REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY *(Leave blank,*	2. REPORT DATE	
	Nov 5-8, 1990	Proceedings of a Workshop

4. TITLE AND SUBTITLE
INNOVATIVE APPROACHES TO PLANNING, SCHEDULING AND CONTROL

5. FUNDING NUMBERS

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
CARNEGIE MELLON UNIVERSITY

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ADVANCE RESEARCH PROJECT AGENCY
3701 FAIRFAX DRIVE
ARLINGTON, VA 2203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
The purpose of this workshop is to bring together researchers, concerned with large, real-world Planning, Scheduling and Control problems, to review the latest research results in the field, to keep the government research community abreast of current technology, and to discuss future directions.

DTIC
ELECTE
APR 27 1995
B

14. SUBJECT TERMS
Scheduling
Control Problems
Planning

15. NUMBER OF PAGES

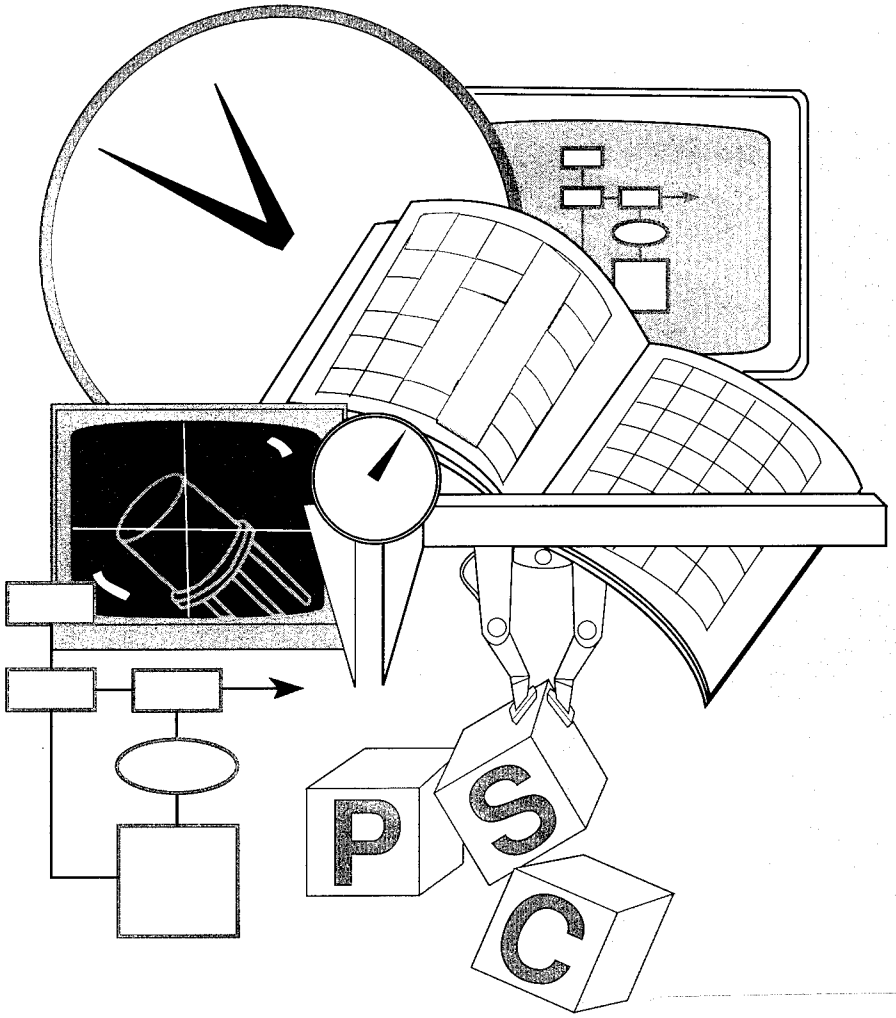
16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
SAR			SAR

P R O C E E D I N G S

Workshop on Innovative Approaches to Planning, Scheduling and Control

November 1990



19950426 050

Sponsored by:

DARPA

Defense Advanced Research Projects Agency
Information Science and Technology Office

Innovative Approaches to Planning, Scheduling and Control

Proceedings of a Workshop
Held at
San Diego, California
November 5-8, 1990

Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Edited by Katia P. Sycara

Sponsored by:

Defense Advanced Research Projects Agency
Information Science and Technology Office

This document contains copies of reports prepared for the DARPA Innovative Approaches to Planning, Scheduling and Control Workshop. Included are Principal Investigator Reports and technical papers from the DARPA sponsored programs.

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

Distributed by

Morgan Kaufmann Publishers, Inc.

2929 Campus Drive

San Mateo, CA 94403

ISBN 1-55860-164-3

Printed in the United States of America

ABCDEFGHIJK-EB-93210

WORKSHOP ON INNOVATIVE APPROACHES
TO PLANNING, SCHEDULING AND CONTROL

TABLE OF CONTENTS

PAGE

AUTHOR INDEX ix

FOREWORD x

PLANNING

Robotic Manipulation Planning with Stochastic Action 3
Alan D. Christiansen, Kenneth Y. Goldberg, *Carnegie Mellon University*

Designing and Analysing Strategies for Phoenix from Models 9
Paul R. Cohen, *University of Massachusetts*

Analogical Planning 22
Diane J. Cook, *University of Illinois*

Rational Distributed Reason Maintenance for Planning and Replanning of Large-Scale Activities (Preliminary Report) 28
Jon Doyle, *MIT Laboratory for Computer Science*
Michael P. Wellman, *USAF Wright R&D Center*

Computational Considerations in Reasoning about Action 37
Matthew L. Ginsberg, *Stanford University*

Issues in Decision-Theoretic Planning: Symbolic Goals and Numeric Utilities 48
Peter Haddawy, University *of Illinois*
Steve Hanks, *University of Washington*

Issues and Architectures for Planning and Execution 59
Steve Hanks, *University of Washington*
R. James Firby, *Jet Propulsion Laboratory*

Envelopes as a Vehicle for Improving the Efficiency of Plan Execution 71
David M. Hart, Scott D. Anderson, Paul R. Cohen
University of Massachusetts

PAGE

Mission Critical Planning: AI on the MARUTI Real-Time Operating System 77
James Hendler, Ashok Agrawala, *University of Maryland*

Responding to Environmental Change 85
Adele E. Howe, Paul R. Cohen, *University of Massachusetts*

Planning in Concurrent Domains 93
Subbarao Kambhampati, Jay M. Tenenbaum, *Stanford University*

Deadline-Coupled Real-Time Planning 100
Sarit Kraus, Madhura Nirkhe, Donald Perlis, *University of Maryland*

Toward an Experimental Science of Planning 109
Pat Langley, Mark Drummond, *Ames Research Center*

Localized Search for Controlling Automated Reasoning 115
Amy L. Lansky, *NASA Sterling Federal Systems*

Transformational Synthesis: An Approach to Large-Scale Planning Applications 126
Theodore A. Linden, *Advanced Decision Systems*

Combining Reactive and Strategic Planning through Decomposition Abstraction 137
Nathaniel G. Martin, James F. Allen, *University of Rochester*

Cooperative Planning and Decentralized Negotiation in Multi-Fireboss Phoenix 144
Theresa Moehlman, Victor Lesser, *University of Massachusetts*

Optimization of Multiple-Goal Plans with Limited Interaction 160
Dana S. Nau, James Hendler, *University of Maryland*
Qiang Yang, *University of Waterloo*

Deferred Planning and Sensor Use 166
Duane Olawsky, Maria Gini, *University of Minnesota*

Exploiting Plans as Resources for Action 175
David Payton, *Hughes Research Laboratories*

PAGE

Responding to Impasses in Memory-Driven Behavior: A Framework for Planning 181
Paul S. Rosenbloom, Soowon Lee, *University of Southern California*
Amy Unruh, *Stanford University*

O-Plan2: Choice Ordering Mechanisms in an AI Planning Architecture 192
Austin Tate, *University of Edinburgh*

Hypergames and AI in Automated Adversarial Planning 198
Russell R. Vane, III, *The Young Guard Company*
Paul E. Lehner, *George Mason University*

Nonlinear Planning with Parallel Resource Allocation 207
Manuela M. Veloso, M. Alicia Perez, Jaime G. Carbonell
Carnegie Mellon University

SCHEDULING

Applying a Heuristic Repair Method to the HST Scheduling Problem 215
Steve Minton, Andrew B. Philips, *Sterling Federal Systems*

Integrating Planning and Scheduling To Solve Space Mission Scheduling Problems 220
Nicola Muscettola, Stephen F. Smith, *Carnegie Mellon University*

Solution of Time Constrained Scheduling Problems with Parallel Tabu Search 231
E. L. Perry, *Ford Aerospace*

Managing Resource Allocation in Multi-Agent Time-Constrained Domains 240
Katia Sycara, Steve Roth, Norman Sadeh, Mark S. Fox
Carnegie Mellon University

Anytime Rescheduling 251
Monte Zweben, *NASA Ames Research Center*
Micheal Deale, Robert Gargan, *Lockheed AI Center*

PAGE

Becoming Increasingly Reactive 459
Tom M. Mitchell, *Carnegie-Mellon University*

A Preliminary Analysis of the Soar Architecture as a Basis 468
for General Intelligence
Paul S. Rosenbloom, *University of Southern California*
John E. Laird, Robert McCarl, *University of Michigan*
Allen Newell, *Carnegie Mellon University*

An Implementation of Indexical/Functional Reference for 490
Embedded Execution of Symbolic Plans
Marcel Schoppers, Richard Shu, *Advanced Decision Systems*

OPIS: An Integrated Framework for Generating and Revising 497
Factory Schedules
Stephen F. Smith, Peng Si Ow, Nicola Muscettola, Jean-Yves Potvin,
Dirk C. Matthys, *Carnegie Mellon University*

AUTHOR INDEX

Agrawala, A.	77
Allen, J. F.	137
Allen, J. A.	301
Anderson, S. D.	71
Basye, K.	271
Bennett, S.	313
Bonissone, P.	379
Carbonell, J. G.	207
Christiansen, A. D.	3
Cohen, P. R.	9, 71, 85
Converse, T.	354
Collinot, A.	263
Cook, D. J.	22
Deale, M.	251
Dean, T.	271, 290
Decker, K.	396
DeJong, G. F.	325, 337
Doyle, J.	28
Drummond, M.	109, 408
Durfee, E. H.	277
Dutta, S.	379
Firby, R. J.	59
Fox, M.	240, 412
Gargan, R.	251
Georgeff, M. P.	284
Gini, M.	166
Ginsberg, M. L.	37
Goldberg, K. Y.	3
Gratch, J. M.	337
Grenfenstette, J. J.	348
Haddawy, P.	48
Hammond, K.	354
Hanks, S.	48, 59
Hart, D. M.	71
Hayes-Roth, B.	263, 422
Hendler, J.	77, 160
Howe, A. E.	85
Ingrand, F. F.	284
Kaelbling, L. P.	408, 483
Kambhampati, S.	93
Kibler, D.	366
Kraus, S.	100
Laird, J. E.	468
Langley, P.	109, 301
Lansky, A. L.	115
Lee, S.	181
Lehner, P. E.	198
Lejter, M.	271
Lesser, V.	144, 396
Linden, T. A.	126
Lowrance, J. D.	439
McCarl, R.	468
McDermott, D.	450
Marks, M.	354
Martin, N. G.	137
Matthys, D. C.	497
Minton, Steve	215
Mitchell, T. M.	459
Moehlman, T.	144
Muscettola, N.	220, 497
Nirkhe, M.	100
Nau, D. S.	160
Newell, A.	377, 468
Olawsky, D.	166
Ow, P. S.	497
Payton, D.	175
Perez, M. A.	207
Perlis, D.	100
Perry, E. L.	231
Philips, A. B.	215
Rosenbloom, P. S.	181, 468
Roth, S.	240
Ruby, D.	366
Schoppers, M.	490
Shu, R.	490
Simmons, R.	292
Smith, S. F.	220, 497
Sadeh, N.	240
Sycara, K.	240, 412
Tate, A.	192
Tenenbaum, J. M.	93
Unruh, A.	181
Vane, R. R III	198
Veloso, M. M.	207
Wellman, M. P.	28
Yang, Q.	160
Zweben, M.	251

FOREWORD

This workshop was organized at the direction of Major Stephen Cross, Program Manager for Research in Planning at the Information Science and Technology Office of the Defense Advanced Research Projects Agency (DARPA). The purpose of the workshop is to bring together researchers concerned with large, real-world Planning, Scheduling and Control problems, to review the latest research results in the field, to keep the government research community abreast of current technology, and to discuss future directions.

All submitted papers were subject to a stringent refereeing process. Each paper was reviewed by three Program committee members or additional selected reviewers. Any variations among reviewers were thoughtfully discussed by program committee members. The Proceedings contains the accepted papers plus invited reports of DARPA Principal Investigators summarizing their work.

The technical chair for the workshop responsible for coordination of the reviewing process, the setting of the workshop program and general organization of the workshop was Dr. Katia P. Sycara of Carnegie Mellon University. Dr. Sycara was assisted by a program committee consisting of:

Pierro Bonissone, General Electric
Tom Dean, Brown University
Northrup Fowler III, Rome Air Development Center
Barbara Hayes-Roth, Stanford University
Phyllis Koton, MITRE
Amy Lansky, NASA/AMES
Drew McDermott, Yale University
Tom Mitchell, Carnegie Mellon University
David Tseng, Hughes AI Research Center

Special thanks are due Dr. Paul Cohen and Dr. Jim Hendler for their assistance with the refereeing process. Local arrangements were coordinated by Ms. Romina Fincher who worked tirelessly for the success of the workshop. The cover was designed by Ms. Mary Jo Dowling of Carnegie Mellon University. Arrangement and editing of these proceedings was done by Katia Sycara of Carnegie Mellon University.

PLANNING

Robotic Manipulation Planning with Stochastic Actions

Alan D. Christiansen **Kenneth Y. Goldberg**

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper addresses automatic planning for task domains with stochastic actions. When actions are viewed as stochastic, the conventional planning paradigm must be modified to search for plans that achieve a goal with high probability. We identify two properties of stochastic actions: *state divergence* and *state convergence*. Considering these properties leads to two planning methods. One method, based on the control of Markov chains, uses exhaustive forward search to find optimal length-bounded plans. The second method uses a heuristic search method to find good but not necessarily optimal plans. We evaluate the planners on a particular robotic manipulation task called tray-tilting. The stochastic model for our actions is derived from physical experiments performed by a real robot. In addition to comparisons of estimated plan quality, we report on actual success rates observed when our robot carried out the computed plans.

1 Introduction

To plan, a robot must be able to predict the outcome of its actions. In many physical manipulation tasks, it is not reasonable to model actions as deterministic mappings from one state to another. Non-determinism can arise from any of a number of sources: A robot may have imperfect effectory and sensory capabilities, or it may have an imperfect model of the task (such as the effects of friction or impact dynamics). A reasonable approach in such cases is to develop *stochastic* action models. In such models, when an action is applied to a particular world state, any of a number of world states may occur. The probability that a particular world state will occur is governed by a probability mapping associated with the action.

Stochastic transition models can be developed either analytically, as in [Erdmann, 1989, Goldberg and Mason, 1990], or empirically, as in [Christiansen *et al.*, 1990]. In the present paper, we focus on the planning problem: given a stochastic model of actions, a known initial state and desired final state, find a sequence of actions—a plan—to reach the desired final state. In this paper, we consider only open-loop plans, where there is no plan execution monitoring. If we start with a known world state, and then execute one or more stochastic actions, the predicted result is not a single state. Instead, the result is a *hyperstate* that assigns "probability mass" to each state such that the mass sums to one. The notion of prediction can be extended so that given a particular hyperstate, we can predict what the "resulting" hyperstate will be when an action is applied. We can think of planning as finding a sequence of actions that transfer probability mass from the initial state to the desired final state.

Planning is affected by two contrasting properties of stochastic actions. Some actions have the property that a single initial state can map onto more than one final state. We say that the action exhibits *state divergence* when, in executing the action, there is a tendency to distribute probability mass. Alternatively, some actions have the property that multiple initial states map into a single final state. In these cases, we say that an action exhibits *state convergence*—it tends to focus distributed probability mass into a state. An action can exhibit both divergence and convergence if it spreads out probability mass from one state while combining mass from other states. Of course, an action may exhibit neither state divergence nor state convergence.

A stochastic planner evaluates plans by tracking probability mass as each plan proceeds. The objective is to find a plan that transfers a maximal amount of probability mass into the desired final state. This can be accomplished in different ways. One way is to use actions that cause state divergence followed by actions that cause convergence toward the desired state. Another way to plan is to *avoid* actions that cause state divergence. These two ways of transferring probability mass suggest two planning methods: Method I propagates hyperstates, using a forward exhaustive search (to a fixed plan length) to find a plan most likely to produce the desired goal. Method II propagates only the most likely state, using a backward best-first search to find a plan that maximizes a lower bound on the probability of reaching the goal.

Planning with stochastic actions has been considered by several researchers previously. Stochastic approaches to domain-independent planning have been described in the AI literature [Feldman and Sproull, 1977, Cheeseman, 1988, Russell and Wefald, 1988, Pearl, 1988, Drummond and Bresina, 1990, Hansson *et al.*, 1990], in texts on stochastic control [Bertsekas, 1987] and in texts on decision theory [Berger, 1985]. The present paper focuses on the stochastic properties of physical actions and the demands that these properties place on the planning problem.

In this paper, we describe our experiences with applying two planning methods to a particular manipulation task: tray-

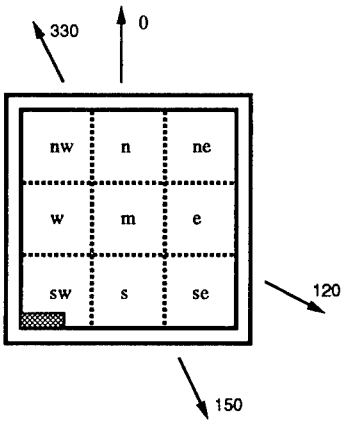


Figure 1: The tray (viewed from above), its states, and examples of tilt azimuths.

tilting. By using a concrete example, we hope to illustrate characteristics that are common in robot manipulation. We use a real robot system to generate the stochastic action model and to test the resulting plans. We describe the two planning methods in detail and analyze the computational complexity of each method. We compare the plans produced by each method and the planning time required by each planner. We conclude the paper by revisiting the concepts of state divergence and state convergence and their importance to planning with stochastic actions.

2 The Manipulation Task

The physical task that is considered in this paper is called tray-tilting. The object of the task is to manipulate planar objects in a walled tray by tilting the tray so that the objects slide to a desired position and orientation. The physics of this task are subtle and include frictional and impact effects. Tray-tilting has been studied before, both analytically [Erdmann and Mason, 1988, Taylor *et al.*, 1987] and empirically [Christiansen *et al.*, 1990].

Our robot arm has been programmed to perform tilts in any desired direction. As the tray is tilted by the robot, gravity acts on the object and causes it to slide. For the purposes of this paper, we define tilts by a parameter which we call the tilt *azimuth*. This angle is the direction, in the plane of the tray bottom, of the force of gravity on the object during the tilt. (See Figure 1.) Other parameters of the tilt, such as steepness of the tilt and tilt speed, have constant values during all tilts. Although there is a an infinite number of possible tilt azimuths, and therefore an infinite number of tilts, we sampled this space of azimuths at 30 degree intervals. In all cases, the robot restricted its actions to one of these 12 tilts.

In addition to defining tilts by a single parameter, we have simplified the description of task *state* by discretizing the object's position and orientation. As indicated in Figure 1, we have divided the tray (conceptually) into nine regions, corresponding to the four corners, the four sides, and the middle of the tray. We have given each of these regions symbolic names, as shown in Figure 1. We describe an object's position by one of these names—the name of the region in which the object's center is located. A camera above the tray and an industrial image processor provide this position information. We describe orientations by a similar discretization—our objects are described as being either horizontal *(h)* or vertical *(v)*, depending on the orientation of the object's major axis, as reported by our vision system. The rectangular tile of Figure 1 would be classified as *(sw h)*. This discretization is coarse and somewhat arbitrary. However, states defined by this decomposition correspond to qualitatively distinct physical configurations.

3 Developing a Stochastic Model from Observations

Given a set of observations from physical experiments, what is an appropriate stochastic model of actions? If the probabilities of future states depend only on the current state, then we can use a *Markov chain* to represent the actions. We represented each tilting action u with a stochastic transition matrix, $\mathbf{P}_u$, where p_{ij} is the (conditional) probability that the system will be in state j after action u is applied to state i. We assume that the set of states and the set of actions are finite.

In the physical experiments, each observation consists of an initial state, a tilting action, and a final state. For each tilting action u, consider the matrix $\mathbf{X}_u$, where x_{ij} is the number of observations with initial state i and final state j. Given an observation matrix $\mathbf{X}_u$, how do we generate a stochastic transition matrix $\mathbf{P}_u$?

One possible approach is to use the observed frequencies. The difficulty is that some observed transition frequencies may be zero. For such a transition, it isn't clear whether the transition is truly impossible—maybe the transition just has a low probability and hasn't yet been observed. A standard approach from statistical estimation is to use the following estimator for each action u,

$$\hat{p_{ij}}(\mathbf{x}) = \frac{\alpha_{ij} + x_{ij}}{\sum_j \alpha_{ij} + x_{ij}}, \qquad (1)$$

where the numbers α_{ij} for $i = 1, 2, ..., k$ are Dirichlet parameters based on *a priori* assumptions. This is equivalent to a Bayes' estimator using a squared error loss criterion, see [DeGroot, 1970].

We could set $\alpha_{ij} = 1.0$ to represent the prior assumption that the conditional probability distribution is uniform: after an action is applied, the system is as likely to be in any one state as in any other. For the tray tilting problem, we set $\alpha_{ij} = .01$ to represent our prior assumption that the conditional probability distribution for each action will not be uniformly distributed, but will in fact be skewed toward some subset of states.

We generated 2000 tilt azimuths by random selection (with replacement) from the set of twelve azimuths described previously. Our robot performed the corresponding tilts of the tray, and observed the outcome of each tilt. The $\mathbf{X}_u$ matrices were defined by this data and we used equation 1 to generate the corresponding stochastic transition matrices.

4 Planning with Method I

We now turn to the planning problem: given a stochastic model of actions, a known initial state and desired final state, find a sequence of open-loop tilting actions—a plan—to reach the desired final state. We will present two planning methods and their resulting plans. In this section we describe a method

that maximizes the probability of reaching the desired final state.

Method I is based on the control of Markov chains. Consider a system with finite state space. Let us refer to a probability distribution on this state space as a *hyperstate*. Each action is a mapping between hyperstates. A plan is a sequence of actions and hence is also a mapping between hyperstates. For a given initial hyperstate, each plan generates a final hyperstate. To compare plans, we compare their final hyperstates. To rank hyperstates, we introduce a function that maps each hyperstate into a real number called its *cost*. The best plan is the plan with the lowest cost. See [Goldberg, 1990] for details on Method I and its application to programming parts feeders.

Let the vector λ refer to a hyperstate. In a Markov chain, the hyperstate that results from applying action u to λ is given by post-multiplying λ by $\mathbf{P}_u$,

$$\lambda' = \lambda \mathbf{P}_u.$$

A *plan* is a sequence of actions. The outcome of a plan is the composite effect of its inputs; the transition matrix for a plan is the product of the transition matrices of its actions. That is, for a plan consisting of the sequence of actions $< u_1, u_4, u_2 >$, the final hyperstate is

$$\lambda' = \lambda \mathbf{P}_{u_1} \mathbf{P}_{u_4} \mathbf{P}_{u_2}.$$

For the tray tilting task we are given a known initial state and desired final state. In this case the initial hyperstate is a vector with a 1 corresponding to the initial state and zeros elsewhere. Each action (and hence plan) defines a final hyperstate using the stochastic transition matrices described in Section 3. The cost function depends on the desired final state. If we want to reach state i, let

$$C(\lambda) = -p_i,$$

so that the minimum cost hyperstate corresponds to the highest probability that the system is in state i. Note that there may be more than one minimum-cost hyperstate.

To find the best plan, we consider all plans and find one with minimum cost. The difficulty is that there is an infinite number of plans to consider. So finding the best plan can take a long time, even on a supercomputer. We compromise and ignore plans longer than some cutoff threshold. This threshold depends on how much time we have and how fast we can evaluate plans, which in turn depends on how fast we can multiply matrices. In our case we considered all plans with length ≤ 3 to find a plan with minimal cost.

5 Planning with Method II

An alternative planning method is based on heuristic graph search. The transition probability matrices described in Section 3 define a graph, or more correctly, a multi-graph. The vertices of the graph are the *states* that were defined previously (the object configurations) and the graph's edges are the *actions* that cause one state to be changed to another. Associated with each edge is an estimate of the probability that the action will provide the indicated state change. The graph is a multi-graph because there may be multiple edges possible between a single pair of vertices.

Figure 2 shows a portion of the graph defined by our data. Above each edge is a tilt azimuth. Below each edge is the

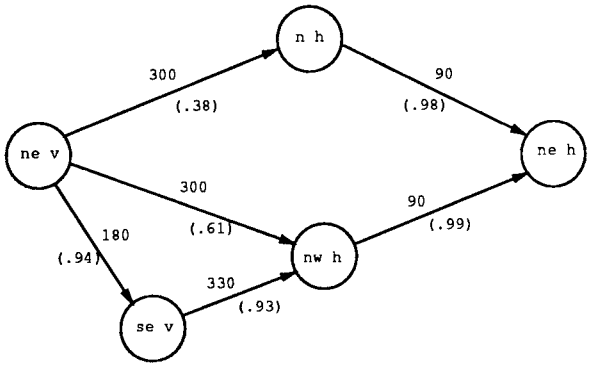


Figure 2: A portion of the robot's task model.

associated transition probability. Note that when the object is in state *(ne v)*, and a tilt of azimuth 300 is applied, the object's new state is uncertain. It is estimated that it has a 61% chance of achieving the *(nw h)* configuration, and a 38% chance of moving to the *(n h)* configuration. Of course, not all transitions are shown in Figure 2. The sum of the probabilities for a particular action executed from a particular state must be equal to one.

The planning problem of finding a sequence of actions that will transform a given current state to a desired goal state can now be viewed as finding a path in the graph that links the two states. Since our model of state transitions is stochastic, we naturally wish to find a plan (path) with high probability of achieving the goal. When a plan is executed, several actions are performed in succession. The action probabilities for the plan steps must be combined to give an estimate of the probability of success for the whole plan.

A *lower bound* on the probability that a plan will reach a desired state can be computed by multiplying the probabilities along one path between initial and desired states. Such a computation produces only a lower bound because there can exist multiple paths between the initial and desired states that share the same sequence of actions. We can find a plan that maximizes this lower bound by using shortest-path graph search. This leads to an efficient algorithm for finding plans that we will call Method II.

Method II is an example of *uniform-cost search* [Pearl, 1984]. While uniform-cost search is usually cast as finding a minimum cost path in a graph (where the cost of a path is the sum of the costs of the edges in the path), it is easy to adapt the algorithm to our problem by changing the evaluation function. In finding a minimum cost path in a graph, one desires to find a sequence of edges linking the start and goal vertices such that the sum of the costs of those edges is as small as that of any other such sequence. Our problem is to find a sequence of edges linking the start and goal vertices such that the *product* of the *probabilities* is as *large* as any other such sequence. It is possible to map our problem exactly onto a shortest path problem by transforming the values associated with edges. For each edge probability p, we consider a new edge value $(-\log p)$. In this way, we guarantee that finding a shortest path in the transformed graph will correspond exactly to a maximum product probability path in the original graph.

Since Method II only considers single paths, it sometimes

misses good plans. Consider when Method II is applied to the problem of getting from state *(ne v)* to *(ne h)*, and its stochastic action model is defined by Figure 2. Method II returns the plan *(180 330 90)*. This path's product of action probability estimates is larger than any other path in the graph. Method II has found a good plan, but note that the plan *(300 90)* would be better. Under this plan, the configuration achieved after the first tilt is very likely to be either *(n h)* or *(nw h)*, and the second tilt is highly likely to achieve the goal no matter which intermediate configuration was actually achieved. The combination of paths yields a high probability even though neither single path has higher probability than the path *(180 330 90)*. Note that Method I would return the better plan in this case but we shall show that Method I requires significantly more computational effort to achieve this rigor.

6 Computational Complexity

Finding optimal open-loop plans with stochastic actions has been shown to be *NP-Complete* [Papadimitriou and Tsitsiklis, 1987]. This suggests that an algorithm with good worst-case running time may not exist.

Recall that Method I considers all plans up to some length limit, k. Let n be the number of states and m be the number of actions. There are m^k k-step plans. We can visualize the search for an optimal strategy as proceeding through a tree, where the root node contains the initial hyperstate and has a branch for each action in the action space. Each branch leads to a new hyperstate which in turn has branches for each action. We expand the tree to some fixed depth (horizon) and select the optimal path. To generate each node in the tree we must perform $O(n^2)$ multiplications. The total time for finding the best k-step plan is $O(n^2 m^k)$.

Recall that Method II is based on uniform-cost graph search. Because the edge values $(-\log p)$ are nonnegative, uniform cost search on this problem has a monotone heuristic, which implies that whenever a node is expanded, a best path to that node has been found. This means that nodes will never have to be re-expanded, and in a finite graph of n vertices, there can be no more than n node expansions. If there are m actions available at each state, then m is the maximum number of edges that can be between any pair of vertices. Therefore, the maximum amount of work that will have to be done at each node expansion is $O(nm)$, and the complexity of the algorithm is $O(n^2 m)$. The implemented algorithm deals with probabilities and products directly instead of transforming the problem to a shortest path problem, but it performs analogous steps to those of uniform cost search on the transformed graph. So, the complexity of Method II is also $O(n^2 m)$.

7 Empirical Comparisons of the Planners

The two planning methods were implemented in Common Lisp. To explore performance tradeoffs, we performed several experiments with the tray-tilting task. In all experiments reported in this paper, we used an 11 inch square tray and a 1 by 3 inch rectangular tile (Figure 1).

In Section 2, we described the state space of possible tile configurations for the tray-tilting tasks. There are nine possible discrete positions and two discrete orientations, making a total of eighteen possible configurations. It turns out that only twelve of these configurations occur in practice. When the tile is in one of the four corners, both orientations are possible, for a total of eight configurations. The two configurations in the middle of the tray are impossible. For the remaining positions—where the rectangular tile is against a side of the tray—only the orientation where the tile's major axis parallels the tray wall occurs in practice. This adds four more possible configurations, for the total of twelve.

Problem		Method I		Method II	
Start	Goal	Plan	P	Plan	P
(n h)	(ne h)	(90)	.98	(90)	.98
(n h)	(ne v)	(240 180 60)	.97	(240 180 60)	.96
(n h)	(e v)	(120)	.97	(120)	.97
(n h)	(se h)	(90 180 90)	.98	(270 150)	.96
(n h)	(se v)	(150 30 180)	.98	(240 0 120)	.97
(n h)	(s h)	(240 150)	.97	(240 150)	.97
(n h)	(sw h)	(270 270 180)	.97	(270 180)	.96
(n h)	(sw v)	(240 180)	.98	(240 180)	.97
(n h)	(w v)	(240)	.99	(240)	.99
(n h)	(nw h)	(270 30 270)	.97	(270)	.97
(n h)	(nw v)	(240 0)	.98	(240 0)	.98
...					
(sw v)	(n h)	(120 300 30)	.62	(60 180 240 300 30)	.75
...					
(ne v)	(n h)	(270 30 60)	.67	(180 240 300 30)	.75
(ne v)	(ne h)	(300 90)	.98	(180 330 90)	.87
...					

Table 1: Comparison of plans generated by the two methods.

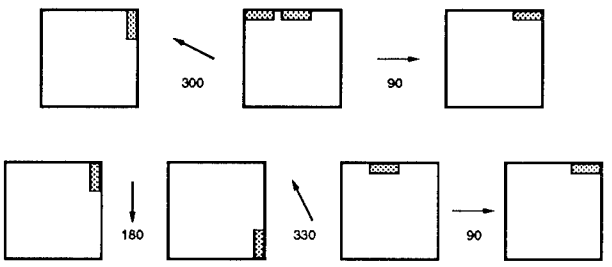


Figure 3: Two tray-tilting plans. The plans are read from left to right, with intermediate states shown as views from above the tray. Tray tilt directions are shown between the states that they link. The problem is to re-orient the object from vertical to horizontal, leaving the object in the upper right corner of the tray. Above: Method I's plan. Below: Method II's plan.

For twelve possible tile configurations, there are 144 pairs of configurations defining start state and goal state. Let us refer to each of these pairs as a *problem*. There are 132 non-trivial problems for our tray domain. (The twelve problems with start state and goal state equal to each other are trivial, since a null plan always solves the problem.) We ran the two planners on each of the 132 non-trivial problems. Table 1 lists some of the resulting plans.

For 51 (39%) of the problems, the two planners produced identical plans. In many other cases, the two planners produced similar or symmetrically equivalent plans. In nearly every case, the estimated success ratios of the two methods were within a few percentage points of each other. Note that for plans of length ≤ 3, the estimated success ratio for Method II is less than or equal to that for Method I, since in those cases, Method II's estimate is a lower bound on the Method I estimate.

On some problems, the planners did not agree. In Table

Start State	Goal State	Planning Approach	Best Plan	Estimated Successes	Measured Successes
(ne v)	(n h)	Method I	(270 30 60)	134 (0.67)	170 (0.85)
(ne v)	(n h)	Method II	(180 240 300 30)	150 (0.75)	192 (0.96)
(ne v)	(ne h)	Method I	(300 90)	196 (0.98)	198 (0.99)
(ne v)	(ne h)	Method II	(180 330 90)	174 (0.87)	171 (0.855)

Table 2: Summary of execution trials for four plans. The estimated and measured success columns show the number of successful plan executions (out of 200 trials) along with the corresponding decimal fractions.

1 we have listed three such problems (the last three entries). Since Method I was limited to searching for plans of three steps or shorter, there were occasions when Method II was able to search deeper and find a superior plan. The four and five step plans listed in the table are two such examples. On some problems Method I was able to find a superior plan within its length bound by taking advantage of state convergence. The plan *(300 90)* is an example. In this problem, the initial state is the northeast corner of the tray in a vertical configuration, and the desired goal is the same corner, but in a horizontal configuration. Figure 3 shows a trace of the anticipated object locations as each of the plans proceeds. Method II (below) finds an adequate plan: It tilts the tray at 180, moving the object to the *(se v)* configuration. Then it tilts at 330, moving the object nominally to *(n h)*. Finally, it tilts at 90, moving the object to *(ne h)*. This plan is good, but it can fail by (for example) the tilt 330 not aligning the object horizontally. Method I's plan (above) is better since its intermediate hyperstate aligns the object horizontally but causes divergence of its position. The second tilt of the plan causes all such intermediate states to converge to *(ne h)*.

Both planners were implemented as compiled Common Lisp programs and were tested on the same computer, a Sun 3/280. Method I's search was truncated at depth three, and so all of its plans were between one and three steps in length. Method II's plans were all between one and six steps in length. Over the 132 problems, Method I took an average of 62 seconds real time per problem, with a standard deviation of 2.8 seconds. The average time for Method II on the same problems was 0.46 seconds, with a standard deviation of 0.62 seconds. The minimum planning time for Method I was 59 seconds and the maximum was 87 seconds. The minimum planning time for Method II was 0.040 seconds and the maximum was 6.5 seconds. The average length of plans found by Method I was 2.4 tilts with a standard deviation of 0.75 tilts. The average plan length for the Method II planner was also 2.4 tilts, but with a standard deviation of 1.2 tilts.

7.1 Physical Test of Resulting Plans

We tested the last two problems of Table 1 with the robot. These two problems represented cases where the methods produced significantly different plans. In terms of predicted success ratios (probability of reaching the goal), Method II found a better plan for the first problem because it was able to search deeper. In the second problem, Method I found a better plan because it considered state divergence and state convergence.

Table 2 summarizes the head-to-head competition. Each of the four plans was executed 200 times. In the first problem, the estimated success ratio was lower than the observed success ratio. Evidently the estimates for these plans were low because insufficient data had been collected to predict the true outcomes of the actions comprising the plans. In the second problem, the estimated and observed success rates were nearly equal. In both cases, the plan with higher estimated success rate performed better.

8 Discussion

We began this paper by acknowledging that it is not always possible to predict the exact outcome of actions in robotic manipulation due to factors such as control error, friction, and dynamics. In response, we considered *stochastic* models of action. When actions are viewed as stochastic, the conventional planning paradigm must be modified to search for plans that achieve a goal with high probability. We can view planning as finding a sequence of actions that transfer probability mass from the initial state to the desired final state.

Planning is affected by two contrasting properties of stochastic actions. We say that an action exhibits *state divergence* when it distributes probability mass. We say that an action exhibits *state convergence* when it focuses probability mass. These properties affect the choice of planning method. We considered two methods.

Method I, based on the theory of Markov chains, uses a forward exhaustive search (to a fixed plan length) to find a plan most likely to produce the desired goal. Method II uses a backward best-first search to find a plan that maximizes a lower bound on the probability of reaching the goal. Method I keeps track of *all* probability mass as it evaluates plans, monitoring both state divergence and state convergence as the plan progresses. However, keeping track of all probability mass requires substantial computation; Method I can only consider short plans. Method II keeps track of only the most probable trajectory, monitoring state divergence and ignoring state convergence. Accordingly, Method II is faster and can consider longer plans, but it sometimes misses good plans.

Which method is better? It depends on the available planning time and the degree of state divergence in the available actions. Method I is generally much slower than Method II. The exponential factor in Method I's complexity makes it impractical when a large number of actions are available at each state. If we are not concerned with computation time, Method I is to be preferred. Method II is faster, and is preferable when we can avoid or minimize divergence.

Sometimes divergence is unavoidable. For example, consider a tray-tilting problem where we want to achieve a particular object configuration but the initial configuration is unknown. This is equivalent to a highly divergent action occuring before plan execution. It is as if someone randomly shook the tray prior to the robot carrying out its plan. Consider a case where the initial hyperstate reflects a uniformly distributed state probability. Method I can find the best plan for getting the tile into the northeast corner. Method II can't solve this problem. As another example, consider programming a robot to pick up parts on an assembly line. Each time a part arrives, its initial position and orientation relative to the robot will be slightly different. This results from a divergent action earlier in the assembly line.

Sometimes state divergence is desirable. For example, it may be be more efficient to allow divergence followed by ef-

fective convergence. Consider a typical plan for causing two gears to mesh: we jiggle the gears at random (divergence), until the gears fall into alignment (convergence). This plan is more efficient than trying to avoid divergence by carefully aligning the gears. The process of deliberately incurring divergence is known as *randomization*, and has been recently identified as an important component of manipulation [Erdmann, 1989].

For most problems in the tray-tilting domain, Method II worked as well as Method I. Studying the resulting plans, we discovered it was often possible to avoid significant state divergence. For a few problems, it was better to incur state divergence followed by state convergence. An example of such a problem was given in Figure 3. On a problem like this, Method I is superior.

We believe that the study of planning with stochastic actions is a research area that will become increasingly important as AI planning techniques are employed in robotics. It seems possible to build an efficient planner that considers some state convergence. Instead of tracking the probabilities for every state, as Method I does, or tracking the probability of only the most likely state, as Method II does, maybe the hypothetical planner could track the probabilities of the two or three most likely states. Like Method II, the hypothetical planner would, in most cases, find a plan of near-optimal quality. Even more desirable would be an approximation algorithm, where we might be able to guarantee, for all problems, that the plans produced would be suboptimal by no more than a fixed constant factor.

Acknowledgements

This work was supported by the National Science Foundation under grant DMC-8520475 and by an AT&T Bell Laboratories Ph.D. Scholarship supporting the first author. We thank Matt Mason for suggesting this collaboration. We also thank Matt, Rob Kass, Kevin Lynch, Tom Mitchell, Steve Shreve, and Manuela Veloso for helpful comments.

References

[Berger, 1985] James O. Berger. *Statistical Decision Theory and Bayesian Analysis*. Springer-Verlag, 1985.

[Bertsekas, 1987] Dimitri P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, 1987.

[Cheeseman, 1988] Peter Cheeseman. An inquiry into computer understanding. *Computational Intelligence*, 4(1), February 1988.

[Christiansen *et al.*, 1990] Alan D. Christiansen, Matthew T. Mason, and Tom M. Mitchell. Learning reliable manipulation strategies without initial physical models. In *IEEE International Conference on Robotics and Automation*, pages 1224–1230, May 1990.

[DeGroot, 1970] Morris H. DeGroot. *Optimal Statistical Decisions*. McGraw-Hill, 1970.

[Drummond and Bresina, 1990] Mark Drummond and John Bresina. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *AAAI-90*, 1990.

[Erdmann and Mason, 1988] Michael A. Erdmann and Matthew T. Mason. An exploration of sensorless manipulation. *IEEE Journal of Robotics and Automation*, 4(4):369–379, August 1988. Originally appeared in 1986 IEEE International Conference on Robotics and Automation.

[Erdmann, 1989] Mike A. Erdmann. *On Probabilistic Robot Strategies*. PhD thesis, MIT, 1989.

[Feldman and Sproull, 1977] J. A. Feldman and R. F. Sproull. Decision theory and artificial intelligence ii: The hungry monkey. *Cognitive Science*, 1:158–192, 1977.

[Goldberg and Mason, 1990] K. Y. Goldberg and M. T. Mason. Bayesian grasping. In *International Conference on Robotics and Automation*. IEEE, pages 1264–1269, May 1990.

[Goldberg, 1990] Kenneth Yigal Goldberg. *Stochastic Plans for Open-Loop Manipulation*. PhD thesis, CMU School of Computer Science, 1990. To appear.

[Hansson *et al.*, 1990] Othar Hansson, Andrew Mayer, and Stuart Russel. Decision-theoretic planning in bps. In *AAAI Symposium on Planning*, 1990.

[Papadimitriou and Tsitsiklis, 1987] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of markov decision processes. *Mathematics of Operations Research*, 12(3), August 1987.

[Pearl, 1984] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[Pearl, 1988] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[Russell and Wefald, 1988] Stuart Russell and Eric Wefald. Decision-theoretic control of reasoning: General theory and an application to game playing. Technical Report UCB/CSD 88/435, UC Berkeley, October 1988.

[Taylor *et al.*, 1987] Russell H. Taylor, Matthew T. Mason, and Kenneth Y. Goldberg. Sensor-based manipulation planning as a game with nature. In *International Symposium on Robotics Research*, pages 421–429, August 1987.

Designing and Analysing Strategies for Phoenix from Models

Paul R. Cohen
Experimental Knowledge Systems Lab
Department of Computer and Information Science
University of Massachusetts, Amherst, Mass 01003
email: cohen@cs.umass.edu

Abstract

This paper illustrates how aspects of the design of a planner can be derived from a formal model of the planner's environment and the desired planner behaviors. Specifically, I show how the order of execution of multiple fire-fighting plans is determined by a model of the dynamics of the Phoenix environment. More generally, I introduce an *ecological* approach to the design and analysis of intelligent agents. Some tenets of this approach are familiar; for example, the behavior of an agent arises from its interactions with its environment (e.g., [12, 11, 6, 1, 8]); and some agent designs are better adapted to particular environments than others (e.g., [9, 7, 3, 10]). My purpose here is not to discuss these foundations, but to demonstrate how models of the interactions between an agent and its environment can facilitate design and analysis.

1. An Ecological View

Our goal is to understand the functional relationships between three complex structures: the architecture and knowledge of agents, the structure and dynamics of environments, and the behaviors that result from the interactions between agents and their environments (Fig. 1). Borrowing from the literature on animal behavior, we call these relationships the *behavioral ecology* of an agent.

The terms *agent*, *architecture*, *environment*, and *behavior* are open to interpretation. Without implying that our interpretations are consensual, our view is that agents sense their environments and decide autonomously how to act, and that these actions, moderated by the environment over time, produce behavior. The agent's architecture is a collection of sensors, effectors, and internal data structures and processes. The ecological view in Figure 1 suggests seven research activities that AI researchers currently engage in or would like to engage in:

Environment assessment: Determining which aspects of the environment must be represented in a model for design and analysis

Modelling: Formally specifying the functional relationships from which to predict behavior, given the architecture and environment of an agent.

Design: Inventing or adapting architectures that are predicted to behave as desired in particular environments. In addition, redesign involves modifying a design when it is shown, by way of a model, to perform less well than it might.

Prediction: Inferring from the functional relationships in a model how behavior will be affected by changing the architecture of the agent or its environment.

Experiments: Testing the veracity of predictions by running the agent in its environment.

Explanation: Finding the source of incorrect predictions in a model, and revising the model, when unexpected behaviors emerge from the interactions between an agent and its environment.

Generalization: Whenever we predict the behavior of one agent in one environment, we should ideally be predicting similar behaviors for agents with related architectures in related environments. In other words, our models should generalize over architectures, environmental conditions and behaviors.

In the following sections I will illustrate each of these activities. But first, some disclaimers, an opportunity to say what this paper is *not* about. This

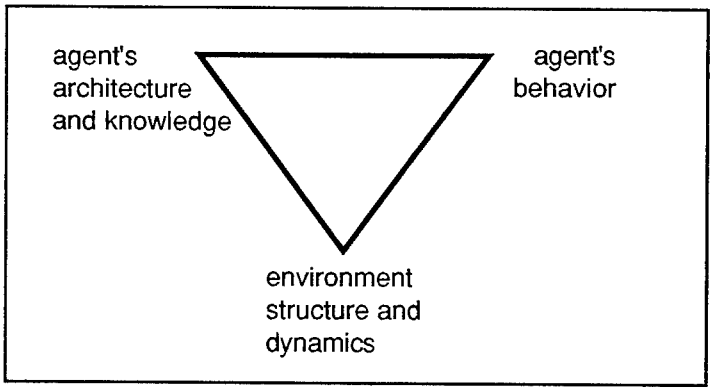


Figure 1. Three components of an agent's behavioral ecology

is not a collection of formal models from which to derive the design of the entire Phoenix planner, complete with predictions and experiments. Such models are the long term goal of the Phoenix project. Nor does this paper present models of the "innards" of the Phoenix planner—the "fixed part," or the "architecture" of the planner. The Phoenix planner relies on stored skeletal plans for rough guidance about how to fight fires.These plans—the "flexible part" or "knowledge" of Phoenix planners—are the subject of this paper. I show how to determine the order in which to fight several simulated forest fires. While this question is not intrinsically interesting, it has provided an opportunity to develop the design methodology represented by the seven steps, above. And the methodology itself *is* interesting, or so it seems to me, because it bases design decisions on formal models from which specific predictions about performance can be derived. This is qualitatively different (and better, it seems to me) than the modal AI design methodology of basing design decisions on the intuitions of the designer and making *no* specific predictions about performance.

This paper has been written as a record (and a demonstration) of this model-based design methodology, as it is applied to a single design decision for the Phoenix planner. In the following sections, I describe the decision, derive a model, prove that the correct decision for an arbitrary number of simulated forest fires is to fight the youngest fire first, predict how the Phoenix planner will actually perform, explain why the predictions are incorrect in some cases, and describe how the design of the Phoenix planner has been modified (and improved) as a result. This final step has involved modifying the architecture, or fixed part of the Phoenix planner.

2 A Design Problem in Phoenix

Phoenix is an environment—a simulation of forest fires—and a collection of simulated agents [4]. Many factors that affect forest fires also affect the fires in the Phoenix environment, including wind speed and direction, elevation gradients, fire temperature and flame height, ground cover, and natural and artificial boundaries such as rivers and roads. Fires, which are implemented as cellular automata, "burn" an array that represents the topographical features of Yellowstone National Park. Nontopographical factors (e.g., weather) are set and changed manually, or randomly, or by prespecified scripts. Phoenix agents contain fires by cutting fireline around them. Several agents are usually required to contain a fire, so they must coordinate or be coordinated. Currently, a single fireboss agent directs several semi-autonomous bulldozer agents, which plan individually how to carry out the directions. The current Phoenix agent architecture includes sensors, effectors, reflexes, and a cognitive scheduler. Reflexes respond immediately to situations such as encroaching fire. The cognitive scheduler coordinates all the agent's activities except its reflexes, including selecting skeletal plans, expanding plans into subplans, assigning appropriate execution methods to actions, monitoring, and fixing plans when actions cannot be executed.

Let us consider a situation in which several fires are burning simultaneously. The Phoenix fireboss decides to send all its resources to each fire in sequence, rather than dividing resources between the fires (this has been shown to be the best strategy in most cases). The question remains, in which order should the fires be fought?

2.1 Environment assessment.

Environment assessment is the informal process of deciding which aspects of an environment have important effects on particular behaviors, which can be safely left out, and which have unknown effects. The behavior that interests us here is the order in which multiple fires are fought, and the consequent loss of acreage of forest. A good ordering minimizes the loss of acreage. After watching fires in Phoenix for a long time, one gets a sense of the factors that affect how much area burns, and, thus, the factors that influence the fireboss's decision about the order in which to fight the fires. The factors that should most influence the ordering decision are probably wind speed, ground cover, the initial size of the fires, and the force one can bring to bear on the fires (see Table 1). It also matters whether bulldozers work directly at the fire edge (direct attack), or at a distance (indirect attack). The direction of the wind probably does not affect the ordering decision, nor do small fluctuations in wind speed and direction, which cancel out over time. Fires in the Phoenix environment generally have lumpy elliptical shapes, but the exact shape probably has little effect on the ordering decision, and it probably does not matter where bulldozers start working. Natural and artificial boundaries are currently exploited by the Phoenix planner, but it is unclear how these should affect the ordering decision. Another uncertainty is whether the fire perimeter can ever increase at a nonlinear rate that is high enough to affect performance. Preliminary data tell us that perimeter growth is linear, but when convective fires are implemented in Phoenix, they will probably influence the ordering decision.

This assessment leads to some assumptions, and then to a model. I will assume that fireline cut by bulldozers is contiguous and its position around the fire is irrelevant. A will also assume that the fire grows by the same amount at all points on its perimeter that are not constrained by fireline or other boundaries, and that travel time between fires is negligible.

2.2 Modelling

It turns out that if the Phoenix fireboss can minimize the total amount of time that agents require to contain a sequence of fires, it will also minimize the total area burned [3].

Fire growth is roughly linear. The radius of the fire grows by a constant at each time unit:

$$r(t+1) = r(t) + k$$

A fire is usually not noticed immediately by the Phoenix fireboss because its subordinate watchtowers require a significant interval to scan an area. Also,

Probably Influences Fireboss's Ordering Decision	**Probably Doesn't Influence Decision**	**Unknown or Uncertain Influence on Decision`**
Wind speed	Wind direction	Boundaries
Ground cover	Shape of the fire	Nonlinear fire growth
Elevation gradients	Where on the perimeter	
Direct or indirect attack	bulldozers work first	
Initial size of fires	Fluctuations in wind	
Number of bulldozers	speed and direction	

Table 1. Assessment of factors with respect to the fireboss's ordering decision.

bulldozers need time to reach the fire. So by the time bulldozers begin working on a fire, it already has a significant perimeter, which I denote p_0.

Now, imagine an agent constructs a circular line at a constant rate around a growing fire, as shown in Figure 2, so that when it has finished constructing the line, the fire is completely within the line. This is called indirect attack.The time it takes to construct such a line depends on p_0, the initial perimeter of the fire; s, how fast the fire grows; and C, how fast bulldozers can construct fireline. The fire is contained by the line when

$$t = \frac{p_0}{C - s} \qquad (1)$$

or, if we build fireline so that a "corridor" remains between the perimeter of the fire and the fireline, such that the length of the fireline is r times the length of the fire perimeter, then

$$t = \frac{p_0}{\frac{C}{r} - s} \qquad (2)$$

The situation is as shown in Figure 3, which plots the perimeter of the fire (y axis) against time (x axis). As long as C > s, the agent's line will eventually contain the fire. If r = 1, the containing fireline and the fire perimeter will equal p_{f1}. If r > 1, then the length of the fireline and the containing perimeter will be p_{fr} and r p_{fr}, respectively. At some time in the past, the fire was very small, perhaps just a single tree or patch of grass. This point, Q, proves significant later.

Equations 1 and 2 can be extended to multiple fires that are fought in succession (and also to fires fought simultaneously, though I will not describe that here). For the immediate discussion, I will work with Eq. 1, assuming a "snug" fit between the fireline and the fire. In the Phoenix environment, different fires have different initial perimeters and spreading rates. I denote the initial perimeters of fires $f_1,f_2,...,f_n$ as $p_0(f_1)$, $p_0(f_2)$,...$p_0(f_n)$, the rates of spread as $s(f_1)$, $s(f_2)$,...,$s(f_n)$. I denote the time required to contain fires $f_1,f_2,...,f_n$ in that order as $T(f_1,f_2,...,f_n)$. It is easy to show that

$$T(f_1,f_2,...,f_m,f_n) = t(f_n) + (1 + g(f_n))\, T(f_1,f_2,...,f_m) \qquad (3)$$

where $g(x) = s(x) / (C - s(x))$. The derivation is in the Appendix.

One can see from Eq. 3 that the order in which one fights fires affects the time required to fight them. Imagine two fires, a and b, with $p_0(a) = 100$, $p_0(b) = 150$, $s(a) = 12$, and $s(b) = 4$; and the maximum rate at which bulldozers can cut fireline is $C = 20$. If we fight fire a first (i.e., assign $\{a \rightarrow f_1, b \rightarrow f_2\}$) then $T(a,b) = 25$. Alternatively, if we fight b first (i.e., assign $\{b \rightarrow f_1, a \rightarrow f_2\}$) then $T(b,a) = 35.9$.

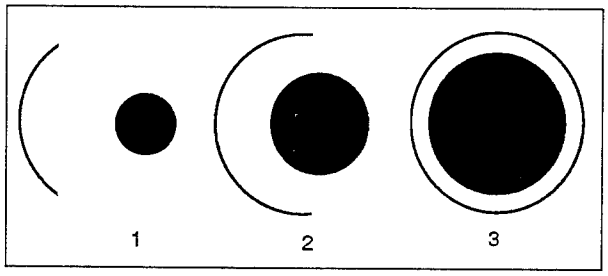


Figure 2. A schematic view of indirect attack.

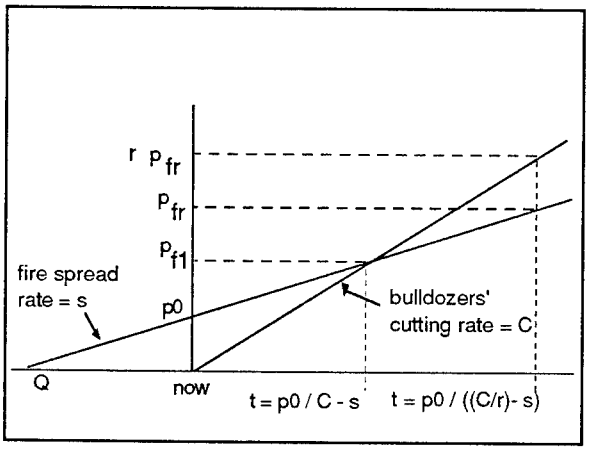


Figure 3. If C > s, the bulldozers will eventually "catch" the growing fire at time t.

The Phoenix fireboss should fight fires in the order that will minimize the sum of the times it spends on

each. Given a set of fires A = {a,b,...}, we need a rule that maps A onto the ordered set F = $\{f_1,f_2,...\}$ so that $T(f_1,f_2,...)$ is minimized. One approach would be to have the fireboss calculate the time required for each order, then select the order with the lowest time. Because the number of orders increases as the factorial of the number of fires, this approach could be expensive. In fact, there is a much quicker way to find the best order: Each fire f_1 can be characterized by a function of its initial perimeter and its rate of pread:

$$Q(f_i) = \frac{p_0(f_i)}{s(f_i)}.$$

If Q(a) < Q(b) < ... < Q(k), and the fireboss fights fires in the order {a,b,...k} then it will minimize the time required to fight the fires. I call this the "youngest first" rule because $Q(f_i)$ is the age of fire i. I prove in the Appendix that fighting fires in the order youngest first minimizes the total time required to contain all the fires. The rule is illustrated graphically in Figure 4. In the top pane, starting at the point labelled "now," the Phoenix agents build fireline around fire a until it is contained. This is shown as the intersection of the heavy "fireline" line and the thinner "fire a" line, at t(a). The agents then start work on fire b (assuming that travel time from fire a to fire b is negligible). Fire b is contained at time T(a,b). The intervening period between t(a) and T(a,b) is t(a,b). In the bottom pane, this pattern is reversed. Agents work on fire b first. Note that T(a,b) < T(b,a), as predicted by the youngest first rule.[1]

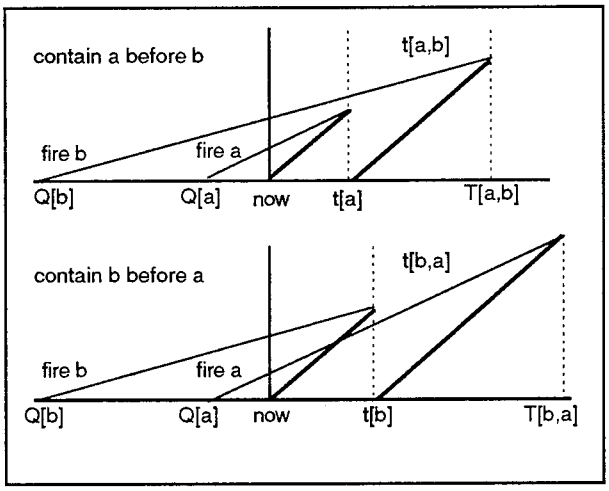


Figure 4. A geometric interpretation of the ordering rule.

2.3 Design and Redesign

The model developed above says that if Phoenix fights fires in youngest first order, it will minimize the time to put out all the fires. Implementing this strategy—making it a permanent part of the design of indirect attack plans—is very easy if the assumptions upon which the model is based are true. If the assumptions are not true, then the model will make false predictions, and we are left with this question: Should we rework the model, or should we change the design of the Phoenix planner so that the assumptions are true? Here is an example of each:

Rework Phoenix: The model is based on the idea that Phoenix figures out how long a fire will take to contain, then selects an appropriate perimeter around the fire to cut line, so that when it is done the perimeter of the cut fireline is r times the perimeter of the fire (see Figure 3). In the model, r = 1.0; that is, there is no "corridor" between the fire and the cut fireline. The assumption is that Phoenix has infinite control over the perimeter of its fireline. In fact, this assumption is wrong. The Phoenix planner does *not* determine t as shown in Figure 3, but rather quantizes t, deciding that the fire will take 500, 1000, 2000, 3000...minutes to contain. As a result, Phoenix cannot ensure a constant corridor around a fire; for

[1]It surprised me that the best strategy should be to fight the *youngest* fires, not necessarily the smallest, or slowest, first. Had we relied on intuitive criteria to select fires to fight, we would probably have programmed Phoenix to fight the fastest-moving fire first. I suspect that many other intuitively correct design decisions in AI planners are, in fact, incorrect.

example, a "1000-minute shell" may provide a snug fit to some fires and very loose fit to others. Since the model is based on a constant corridor (i.e., r = 1.0), the strategy derived from the model ("youngest first" is best) may be wrong. If these wrong predictions can be laid at the door of the quantizing problem, and the model appears to be otherwise correct, then the appropriate response is to rework Phoenix to eliminate the quantizing problem, to give it infinite control over the perimeter it decides to cut, to make it conform to the assumptions of the model. This tack is using the model *prescriptively*, to drive redesign.

Rework the model: The model assumes that as soon as one fire is contained, work begins on the next, with no travel time. If the prediction ("youngest first" is the best strategy) is not borne out, it may be due to travel times between fires. There is no meaningful way to rework the Phoenix planner to make it conform to the "no travel time" assumption: travel time is inherent in the Phoenix environment. The model is *descriptively* wrong, and must be reworked.

Now I will show how experiments with Phoenix proved wrong the prediction that we should always fight the youngest fire first, and how I showed that Phoenix, not the model, needed reworking.

2.4 Predictions

The principal prediction under examination is that fighting fires in the order "youngest first" minimizes the time it takes to fight all the fires. Related predictions are quantitative in nature: how long it should take to fight a sequence of fires, how large is the difference in times between one strategy and another, and so on. Figure 5 summarizes the predicted difference in times to fight a pair of fires under the "oldest first" strategy and the "youngest first" strategy, as a function of the difference in the age of the fires (the curves correspond to differences of 20, 16, 12, and 8 hours, from top to bottom, respectively); and the rates of spread of the fires.

Figure 5 assumes one fire spreads at 100 m/hr and the other spreads at rates between 100 and 300 m/hr. It predicts a 1.75 hour advantage for fighting the youngest fire first, if the delay between the fires is 20 hours and one fire spreads at 300 m/hr and one spreads at 100 m/hr. In Phoenix, softwood fires spread at roughly 200 m/hr and hardwood at roughly 100 m/hr in a 3km wind. Given these figures, the expected advantage of the youngest-first strategy, for an eight hour delay, is roughly twenty minutes. The advantage of the youngest-first strategy is predicted to be much higher for fires that burn faster.

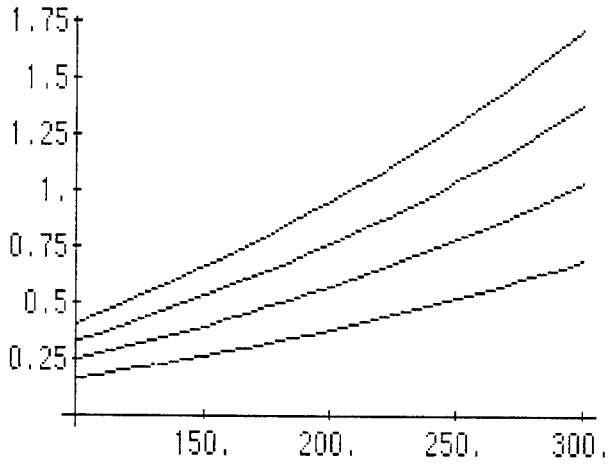


Figure 5. Predicted advantage, in hours, of the youngest first strategy for fires burning in 3km winds

2.5 Experiments

The basic experiment went like this. A trial involved two fires, fought in the yougest-first or oldest-first order. For each trial, the groundcover in each of two sectors was selected and "painted" with hardwood or softwood according to the experimental protocol. Then the watchtowers in each sector were deactivated. Next, a fire was set at a random location in one of the sectors. A delay of 8, 12, 16, or 20 hours ensued, depending on the protocol. Then the second fire was set in the other sector. If the strategy under test was "fight the youngest fire first," then the watchtower in the sector with the most recent fire would be activated; it would send a report to the fireboss, who would begin planning to fight the fire. If the strategy was "fight the oldest fire first," then the other

$Q_2 - Q_1$	predicted time	actual time	disparity
8 hours	8.6	9.4	9%
12 hours	9.84	10.7	9%
16 hours	11.1	12.0	8%
20 hours	11.6	12.66	9%

Table 2. Predicted and actual times to contain fires as a function of the delay between the start times of the fires.

	8 hour	12 hour	16 hour	20 hour
YF first fire				
final fire perim.	4990	5123	5015	5154
final line perim	8104	8106	8088	8097
final line/fire = r	162%	158%	161%	157%
OF first fire				
final fire perim.	7120	8495	10396	11840
final line perim	8335	9969	12259	13074
final line/fire = r	117%	117%	118%	110%

Table 3. Fires fought under the youngest-first strategy have a higher fireline to fire perimeter ratio than fires fought under the oldest-first strategy, indicating a "sloppier" shell.

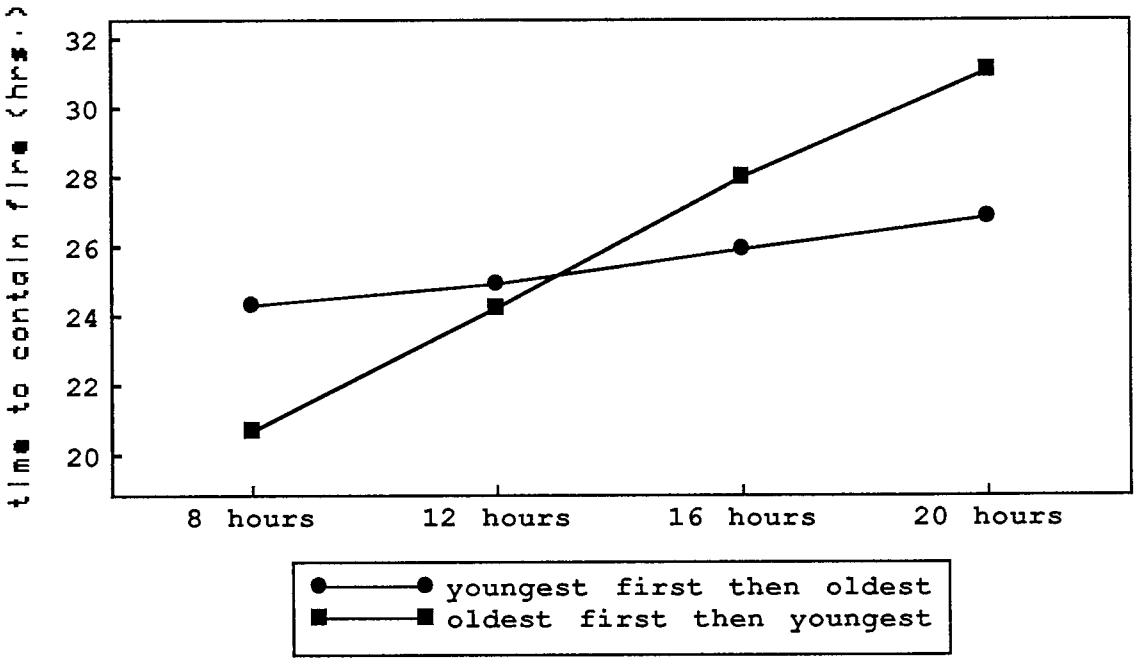


Figure 6. Mean summed time to contain both fires as a function of delay between start times of the fires and fire-fighting strategy.

watchtower would be activated, instead. After a period of hours, the as-yet-inactive watchtower would be activated, and would send a report to the fireboss. The experiment would be allowed to run until both fires were out or 65 simulated hours elapsed. Then the identical experiment would be run again, except with the opposite strategy. Following this pair of trials, new locations for the fires would be selected. We fully counterbalanced for local terrain and other factors that affected fire growth and bulldozer cutting rates.

We ran 470 pairs of fires. One basic result is that Eq.1 does a good job of predicting the time it takes to put out the *first* fire. Table 2 contains data for the fire under both strategies. Empirically, I found that for all first fires, under both strategies, Phoenix was building fire lines in such a way that the final perimeter of the fireline was on average 1.37 times the final perimeter of the fire. That is, over all first fires, r = 1.37. However, it became clear that this "corridor" between the fireline and the fire was bigger for fires fought under the youngest first strategy than for fires fought under the oldest first strategy. Table 3 contains the data: Whereas r ranges from 1.1 to 1.18 for first fires fought under the oldest first strategy, it ranges from 1.57 to 1.62 for those fought under the youngest first strategy. This means that the "shell" selected to fight the latter fires is much "sloppier" than the shells selected for the former. As a consequence, the first fire under the youngest first strategy takes longer to fight than it should.

The data on the time to fight *both* fires under the youngest first and oldest first strategies are given in Figure 6, which shows the time to contain both fires on the y axis. As predicted, the advantage of the yongest-first strategy increases as the delay between the fires increases, but, counter to prediction, the youngest-first strategy does *worse* than the oldest first strategy when the delay is 8 hours. Thus, the principal prediction of the model is wrong!

2.6 Explanation

As noted earlier, given wrong predictions, we have to decide whether to rework the model or rework Phoenix. In either case, the first step is to discover why the model is making wrong predictions. A good way to proceed is to modify the model so it makes correct predictions, and then see whether the modification points to an aspect of the Phoenix environment that cannot be changed (e.g., travel time) or an aspect of the Phoenix planner, which can be changed. The model as derived assumes r is a constant: Phoenix keeps the same size "corridor" between its cut fireline and the fire perimeter. In fact, as we saw above, this is false. The corridor is much bigger for the first fire under the youngest-first strategy than it is for the first fire under the oldest-first strategy. In another experiment, we discovered the reason for this: Phoenix plans the perimeter of its fireline based on projections of where the fire will be in 500, 1000, 2000... minutes. *Every* first fire fought under the youngest first strategy was fought with a 500-minute projection. Since this is the smallest possible projection, it seems to be too large for many of these fires.

When the model is modified to incorporate the fact that r is larger for the youngest first strategy than for the oldest first, the predicted and actual times (in minutes) to contain both fires under each strategy are shown in Table 4. Clearly, the modified model makes very good predictions. It lends strong support to the argument that youngest-first would be the best strategy in all cases, if only Phoenix wasn't forced to use a 500-minute projection for the first fires under the youngest first strategy. In general, the problem with quantizing the projections is that r will not be constant, and so the youngest-first strategy will not always be best.

	8 hour	12 hour	16 hour	20 hour
YF both fires				
predicted	2281	2639	2996	3353
actual	2287	2583	2964	3319
disparity	100%	98%	99%	99%
OF both fires				
predicted	2110	2519	2928	3336
actual	2017	2496	2968	3377
disparity	104%	101%	99%	99%

Table 4. Predictions from the revised model are more accurate.

It seems to me that when one has a simple model from which one can derive a general rule, such as fight the youngest fire first, but the rule fails because the program is doing something that is demonstrably inefficient, such as constructing an excessively large fireline around one kind of fire, then changing the model does no more than describe the inefficiency formally, whereas changing the program eliminates the inefficiency and restores the rule. For this reason, we decided to modify Phoenix to allow it to plan to cut a fireline of any perimeter, not an arbitrarily quantized perimeter. We have not yet re-run the previous experiments to determine whether the predictions of the original model hold.

3.0 Conclusion: The Issue of Generalization

Let me summarize the discussion to this point. The premise of the ecological view is that behavior results from the interaction between an agent and its environment, and that we should design agents from models of these interactions. Equations 1 and 3 are models of this kind because they represent how a measure of behavior (the time to control a sequence of fires) results from the interaction between environmental factors (the rate at which fires spread, s and g), and aspects of an agent's architecture (the rate at which line can be cut, C).[2] From these models, it was possible to design an ordering scheme to miminize time.We are currently engaged in other modelling effort, also:

3.1. Direct attack. The current model assumes agents work at a distance from the fire. In fact, Phoenix agents can also work directly at the fire edge. In this case, called direct attack, the perimeter growth is not linear, because as agents control more of the perimeter its growth rate decreases. Empirically, this nonlinearity is quite small, but we need to know whether it can obviate the strategy to order fires by Q.

3.2. Nonlinear growth. As fires get bigger they generate convective winds, which increase the rate at which the fires grow. We have not yet implemented convective fires in the Phoenix environment. When we do, we may need to rethink the strategy for ordering fires, because the oldest fires, not the youngest, are the most likely to become convective. Nonlinear growth adds a previously absent dimension to the fireboss's scheduling problem: hard deadlines. In the *linear* model, as long as C > s, delays in fighting a fire have linear effects—the bulldozers will

[2] p_0, the size of the fire when the bulldozers reach it, can be treated either as an environmental factor or as another measure of behavior, since it results from the interaction of agent features, such as the time it takes the fireboss to respond to a reported fire, and environmental conditions.

"catch" the fire eventually (Fig. 7.a). But if fire growth is not linear, a delay may ensure that the bulldozers can never contain the fire. When the "C line" is tangent to the "s line" (Fig. 7.b), its x intercept is a hard deadline.

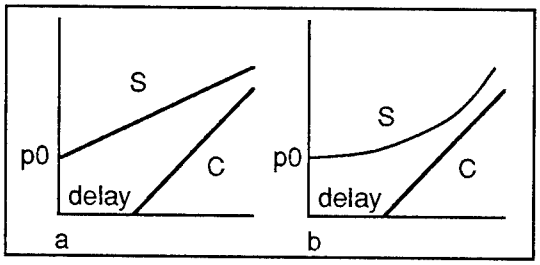


Figure 7. Linear fire growth implies soft deadlines; nonlinear growth implies hard deadlines.

3.3. Other performance measures. I have assumed that the fireboss wants to minimize the area burned, but it may want to minimize the total cost of the area burned and the resources it requires to fight the fires. Assuming that the acreage burned increases with the square of time, the cost of a firefighting operation can be modelled as the following sum:

$$\Sigma = (t^2 \cdot \text{cost(acre)}) + (N \cdot \text{cost(bulldozer)})$$

given that N is the number of bulldozers and t is defined by Eq. 1. As N decreases, t increases. We can find the value of N that minizes this sum by setting its first derivative with respect to N to zero and solving for N:

$$N = s + \frac{\text{cost(acre)}^{1/3}\, p_0^{2/3}}{.5^{1/3}\, \text{cost(bulldozer)}^{1/3}}$$

The efficacy of designing from models depends on whether models can predict how designs will behave. As AI researchers work with more complex environments, and with architectures that produce complex behaviors from interactions of simpler behaviors, the goal of predicting behavior seems increasingly remote. Some researchers claim that behavior is in principle unpredictable, so the only way to design systems is as Nature does, by mutation and selection (e.g., [9], p. 25). I think this is going too far: We can often predict the behavior of a system at a level of abstraction or aggregation that is useful for design, even if we cannot predict details of the behavior. For example, I do not expect Eq. 1 to predict precisely t, the time it takes to contain a fire. No model can, because t depends on the interactions of hundreds of events over time. But as Figure 8 shows, Eq. 1 does predict the general form of the relationship between t and (C – s): t increases as (C – s) approaches zero. Nonlinearities in performance, like this one, alert designers to diminishing returns: Increasing the number of bulldozers that are sent to a fire (increasing C), will have a increasingly smaller effects on t; whereas decreasing the number of bulldozers will have an increasingly larger effects. I am confident in these trends in the values of t, if not in the values themselves. So the question is not whether predicting behavior is possible in principle, but whether prediction is useful in practice.

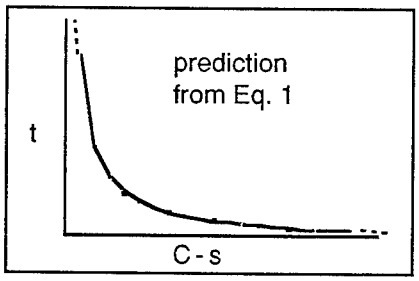


Figure 8. As (C – s) approaches zero, t increases as a harmonic series

Quite apart from their utility to design, modelling and prediction are useful for analysis of systems that already exist—for explaining why systems behave as they do. And here, predictions can be equally useful if they are wrong, because they prompt revisions in our models. For example, we recently noticed that Phoenix's predictions about the fire spread were wrong, adversely affecting its performance. The problem was that fires spread much more quickly upwind than predicted. Scott Anderson discovered that the fire tacks upwind, moving from point A to point B by tacking first from A to C, then from C to B; and it does this much more quickly than going from A to B directly.

But, returning to design for a moment, how can we design from models that sometimes make wrong

predictions? I acknowledge this concern, but what is the alternative? Currently, we design AI programs to vague specifications; for example, we designed Phoenix agents for "unpredictable, real-time, dynamic" environments, and endowed them with "reactivity, approximate processing, responsiveness," and so on. With such vague design goals, it is hard to know whether we succeeded. Phoenix agents put out fires, of course; but as Adele Howe and I have discussed, demonstrations are not adequate evaluations because they don't tell us *why* a system works or doesn't work [2,5]. And even good evaluations cannot ensure that our results are general, because you and I may interpret the informal terms we use as design goals, such as "real time," differently.

Modelling might provide a solution to these problems. A model is a formal summary of our understanding of how behavior emerges from the interactions between an agent and its environment. Models are no less useful for being formal. I think we are better off designing from models like Eqs. 1 and 3 than we are designing from informal terms like "real time." And if a model is inaccurate, if it makes wrong predictions, that is because our understanding of the environment is wrong; it has nothing to do with whether we express that understanding formally or informally. On the other hand, formal models like Eqs. 1 and 3 make our design goals precise. This means that we can say exactly what a system does, instead of relying on demonstrations that a system "works." This is essential to achieving general results in AI. For example, Figure 7 shows a sketch of one definition of "hard deadline": the tolerable delay before a linear process starts "chasing" a superlinear one. You may have another definition. If our definitions are precise, and encapsulated in models, then we can reason about our respective models, instead of guessing what our respective programs might or might not do.

The principal challenge for the model-based design methodology is coming up with *general* models. The model developed in Section 2 is of limited interest because it applies only to linear processes chasing other linear processes. It is difficult to see how to generalize the strategy "work on the linear process that started most recently," to other domains and tasks. But engineers in virtually every discipline *have* developed general models, from queueing theory to stress analysis, and these are used in model-based design. There is no reason to suppose we cannot do the same in AI.

4.0 Appendix.

Derivation of Eq 3:

$$T(f_1,f_2,\ldots,f_m,f_n) = t(f_n) + (1 + g(f_n))\, T(f_1,f_2,\ldots,f_m)$$

I denote the first fire as f_1, the second as f_2, and so on. I will often refer to the time it *would have required* to contain fire f_i *if* the fireboss had attacked it first. I denote this $t(f_i)$.

Consider the time it takes to contain fire f_2 *after* containing fire f_1.

$$t(f_1, f_2) = \frac{(p_0(f_2) + t(f_1)\, s(f_2))}{(C - s(f_2))} \qquad (A.1)$$

When the buldozers finally start work on fire f_2 (after containing fire f_1), its perimeter will be larger than it would have been if they had worked on it first. $p_0(f_2)$ will have grown at rate $s(f_2)$ for as long as it took to contain fire f_1. Therefore, the perimeter of fire f_2 at time $t(f_1)$ will be $p_0(f_2) + t(f_1)\, s(f_2)$. Expanding Eq. A.1 gives

$$t(f_1,f_2) = \frac{p_0(f_2)}{(C - s(f_2))} + \frac{\frac{p_0(f_1)}{C - s(f_1)}\, s(f_2)}{(C - s(f_2))}$$

Now let us introduce a function, $g(x) = s(x) / (C - s(x))$. Rewriting the previous expression in terms of g gives

$$t(f_1,f_2) = \frac{p_0(f_2)}{(C - s(f_2))} + g(f_2)\, \frac{p_0(f_1)}{(C - s(f_1))}$$
$$= t(f_2) + g(f_2)\, t(f_1) \qquad (A.2)$$

This is the time to contain fire f_2 after fire f_1. The time to contain *both* fires is

$$T(f_1,f_2) = t(f_1) + t(f_1,f_2)$$
$$= t(f_2) + (1 + g(f_2))\, t(f_1) \qquad (A.3)$$

We can extend this model to three fires, and then to any number of fires:

$$t(f_1,f_2,f_3) = \frac{(p_0(f_3) + T(f_1,f_2)\, s(f_3))}{(C - s(f_3))} \qquad (A.4)$$

$$T(f_1,f_2,f_3) = T(f_1,f_2) + t(f_1,f_2,f_3)$$
$$= T(f_1,f_2) + t(f_3) + g(f_3)\, T(f_1,f_2)$$
$$= t(f_3) + (1 + g(f_3))\, T(f_1,f_2) \qquad (A.5)$$

Eq. A.4 is analogous to Eq. A.1 and Eq. A.5 is analogous to Eq. A.3. In general,

$$T(f_1,f_2,...,f_m,f_n)$$
$$= t(f_n) + (1 + g(f_n))\, T(f_1,f_2,...,f_m) \qquad (A.6)$$

Proof that the fires should be fought in "youngest first" order:

If Q(a) < Q(b) < ... < Q(k), and the fireboss fights fires in the order {a,b,...k} then it will minimize the time required to fight the fires. To prove this, I will show that if Q(a) < Q(b) < , ... , < Q(k), then the fireboss cannot minimize time unless it fights fire a first. The rule is to fight first the fire with the smallest value of Q. Then, after it has fought the first fire, the fireboss can reapply this rule to the remaining fires. To prove this rule, I will prove two lemmas:

Lemma 1: $T(i,j) < T(j,i)$ iff $Q(i) < Q(j)$

Lemma 2: $T(i,j) < T(j,i) \rightarrow T(S_1, i, j, S_2) < T(S_1, j, i, S_2)$ where S_1 and S_2 are subsequences of zero or more fires.

Lemma 1: From the definitions of Q, t, and g, it follows that $Q(f_i) = t(f_i) / g(f_i)$. I will show that

$$T(i,j) < T(j,i) = \frac{t(i)}{g(i)} < \frac{t(j)}{g(j)} = Q(i) < Q(j)$$

Expanding $T(i,j) < T(j,i)$ according to Eq. A.3 gives

$$t(i) + t(j) + g(j)t(i) < t(j) + t(i) + g(i)\, t(j)$$

Eliminating the common terms gives

$$g(j)\, t(i) < g(i)\, t(j) = \frac{t(i)}{g(i)} < \frac{t(j)}{g(j)} = Q(i) < Q(j)$$

So $T(i,j) < T(j,i) \equiv Q(i) < Q(j)$.

Lemma 2: It is intuitively clear that if $T(i,j) < T(j,i)$, then $T(i, j, S_2) < T(j,i, S_2)$. All fires in the subsequence S_2 are growing while fires i and j are being fought, and they will grow more, and thus take longer to contain, if those fires are fought in the order j,i than if they are fought in the order i,j. Similarly, if $T(S_1, i, j) < T(S_1, j,i)$ then, by the same reasoning, $T(S_1, i, j, S_2) < T(S_1, j,i, S_2)$. It remains to prove that $T(i,j) < T(j,i)$ implies $T(S_1, i, j) < T(S_1, j,i)$. To begin, we rewrite Eq. A.6 with all its terms fully expanded:

$$T(f_1,f_2,...,f_m,f_n) =$$
$$t(f_n) + (1 + g(f_n))\, (t(f_m)$$
$$+ (1 + g(m))(.... (t(f_2) + g(f_2)(t(f_1))) ...)))$$

Rearranging terms and multiplying through, we get

$$T(f_1,f_2,..., f_m,f_n) =$$
$$t(f_1)\, (1 + g(f_2))(1 + g(f_3)) ... (1 + g(f_m))(1 + g(f_n))$$
$$+ t(f_2)\, (1 + g(f_3)) ... (1 + g(f_m))(1 + g(f_n))$$
$$+ ...$$
$$+ t(f_m)\, (1 + g(f_n))$$
$$+ t(f_n)$$

Note that the last two terms in this sum are $(1 + g(f_n))t(f_m) + t(f_n)$, which is just $T(f_m,f_n)$. If we decide to fight these fires in the opposite order, then the last two terms of the sum become $(1 + g(f_m))t(f_n) + t(f_m) = T(f_n,f_m)$, *but the previous terms in the sum do not change.* If you swap the last two fires, f_m and f_n, in a sequence of fires, the resulting difference in time is just $T(f_m,f_n) - T(f_n,f_m)$. Therefore, the lemma is proved: $T(i,j) < T(j,i) \rightarrow T(S_1, i, j) < T(S_1, j, i)$

Now, if we know Q(a) < Q(b) < Q(c) < Q(d), we can use lemma 1 to show

$$T(a,b) < T(b,a),\ T(a,c) < T(c,a),\ T(a,d) < T(d,a),$$
$$T(b,c) < T(c,b),\ T(b,d) < T(d,b),$$
$$T(c,d) < T(d,c) \qquad (A.7)$$

The fireboss should not fight d first: If it does, then it will next have to fight either a, b, or c, giving the sequences T(d,a,...) or T(d,b,...) or T(d,c,...). Because we know from Lemma 2 that if T(i,j) < T(j,i) then T(i,j,S2) < T(j,i,S2), and we know from (A.7) that T(d,x) > T(x,d) for all x, it follows that for every sequence that begins {d,x,...} there is another with a lower value of T that begins {x,d,...}.

Similar reasoning shows that the fireboss must fight fire a first. If it doesn't, then at some point in the future, it will generate a sequence $\{S_1, x, a\}$, and we know from Lemma 2 and (A.7) that this has a higher value of T than $\{S_1, a, x\}$. Now we start "unpacking" the subsequence S_1: for each of its components, T would be smaller if fire a preceded that component. So if Q(a) < Q(b) < Q(c) < Q(d), the time to fight all the fires cannot be minimized unless fire a is fought first. This reasoning applies anew to fire b: if Q(b) < Q(c) < Q(d), then the fireboss must fight fire b before the others. Thus, Q(a) < Q(b) < ... < Q(k) implies that T(a,b,...,k) is the smallest time to fight fires a,b,...,k.

5. Acknowledgments

This work was supported by ONR Contract N00014 - 88 - K - 004 and University Research Initiative grant, ONR N00014-86-K-0764

6. References

1. Chapman, D. and P. E. Agre. Abstract Reasoning as Emergent from Concrete Activity. Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon. 411-424, 1987.

2. Cohen, P. R., Howe, Adele E. "How evaluation guides AI research." AI Magazine. **9**(4): 35 - 43, 1988.

3. Cohen, P. R. Discovering Functional Relationships that Model AI Programs. 1989.

4. Cohen, P. R., Greenberg, M. L., Hart, D.M., Howe, A. E. "Trial by Fire: Understanding the Design Requirements for Agents in Complex Environments." AI Magazine. **10**(3): 32-48, 1989.

5. Cohen, P. R. and A. E. Howe. "Toward AI research methodology: Three case studies in evaluation." IEEE Transactions on Systems, Man and Cybernetics. **19**(3): 634-646, 1988.

6. Cohen, P. R., A. E. Howe and D. M. Hart. Intelligent Real-time Problem Solving: Issues and Examples. Intelligent Real-Time Problem Solving (IRTPS). 1989.

7. Dean, T. L. Intractability and Time-dependent Planning. Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon. 245-265, 1987.

8. Kaelbling, L. P. An Architecture for Reactive Systems. Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon. 395-410, 1987.

9. Langton, C. Artificial Life. Santa Fe Institute Studies in the Sciences of Complexity. 1989.

10. Neisser, U. "Cognition and Reality." 1976 Freeman Press. San Francisco.

11. Rosenschein, S. J., B. Hayes-Roth and L. D. Erman. Notes on Methodologies for Evaluating IRTPS Systems. Intelligent Real-Time Problem Solving (IRTPS). 1989.

12. Simon, H. A. "The Sciences of the Artificial." 1981 The MIT Press. Cambridge, MA.

Analogical Planning

Diane J. Cook
Beckman Institute for Advanced Science and Technology
Department of Computer Science
University of Illinois
Urbana, IL 61801
Arpanet: cook@gaea.cs.uiuc.edu

Abstract

Analogical reasoning provides a powerful method for learning plans where other machine learning methods fail. Unlike many machine learning paradigms, analogy does not require numerous previous examples or a rich domain theory. Instead, analogical reasoning utilizes knowledge of solved problems in similar domains, adapting the knowledge to the current problem.

This paper describes the ANAGRAM system which performs parallel analogical planning using a graph match technique. Given a target problem specification, ANAGRAM finds a similar problem from the database from which a solution can be derived. This paper describes the parallel implementation of ANAGRAM on the Connection Machine and addresses the difficulties that arise when an analogy fails because the base case is only partially applicable to the current problem. ANAGRAM offers a solution to this problem by merging the graphs representing several similar base cases, resulting in a virtual base graph that generalizes the individual cases enough to cover the target problem.

Two examples are presented in this paper that illustrate the use of parallel analogical planning and graph merge in the domain of automatic programming. The techniques described in this paper can similarly be applied to planning tasks in a variety of complex domains.

1 Introduction

When solving a problem in a relatively new and unfamiliar domain, an planner often relies on experience with similar problems to suggest ways of attacking the current problem: adapting known techniques, mapping appropriate constraints from a solved problem to the problem at hand, and modifying existing programs to include new capabilities. Application of many machine learning paradigms to engineering problems requires knowledge of the domain despite the fact that knowledge is available from similar domains. Induction requires numerous examples within the problem class. Explanation-based approaches require a rich theory of the problem domain. In the absence of numerous examples or a rich domain theory, analogy can be used to transfer knowledge of a similar domain to the current problem domain. Given a novel problem (the *target* case), analogy selects a similar, solved problem (the *base* case), computes a mapping between the base and target problem descriptions, and uses the mapping to adapt the base solution to the current domain.

The ANAGRAM system solves novel problems by constructing analogical plans. Given a target goal, ANAGRAM finds a similar goal in the base domain from which a solution can be derived. ANAGRAM expresses plans as graphs and uses a graph match algorithm to identify potential base cases and form the mapping between base and target problems. To approach the benefits offered by other machine learning paradigms, ANAGRAM is capable of merging several similar base cases when a single base is only partially applicable to the current problem.

This paper describes ANAGRAM's efficient approach to constructing plans using parallel analogical graph match, and introduces congruent graph merge as a technique for increasing the effectiveness of an analogy by merging several similar base cases. The approach is illustrated with several examples from the domain of automatic programming.

The theories described here are beneficial to scientific and industrial planning applications in several ways:

- Analogy is a central approach to learning. Skilled designers rarely attempt to solve a problem from scratch. Instead, they build on their expertise, comparing current problems to ones previously solved.
- The inability to utilize all necessary information is a limitation that has always plagued analogical reasoning systems. Merging congruent bases to form more general virtual bases is one solution that will impact all areas in which analogical planning can be used.
- The complexity of the analogical reasoning task has long prevented its automation. Using the massively parallel architecture to reduce the complexity of base selection and map formation makes the task tractable.

- Automatic programming problems have features common with other problems in engineering. All require reasoning about the structure of the problem, its solution, and the ordering of sub-tasks.

Section 2 defines the area of research: planning by analogy. Section 3 describes the parallel implementation of this research in the ANAGRAM system. The next section introduces the notion of merging congruent base cases when forming an analogy, followed by an example from the domain of automatic programming.

2 Planning by Analogy

Analogy uses knowledge about one problem or domain to infer knowledge about a similar problem or domain. Analogy is a central approach to learning. Skilled designers and talented students rarely try to learn about a new area or solve a new problem from scratch. Instead they build on their expertise, comparing current problems to ones previously solved.

Gentner [Gentner, 1988] has shown that people form analogies between concepts that have structural similarities, rather than surface similarities. Representing plans as graphs encodes the structure of the plans, and forcing the base and target graphs to match ensures that the structure of the base and target plans are the same.

Much of analogical reasoning research uses analogies to produce a detailed description of a concept [Gentner, 1988, Greiner, 1988]. It is difficult in these systems to determine the type of information that should be mapped to the target. As Holyoak [Holyoak, 1984] and Carbonell [Carbonell, 1983] have pointed out, goals provide an essential constraint in problem solving. Using analogical reasoning in the problem area of planning provides a focus for the analogical learning task and offers a method of generating plans in unfamiliar domains.

3 Overview of ANAGRAM

The ideas mentioned in this paper are implemented in a system called ANAGRAM (ANAlogical GRAph Matching). Given a target problem specification represented in graph form, ANAGRAM uses a colored graph match technique to select a base case from a database of previously-solved problems. ANAGRAM uses the selected base case to generate a plan which will achieve the target goal.[1]

The system accepts as input two subgraphs, representing the target problem's initial state description and goal state specification. ANAGRAM then searches through the database, finding the best match for both subgraphs. Using the output of the individual graph matches, ANAGRAM then maps over the base plan to the target domain to generate a solution to the target problem. If the resulting plan is unsuccessful, or if no sufficiently similar base cases are found, the system then attempts to merge several base cases that are all similar to the target problem. The result is a virtual base graph that eliminates anomalies and generalizes various options in the plan to an extent that covers the target domain.

[1] See [Cook, 1989] for a complete description of the ANAGRAM system.

3.1 The Graph Match Algorithm

Perhaps the biggest factor that currently prevents machines from making extensive use of learning by analogy is the complexity of the task. It is difficult to understand why analogical reasoning is performed so often and so easily by humans, yet is difficult and costly to perform on a machine. Part of the problem is not fully understanding the nature of analogical reasoning and the algorithms humans use to perform it. However, much of the problem is speed. To make analogical planning tractable, it must be able to examine many base cases in parallel and efficiently form correspondences between base and target. This is possible if the algorithm takes advantage of the massively parallel architecture of such machines as the Connection Machine

This section describes a method of efficiently performing analogical planning by performing base selection and map formation in parallel on the Connection Machine. The algorithms used by ANAGRAM were performed on a Connection Machine-2 with 32,768 nodes.

ANAGRAM employs a colored graph match on directed acyclic graphs (DAGs). The arcs as well as the nodes are labeled. These labels provide an additional constraint on the matching process. Node labels may be different between two graphs, but the arc labels must correspond exactly. The data describing each node of a graph is stored in a separate processor. To perform the graph match, the nodes in the base graph look for a match in the target graph in parallel.

Two nodes match if they are at the same level in the graph (leaves are at level 0, and their parents are at level 1), and the structures of the nodes' children and parents (encoded by the integer assigned to each child and parent) match. Each node is described by the tuple (level [(child-integer out-link) ...] [**(parent-integer in-link)** ...] node-label). Initially, no integers have been assigned to the nodes, so each so each child-integer and parent-integer slot is set to "?".

Each node from the first graph looks in parallel for a match with a node at the same level in the second graph. If two complete tuples match (the tuples are complete if they have no ?s), the match is added to the gmap and a unique integer is assigned to the two nodes. If a tuple is incomplete, it generates a list of partial matches (every non-? matches). After every search pass, each

node in both graphs simultaneously updates its tuples. Assigned integers are propagated across the links. Once the tuples are updated, matches between incomplete tuples are checked once again — if they no longer match, the algorithm returns failure. If no unique matches are found for any of the nodes on a given pass, the algorithm takes one node from the list of nodes with more than one candidate match, and randomly selects a match for the node. If there are nodes from the first graph that cannot be matched with any node from the second graph, the algorithm returns failure. When this is done, the entire process is repeated. The process is successfully completed when a match is found for each node in the first graph.

The complexity of the graph match algorithm is proportional to the greatest number of nodes found at any level in the target graph, because each base node sequences through the target nodes at the same level to look for a match. Let n represent the number of nodes in the each graph, and let h represent the height of the graph. The complexity of the graph match is thus $O(n-h)$.

The base selection process enjoys a tremendous speedup by being parallelized. Normally, the base selection process is extremely time consuming because each potential base solution must be compared with the target problem specification. Fortunately, each of these comparisons is independent of the others, so the bases can be examined in parallel. To perform base selection, each node from each base graph is stored in a separate processor and looks for a match in parallel. Assuming there are enough processors to store all of the base cases, the complexity of the base selection task is the same as for graph match, or $O(n-h)$.

3.2 Example 1

This example is borrowed from Dershowitz [Dershowitz, 1986], who uses analogies between program specifications to modify existing programs in a way that allows them to perform different tasks. The target problem is to generate a function to compute c/d within an accuracy ϵ:

assert $c \leq 0 < d$, $\epsilon > 0$;; *Initial State*
goal $|c/d - q| < \epsilon$;; *Goal State*

The base case is a program that computes the cube root of a within an accuracy ϵ:

begin cube-root
  assert $a \geq 0$, $\epsilon > 0$;; *Initial State*
  goal $|a^{1/3} - r| < \epsilon$;; *Goal State*
  $(r, s) := (0, a+1)$;; *function body*
  loop L_3 : **until** $s \leq \epsilon$
    $s := s/2$
    if $(r+s)^3 \leq a$ **then** $r := r+s$ **endif**
    repeat
end

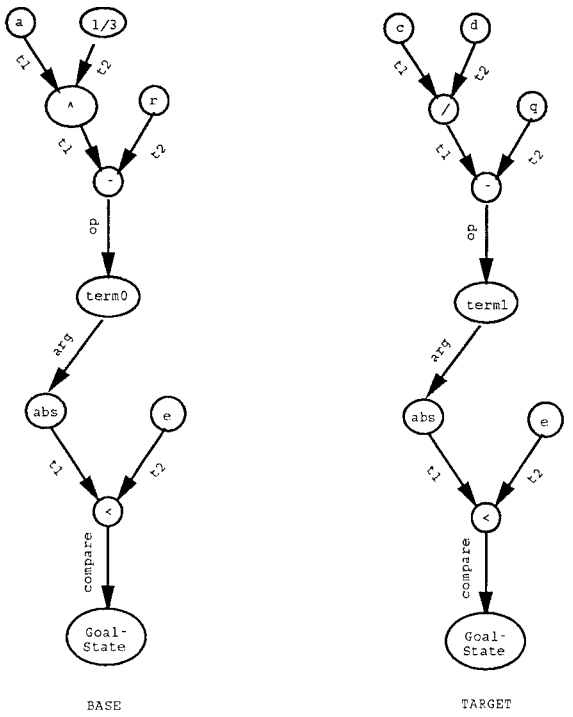


Figure 1: The Base and Target Initial State Subgraphs

ANAGRAM performs a graph match on the subgraphs representing the initial states and goal states of the base and target. The goal-state subgraphs for the two problems are shown in figure 1.

In the first pass, each node from the base graph looks for a partial match from the target graph in parallel. An incomplete match is found for every tuple. A unique integer *int* is assigned to the nodes in each unique match. These integers are then propagated to parents and sons through the graph, and corresponding tuples are updated. The final values of the tuples are shown in tables 1 and 2.

int_t	$tuple_t$
1	(1 [] [**(compare 2)**] GOAL-STATE)
2	(2 [(compare 1)] [**(t1 3) (t2 4)**] <)
3	(3 [(t1 2)] [**(arg 5)**] ABS)
4	(3 [(t2 2)] [] E)
5	(4 [(arg 3)] [**(op 6)**] TERM0)
6	(5 [(op 5)] [**(t1 7) (t2 8)**] -)
7	(6 [(t1 6)] [**(t1 9) (t2 10)**] ^)
8	(6 [(t2 6)] [] R)
9	(7 [(t1 7)] [] A)
10	(7 [(t2 7)] [] 1/3)

Table 1: Updated tuples for the base goal subgraph

int_b	$tuple_b$
1	(1 [] [**(compare 2)**] GOAL-STATE)
2	(2 [(compare 1)] [**(t1 3) (t2 4)**] <)
3	(3 [(t1 2)] [**(arg 5)**] ABS)
4	(3 [(t2 2)] [] I)
5	(4 [(arg 3)] [**(op 6)**] TERM1)
6	(5 [(op 5)] [**(t1 7) (t2 8)**] -)
7	(6 [(t1 6)] [**(t1 9) (t2 10)**] /)
8	(6 [(t2 6)] [] Q)
9	(7 [(t1 6)] [] C)
10	(7 [(t2 7)] [] D)

Table 2: Updated tuples for the target goal subgraph

Combined with the mapping produced by comparing the initial-state subgraphs, the global mapping between the base and target graphs is shown below.

{Initial-State → Initial-State, Goal-state → Goal-state,
< → < abs → abs, - → -, r → q, 1/3 → d,
^ → /, term0 → term1, a → c}

Using these matches, ANAGRAM maps the complete base graph over to the target domain, resulting in the following function that successfully meets the target goal:

```
begin target
    assert c ≤ 0 < d, ε > 0              ;; Initial State
    goal |c/d − q| < ε                   ;; Goal State
    (q, s) := (0, 2)                     ;; function body
    loop L2 : until s ≤ ε
        s := s/2
        if (a + s) × d ≤ c then q := q + s endif
        repeat
end
```

4 Merging Congruent Base Cases

One basic difference between analogical learning and learning by induction is that induction requires several input examples of the concept, and analogy generally uses only one example, the base case. However, there are many instances in which multiple base cases would strengthen an analogy.

One example of using multiple base cases is incremental analogy [Burstein, 1988]. One base case may provide some of the information needed for the target, but not all. Another base case may provide the remaining needed information, but nothing else. An analogy formed between the target and either one of these bases would be insufficient, but the merging of the two separate analogies results in a complete, useful analogy.

A second way of using multiple base cases is to merge similar base cases, resulting in a "virtual" base case. This virtual base case is more beneficial to the analogy than a single case, because it removes anomalies and generalizes alternative operations. Furthermore, merging

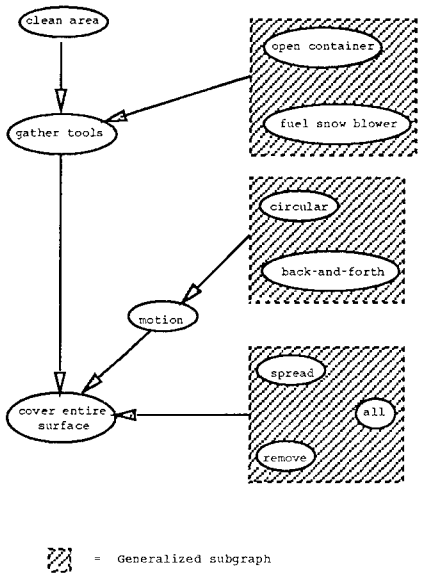


Figure 2: Virtual Base Case

base cases serves to focus on the relevant aspects of the base cases, because those aspects of previous plans that are beneficial to the target are retained in the virtual base. Base cases that are sufficiently similar in structure to be merged together are termed *congruent* base cases. Note that this term is borrowed from geometry, where two triangles are congruent if they have the same angles and proportions of side lengths.

Consider the task of mowing the lawn. A person who has never mowed before may compare cutting the grass to painting a large area with only a small brush. The person could also form an analogy with shoveling snow, waxing the floor, or wallpapering the wall. The analogy formed between the target case (mowing the lawn) and each of the base cases is insufficient, for various reasons. The waxing motion is circular instead of the long back-and-forth movements needed for mowing the lawn. When wallpapering, care must be taken to line up the strips so the designs meet. Painting a wall involves putting on more than one coat of paint. Shoveled snow is dumped to the side of the sidewalk, and the ultimate goal is to remove all of the snow, not just give it a trim. Figure 2 depicts a virtual base case generated by generalizing part of each of the base cases. The analogy between the virtual base case and the target case provides enough information to solve the problem of mowing the lawn.

4.1 The Merge Algorithm

This section describes how ANAGRAM uses the technique of merging congruent base cases to enhance its analogy-formation and problem-solving capabilities. First, the issue of deciding when to merge base cases is addressed, and then the merging algorithm is described.

ANAGRAM using the graph-merge algorithm in the following cases:

- A match is found between the target initial/goal states and the base initial/goal states, but the intermediate steps in the base case are not mappable or can not be applied in the target domain.

- No base case matches perfectly, but several match closely. Moreover, the unmappable parts either 1) are generalizable in a way that map to the target case, or 2) they do not overlap (the base cases fail to match the target at distinct parts of the target graph).

In all of the cases described above, it is possible to merge base cases only if congruent base cases exist in the database. Because trying to find a fit between an arbitrary number of graphs in a given database is an arduous task, the selection process only considers cases within the same index category. The categories are determined when base cases are entered into the database, and are formed according to key parts of the graph, such as the type of operators and objects involved in the plan and the size of the plan.

When selecting cases for merging, the algorithm chooses cases based on ease of generalization. The types of graph merges attempted are (in order of preference):

1. *Merging graphs with distinct base/target differences.* The simplest and most beneficial way to merge congruent graphs occurs when the graphs match each other and their differences with the target do not overlap. Let B_1 represent a base graph which is composed of the subgraphs $S_1^1, \ldots, S_1^i$, let B_2 represent the base graph composed of subgraphs $S_2^1, \ldots, S_2^i$, and let T represent the target graph composed of subgraphs $S_T^1, \ldots, S_T^j$. If the subgraph S_1^x cannot be mapped to S_T^x, S_2^y cannot be mapped by S_T^y, $S_T^x \neq S_T^y$, and a match exists between B_1 and B_2, then a merge of the two graphs is possible. Let M represent the set of mappings output from the match between B_1 and B_2. The algorithm replaces the "defective" subgraph S_1^x in B_1 with the corresponding mapped subgraph $\mathrm{M}(S_2^x)$. The new base graph is mapped to the target (and guaranteed to work). This merging process is easily extended to 3 or more graphs. The restrictions are that the "defective" subgraphs be distinct and all of the base cases are mappable to each other.

2. *Relaxing order constraints.* The graph match algorithm looks for matches between nodes at corresponding levels in the graphs. This precludes a match between plans with different orderings of operators. This is unfortunate, because changing the order of operations in the base case can often solve the target problem (see the description of the inorder tree traversal problem described in the next section).

   Comparing multiple base graphs helps to focus on relevant aspects of a problem. When comparing two plans whose only difference is the order of operations, it becomes apparent that maneuvering the operators may also solve the target problem. When this situation exists, ANAGRAM looks for correlations between the order of operations and ordering constraints in the initial or goal state. Such constraints help focus the mapping to the target problem, preventing trials of all possible operator combinations.

3. *Disjunction/generalization of subgraphs.* The methods of merging base graphs discussed in the previous paragraphs are considered first, because the generated target plan (based on the virtual base plan) is guaranteed to be successful. If neither of the methods is applicable, the system generalizes portions of the graph.

   When comparing base graphs whose "defective" subgraphs *do* overlap, ANAGRAM generalizes the overlapping subgraphs. The methods of generalization correspond to those found in most induction systems, such as adding disjunctions and climbing ISA trees. The purpose of the generalization is to abstract the non-mappable parts to the extent that the generalized subgraph will cover the target case. The more base cases found to support the generalization, the better. As the number of base cases increases, this form of merging base graphs begins to look like pure induction.

4.2 Example 2

This section describes the application of ANAGRAM's graph match algorithm and congruent base case merging algorithm to an example in the domain of automatic programming. In this example, the target problem is to construct a program that uses inorder traversal to traverse a given binary tree. The initial and goal state descriptions are given:

assert *tree* $\in$ *binary − trees* **and** *null(vlist)*
goal $\forall(y \in vlist)$ [left-son(y) **before** y **before** right-son(y)]

Among the base cases in the database are the algorithms for preorder and postorder tree traversal. The matches are equally good with either base case, so the selection process arbitrarily chooses the preorder case. However, the resulting plan is

begin inorder
  $x := root(tree)$

```
    unless null(tree)
        vlist := append(vlist, x)
        inorder(left-son(x), vlist)
        inorder(right-son(x), vlist)
end
```

which does not solve the problem (remember that the graph matcher does not consider re-ordering the operators). The system then compares the preorder and postorder algorithms, and notices that the operators are the same in the two algorithms, but the order of application is different. The goal description in both base cases places ordering constraints on the output:

goal-preorder:
$\forall(y \in vlist)$ [y **before** left-son(y) **before** right-son(y)]

goal-postorder:
$\forall(y \in vlist)$ [left-son(y) **before** right-son(y) **before** y]

By comparing the order of operators with the order imposed by the goal description, ANAGRAM observes that the placement of y corresponds with the push-end operation, *left-son(y)* with *recursive-call(left-son(x))*, and *right-son(y)* with *recursive-call(right-son(x))*. ANAGRAM is able to generate a virtual base graph that contains the correspondences between the three operators and the desired order of elements in vlist. The resulting target plan is successful:

```
begin inorder
    x := root(tree)
    unless null(tree)
        inorder(left-son(x), vlist)
        vlist := append(vlist, x)
        inorder(right-son(x), vlist)
end
```

5 Conclusion

The ANAGRAM system demonstrates the power of analogy for planning in unfamiliar domains. Analogical reasoning allows a system to hypothesize plans to solve problems which lack a rich domain theory and have few similar examples from which to generalize. Using both the structure of the plan descriptions and the goal of the target instance, ANAGRAM can select a base case, find correspondences between base and target, and map the base solution to the target domain. Because of the parallel nature of the algorithm, the task is performed in time sublinear in the average size of the base graphs.

In many cases, an analogy will fail because of an anomaly in the base case, because operators used in the base solution cannot be applied to the target problem, or because not all information maps from the base to the target. ANAGRAM proposes a solution to this situation by examining several similar base cases. If two or more base cases can be found which are similar to each other and to the target, they can be merged into a generalized virtual base graph which is more likely to cover the target situation.

In this paper, ANAGRAM's graph match algorithm and congruent graph merge algorithms are described and illustrated using examples from the domain of automatic programming. The contributions that ANAGRAM makes in this domain are indicative of the benefits analogical reasoning can provide to planning in many new and complex domains.

6 References

[Burstein, 1988] M. H. Burstein, "Incremental Learning from Multiple Analogies," in *Analogica*, A. Prieditis (ed.), Morban Kaufmann Publishers, Inc., Los Altos, California, 1988, pp. 37-62.

[Carbonell, 1986] J G Carbonell, "Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition", in *Machine Learning: An Artificial Intelligence Approach, Vol. II*, R. S. Michalski, J. G. Carbonell and T. M. Mitchell (ed.), Morgan Kaufmann Publishers, Inc., Los Altos, California, 1986, pp. 371-421.

[Cook, 1989] D. Cook, "ANAGRAM: An Analogical Planning System," Technical Report, University of Illinois, 1989.

[Dershowitz, 1986] N. Dershowitz, "Programming by Analogy," in *Machine Learning: An Artificial Intelligence Approach, Vol. II*, R. S. Michalski, J. G. Carbonell and T. M. Mitchell (ed.), Morgan Kaufmann Publishers, Inc., Los Altos, California, 1986, pp. 393-421.

[Gentner, 1988] D. Gentner, "Analogical Inference and Analogical Access," in *Analogica*, A. Prieditis (ed.), Morgan Kaufmann Publishers, Inc., Los Altos, California, 1988, pp. 63-88.

[Greiner, 1988] R. Greiner, "Learning by Understanding Analogies," in *Analogica*, A. Prieditis (ed.), Morgan Kaufmann Publishers, Inc., Los Altos, California, 1988, pp. 1-36.

[Holyoak, 1984] K. J. Holyoak, "The Pragmatics of Analogical Transfer," in *The Psychology of Learning and Motivation, Vol. I*, G. H. Bower (ed.), Academic Press, New York, 1984.

Rational Distributed Reason Maintenance for Planning and Replanning of Large-Scale Activities (Preliminary Report)

Jon Doyle*
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
doyle@zermatt.lcs.mit.edu

Michael P. Wellman
USAF Wright R&D Center
WRDC/TXI
Wright-Patterson AFB, OH 45433
wellman@wrdc.af.mil

Abstract

Efficiency dictates that plans for large-scale distributed activities be revised incrementally, with parts of plans being revised only if the expected utility of identifying and revising the subplans improve on the expected utility of using the original plan. The problems of identifying and reconsidering the subplans affected by changed circumstances or goals are closely related to the problems of revising beliefs as new or changed information is gained. But the current techniques of reason maintenance—the standard method for belief revision—choose revisions arbitrarily and enforce global notions of consistency and groundedness which may mean reconsidering all beliefs or plan elements at each step. We outline revision methods that revise only those beliefs and plans worth revising, and that tolerate incoherence and ungroundedness when these are judged less detrimental than a costly revision effort.

1 Introduction

Planning is necessary for the organization of large-scale activities because decisions about actions to be taken in the future have direct impact on what should be done in the shorter term. But even if well-constructed, the value of a plan decays as changing circumstances, resources, information, or objectives render the original course of action inappropriate. When changes occur before or during execution of the plan, it may be necessary to construct a new plan by starting from scratch or by revising a previous plan. In fact, replanning may be worthwhile even when the new situation does not deviate significantly from prior expectations. The original plan may have been constructed to perform acceptably over a wide range of possible circumstances, and knowing more about the particular situation encountered may enable construction of strategies which are better suited to the case at hand.

There are two central decisions surrounding the replanning process. First, given the information accrued during plan execution, which remaining parts of the original plan should be salvaged and in what ways should other parts be changed? Incremental modification is more efficient than wholesale replanning, but a restriction to local changes can compromise the value of the revised plan. Second, to what extent should the planner attempt to avoid the need for replanning by anticipating contingencies and providing for them in the original plan? Contingency planning improves the capacity for response when replanning time is limited, but the return on up-front investment rapidly diminishes as the likelihood of particular contingencies decreases.

In the following, we describe an approach to replanning which addresses the first question by applying the decision-theoretic conception of rationality to the plan revision tradeoff. Characterizing the computational costs and performance of the revision process contributes toward solutions to the second problem, development of a contingency planning strategy. Our techniques center on a reason maintenance system or RMS (also known as TMS for "truth maintenance system" [de Kleer, 1986; Doyle, 1979]), redesigned for more rational and flexible control.

2 Rational replanning

To replan effectively in crisis situations, replanning must be *incremental*, so that it modifies only the portions of the plan actually affected by the changes. Incremental replanning first involves *localizing* the potential changes or conflicts by identifying the subset of the extant beliefs and plans in which they occur. It then involves *choosing* which of the identified beliefs and plans to keep and which to change. For greatest efficiency, the choices of what portion of the plan to revise and how to revise it should be *rational* in the sense of decision theory. This means that the replanner employs expectations about

*Jon Doyle is supported by National Institutes of Health Grant No. R01 LM04493 from the National Library of Medicine.

and preferences among the consequences of different alternatives to choose the best one.

2.1 Explicit and implicit rationality

According to decision theory, a choice is rational if it is of maximal expected utility among all alternatives. But planning and replanning involve at least two different sorts of decisions, and applying the standard of rationality to each yields different notions. The fundamental distinction is that between *result* rationality and *process* rationality. Rationality of result measures how efficiently the plan achieves specified objectives. Complementing this, rationality of process measures how efficiently the planner expends its efforts in constructing the plan. While most investigations of planning have focused on one or the other, both elements are essential to the overall rationality of the planning system.

Making any process rational is not easy, for straightforward mechanizations of decision-theoretic definitions can require more information than is available and more computation than is feasible to use that information. Sophisticated mechanizations are more tractable, but the main tool for achieving rationality in reasoning is to distinguish between *explicit* and *implicit* rationality in processes. Computational mechanisms may calculate and compare expected utilities in order to make explicitly rational choices. Explicit rational choice promises to be most useful in guiding some of the larger meta-level decisions about whether to replan globally or incrementally, and in choosing which contingencies call for planned responses. For the more numerous small decisions that arise, however, explicitly representing and calculating expected utilities may not be worth the cost. Instead, the more useful approach is to apply non-decision-theoretic reasoning mechanisms whose results may be justified as rational by separate decision-theoretic analyses. Such mechanisms may be viewed as "compiling" the results of explicit rational analysis into directly applicable forms. Each of these ways of implementing rationality is best in some circumstances, since compilation is not always possible or worthwhile.

Examples of implicitly rational procedures abound in AI under the name of heuristics. For instance, the "status quo optimality" heuristic [Wellman, 1990a, Section 6.4.1] constrains the set of possible revisions under the assumption that the current plan is optimal. In particular, the replanner need only respond to the specific changes. A related example is application of the basic theorem of optimization that says that if the only change is a tightening of constraints, the currently optimal plan remains optimal if it remains feasible. Another example, of somewhat different character, is provided by the assumptions made by nonmonotonic reason maintenance systems. The default rules or reasons justifying these assumptions are important forms of heuristics, and the RMS examines them to come up with a coherent set of assumptions and logical conclusions. Though the algorithms for determining these sets of conclusions do not involve any explicit rationality calculations, the conclusions drawn by the RMS can be shown to be Pareto optimal sets, that is, rational choices of conclusions when the reasons are interpreted as preferences over states of belief [Doyle, 1985]. Viewed this way, default rules or reasons encode compiled preferences, and reason maintenance is an example of an implicitly rational choice mechanism.

Thus one approach to the application of rationality principles in planning and replanning is to identify the principles and look for computational mechanisms that implement them, preferably implicitly. Another is to develop seemingly effective computational mechanisms and then figure out under what conditions they are rational. We are pursuing both approaches.

2.2 Rational guidance of replanning

Process rationality enters the task of planning in numerous ways. For example, in the development of a plan, contingency plans should be included only when the expected utility of preparing them is sufficiently great: if the contingency is likely to occur and if the costs of developing it in advance are less than the costs of constructing it under the tighter constraints existing while executing the enclosing plan. Similarly, a portion of a large plan should be revised only if, given the new information, the expected costs and benefits of identifying which plan elements need revising outweigh those expected for either using the original portion or replanning from scratch.

Making these judgments requires information about the likelihoods, costs, and benefits of different sorts of contingencies and planning responses. This includes the likelihood of specific contingencies arising, their importance if they do arise, and the costs of planning for them; similarly, the likelihood of one part of the plan being affected by changes in another, the importance of those changes, and the costs of determining and effecting them.

While many of the likelihoods involved in planning derive from the specifications of the task, the costs and benefits of reasoning steps involved in planning are functions of the underlying representational and reasoning architecture. The theory of computation supplies some abstract notions of computational costs, such as worst-case time and space taken by Turing machines. However, significant differences in reasoning time and space can be lost in the translation to Turing machines, and the worst case is not the only one of interest. Use of the theory of rational decisions effectively in making judgments about plan revision requires realistic measures of computational costs and benefits appropriate to the particular architecture of the planner, as well as expectations appropriate to the domain of planning. Our development of the planning architecture attempts to make formalization and estimation of these measures more direct.

Process rationality must be evaluated with respect to the combined planning/replanning system. In our model of the plan construction process, depicted in Figure 1, the planner and replanner continually evaluate and re-

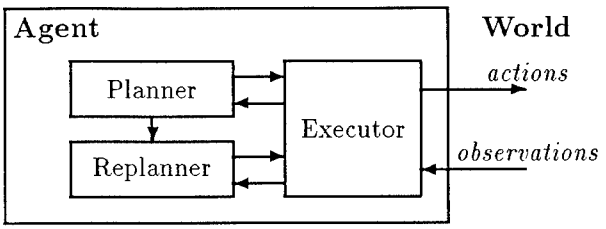


Figure 1: An integrated planning, replanning, and execution system.

vise the existing plan in light of what happens in the world. The distinction between planning and replanning is that the latter uses the existing plan to focus attention on a restricted set of decisions about actions to be performed. The tight coupling of the planning and replanning modules is indicative of the strong interactions between their designs. Knowledge about the capability of the replanner dictates where up-front planning effort should be spent anticipating particular contingencies. And the replanner requires access to the planner's reasons for adopting the current strategy in order to intelligently adapt it for changing situations.

To do this, the planning procedures routinely identify the assumptions made during planning and connect plan elements with these assumptions. In addition, to achieve true flexibility in the sorts of changes the replanner can accommodate, we permit any element of information to change, including the problem specification, background knowledge, and preferences. This allows the replanner to benefit from knowledge of what other specifications, beliefs, and preferences were adopted as consequences or choices from the changed items. This also makes it important that implicitly rational planning procedures indicate the original expectations, preferences, and subplans from which they were "compiled."

3 Planning framework

Our approach combines a dominance-proving architecture for planning [Wellman, 1990a] with a reason maintenance facility for replanning. We start from a constraint-posting view of the plan construction process. Plans consist of a set of *actions*, which can be specified at varying levels of detail. Constraints posted by the planner dictate the inclusion or exclusion of particular actions, and specify features of the actions included. For example, unary constraints on an action may determine the resources allocated to it, its spatiotemporal location, or some other details about its implementation process. Inter-activity constraints may identify shared objects or establish temporal relations among actions. The class of expressible constraints defines the plan construction language. The planning language itself is a restricted subset of this, limited by input requirements of the execution module.

Each posted constraint represents a *decision* made by the planner, choosing the class of plans satisfying the constraint over those that do not. To support rationality in planning, we require that every decision be associated with a reason, of one of the following types:

1. *Dominance* reasons indicate decision-theoretic arguments that plans violating the constraints are inadmissible [Wellman, 1987].
2. *Feasibility* reasons justify posting constraints because they are required for plan executability. For example, we must enforce preconditions of included actions.
3. *Completeness* reasons indicate the constraints are required to fill out plans so that they can be interpreted by the execution module. For example, all shipment actions must specify a source and destination. The degree of incompleteness permitted depends on the reactive capabilities of the executor.
4. *Default* reasons directly associate decisions with other conditions on the planning situation. While all planning decisions are defeasible, we distinguish those not based on explicit rationality arguments.

All reasons specify the beliefs, preferences, and other planning decisions on which they depend. Because these elements in turn are supported by reasons, the composite argument for a planning decision can include a variety of these justification types. For example, a decision might be derived from a decision-theoretic dominance proof with some premises representing default intentions premised on some default intentions which in turn were triggered by the need to complete an insufficiently specified action description.

The dominance-proving architecture offers several advantages as the basis for a rational replanning system. Foremost, it accommodates use of decision-theoretic criteria for choice among plans, which is the central basis of result rationality. In addition, its dominance relation is defined over abstract plan classes, so that these criteria can be associated with isolated planning decisions (that is, individual constraints). Attaching reasons to dominance conditions generalizes the current architecture and directs the replanner to the appropriate regions for modification when things change.

Though recording the reasons for plans is a first step towards efficient incremental replanning, this alone is not sufficient, as we see by a closer examination of reason maintenance techniques.

4 Replanning and reason maintenance

The problem of revising plans to account for changed conditions has much in common with backtracking and the problem of revising beliefs in light of new information. In both cases, one must determine which existing beliefs or plans are in conflict with the new information, what these existing beliefs or plans depend on, and what gaps in plans or beliefs appear as the revisions or

updates are made. That is, one must localize the potential changes or conflicts by identifying the subset of the extant beliefs and plans in which they occur. Similarly, both belief revision and plan revision involve choosing which of the identified beliefs and plans to keep and which to change. In addition, the problem of providing for contingencies has much in common with the problem of choosing rules for reasoning by default, for both involve setting up primary plans or beliefs and the secondary plans or beliefs to use when the primary ones are not applicable. In both plan revision and belief revision, we seek to make these choices of where and how to revise rational in the sense of decision theory.

The standard approach to belief revision, backtracking, and default reasoning is to use a reason maintenance system to connect original information with derived conclusions and assumptions. Reason maintenance may be used in a similar way to revise plans as well as beliefs by indicating the dependence of plans on beliefs and on other plans, thus indicating the relevant portions for revision and the conflicts between prior plans and new circumstances. This possibility was, in fact, one of the original motivations for reason maintenance systems (see [de Kleer *et al.*, 1977]).

4.1 Rational reason maintenance

But the extant architectures for reason maintenance require reassessment. In the first place, essentially all the choices made by current RMSs are irrational since they are made without reference to any preferential information about what choices are better than others. The most obvious decisions concern backtracking: whether observed conflicts warrant resolution and if so, which assumption to retract in order to resolve them. Approaches to each of these decisions play prominent roles in the design of different reason maintenance systems. But if we are to achieve the efficiency required for revising large plans, reason maintenance must be redesigned to make these choices rationally whenever possible. Accordingly, we have begun to develop formal foundations for the theory of rational belief revision [Doyle, 1988; Doyle, 1990], and are developing techniques for encoding probabilistic and preferential information within the RMS and methods by which the RMS can use this information to backtrack in a rational manner. In this, we build on techniques for qualitative representation of probabilistic information [Wellman, 1990b].

But to really make reason maintenance techniques efficient, we must do more than choose rationally among assumptions in backtracking. We must in addition undertake a fundamental reconsideration and redesign of reason maintenance systems to make them much more incremental than extant architectures. Current algorithms for revising beliefs are based on making unbounded (potentially global) optimizing computations that in some cases may reconsider the status of *every* item in the plan and knowledge base, even though very few of these statuses may change as the result of the revision. Put another way, extant systems maintain global coherence (propositions are believed if and only if there is a valid argument for them) and global groundedness (all believed propositions have a well-founded argument from premises). While these unbounded computations have been manageable in the relatively small knowledge bases explored to date, they would appear to be infeasible for use in systems manipulating very large plans. Instead of global computations, we need some way of controlling how much effort is spent on revision. If reason maintenance is to be rational, the system must be able to trade off coherence and groundedness for time or other resources. Specifically, it must be able to decide whether the benefits of updating some arguments or consequences justify the costs of updating them.

To make the RMS amenable to rational control, we divide the knowledge base into parts, each of which may be revised or preserved separately. Each module of this *distributed* RMS contains its own set of beliefs and plans (as well as other information) corresponding to different elements and purposes of the overall plan or to different dimensions of structure (hierarchical abstraction, overlapping views, spatial separation, temporal separation, flow of material and information, etc.). Decomposition of knowledge in this way is a familiar element of many representational schemes (e.g., those based on Minsky's [1975] original frame-systems idea). The use of locality in planning is illustrated most explicitly by the encapsulation mechanisms of Lansky's [1988] GEMPLAN system.

4.2 Distributed reason maintenance

Along with the general benefits of decomposition, there are several additional reasons for distributing reason maintenance across different processors. In the first place, the information and effort required may be too great to store or perform on a single machine. In managing very large activities, for example, the most effective representations may spread information across machines or storage media of different speeds and access times (e.g., disk storage, large spatial separations). Even when the information resides on a single processor, the most convenient representation may be a modular, distributed organization as described above. But more generally, the information and actions involved in some task may be naturally distributed. For example, the necessary information may come from geographically separated sensors or databases. If communication is either unreliable or costly, effective action may require on-site processing. Similarly, there may be numerous people or devices carrying out parts of the task. For example, in the task of operating a large manufacturing complex, plans are executed by line or cell managers acting independently except as coordinated by the plan. When changes occur, at least some of the changes in plan must be determined by the line or cell managers, since the complex manager will not be able to keep track of all of the activities or to respond quickly enough. Because authority is dele-

gated and distributed, reactions to deviations may be completely decentralized and uncoordinated.

In addition, distributed reason maintenance may be valuable because different beliefs and plans may serve different purposes. These purposes may dictate careful maintenance of some beliefs and more casual maintenance of others. A common case of this arises when reasoning is accomplished by different modules operating at different rates. Even if they share a common database, it is often natural to view each module as having distinct inputs, outputs, and local state. In this setting, different rates of inference or action in the modules call for differing treatment of the information in computing updates and checking support. For example, outputs which change rapidly compared with how often they are used as inputs need not demand reconsideration of consequences each time they change. Instead, it may be much more efficient to leave the consequences untouched and to have the consuming module recheck the support immediately prior to use—and then only if the risks of unjustified action are great enough. In many cases, we may expect that the success of the overall plan will not be adversely affected if the beliefs of one module about plans involving some distant module are mistaken.

For example, suppose one part of a manufacturing plan calls for receiving parts from San Diego at Los Angeles and then flying them to Detroit. If local difficulties promise to delay the parts from San Diego, the origination portions of the plan might be revised to reroute similar parts in San Francisco to Los Angeles. As long as this plan patch attaches appropriate shipping orders for the Los Angeles authorities, there is no need to notify them in advance about the change in plans. Indeed, if the origination plans change several times (say from San Diego to San Francisco, back to San Diego, etc.), notifying Los Angeles in advance just leads to wasted effort in revising the latter portion of the plan.

4.3 The reason maintenance service

The extant RMS architectures make reason maintenance the base-level stratum upon which all other reasoning procedures are erected. To enable belief revision, one must encode every bit of information that might change in reasons and tell these reasons to the RMS (cf. [Rich, 1985; Vilain, 1985]). This can present an excessive burden, as manifest by the observation that the RMSs supplied in expert system shells all too often go unused. If one must apply it to every step of reasoning, at every level down to the smallest inference, reason maintenance becomes a demanding duty rather than a flexible service to use or ignore as appropriate. To integrate existing application tools and systems that do not use reason maintenance into AI systems that do, the RMS must be able to use other databases and processes to effect its revisions. In particular, the RMS must be able to treat external databases as the authorities about certain beliefs, and it must be able to operate even though other processes may be changing these databases independently of the RMS. This makes the RMS just one of a set of distributed databases.

5 Rational distributed reason maintenance

Putting these observations together, we seek to facilitate revision of large plans by employing a *rational distributed reason maintenance service*, or RDRMS. The purpose of the RDRMS is to maintain a description of the overall system's state of belief that is as good as possible given the reasoner's purposes and resources. This description may be approximate, partial, or imperfect, and it may be improved by performing further computation as the resources supplied to the RDRMS increase.

There are many motivations for using an RMS: as a way of providing explanations, as a way of answering hypothetical questions, and as a way of maintaining coherence, groundedness, and consistency. These also motivate the RDRMS, but its primary purpose is to enable the reuse of past computations in whole or in part without having to repeat the possibly lengthy searches that went into constructing their results. That is, we view reasons as information about past computations or conditions which may be used to reconstruct results in changed circumstances, either exactly or in modified form (as in derivational analogy [Carbonell, 1986] or case-based reasoning). Treating reasons as aids to recomputation is in marked contrast with the traditional use of reasons in RMSs, where they are treated as rigid requirements that belief states must satisfy instead of information which may be used or ignored as suits the reasoner's purposes. Naturally, in this setting the RDRMS is not expected to determine completely and accurately what the system believes. Instead, it only offers a theory of what the overall system believes—an "autoepistemic" theory, in the sense of Moore [1985], but not necessarily a complete or correct one.

5.1 RDRMS Operations

The basic operation of the RDRMS is to record reasons and other information, and, when so instructed, to revise beliefs in accordance with the expectations and preferences supplied by the reasoner. Put another way, the default operation of the RDRMS is to ignore the information it records until it is told to revise beliefs, and then to revise them only as far as can be justified by purposes of the reasoner. We do not require that all inference be rationally controlled. Some amount of automatic inference is acceptable if it represents strictly bounded amounts of processing.

In the RDRMS, reasons are ordinarily partial. That is, the reasoner need not register all inferences with the RDRMS. The RDRMS will therefore be unable to track all the consequences of all beliefs. Although knowledge is usually preferable to ignorance, this incompleteness of the beliefs of the RDRMS need not be detrimental since

the underlying knowledge and inferences of the reasoner are incomplete anyway. Moreover, these consequences may not influence the reasoner's actions, in which case all effort expended in recording them would be wasted. The only discipline required of the reasoner is that any inferences that will not be performed by some other agency and that cannot be determined after the fact during backtracking should be described to the RDRMS.

Correspondingly, reasons may be incorrect in the RDRMS. That is, the reasoner may use a reason to describe the result of a computation, but may leave out some underlying assumptions. The result is a reason that is valid when those unstated assumptions hold, but which may be invalid otherwise. Incorrect reasons can be very troublesome in a traditional RMS, since they would be enforced as requirements on the state of belief, but they need not cause special problems in the RDRMS. Since the RDRMS may obey or ignore reasons depending on its instructions and experience, all reasons are implicitly defeasible. Thus incorrect reasons pose no problems not already present in explicitly defeasible nonmonotonic reasons.

Just as reasons may be incomplete, so may be the theories of belief states constructed from them, since if reasons are ignored, their consequences will not be believed. More generally, the RDRMS makes it possible to vary how many conclusions are drawn from reasons. For example, the system will ordinarily use reasons to construct a single global set of beliefs, as in the original RMS. But for some specific sets of reasons, say those corresponding to a circumscribed problem, the RDRMS may determine all consistent sets of beliefs as in the ATMS [de Kleer, 1986]. Alternatively, only some consistent interpretations may be constructed, such as those maximal in some order (as in preferential nonmonotonic logics [Shoham, 1988]). In general, the aim is to use the recorded reasons to draw as many conclusions as the reasoner needs.

Similarly, the revisions performed by the RDRMS may be incomplete. In the absence of more specific instructions, the default revision is trivial, simply adding the new reasons and their immediate conclusions to the belief set. (In recognition of the partiality of reasons, the RDRMS also accepts commands to simply believe some proposition, independent of reasons. This corresponds to the "revision" operation in philosophical treatments of belief revision [Gärdenfors, 1988].) Specifically, without explicit instructions, the RDRMS does not propagate changes, does not ensure beliefs are grounded, and does not automatically backtrack to remove inconsistencies. To give some structure to these operations, we define revision instructions relative to the modules of the knowledge base. These instructions may indicate that changes should propagate within the module containing the belief, or to its neighbors, or globally; or that all beliefs in the module should be grounded with respect to the module, with respect to its neighbors, or globally; or that backtracking should be confined to the module, or should look further afield for assumptions to change.

5.2 RDRMS Behavior

One consequence of the incompleteness and incorrectness of reasons is that beliefs of the system may be inconsistent in routine operation. The overall set of beliefs may exhibit inconsistencies by including conflicting beliefs from different modules. Ordinarily the specialized beliefs corresponding to specific problems or subjects will be represented in modules that are internally consistent, but the RDRMS need not be forced to keep all these modules consistent with each other. In this case, the locally coherent modules can be interpreted as "microtheories" [Hewitt, 1986] (related to the idea of "small worlds" in decision theory [Savage, 1972]). But inconsistency can arise even within a module if too little inference is specified.

Another consequence is that the beliefs of the system may not be fully grounded. In the first place, the set of beliefs may be so large as to make global groundedness too costly. More fundamentally, large sets of beliefs always contain interderivable sets of propositions—alternative definitions provide the most common example—and which of these sets to choose as axioms can depend on the specific reasoning task being addressed. For example, the standard definition of nonplanar graphs is best for some purposes (e.g., teaching the concept), but Kuratowski's characterization is best for other purposes (e.g., recognition algorithms). Thus lack of global groundedness need not be cause for alarm. Ordinarily, however, specialized modules corresponding to specific problems will be kept grounded in the axioms formulating these problems. The system of beliefs can thus be thought of as "islands" of groundedness floating in a sea of ungrounded beliefs.

The aim of the RDRMS is to make all of its choices as rationally as possible. These include the choices of which reasons to use in reconstructing results, whether to propagate changes, whether to ground a conclusion, and whether to backtrack. Since reasons merely record some of the inferential history of the reasoner, they do not by themselves determine whether consequences are updated or supports are checked. Instead, to make these decisions the RDRMS uses annotations supplied by the reasoner which give instructions, expectations, and preferences about alternative courses of action. These include specification of the conditions under which the RDRMS should pursue consequences and check support. For example, local propagation may be expressed as processing changes within the module containing the changed belief, but not externally. Alternatively, changes might be communicated to neighboring modules (with or without local propagation). Other regimes are possible too, including the extreme of propagating the change globally. Similarly, the annotations may indicate to persist in believing the proposition without reevaluating the supporting reason, to check that the reason is not invalidated by beliefs within the containing module, or to check validity with respect to external beliefs.

It is this limited scope, along with the variety and

fine grain of RDRMS operations, that makes the service amenable to rational control. For decisions about updating consequences and checking support, it is important that the individual operations be well-characterized computationally. Domain knowledge of probabilities and preferences should also be reflected in the revision policies. Because such information is not always available, the architecture provides default choices for each of these classes of decisions. Each domain may override these with other defaults that are more appropriate in its specific area. These default choices are then used whenever there is no evidence that a decision requires special treatment.

In addition to these decisions within the RDRMS, there are choices about whether to record specific reasons and about which propositions to adopt or abandon as premises of different modules. At present, the RDRMS embodies the same approach as do traditional RMSs, namely that these decisions are the responsibility of the external system (or systems). But since these decisions sometimes can depend on what reasons have already been recorded, we are investigating techniques by which the RDRMS can make some of these decisions for the external reasoner when the external reasoner informs the RDRMS of its purposes. Of course, these decisions may also depend on other facts, such as how hard it was for the reasoner to discover the belief, so we cannot expect the RDRMS to make all such decisions on its own.

6 Comparison with other work

Reason maintenance is the standard approach to belief revision, backtracking, and default reasoning [de Kleer *et al.*, 1977; Doyle, 1979; Goodwin, 1987]. Morris [1988] has shown that a standard RMS can support planning and dependency-directed replanning within the classical planning framework. But developing an architecture for reason maintenance and replanning subject to rational control will require significant modification of existing techniques.

As mentioned above, we use the RDRMS to extend the dominance-proving architecture for planning with partially satisfiable goals [Wellman, 1990a]. This decision-theoretic approach fits well with our goal of rational planning. We also make use of the methods, currently under active investigation, for decision-theoretic control of reasoning, in which the reasoner explicitly estimates and compares the expected utilities of individual search or inference steps [Dean, 1990; Horvitz *et al.*, 1989; Russell and Wefald, 1989]. These are very important in making some of the larger, nonroutine decisions arising in the planning and belief revision tasks. But our aim is to identify implicitly rational decision-making procedures whenever possible by separate, off-line decision-theoretic analyses based on the computational tradeoffs associated with RDRMS operations.

In the traditional, generative approach to planning, the planner takes an initial state and a goal, and constructs a sequence (or partially ordered set) of actions to achieve the goal. Most work on generative planning has concentrated on planning from scratch, though the replanning task has been studied off and on over the years [Fikes *et al.*, 1972; McDermott, 1978; Wilkins, 1988] with some success. But generative planning has focused—with a few recent exceptions—on planning without probability or utility information. Many of these techniques therefore require some reworking before they can be said to produce rational plans, and the issue of rational control of the planning process is just now beginning to be studied [Dean, 1990; Smith, 1988].

Another approach is the "reactive" approach to planning and action, which seeks to avoid execution-time planning by "compiling" all necessary behaviors into directly applicable forms [Brooks, 1986; Georgeff and Lansky, 1987; Rosenschein and Kaelbling, 1986; Schoppers, 1987]. Our approach fits well with such compilation, since we seek to develop implicitly rational planning and decision-making procedures. More specifically, decision-theoretic analyses of planning and replanning apply also to the tradeoff between planning and reacting. Since providing a compiled response for every contingency is usually not feasible, our approach is to provide explicit contingency procedures only when they increase the expected utility of the overall plan, taking both planning effort and execution-time utilities into account. In addition, our assumption of distributed execution authorities and replanners explicitly accounts for the reactive abilities of the distributed execution modules.

The constraint-based approach to scheduling [Fox, 1987] complements the generative planning approach in many ways, as it does focus on issues of utility and optimization. At the same time, it has somewhat lower aspirations, since the focus is on scheduling activities within the confines of an overall plan, rather than on selecting the activities in the first place. In addition, it has generally not addressed issues of probability, and its concepts of preference are not directly translatable to expected utility. But many of the fundamental optimization techniques have been refined and integrated with AI reasoning techniques in this area, and we will draw on these in constructing methods for rational control of the planning process and construction of rational plans.

The case-based approach to planning [Collins *et al.*, 1989; Hammond, 1986; Minton *et al.*, 1989] shares with ours the aim of incremental construction and repair of plans. Some case-based reasoners make significant use of recorded reasons for beliefs and plans, for example Carbonell's [1986] method of derivational analogy. By and large, however, most work on case-based reasoning focuses on issues of conceptual organization and retrieval rather than reason maintenance. In addition, it is not too inaccurate to say that research on case-based reasoning has largely ignored issues of rationality. Work in this area has generally aimed to make all planning operations habitual, so that plans are constructed simply by remembering old plans or plan fragments, along

with patches that should be applied to these plans for specific circumstances. We also aim to develop habitual rules for plan construction whenever possible (for example, default plans and default decisions in guiding planning), but to produce and apply these rules in a principled way amenable to formal analysis and directed improvement. In particular, we use the same probabilities that guide decision-making in novel circumstances to also guide the formation and memorization of habitual rules, remembering and forgetting rules and past plans based on estimates of their incremental expected utility. We believe our approach will make it easier to combine techniques from the case-based literature with the more formal techniques developed in the generative planning and constraint-based scheduling literatures.

Most work on distributed AI has not addressed issues of belief or plan revision, focusing instead on distributing the effort involved in ordinary reasoning and planning [Bond and Glasser, 1988]. Very recently, however, some distributed RMSs have been developed. While these represent important first steps, they are not at present suitable bases for rational plan revision. For example, the distributed nonmonotonic TMS of Bridgeland and Huhns [1990] ensures global consistency among different agents about the information they share. Maintaining this degree of coherence is not always feasible in large databases, nor even desirable in cases in which the various agents have different information sources and perspectives. Another relevant work is the distributed ATMS of Mason and Johnson [1989]. This system permits a large degree of inconsistency among the different knowledge-bases, and so is closer to the aims of the RDRMS. But their system also does not address the issue of rationality, and limits the representation of reasons to monotonic implications.

7 Conclusion

Reason maintenance promises to play an important role in replanning, but to prove useful for large-scale activities, the techniques must be capable of incremental application that does not incur the costs of global reconsideration. Furthermore, reasons must reflect likelihoods and preferences about events related to the activity, and revision policies must be sensitive to computational tradeoffs inherent in the process of modifying plans and beliefs.

To support this behavior, we extend traditional reason maintenance techniques to make use of instructions, expectations, and preferences in deciding how to establish and revise beliefs and plan elements. In our conception, the *rational distributed reason maintenance service* maintains only as much coherence and grounded support as is called for by the planner's purposes. In essence, the fundamental operations of finding supporting arguments and pursuing consequences become flexible rather than routine, with different sorts of reasons indicating different sorts of processing during revisions.

Together with the dominance-oriented approach to decision-theoretic planning, the RDRMS represents a general architecture for reasoned replanning of large-scale activities. Although much remains to be worked out, the RDRMS concept provides both a tool for investigating representational issues in belief and preference specification and an analytical framework for studying computational issues in revising beliefs and plans. Because the issues of rationality highlighted by this approach are generally not even expressible within standard RMSs and classical models of planning, we expect this line of research to yield some new insights into the dynamics of the planning and replanning process.

References

[Bond and Glasser, 1988] Alan Bond and Les Glasser, editors. *Readings in Distributed Artificial Intelligence.* Morgan Kaufmann, San Mateo, CA, 1988.

[Bridgeland and Huhns, 1990] David Murray Bridgeland and Michael N. Huhns. Distributed truth maintenance. In *Proceedings of the National Conference on Artificial Intelligence.* AAAI, 1990.

[Brooks, 1986] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14–23, 1986.

[Carbonell, 1986] Jaime G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning 2.* Morgan Kaufmann, 1986.

[Collins *et al.*, 1989] Gregg Collins, Lawrence Birnbaum, and Bruce Krulwich. An adaptive model of decision-making in planning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 511–516, 1989.

[de Kleer, 1986] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[de Kleer *et al.*, 1977] Johan de Kleer, Jon Doyle, Guy L. Steele Jr., et al. AMORD: Explicit control of reasoning. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, pages 116–125, 1977.

[Dean, 1990] Thomas Dean. Decision-theoretic control of inference for time-critical applications. Technical Report CS-90-44, Department of Computer Science, Brown University, Providence, RI, 1990.

[Doyle, 1979] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12(2):231–272, 1979.

[Doyle, 1985] Jon Doyle. Reasoned assumptions and Pareto optimality. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 87–90, 1985.

[Doyle, 1988] Jon Doyle. Artificial intelligence and rational self-government. Technical Report CS-88-124,

Carnegie-Mellon University Computer Science Department, 1988.

[Doyle, 1990] Jon Doyle. Rational belief revision. In *Proceedings of the Third International Workshop on Nonmonotonic Reasoning*, Stanford Sierra Camp, CA, June 1990.

[Fikes *et al.*, 1972] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[Fox, 1987] Mark S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Pitman and Morgan Kaufmann, 1987.

[Gärdenfors, 1988] Peter Gärdenfors. *Knowledge in Flux: Modeling the Dynamics of Epistemic States.* MIT Press, Cambridge, MA, 1988.

[Georgeff and Lansky, 1987] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 677–682, 1987.

[Goodwin, 1987] James W. Goodwin. *A theory and system for non-monotonic reasoning.* PhD thesis, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1987. Linköping Studies in Science and Technology, No. 165.

[Hammond, 1986] Kristian Hammond. *Case-based Planning: An Integrated Theory of Planning, Learning and Memory.* PhD thesis, Yale University, 1986.

[Hewitt, 1986] Carl Hewitt. Offices are open systems. *ACM Transactions on Office Information Systems*, 4:271–287, 1986.

[Horvitz *et al.*, 1989] Eric J. Horvitz, Gregory F. Cooper, and David E. Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1121–1127, 1989.

[Lansky, 1988] Amy L. Lansky. Localized event-based reasoning for multiagent domains. *Computational Intelligence*, 4:319–340, 1988.

[Mason and Johnson, 1989] Cindy L. Mason and Roland R. Johnson. DATMS: A framework for distributed assumption based reasoning. In Les Gasser and Michael N. Huhns, editors, *Distributed Artificial Intelligence*, chapter 13, pages 293–317. Morgan Kaufmann, San Mateo, CA, 1989.

[McDermott, 1978] Drew McDermott. Planning and acting. *Cognitive Science*, 2:71–109, 1978.

[Minsky, 1975] Marvin Minsky. A framework for representing knowledge. In Patrick Henry Winston, editor, *The Psychology of Computer Vision*, chapter 6, pages 211–277. McGraw-Hill, 1975.

[Minton *et al.*, 1989] Steven Minton, Jaime Carbonell, Craig Knoblock, et al. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40:63–118, 1989.

[Moore, 1985] Robert C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[Morris, 1988] Paul Morris. Truth maintenance-based planning with error recovery. In *Proceedings of the Rochester Planning Workshop*, pages 18–19, 1988. Extended Abstract.

[Rich, 1985] Charles Rich. The layered architecture of a system for reasoning about programs. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1985.

[Rosenschein and Kaelbling, 1986] Stanley J. Rosenschein and Leslie Pack Kaelbling. The synthesis of digital machines with provable epistemic properties. In Joseph Y. Halpern, editor, *Theoretical Aspects of Reasoning About Knowledge: Proceedings of the 1986 Conference*, pages 83–98. Morgan Kaufmann, 1986.

[Russell and Wefald, 1989] Stuart Russell and Eric Wefald. Principles of metareasoning. In *First International Conference on Principles of Knowledge Representation and Reasoning*, pages 400–411, 1989.

[Savage, 1972] Leonard J. Savage. *The Foundations of Statistics.* Dover Publications, New York, second edition, 1972.

[Schoppers, 1987] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039–1046, 1987.

[Shoham, 1988] Yoav Shoham. *Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence.* MIT Press, 1988.

[Smith, 1988] David E. Smith. A decision theoretic approach to the control of planning search. Technical Report LOGIC-87-11, Department of Computer Science, Stanford University, 1988.

[Vilain, 1985] Marc B. Vilain. The restricted language architecture of a hybrid representation system. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 547–551, 1985.

[Wellman, 1987] Michael P. Wellman. Dominance and subsumption in constraint-posting planning. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 884–890, 1987.

[Wellman, 1990a] Michael P. Wellman. *Formulation of Tradeoffs in Planning Under Uncertainty.* Pitman and Morgan Kaufmann, 1990.

[Wellman, 1990b] Michael P. Wellman. Fundamental concepts of qualitative probabilistic networks. *Artificial Intelligence*, 1990.

[Wilkins, 1988] David E. Wilkins. *Practical Planning.* Morgan Kaufmann, Los Altos, CA, 1988.

Computational Considerations in Reasoning about Action

Matthew L. Ginsberg*
Computer Science Department
Stanford University
Stanford, California 94305
ginsberg@cs.stanford.edu

Abstract

Any first-principles declarative planner will need to be constructed from an underlying declarative system that reasons about action. In this paper, we point out that if such a planning system is to be computationally viable, the associated declarative description of actions must satisfy at least two broad conditions. First, it will need to be *event-driven*, so that fluents that hold in a particular situation can be propagated into the future at reasonable computational cost. Second, it will need to be *anytime* in the sense that partial or approximate answers to queries can be provided in the presence of computational resource constraints. We suggest that the first these goals can be achieved by taking the truth values assigned to sentences to be functions from the temporal elements into a set of basic values, and that the second can be achieved by viewing temporal operators as functions on these functional truth values.

1 Introduction

Existing planning systems can be grouped into three broad categories: expert planners, general-purpose planners, and first-principles planners.

Expert planners, of which there are many, are essentially applications of expert-systems technology to planning problems. The situation in which a particular agent finds itself is classified to determine which of a predetermined set of actions is most likely to be effective in achieving the agent's goals. There has been some interest recently in constructing the expert decision rules automatically [Drummond, 1988], but the approach itself must inevitably be limited by the fact that the agent involved has no real idea what's going on – it is simply mindlessly applying rules that govern its behavior. The ensuing brittleness is typical of expert systems generally.

General-purpose planners, of which there are few, attempt to address this difficulty by working with a set of action descriptors that describe the possible actions in some domain, and then constructing a plan to achieve a particular goal using methods that are independent of the domain in which the agent finds itself. This work began with STRIPS [Fikes and Nilsson, 1971]; the most successful existing planner of this sort is arguably Wilkins' SIPE system [Wilkins, 1988].

There are two difficulties with the general-purpose approach. The first is that the computational complexity of planning problems is typically very high, making it impractical to generate a complete plan that is guaranteed to achieve a particular goal. SIPE addresses this difficulty by restricting the form of the actions it can consider.

Unfortunately, the nondeclarative restrictions placed on the form of the actions being considered generally make these planners nonuniversal; there are domains for which any particular restriction is inappropriate. This is the essence of the second difficulty: General-purpose planners, by committing at a fundamental level to a specific description of actions, inherit some (but by no means all) of the brittleness of their expert-planning predecessors. "General-purpose" planners are only general-purpose within the bounds established by assumptions embodied in the form of the action descriptors.

First-principles planners (of which there are none) attempt to address these difficulties by viewing planning as a purely declarative activity, specifically, by viewing it as theorem proving set against the background of a declarative system that describes actions in a particular domain.

This idea is an old one, dating back to Green's QA3 system [Green, 1969]; as work on declarative systems generally has advanced, the attractiveness of the approach has remained. With the development of nonmonotonic reasoning, for example, it was suggested that this general declarative notion could be applied to construct a planner that would be able to jump to conclusions while building its plans. It was later suggested that assumption-based truth maintenance [de Kleer, 1986], another general declarative technique, might bear on the problem of debugging plans that appear to be nonmonotonically sound but that closer inspection reveals to be flawed in some way [Ginsberg,

*This work has been supported by the Rockwell Palo Alto Laboratory, by General Dynamics and by NSF under grant number IRI89-12188.

1990b].

The reason that there are no established planners of this type is that the underlying declarative descriptions of action are themselves lacking. The best-known reason for this is the infamous Yale shooting problem [Hanks and McDermott, 1987], although a variety of researchers have found solutions to this particular difficulty.

A more fundamental problem with declarative descriptions of action is that they are simply unsuitable for inclusion in planners. The approach suggested in [Green, 1969] and reiterated in [Genesereth and Nilsson, 1987] is still a valid one – given a monotonic description of a domain, it is indeed possible to view planning as theorem proving. The difficulty is that it is not practical to do so.

The reason for this can be seen by considering the frame axiom. Here is a typical nonmonotonic rendering of it:

$$\texttt{holds}(f,s) \wedge \neg\texttt{ab}(a,f,s) \supset \texttt{holds}(f,\texttt{result}(a,s)) \tag{1}$$

Informally, this axiom says that if some fluent f holds in a situation s and the action a is not abnormal in that it reverses f when executed in the situation s, then f will continue to hold after the action is completed.

There are technical problems with this definition, but they can be avoided [Baker and Ginsberg, 1989, and many others]. But an overwhelming *computational* difficulty can be seen if we imagine using (1) to propagate a set of fluents through a long sequence of actions. The application of (1) for each action and to each fluent will result in a prohibitively large number of consistency checks, making the system unusable in practice.

This problem is avoided in general-purpose planning systems by using a nondeclarative description of action that has more attractive computational properties. In STRIPS, for example, actions are described in terms of add and delete lists, reducing the complexity of the reasoning enormously. The STRIPS formalism cannot deal with the inferred consequences of actions, however, as was observed in [Lifschitz, 1986]. (This is called the *ramification problem* in [Finger, 1987].) A partial solution to this difficulty can be found in [Ginsberg and Smith, 1988], but the approach presented there continues to describe actions in nondeclarative terms.

The intellectual foundation for the work described in this paper lies in an attempt to present a declarative description of the work in [Ginsberg and Smith, 1988]; we have tried to develop a formalization of action that will be computationally viable in the situations likely to arise in planning. The two specific heuristic commitments that we will make are the following:

First, we will assume that fluents typically survive long sequences of actions before being needed; a robot should be able to put a wrench in its toolbox, perform most of its day's activities, and conclude at a single stroke that the wrench is still in the toolbox. We will describe this by saying that our formalization of action needs to be *event-driven* in the sense that propagating fluent values through idle periods does not incur significant computational expense.

Second, we will commit ourselves to a system that can reason about actions in an *anytime* fashion; the word appears to originate in [Dean and Boddy, 1988]. By this we mean that the system, when asked the value of a fluent in a specific situation, will produce some answer quickly, perhaps modifying that answer as necessary if allowed to consider more subtle features of the situation involved. It is generally recognized that planning problems are sufficiently difficult that approximate answers are inevitable; we are simply requiring that this sort of computational response be present in the description of action that underlies the planner itself.

The reason that we have chosen to discuss these two problems in this paper is not that there are no others (there are), but that the solutions to them are linked. Roughly speaking, both difficulties can be addressed by taking the truth value assigned to a sentence to be not a single value such as "true" or "false," but a function from a set of time points into such values.

The reason that this approach leads to an event-driven description is that it allows us to conveniently describe the expected future behavior of fluents in a compact fashion. Instead of saying, "The wrench is in the toolbox at 9:15," and, "Things in toolboxes tend to remain there," we can simply say, "The wrench is expected to be in the toolbox for the rest of the day," meaning that the truth value assigned to the sentence

$$\texttt{in}(\texttt{wrench},\texttt{toolbox})$$

is a function that maps the entire temporal interval from 9:15 to 5:00 to the value t (or perhaps dt – true by default – if we are prepared to admit the possibility of subsequent information reversing our conclusions). The problem of making our description event-driven now becomes essentially a matter of finding a data structure for the functional truth values that efficiently encodes the behavior of fluents that change only infrequently.

The idea of taking truth values to be temporal functions also bears on our requirement that the implementation of our formalism exhibit anytime behavior. As an example, consider a sentence such as, "One second after the valve is closed, the pressure will increase," which we will write somewhat schematically as

$$\texttt{delay}(\texttt{closed-valve}) \supset \texttt{pressure} \tag{2}$$

where `delay` is an operator that we will use to push the temporal description of the valve one second into the future.

From a formal point of view, the `delay` operator appearing in (2) is a *modal operator*, since it accepts as an argument not an object in our language, but a declarative sentence. It is shown in [Ginsberg, 1990a]

that when truth values are taken to be more descriptive than simply elements of the two-point set $\{t, f\}$ (true and false, respectively), it is possible to view modal operators as functions on the truth values of their propositional arguments. In (2), the modal operator `delay` corresponds to the *function* `delay` that is given by

$$[\texttt{delay}(f)](t+1) = f(t) \qquad (3)$$

Note that `delay` accepts a function as an argument and returns a function as its result, since the truth values that we are using are themselves functional.

To see that this interpretation leads to anytime behavior, we need to make one more observation: The basic purpose of a deductive system is to determine what truth value should be assigned to a particular query. Now note that when considering a query q, we may well encounter a modal operator, requiring us to apply the corresponding function (as in (2) or (3)) to the truth values of the propositional arguments (`closed-valve` in (2)). But what are we to use for these truth values? We can use either the result of invoking the theorem prover recursively on the propositional arguments themselves, or use the values that can be obtained by simply searching for the given propositions in the database. Using these latter values as approximations for the former leads to a system that produces some answer quickly, but may modify that answer on further consideration. Perhaps there is a deductive demonstration that the valve in (2) will be open at some particular future time, and so on. If the analysis of the embedded sentences produces still further modal expressions, anytime behavior will result as the system makes and then examines assumptions about the truth values assigned to these embedded sentences.

The remainder of this paper will consider each of these ideas in turn, and then show an example of an implementation of them being used to analyze a shooting scenario similar to that appearing in [Hanks and McDermott, 1987]. The implementation is built using the multivalued theorem proving system MVL [Ginsberg, 1988, Ginsberg, 1989].

2 Truth values

We remarked in the introduction that we intend to label sentences in our declarative database with functional truth values that include information about the truth or falsity over time (or default truth/falsity, etc.) of the sentence involved. The reason that we are comfortable doing this is that the labels so constructed retain the mathematical structure of the original "instantaneous" labels, in that we can combine them, negate them, disjoin and conjoin them, and so on.

It is these operations of conjunction, disjunction and so forth that underlie the semantics of any declarative system. Specifically, if we have labels x and y for sentences p and q respectively, we need a way to construct a label $x \vee y$ for the disjunction of p and q, and so on. In [Ginsberg, 1988], a mathematical structure called a *bilattice* was introduced that consists of a set together with just these combining operations. Although the formal definition will not play a significant role in the remainder of this paper, we include it here in the interests of completeness:

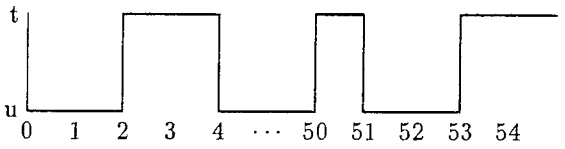


Figure 1: A fluent that is true three times

Definition 2.1 *A* bilattice *is a sextuple* $(B, \wedge, \vee, \cdot, +, \neg)$ *such that:*

1. $(B, \wedge, \vee)$ *and* $(B, \cdot, +)$ *are both complete lattices, and*
2. $\neg : B \rightarrow B$ *is a mapping with:*
   (a) $\neg^2 = 1$, *and*
   (b) $\neg$ *is a lattice homomorphism from* $(B, \wedge, \vee)$ *to* $(B, \vee, \wedge)$ *and from* $(B, \cdot, +)$ *to itself.*

The bilattice operations $\wedge$, $\vee$ and $\neg$ all correspond to the usual logical notions, while $+$ corresponds to combination of evidence and is used to combine the truth values obtained from separate lines of reasoning to a single conclusion. Many more details of the bilattice work, together with a discussion of the philosophical ideas underlying the approach, can be found in [Ginsberg, 1988].

What is of interest to us here is the fact that if B is some bilattice, then B^2, the collection of ordered pairs of elements of B, inherits a bilattice structure from B where all of the bilattice operations are computed pointwise. (The construction is analogous to the construction of the Cartesian plane $\mathbb{R}^2$ as the product of two copies of the real line.) More generally, for any set S, the set B^S of functions from S into B inherits a bilattice structure from the set B.[1]

It follows that if we have some set T of time points, then the set B^T of functions from T into the "base" set of truth values B has the structure required of a set of truth values. As an example, if we take T to be the integers, then the graph in Figure 1 shows the truth value assigned to a fluent that is true for two units of time at $t = 2$, for one unit of time at $t = 50$, and for all time after $t = 53$.

Our event-based philosophy now corresponds simply to a data structure that represents these truth values by listing the points at which the value changes. In Figure 1, for example, we record the fact that the fluent is unknown at time 0, true at time 2, unknown at time 4, and so on; values at a total of six points are recorded.

[1] There is no real difference between viewing the set B^2 as the set of ordered pairs of elements of B, or as the set of functions from the two-point set $\{1,2\}$ into B.

Determining the value of the fluent at any intermediate time t is a matter of walking along the graph until the next event is later than t, and taking the value at the last time encountered. Note that the computational effort required to determine the value of the fluent in Figure 1 is completely independent of the length of the gap between times 4 and 50.

Extensions

The ontological shift that we are proposing does not in and of itself commit us to any specific computational or representational strategy. As an example, the simple representation scheme that we described in the previous paragraph can easily be extended in a variety of ways:

1. The set T of time points only needs the structure of a partial order in order for the above approach to work; to determine the value of a fluent f at some particular time t, we walk our way along the function until we find ourselves between two points t_0 and t_1 such that
$$t_0 \leq t < t_1,$$
so that t is no earlier than t_0 and t_1 is later than t. The value of f at t is then the value taken at t_0.
2. As an example, taking the above partial order to be the continuous real line allows us to avoid our earlier implicit assumption that time was discrete. This particular choice commits us to fluents being true over half-open intervals $[x, y)$ only, but this can be avoided by introducing auxiliary elements x^+ for each $x \in \mathbb{R}$ such that the half-open interval $[x, y^+)$ in fact denotes the closed interval $[x, y]$.
3. Another example involves taking the elements of the partial order to be action sequences, where an action sequence a_2 temporally follows a sequence a_1 whenever a_2 is an extension of a_1. Nonlinear action sequences can be handled by weakening the partial order to cater to possible linear action sequences consistent with a given nonlinear one.
4. It is also possible to extend the scheme by introducing "decay functions" that describe how a fluent's truth value is expected to change as time goes by. In Figure 1, for example, the fluent's truth values do not change at all as time passes; a more realistic example might involve the truth value of the fluent falling from t to dt at times 3 and 54, as shown in Figure 2. Here, our confidence in the truth of the fluent decays as time passes, corresponding to the application of a nonmonotonic frame axiom. As before, information is recorded only when the truth value of the fluent changes from the expected one, so we still need to record information only about the "events" at times 0, 2, 4, 50, 51 and 53. By changing the set of base truth values to which the temporal functions map, this idea can be extended to include a wide variety of temporal behaviors, such as the probabilistic decay functions discussed in [Dean and Kanazawa, 1988].

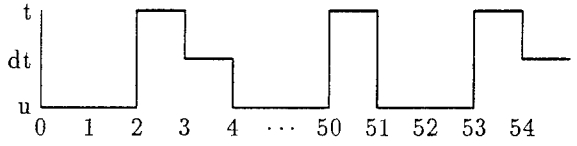


Figure 2: A default frame axiom

In all of these cases, the computational efficacy of the scheme that we have proposed is preserved.

The frame axiom

The approach we have suggested encodes the information that is normally captured by a frame axiom such as (1) in two separate ways. First, the actual default behavior assigned to some particular fluent f is encoded in a truth value such as that appearing in Figure 2, which explicitly indicates the default truth of the fluent at times following times when it is known to be true with certainty.

We have thus far avoided the question of how we obtain truth values such as that appearing in Figure 2 in the first place. What we would like to do is to have a sentence in our database such as the following:

> If the robot puts a tool in its toolbox, then the tool is definitely in the toolbox at that time, and can be assumed by default to be in the toolbox at subsequent times.

Note first that the truth value to be assigned to the conclusion of the rule (that the tool is in the toolbox) is not the same as the truth value of the premise; the premise (that the tool is being put into the toolbox) holds only instantaneously, while the conclusion (that the tool is actually located in the toolbox) holds over a wide range of times. This is a technical issue that we will consider in the next section and is identical in principle to the appearance of the `delay` operator in the introduction.

More importantly, note that the future behavior of any particular fluent (in this case, the location of the tool) is determined not by applying some blanket frame axiom such as (1), but instead by an axiom describing this future behavior when the fluent is first asserted (i.e., when the robot puts the tool in the toolbox).

This is an important distinction between our approach and the conventional one. Computational issues aside, we prefer our approach on purely philosophical grounds, since it is *not* the case that the frame axiom applies to all, or even most of the fluents we encounter in everyday life.

As an example, consider the problem of entering a crosswalk when there is a car five feet away approaching at 60 MPH. Are we to apply the frame axiom to the fact that the car is five feet from the crosswalk, or to the fact that it is moving at a high speed? Clearly

to the latter, although there is no information in (1) indicating that we should do so.

This is related to the well-known *problem of induction* [Skyrms, 1966]. How is it that we know to apply the frame axiom to a predicate such as "blue" or "green" but not to one such as "grue" (green until July 10th but blue subsequently) or "bleen?" Although we have not provided an answer to this question, we have indicated clearly the declarative point at which such an answer is used – in the description of the expected future behavior of newly established fluents. Similar observations have also been made in [Myers and Smith, 1988].

3 Modal expressions

Let us return to the observation we made in the last section that the truth value to be assigned to the consequent of some rule is often not the same as the truth value of the antecedent. In the introduction, we handled a situation such as this in (2) by introducing a modal operator m and writing

$$m(a) \supset c$$

where a is the antecedent and c is the consequent. The modal operator m changes the truth value of a so that the truth value of c is modified correctly by the above rule. The two examples of this that we have seen thus far involve a modal operator `delay` that delays the truth value of the antecedent by a fixed amount of time, and an operator `propagate` that was hinted at in the previous section and that is responsible for inserting the consequent into the database with a complete "future history" if appropriate.

We will discuss these two operators in some detail shortly, but let us continue to examine general issues first. The idea that modal operators can be viewed truth-functionally (i.e., as functions on the truth values of the sentences on which they operate) is an old one in the philosophical community, but was discarded in favor of Kripke's possible-worlds approach [Kripke, 1971] when it was realized that there simply are not enough functions on the two-point set $\{t, f\}$ to correspond to all of the interesting modal expressions that one might wish to consider.

In [Ginsberg, 1990a], however, it was pointed out that if truth values are taken from an arbitrary bilattice instead of from the set $\{t, f\}$, it becomes practical to view modal operators truth-functionally after all; in fact, the resulting construction is a generalization of Kripke's.

Anytime behavior

An additional advantage of viewing modal operators truth-functionally is that the associated declarative systems naturally exhibit anytime properties; this can probably be made clearest by an example from PROLOG. Consider the following program:

```
landlubber(X) :- animal(X), not(fly(X)).
fly(X) :- bird(X), not(penguin(X)).
animal(X) :- bird(X).
penguin(X) :- bird(X), tuxedo(X).
bird(opus).
tuxedo(opus).
```

Animals that cannot fly are landlubbers, birds can fly unless they are penguins, and birds in tuxedos are penguins. Opus is a bird wearing a tuxedo. Is he a landlubber?

Ignoring inadequacies in our representation of the domain, the interpreter begins by noting that it can prove that Opus is an animal, and therefore that he is a landlubber unless he can be shown to fly. A new proof process is therefore begun with the intention being to prove that Opus can fly.

Since Opus is a bird, he can fly unless he can be shown to be a penguin. Yet another proof process is begun; since this one succeeds in showing Opus to be a penguin, he cannot be shown to fly and the original query (is Opus a landlubber?) succeeds.

What is proposed in [Ginsberg, 1990a] is that it should be possible to interrupt this procedure at the points where new proof attempts are generated. Thus, when creating the attempt to prove that Opus can fly, we note that since there is nothing in the database indicating explicitly that he can, we can tentatively label

`fly(opus)`

as unknown, and therefore assign `not(fly(opus))` the value of true using PROLOG's negation-as-failure rule. This allows us to tentatively confirm the original query.

Given more time, we can work on the goal `fly(opus)`, noting that this spawns the subgoal `penguin(opus)`. Once again, we break the inference process, using the fact that `penguin(opus)` is missing from our database to conclude tentatively that `not(penguin(opus))` is true and therefore that Opus can fly, so that the original query should fail. Finally, given still more time for reflection, we realize that Opus is a penguin and therefore a landlubber after all.

PROLOG's treatment of negation is as an operator that returns t unless the truth value of the argument is itself t; specifically, if some sentence p is unknown, negation-as-failure treats `not`(p) as true. Viewed in this fashion, PROLOG's negation is a modal operator in our sense. We are proposing two extensions to this idea:

1. Extending the notion of a modal operator to include temporal operators such as those that arise when reasoning about action.
2. Using these modal operators as semantic markers for points at which the inference process can be suspended and an approximate answer computed.[2]

[2]This suggests the introduction of a modal operator 1 that doesn't modify the truth value of its argument at all, but serves only to mark a point where inference can be suspended. This idea is unexplored at this point.

These ideas are described in greater detail in [Ginsberg, 1990a].

Temporal operators

Given that we take the view that temporal operators can be described by giving their functional behavior and then incorporating them into our declarative language, what operators are required in a system that reasons about action?

We will clearly need an operator delay to separate the occurrence of an action from the appearance of its effects, and another operator propagate that allows us to construct temporal truth functions such as the one appearing in Figure 2.

In fact, we appear to need nothing else; the axiomatization appearing in the appendix uses no modal operators other than these two. The description of delay is as appearing in (3), while propagate is defined recursively as[3]

$$[\texttt{propagate}(f)](t) =$$

$$\begin{cases} f(t), & \text{if } t = 0 \text{ or } f(t) \neq u; \\ \texttt{decay}[\texttt{propagate}(f)(t-1)], & \text{otherwise.} \end{cases}$$

The function decay might be given by, for example:

x	decay(x)
t	dt
dt	dt
f	df
df	df
u	u

This decay function maps any truth value into its default version, corresponding to a nonmonotonic frame axiom. A monotonic frame axiom would simply take $\texttt{decay}(x) = x$. As an example, Figure 3 shows the result of applying propagate to a temporal function that changes from t to f. Figure 4 shows the result if the frame operator is chosen to be monotonic.

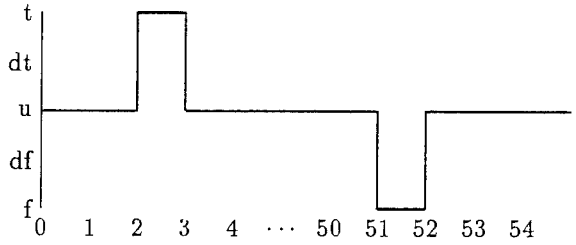
From these two operators, we can build a declarative description of action that has the desired properties of being event-driven and anytime. If a is an action that causes a fluent f to be true in a persistent way (such as putting a tool in the toolbox), we write[4]

$$\texttt{propagate}(\texttt{delay}(a)) \supset f \qquad (4)$$

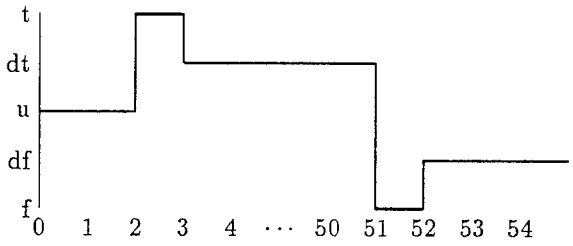
[3] The implementation of propagate does not follow this definition directly, since this would be horrendously inefficient. Instead, we use a monotonic propagation function (as described in the appendix), and simply drop information about any "events" at which a truth function becomes unknown.

[4] The axiom (4) is not quite satisfactory as it stands, because it is awkward to combine it with domain constraints describing ramifications of the action in question. This is handled in the appendix by reifying the fluents so that they can be treated as objects in our language, and replacing (4) with an axiom like

$$\texttt{causes-persistently}(a, f) \wedge$$

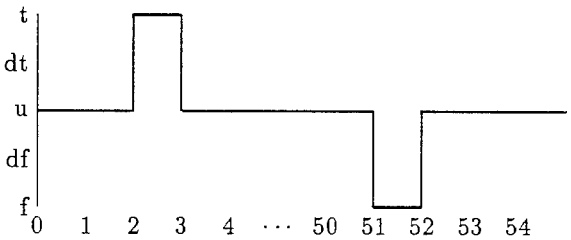


A temporal function f

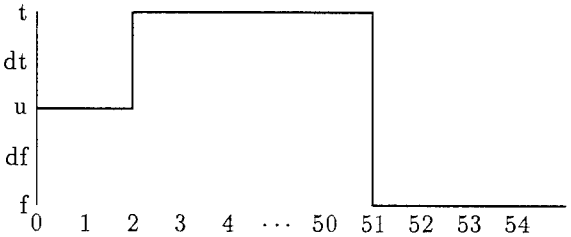


The function propagate(f)

Figure 3: Applying the frame axiom



f



propagate(f)

Figure 4: A monotonic frame axiom

while if a causes f to be true only instantaneously (like striking a match causing a light), we write

$$\texttt{delay}(a) \supset f$$

Efficiency considerations

Suppose that we consider the example in Figures 3 and 4 a bit more closely. In the example in Figure 4, for example, note that there is no point to attempting to show that the fluent holds at time 4 – we already know this by virtue of the application of our monotonic propagation function. This implies the following:

> *When invoking the prover on a sentence p in order to evaluate a modal expression $m(p)$, one should only investigate proofs that will affect not only the truth value of p, but the truth value of $m(p)$ as well.*

In fact, the situation is a bit more subtle still; consider Figure 3.

Suppose that we are interested in showing that the fluent f holds at time 53, perhaps because f is a precondition to an action that we would like to take at that time, or perhaps because the restriction mentioned above implies that this is the only information about f that is of interest to us.

In the initial situation in which we know nothing, it follows that we should try to show that f holds at any time before $t = 53$, since this value will then be propagated to the time of interest. Although showing that f holds at time 54 does effect the value of $\texttt{propagate}(f)$, it does not do so in an interesting way. This means that we should replace the above principle with the following stronger one:

> *When invoking the prover on a sentence p in order to evaluate a modal expression $m(p)$, one should only investigate proofs that will change the truth value of $m(p)$ in a way that will affect the response to the original query.*

Applying this idea can be fairly subtle. In the example we are considering, suppose that we succeed in showing that f holds at time 2, so that $\texttt{propagate}(f)$ holds by default at time 53. Now there is no point in showing that f holds, but there *is* a reason to show that the negation of f holds at some time between 3 and 53, since $\neg f$ will block the propagation of f to the time that is of interest to us.

In the example in the figure, we can show that $\neg f$ holds at time 51; now we must once again change the focus of our proof efforts as we attempt to show that f is true either at $t = 52$ or at $t = 53$.

From a conceptual point of view, this is all quite straightforward. From an implementational point of view, however, it can be rather subtle, especially since we should preserve portions of the proof tree for a fluent f even if they appear not to be relevant to $\texttt{propagate}(f)$. The reason for this is that subsequent developments may change this. In the example we have been considering, perhaps proving f has been reduced (after considerable effort) to proving g and h; when our focus changes to that of proving $\neg f$, we should retain this information in case (as happens in this example) we decide that we need to prove f after all.

$$\texttt{propagate}(\texttt{delay}(\texttt{holds}(a))) \supset \texttt{holds}(f)$$

4 An example

The ideas that we have described have been implemented using the multivalued theorem prover described in [Ginsberg, 1988, Ginsberg, 1989], which allows the user to select truth values from arbitrary bilattices and to include arbitrary modal operators in a declarative database. The precise axiomatization of actions generally and our domain in particular can be found in the appendix.

The domain we are considering involves a gun, which may or may not be loaded, and a victim (Fred), who may or may not be alive. At time 0, the gun is loaded and Fred is alive.

This domain has three actions: loading and unloading the gun (which always succeed), and shooting the gun at Fred. If the gun is loaded, firing it at Fred will kill him.[5] All of the fluents persist in a nonmonotonic fashion except that once Fred dies, he is guaranteed to stay dead.

The course of events in this domain is as follows:

Time	Event
0	Fred is alive and the gun is loaded
1	The gun is unloaded
2	The shooting is attempted and the gun is reloaded
50	The shooting is attempted
52	The gun is reloaded
53	The shooting is attempted

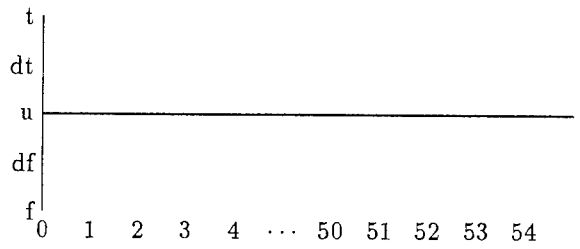
The first shooting action should fail, since the gun has been unloaded at the previous time point. The second shooting action should succeed by default, since the gun has presumably remained loaded between times 3 and 50. The third shooting action will definitely succeed, since it immediately follows a `load` action.

Given this information, the system was asked to investigate the truth or falsity of the fluent `alive` (is Fred alive?) at all times; the results are shown in Figures 5 and 6.

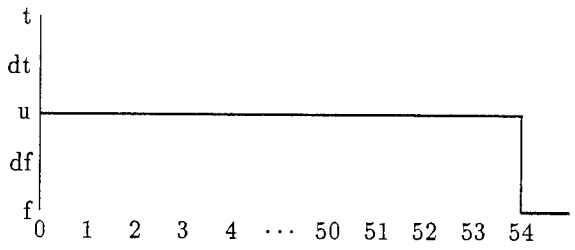
The machine reasoned as follows:[6]

[5] This is the only consequence of the shooting action. Specifically, shooting does not cause the gun to become unloaded.

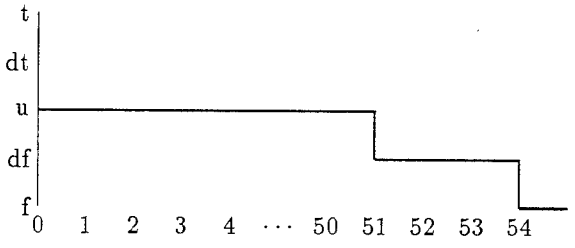
[6] The machine is reasoning backwards in time only because it uses the most recently asserted facts first, and the assertions about what actions took place at what times happen to be in chronological order. (These are the last axioms appearing in the appendix.)



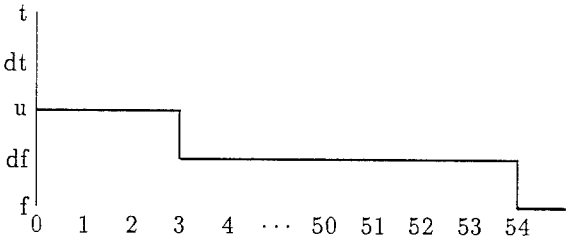
(a) Initial knowledge



(b) The third shooting action succeeds

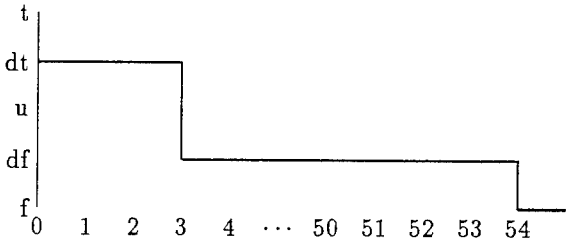


(c) The second shooting action succeeds by default

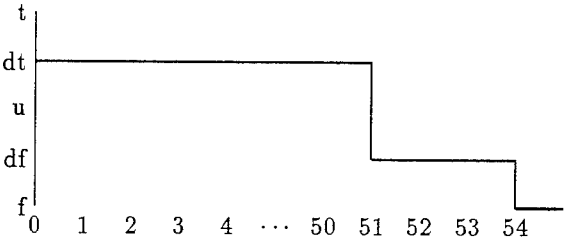


(d) The first shooting action succeeds by default

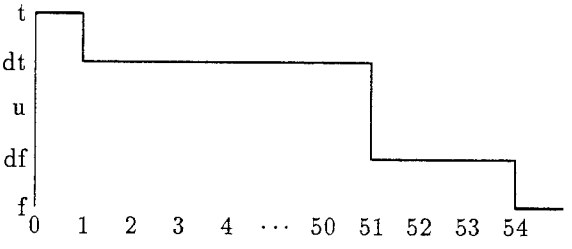
Figure 5: The shooting scenario



(e) Fred is alive by default initially



(f) Unload at $t = 1$ causes first shooting to fail



(g) In the initial situation, Fred is alive for sure

Figure 6: The shooting scenario (ctd.)

1. First, it realized that the shooting at time 53 is guaranteed to succeed, so that at times 54 and subsequently, Fred will be dead.
2. Next, it realized that the shooting at time 50 was expected to succeed, since the gun was loaded at an earlier time and presumably remained so.
3. It also decided that the shooting at time 3 was likely to succeed, since the gun was loaded in the initial situation and that fact was expected to persist.
4. The proof attempts in Figure 5 involve attempts to find times at which Fred is dead; the machine now changed its focus to trying to prove that he is alive, and concluded that he is (by default) from time 0 until the initial shooting.
5. The machine next investigated the modal expression `propagate(loaded)` a bit more closely, since this modal expression was used in its conclusion that the shootings at times 3 and 50 succeed. It discovered that the gun is not expected to be loaded at time 3 after all, and therefore that the first shooting should fail.
6. Finally, the program realized that Fred is guaranteed to be alive in the initial situation, and updated its conclusions to reflect this.

In this example, we can clearly see the two features that have been the focus of this paper – the event-driven nature of the description, evidenced by the lack of computational effort devoted to the "idle" time from $t = 4$ to $t = 50$, and the anytime nature of the analysis, shown in the shifting conclusions displayed in Figures 5 and 6.

5 Conclusion

My intention in this paper has been to argue for two things: First, the use of truth values that directly capture the complete history of fluents that change over time; second, the manipulation of these truth values using truth-functional modal operators. We have presented an implementation of our ideas that correctly analyzes a simple example similar to the shooting scenario presented in [Hanks and McDermott, 1987], but the theoretical justifications for this approach are more compelling:

1. The approach embeds the action descriptions in a full declarative language, and investigates the consequences of actions by proving theorems against this background. A system developed in this way will benefit from developments elsewhere in the theorem-proving community in a way that a more *ad hoc* approach cannot.
2. The event-driven nature of the approach allows us to reason in a computationally viable way about fluents that change value only infrequently. As discussed in Section 2, we do this without committing ourselves to any specific ontology of time or of action.
3. The natural implementation of our ideas exhibits an anytime behavior that we can expect to be present in the associated planning system as well. Furthermore, this ability to incrementally refine our conclusions is grounded in a solid formal foundation.
4. Finally, the approach we have described avoids the use for a blanket frame axiom such as (1), which is likely to fall prey to the problem of induction. Although we have presented no solution to this difficulty, our approach makes clear that such a solution will need be reflected in the declarative description of our domain, since the expected future history of any particular fluent needs to be asserted when the fluent itself is added to the database.

Appendix: Axioms used in Section 4

The axioms used in our description of action involve four separate causal predicates, as follows:

1. `causes`(a, f) means that the action a causes the fluent f to be true instantaneously.
2. `causes-persistently`(a, f) means that the action causes the fluent to be true in a way that is expected to persist into the future.
3. `causes-forever`(a, f) means that the action causes the fluent to be true in a way that is *guaranteed* to be true in the future.
4. Finally, `causes-not`(f_1, f_2) means that fluent f_1 implies the negation of fluent f_2. The fluents `alive` and `dead` are related in this fashion.

Fluents are reified using a `holds` predicate; the insertion of the reified fluents into the database is also reified using a `triggers` predicate, so that `holds` is the result of applying the modal operator `propagate` to `triggers`.

Here are the axioms associated with these predicates, expressed in a PROLOG-like style:

```
holds(F) :- causes(A,F),
            delay(succeeds(A)).
causes(A,F) :- causes-persistently(A,F).
holds(F) :- propagate(triggers(F)).
triggers(F) :- causes-persistently(A,F),
               delay(succeeds(a)), default.
triggers(F) :- causes-forever(A,F),
               delay(succeeds(a)).
not(holds(P)) :- causes-not(Q,P), holds(Q).
not(triggers(P)) :- causes-not(Q,P),
                    triggers(Q).
```

There are a couple of things to note here:

1. The MVL system has a true negation operator in addition to the modal negation-as-failure operator used in PROLOG. The `not` appearing in the heads of the last two of the above rules is true negation.
2. The sentence `default` is inserted into the database with a truth value indicating that it has value

dt at all times. This allows us to use a monotonic propagate operator, with the inclusion of the default sentence serving to distinguish causes-persistently from causes-forever.

We also need axioms expressing conditions under which an action succeeds. We assume that there is a predicate prec(a,p) that holds just in case p is a list of the preconditions of the action a, and another predicate prec-holds(p) that is true just in case every sentence in the list p holds:

```
succeeds(A) :- occurs(A), prec(A,P),
               prec-holds(P).
prec-holds([X|Y]) :- holds(X),
                     prec-holds(Y).
prec-holds([]).
```

Finally, there is a dummy action init that takes place at time -1 and is used to construct the initial situation. This action has no preconditions.

```
occurs(init).        ;true at time -1 only
prec(init,[]).
```

To describe the shooting domain specifically, we first describe the initial situation as a consequence of the init action:

```
causes-persistently(init,alive).
causes-persistently(init,loaded).
```

These axioms say that Fred is alive (and expected to remain so) and that the gun is loaded (and also expected to remain so) in the initial situation.

We also need axioms describing the various actions. Here are load and unload:

```
causes-persistently(load,loaded).
causes-persistently(unload,unloaded).
prec(load,[]).
prec(unload,[]).
```

The fluents load and unload are negations of one another:

```
causes-not(loaded,unloaded).
causes-not(unloaded,loaded).
```

Shooting is similar. It causes Fred to be dead forever and has a precondition of the gun being loaded. The fluents alive and dead are negations of one another:

```
causes-forever(shoot,dead).
prec(shoot,[loaded]).
causes-not(alive,dead).
causes-not(dead,alive).
```

Finally, we need axioms saying what occurs when:

```
occurs(unload).        ;true at time 1
occurs(shoot).         ;true at time 2
occurs(load).          ;also true at time 2
occurs(shoot).         ;true at time 50
occurs(load).          ;true at time 51
occurs(shoot).         ;true at time 53
```

When we indicate "true at time t" above, we mean that these facts are inserted into the database with truth values indicating that they are true at these times; there is no way to represent this using conventional PROLOG syntax. The three occurrences of the shooting action are combined to get a truth function similar to that shown in Figure 1 except for the fact that the occurrences are all of unit time duration.

Acknowledgement

I would like to thank Adnan Darwiche and Don Geddis for various helpful discussions.

References

[Baker and Ginsberg, 1989] Andrew B. Baker and Matthew L. Ginsberg. Temporal projection and explanation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 906–911, 1989.

[Dean and Boddy, 1988] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, 1988.

[Dean and Kanazawa, 1988] Thomas Dean and Keiji Kanazawa. Probabilistic temporal reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, 1988.

[de Kleer, 1986] Johan de Kleer. An assumption-based truth maintenance system. *Artificial Intelligence*, 28:127–162, 1986.

[Drummond, 1988] Mark Drummond. *Situated Control Rules.* Technical Report, NASA Ames Research Center, Moffett Field, CA, 1988.

[Fikes and Nilsson, 1971] R.E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Finger, 1987] Jeffrey J. Finger. *Exploiting Constraints in Design Synthesis.* PhD thesis, Stanford University, Stanford, CA, 1987.

[Genesereth and Nilsson, 1987] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, 1987.

[Ginsberg, 1988] Matthew L. Ginsberg. Multivalued logics: A uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[Ginsberg, 1989] Matthew L. Ginsberg. *User's Guide to the MVL System.* Technical Report 840-88-24, Rockwell International Science Center, 1989.

[Ginsberg, 1990a] Matthew L. Ginsberg. Bilattices and modal operators. *Journal of Logic and Computation*, 1, 1990.

[Ginsberg, 1990b] Matthew L. Ginsberg. The computational value of nonmonotonic reasoning. In *Proceedings 1990 Workshop on Nonmonotonic Reasoning*, American Association for Artificial Intelligence, Lake Tahoe, CA, 1990.

[Ginsberg and Smith, 1988] Matthew L. Ginsberg and David E. Smith. Reasoning about action I: A possible worlds approach. *Artificial Intelligence*, 35:165–195, 1988.

[Green, 1969] C. C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 4*, pages 183–205, American Elsevier, New York, 1969.

[Hanks and McDermott, 1987] Steve Hanks and Drew McDermott. Nonmonotonic logics and temporal projection. *Artificial Intelligence*, 33:379–412, 1987.

[Kripke, 1971] Saul A. Kripke. Semantical considerations on modal logic. In L. Linsky, editor, *Reference and Modality*, pages 63–72, Oxford University Press, London, 1971.

[Lifschitz, 1986] Vladimir Lifschitz. On the semantics of STRIPS. In *Proceedings of the 1986 Workshop on Planning and Reasoning about Action*, Timberline, Oregon, 1986.

[Myers and Smith, 1988] Karen L. Myers and David E. Smith. On the persistence of derived beliefs. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 1988.

[Skyrms, 1966] Brian Skyrms. *Choice and Chance: An Introduction to Inductive Logic.* Dickerson, 1966.

[Wilkins, 1988] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, San Mateo, CA, 1988.

Issues in Decision-Theoretic Planning: Symbolic Goals and Numeric Utilities

Peter Haddawy
Department of Computer Science
University of Illinois
405 N. Mathews Ave.
Urbana IL 61801
haddawy@m.cs.uiuc.edu

Steve Hanks
Dept. of Comp. Sci. & Engr. FR–35
University of Washington
Seattle WA 98195
hanks@cs.washington.edu

Abstract

The planning problem in AI has traditionally been framed as a problem of search plus deduction. But as researchers admit that the planning world may not be controlled perfectly or known completely by the planning agent this model looks less satisfying. One loses the crisp distinction between the provably good and provably bad plans, and is forced to choose among alternatives that offer various tradeoffs between likelihood of success and penalty for failure. Planning becomes a problem of choice under uncertainty.

Decision theory offers a normative model for choice under uncertainty. But applying decision-theoretic analysis to the planning problem raises questions concerning assessment of the probability and utility model. This paper centers on the utility model, paying particular attention to the role played by the agent's explicit goals. In considering how to integrate symbolic goals with numeric utilities we take into account the contribution those goals make to the practical business of constructing plans.

In this paper we explore relationships between the process of planning to achieve symbolic goals and planning to maximize utility, concentrating on relationships that must hold between the goals and the utility function. We do so in three parts: First, we show relationships that ensure consistent solutions to the problem of planning to achieve explicit goals and planning to maximize utility. Then we present a general framework for building goal-oriented utility models that allows the incorporation of explicit goals and at the same time respects the conditions that ensure a consistent relationship between utility maximization and goal satisfaction. Finally, we integrate these two results by showing the relationship between planning to achieve goals and maximizing utility with respect to goal-oriented utility functions.

1 Introduction: Planning under Uncertainty

The planning problem in AI can be expressed as follows: given a set of goals $G = \{g_1, g_2, \ldots, g_n\}$, an initial state of the world S_0, and a set of operators $\{a_j\}$, find a sequence of the a_j that will cause all the g_i to be true if executed beginning at S_0.

Simplifying assumptions about the world—that no other events will occur and that the effects of all the operators are known completely and with certainty—allow classical planning algorithms to prove (in principle at least), for any sequence of operators, that the resulting world state either provably satisfies or provably fails to satisfy all the goals. Thus the problem faced by classical planners is one of search rather than choice: a planner searches for a plan that works, but does not attempt to choose among the feasible alternatives.

As researchers admit that the planning world may not be controlled perfectly or known completely by the planning agent—as the agent is seen to be uncertain as to its past, present, or future environment—the model of search plus deduction looks less satisfying. One loses the crisp distinction between the provably good and provably bad plans, and is forced to choose among alternatives that offer various tradeoffs between likelihood of success and penalty for failure. Planning, in other words, is a problem of choice under uncertainty.

Decision theory offers a normative model for choice under uncertainty. Given again the initial world state S_0, and letting A be a sequence of operator instances $(a_1, a_2, \ldots, a_m)$,[1] we can define the expected utility of executing A in S_0 as follows:

$$\mathbf{EU}(A) \equiv \sum_{s} \mathbf{P}(s|A, S_0)\mathbf{U}(s)$$

where $\mathbf{P}(s|A,S_0)$ is the probability that executing A in S_0 will actually result in state s and $\mathbf{U}(s)$ is the utility the agent associates with world state s. Decision theory dictates that an agent perform that course of action A^* among all possible courses of action A that affords it the highest expected utility. Note that decision theory says nothing about plan generation: it dictates only how to

[1] We will refer to these sequences interchangeably as an "action," "course of action," or "plan."

choose a course of action from among a set of alternatives.

Applying decision-theoretic analysis to the planning problem raises questions in addition to those concerning how to generate alternatives. These problems center respectively around the probability model (computing the probabilities $\mathbf{P}(s|A,S_0)$) and the utility model (computing $\mathbf{U}(s)$ for those members). Important questions that arise are how one assigns probabilities to the occurrence of certain events and numeric utilities to outcome states, as well as how one knows how to "stop projecting." The last problem, known as the *horizon problem*, arises because in principle the result states s might represent the state of the world arbitrarily far in the future—there must be time such that projecting a course of action further into the future would have a negligible effect on the utility associated with that course of action.

Some work has begun on computing the probabilistic planning model—[Hanks, 1990c], for example, confronts this problem. This paper centers instead on the utility model, paying particular attention to the role played by the agent's explicit goals, the set G above.[2]

Clearly the agent's goals must play a role in building the utility model (note that goals receive no explicit mention in the decision-theoretic formalism), in that states in which the goals are satisfied should tend to be assigned higher utility than states in which they are not. Exceptions are certainly possible, however: one could imagine being able to satisfy one's goals, but at an unacceptably high cost.

In considering how to integrate symbolic goals with a utility function we must take into account the contribution those goals make to the practical business of constructing plans:

1. Goals are easily communicated to the agent, whereas numeric utilities are notoriously hard to assess consistently—see, for example, [Keeney and Raiffa, 1976, Hogarth, 1975, Savage, 1971].
2. Goals guide the search for plan alternatives, by providing indices into plan libraries or transformation strategies.
3. Goals guide the projection process in that they identify those aspects of the world that are relevant and allow the planner to ignore all others ([Hanks, 1990b]).
4. Goals solve the horizon problem in that the last time point associated with a goal is the point at which projection can terminate.

So traditional symbolic goals are crucial to the process of plan generation, and play a role in the construction of utility functions, which can then be used to compare alternative plans. In this paper we explore relationships between the process of planning to achieve symbolic goals and planning to maximize utility, concentrating on relationships that must hold between the goals and the utility function. We do so in three parts: First, we show relationships that ensure consistent solutions to the problem of planning to achieve explicit goals and planning to maximize utility. Then we present a general framework for building goal-oriented utility models that allows the incorporation of explicit goals and at the same time respects the conditions that ensure a consistent relationship between utility maximization and goal satisfaction. Finally, we integrate these two results by showing the relationship between planning to achieve goals and maximizing utility with respect to goal-oriented utility functions.

2 Satisfying Goals and Maximizing Utility

The probabilistic analogue to the goal satisfaction problem is to find that course of action that maximizes the probability of goal satisfaction. We might ask, then, for what forms of utility functions does choosing the plan that maximizes the probability of the goal lead to choosing the plan that maximizes expected utility?

The answer is that this relationship holds only for simple step utility functions, functions for which utility is a constant low value for outcomes in which the goal is not satisfied and a constant high value for outcomes in which the goal is satisfied. Such a function is shown in figure 1. Utility is represented along the vertical axis and the space of world states along the horizontal axis. G and $\overline{\mathsf{G}}$ designate the set of all states that satisfy and do not satisfy the goal, respectively.

To demonstrate this fact we first introduce some notation: assume that S_0 is the (known) initial state, and that we can characterize the goal condition G as a set of world states—the set of states in which all the g_i hold. We then define for a course of action A,

$$\mathbf{P}(\mathsf{G}|\mathsf{A}) \equiv \sum_{s \in \mathsf{G}} \mathbf{P}(s|S_0, \mathsf{A}).$$

We now prove this specific class of step functions is the only form of utility functions for which

$$\mathbf{P}(\mathsf{G}|\mathsf{A}_1) > \mathbf{P}(\mathsf{G}|\mathsf{A}_2) \Rightarrow \mathbf{EU}(\mathsf{A}_1) > \mathbf{EU}(\mathsf{A}_2) \quad (1)$$

for any two courses of action A_1 and A_2.

We first show that this form of the utility function is a sufficient condition for condition (1) to hold. Suppose

[2] Previous work in decision-theoretic planning has mostly ignored problems associated with integrating goals and utility. [Feldman and Sproull, 1975], for example, associate a fixed utility with achieving a goal, then go on to define the cost of applying various operators (also in terms of these utility units), but never confront the problem of reconciling the two numeric assignments. [Dean and Boddy, 1988] and [Horvitz, 1988] make similar assumptions: that the application provides the planner with a utility function that identifies the benefit associated with achieving a given world state. Once again, they ignore questions of how goals might give rise to these functions, and how to ensure consistency between the utility benefit and the cost associated with achieving that benefit. [Etzioni, 1989] admits explicit symbolic goals, but he again separates "goal utility" from the cost of achievement, and provides no way to make the two notions compatible. These recent efforts have focused on the problem of decision-theoretic *control*—how an agent decides whether to act or to think about acting—rather than the problem of how to use decision theory to choose among alternatives. We argue in [Hanks, 1990a] that this effort is misguided.

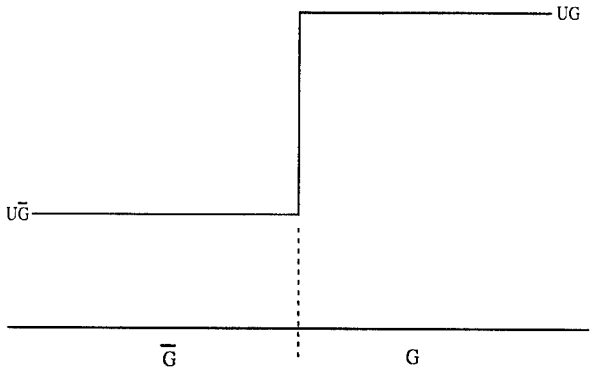


Figure 1: Step utility function.

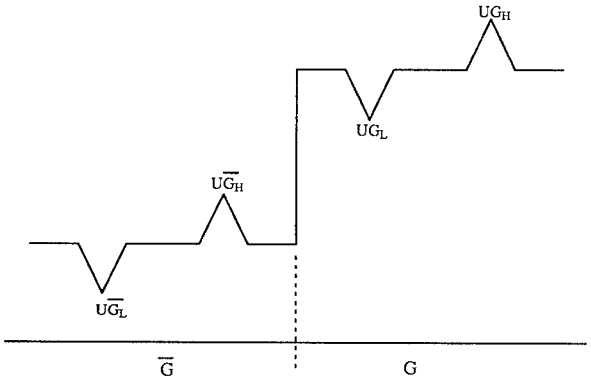


Figure 2: Approximate step utility function.

that p_1 and p_2 are the probabilities that acts A_1 and A_2 achieve the goal, respectively. Suppose further that $p_1 > p_2$ and that we have two constants, UG and $U\overline{G}$ such that $UG > U\overline{G}$. Then by algebraic identity

$$(UG - U\overline{G}) \cdot p_1 + U\overline{G} > (UG - U\overline{G}) \cdot p_2 + U\overline{G}.$$

Rearranging terms,

$$UG \cdot p_1 + U\overline{G}(1 - p_1) > UG \cdot p_2 + U\overline{G}(1 - p_2)$$

So if UG represents the utility associated with satisfying the goal and $U\overline{G}$ represents the utility associated with failing to do so, then $\mathbf{EU}(A_1) > \mathbf{EU}(A_2)$.

Next, we show that the utility function must *necessarily* take the form of a step function for condition (1) to hold. For simplicity, suppose we have only four states, s_1 s_2, s_3, and s_4, and we are choosing between two acts, A_1 and A_2. Suppose that the goal is satisfied in s_1 and in s_2, but not in the other two states. What restrictions on the utility function are necessary to guarantee condition (1)? We start by noting that

$$\mathbf{P}(G|A_1) = \mathbf{P}(s_1|A_1) + \mathbf{P}(s_2|A_1)$$

and likewise for $\mathbf{P}(G|A_2)$, and examine what restrictions on the utility functions over the s_i cause condition (1) to be true.

First, the utility function must be constant over the regions where the goal is satisfied and where it is not satisfied. Suppose we have a utility function in which this is not the case, as in the following scenario:

state	$\mathbf{P}(s\|A_1)$	$\mathbf{P}(s\|A_2)$	$\mathbf{U}(s)$
s_1	0.6	0.0	1
s_2	0.0	0.2	10
s_3	0.4	0.0	0
s_4	0.0	0.8	0
$\mathbf{P}(G\|A_1) = 0.6$		$\mathbf{P}(G\|A_2) = 0.2$	
$\mathbf{EU}(A_1) = 0.6$		$\mathbf{EU}(A_2) = 2$	

So note that $\mathbf{P}(G|A_1) > \mathbf{P}(G|A_2)$ whereas $\mathbf{EU}(A_1) < \mathbf{EU}(A_2)$, thus contradicting condition (1). So it is necessary that $\mathbf{U}(s_1) \geq \mathbf{U}(s_2)$, and the only way to guarantee this in general is if $\mathbf{U}(s_i) = \mathbf{U}(s_j)$ for all s_i and s_j such that $s_i, s_j \in G$. An identical argument shows that $\mathbf{U}(s_i) = \mathbf{U}(s_j)$ over $\overline{G}$ is a necessary condition as well.

The final necessary condition is that the utility of the world states in which the goal is satisfied must be less than the utility of those in which is is not satisfied. This condition is obvious and the proof is omitted.

In practice, only few utility functions will actually take the form of such a simple step function. Utility is likely not to be perfectly flat over any region, and it may contain continuous transitions rather than discontinuities. In the next three sections we discuss how to handle these cases.

2.1 Noise in the step function

What can be said if the utility function has the discontinuity of a step function but is not otherwise constant? Suppose that we have a discontinuous utility function such that the states that satisfy the goal all have relatively high utility and those that do not all have relatively low utility as shown in Figure 2. $U\overline{G}_L$ and $U\overline{G}_H$ are the lowest and highest utility for states that do not satisfy the goal. UG_L and UG_H are the lowest and highest utility for states that satisfy the goal. Suppose we are considering two plans A_1 and A_2 with probabilities of achieving the goal p_1 and p_2, respectively. For what values of p_1 and p_2 can we say that plan A_1 is preferred to plan A_2? Since we don't know the exact outcomes of the plans, we must do a worst-case analysis. The lowest possible expected utility for A_1 is $p_1 \cdot UG_L + (1 - p_1) \cdot U\overline{G}_L$. The highest possible expected utility for A_2 is $p_2 \cdot UG_H + (1 - p_2) \cdot U\overline{G}_H$. A_1 is guaranteed to be preferred to A_2 just in case

$$p_1 \cdot UG_L + (1 - p_1) \cdot U\overline{G}_L > p_2 \cdot UG_H + (1 - p_2) \cdot U\overline{G}_H.$$

Rearranging terms,

$$(UG_L - U\overline{G}_L)p_1 + U\overline{G}_L > (UG_H - U\overline{G}_H)p_2 + U\overline{G}_H$$

and finally

$$p_1 > \frac{(UG_H - U\overline{G}_H)p_2 + (U\overline{G}_H - U\overline{G}_L)}{UG_L - U\overline{G}_L}. \quad (2)$$

By comparing probabilities in this way we can eliminate the set of plans known to be sub-optimal and use more

refined methods (e.g. complete **EU** calculation) to identify the best plan among the candidates left. (Note the similarity to [Wellman, 1988].) Furthermore, if we substitute probability 1 for p_1 we can see that any plan with probability greater than

$$\textbf{cutoff}(G) = \frac{(UG_L - U\overline{G}_H)}{(UG_H - U\overline{G}_H)}$$

is guaranteed to be in the candidate set (that is, among the set of non-dominated plans). The reason is that no plan can have a probability of achieving the goal greater than one, so there can be no plan with probability of achieving the goal high enough to be preferred to any plan that has probability of achieving the goal at least as great as the cutoff value. In order to compare plans in terms of their probability of achieving the goal using inequality (2), we need not calculate precise point probability values. It is sufficient to establish upper and lower bounds on the probabilities. This can result in computational savings (see, for example, [Hanks, 1990c, Haddawy and Frisch, 1987]).

A simple numerical example will help to illustrate how these results can be used. Suppose that

$$U\overline{G}_L = -13 \quad U\overline{G}_H = -4 \quad UG_L = +8 \quad UG_H = +15.$$

Then inequality 2 becomes

$$p_1 > \frac{19p_2 + 9}{21}$$

If we have a plan A_1 with $p_1 = .7$ then it is preferable to any plan with probability p_2 less than

$$\frac{(.7)(21) - 9}{19} = 0.3$$

This means that once we have a lower bound on the probability of any one plan, if this bound is high enough, we can eliminate other plans from consideration based on their upper bounds. Furthermore, the **cutoff**(G) value is

$$\frac{8+4}{15+4} = 0.63,$$

so any plan with a probability of achieving the goal of at least 0.63 is guaranteed to be in the candidate set.

Any plan that is related to *all others* by inequality (2) is guaranteed to be one that maximizes utility. What if there is no such plan? We can still quantify the degree of approximation involved in choosing the plan that has the highest probability of achieving the goal. Suppose the two plans with highest probability of achieving the goal are A_1 and A_2 and that $p_1 > p_2$. But suppose that A_2 actually has a higher expected utility than A_1. To what degree does choosing A_1 approximate maximizing expected utility? (In other words, how far wrong can we go by choosing A_1?) This degree of approximation can be expressed as the percent that A_1 falls short of maximizing expected utility. In the worst case, $\mathbf{EU}(A_1) = p_1 \cdot UG_L + (1 - p_1) \cdot U\overline{G}_L$ and $\mathbf{EU}(A_2) = p_2 \cdot UG_H + (1 - p_2) \cdot UG_H$. The worst-case degree of approximation can then be defined as:

$$\frac{\mathbf{EU}(A_2) - \mathbf{EU}(A_1)}{\mathbf{EU}(A_2)}.$$

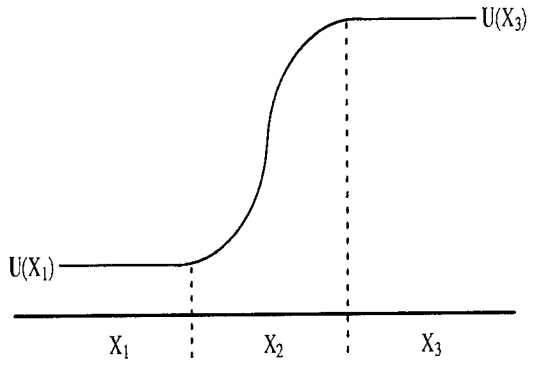


Figure 3: Continuous utility function.

In the previous example, if $p_1 = .7$ and $p_2 = .35$, the degree of approximation in choosing A_1 is

$$\frac{[.35(15) + .6(-4)] - [.7(8) + .3(-13)]}{.35(15) + .6(-4)} = .40$$

So at best A_1 is the act that maximizes expected utility and at worst its expected utility is 60% that of the best plan.

2.2 Continuous utility functions

Suppose now that the utility function has the flat regions characteristic of a step function but the transition between these regions is continuous rather than discontinuous. Figure 3 shows the graph of a univariate utility function that is continuous in the value of the attribute variable. The utility function can be qualitatively described by specifying the three distinct regions of the space of world states over which it is flat, transitionary, and flat again. If we have a symbolic description of each of these regions, plans can be described in terms of their probability of achieving an outcome that satisfies each of the descriptions. That is we can define, for a course of action A_i,

$$\mathbf{P}_i(X) \equiv \sum_{s \in X} \mathbf{P}(s|A_i),$$

where X is one of the utility regions as in Figure 3.

Suppose we are considering two plans A_1 and A_2 with probability distributions $\mathbf{P}_1$ and $\mathbf{P}_2$, respectively. For what sorts of distributions $\mathbf{P}_1$ and $\mathbf{P}_2$ can we say that A_1 is preferred to A_2?

The lowest possible expected utility for A_1 is

$$\mathbf{P}_1(X_3) \cdot \mathbf{U}(X_3) + (1 - \mathbf{P}_1(X_3)) \cdot \mathbf{U}(X_1),$$

and the highest possible expected utility for A_2 is

$$\mathbf{P}_2(X_1) \cdot \mathbf{U}(X_1) + (1 - \mathbf{P}_2(X_1)) \cdot \mathbf{U}(X_3).$$

A_1 is therefore guaranteed to be preferred to A_2 just in case

$$\mathbf{P}_1(X_3) \cdot \mathbf{U}(X_3) + (1 - \mathbf{P}_1(X_3)) \cdot \mathbf{U}(X_1) > \mathbf{P}_2(X_1) \cdot \mathbf{U}(X_1) + (1 - \mathbf{P}_2(X_1)) \cdot \mathbf{U}(X_3)$$

$$\mathbf{P}_1(X_3) \cdot (\mathbf{U}(X_3) - \mathbf{U}(X_1)) + \mathbf{U}(X_1) > \mathbf{P}_2(X_1) \cdot (\mathbf{U}(X_1) - \mathbf{U}(X_3)) + \mathbf{U}(X_3)$$

$$\mathbf{P}_1(X_3) > \frac{\mathbf{P}_2(X_1) \cdot (\mathbf{U}(X_1) - \mathbf{U}(X_3)) + (\mathbf{U}(X_3) - \mathbf{U}(X_1))}{(\mathbf{U}(X_3) - \mathbf{U}(X_1))}$$

and finally

$$\mathbf{P}_1(X_3) > 1 - \mathbf{P}_2(X_1). \quad (3)$$

Notice that the utility values no longer appear in the inequality.

The smaller the X_2 region, the easier inequality (3) is to satisfy. If $\mathbf{P}_1(X_2) = \mathbf{P}_2(X_2) = 0$ then 3 reduces to the simple step function condition

$$\mathbf{P}_1(X_3) > \mathbf{P}_2(X_3).$$

Inequality 3 holds not only for the utility function shown in the figure but for any utility function in which the utilities of states in region X_2 are between $\mathbf{U}(X_1)$ and $\mathbf{U}(X_3)$.

2.3 Continuous utility functions with noise

Now suppose we have a continuous utility function as shown in Figure 3 with noise as in Figure 2. Suppose that $U\overline{G}_L$ is the lowest utility in the X_2 region as well as the X_1 region, and that UG_H is the highest utility in the X_2 region as well as the X_3 region. Then plan A_1 is guaranteed to be preferable to plan A_2 just in case

$$\mathbf{P}_1(X_3) \cdot UG_L + (1 - \mathbf{P}_1(X_3)) \cdot U\overline{G}_L > \mathbf{P}_2(X_1) \cdot U\overline{G}_H + (1 - \mathbf{P}_2(X_1)) \cdot UG_H$$

which simplifies to

$$\mathbf{P}_1(X_3) > \frac{\mathbf{P}_2(X_1) \cdot (U\overline{G}_H - UG_H) + (UG_H - U\overline{G}_L)}{(UG_L - U\overline{G}_L)}. \quad (4)$$

The four forms of utility functions just analyzed are not the only possible forms one might consider. They are prototypical examples of how, by generalizing the notion of goal and relating it to utilities, goals can be used to characterize more general preference structures.

Under the strict definition of goal (which has been a standard for AI), a goal is a logical expression that describes two regions of the outcome space: the region in which the goal is satisfied and the one where it is not. The set of outcome states that satisfy the goal have constant high utility and the set of outcome states that falsify the goal have constant low utility. We have now generalized the concept of goal. Under our new, more general definition, a goal describes a partition of the outcome space such that within each region of the partition, all utility values fall within given bounds. The strict definition of goal is the degenerate case, in which there are only two partitions and the upper and lower bounds in each region of the partition are equal.

3 Utility Functions for Planning Applications

We have so far discussed goal satisfaction and utility in abstract terms. Now we move on to give a more concrete application of these results. We must start, however, by making precise the elements of our utility analysis.

3.1 World states and chronicles

We have been vague to this point about the interpretation of a "world state" s, to which we assign probability and utility values. States can be thought of as representing either a snapshot of the world at a point in time or alternatively as a complete description of the world over all times. The latter is typically called a "chronicle" [McDermott, 1982]. The two are formally equivalent, in that a snapshot-like state can code arbitrary information about the course of events that led to its realization. We adopt a chronicle-based approach, which we argue in [Hanks, 1990c] is preferable for practical reasons: a chronicle constitutes an explicit record of a plan's hypothetical execution, and that record may prove useful in debugging or optimizing that plan.

3.2 Temporally qualified goals

Interpreting world states as chronicles facilitates reasoning about time as well, particularly notions like deadlines by which and intervals over which propositions are to be made true. To accommodate the notion of a deadline or other temporal qualification, we will describe a goal using two components: an atemporal *condition* and a temporal *qualifier* (*e.g.* "have all the blue rocks at the depot by noon," "keep the lights out between midnight and 4AM"). The condition is a logical formula that is either true or false at each point in time in a chronicle, and the temporal qualifier describes the part of the chronicle over which the condition is to hold.

The temporal qualifier component of a goal has two general forms, the *existential* form and the *universal* form. The existential form says that there exists a point within an interval at which the condition is satisfied

$$\exists t : t_1 \leq t \leq t_2 \ \phi$$

(where ϕ is the goal condition), and the universal form says that the condition is satisfied at every point in an interval

$$\forall t : t_1 \leq t \leq t_2 \ \phi.$$

All types of temporal goals can be expressed as special cases of these two general forms. For example, a simple deadline "make ϕ true before t_2" can be expressed as

$$\exists t : now \leq t \leq t_2 \ \phi$$

and a time point goal "make ϕ true *at* t_1" can be written as

$$\exists t : t_1 \leq t \leq t_1 \ \phi.$$

Note that the earliest relevant time for t_1 is *now* while the end point t_2 could extend infinitely into the future. However, we will not consider temporal qualifiers in which $t_2 = +\infty$ since they do not provide criteria for termination of plan elaboration.

Some examples of goals are

- Have block A on block B by noon.

  $\exists t : now \leq t \leq noon$ on(A,B)

- Get all the rocks to the depot by noon.

  $\exists t : now \leq t \leq noon$ all-rocks-at-depot

- Keep my heart rate between 160 and 200 bpm from 1:00 till 1:30.

  $\forall t :$ 1:00 $\leq t \leq$ 1:30 heart-rate-in-range(160,200)

3.3 Goal-related utility

We will associate with each goal $\mathbf{g}_i$ a utility $\mathbf{U}_i(\mathsf{c})$ (where c is a chronicle). This function in turn can be separated into two components: *degree of satisfaction* and *utility of satisfaction.*

3.3.1 Degree of satisfaction

Degree of satisfaction, a real number between 0 and 1, measures the extent to which the goal (logical formula and temporal qualifier) is satisfied in the chronicle. A 1 indicates absolute success, a 0 indicates absolute failure. Utility of satisfaction then measures the utility penalty associated with this degree of success. The reason behind this split—the reason we don't assess goal utility directly—is that degree of satisfaction can generally be measured independent of any particular problem or planning situation, whereas the utility number assigned to a goal depends crucially on tradeoffs between that goal and other goals active in the current situation.

The degree of satisfaction function for a goal can further be split into two components, corresponding to the atemporal condition and the temporal qualifier. The degree of satisfaction function for the atemporal condition measures the degree to which the condition is satisfied. This function will be $\mathbf{DSA}_i(\mathsf{t},\mathsf{c})$. The subscript i refers to the i^{th} goal, which also specifies the goal's condition. The argument c is a chronicle, and the function measures the extent to which the condition is satisfied at time t in chronicle c. Some conditions, *e.g.* "the truck's headlights are on," will always be assigned a value 0 or 1, but others, *e.g.* "all the blue rocks are in the depot," might generate intermediate satisfaction values based on the percentage of rocks that are actually at the depot at time t in chronicle c.

Next we associate a degree of satisfaction with the temporal qualifier—the interval or deadline at or by which the goal is to be satisfied. We do so with a function $\mathbf{DST}_i(\mathsf{t})$, where the index i means that the function depends on the goal's temporal qualifier. Note that there is no chronicle argument, because no logical formula need be evaluated to get the temporal degree of satisfaction—this quantity depends only on the goal's time argument. The meaning of **DST** will depend on whether the temporal qualifier is existential (deadline) or universal (interval). In the former case the degree of satisfaction measures the degree to which the deadline is satisfied, so we can represent penalties for lateness and/or for earliness. For interval qualifiers the function might evaluate whether a time point is in the interval, or how far outside the interval it falls.

We can combine **DSA** and **DST** to get an overall degree of satisfaction associated with a time point, that is

$$\mathbf{DS}_i(\mathsf{t},\mathsf{c}) = f_i(\mathbf{DSA}_i(\mathsf{t},\mathsf{c}), \mathbf{DST}_i(\mathsf{t})).$$

The choice of an appropriate f_i will depend on the particular goal, in particular on the tradeoff we want to express between violating the temporal and atemporal component. We probably want to limit the function to one of the class of *conjunctive operators*, that is the set of functions that satisfy $f(x,y) \leq \min(x,y)$ which is to say that overall degree of satisfaction cannot exceed either

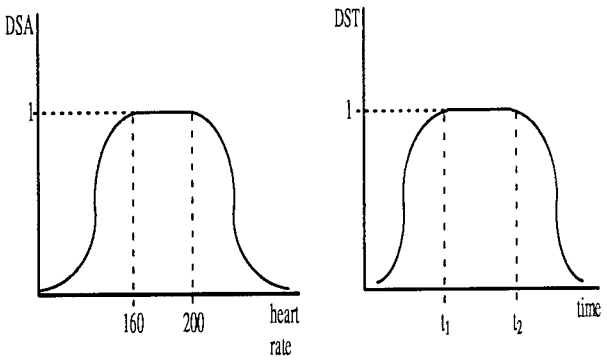


Figure 4: Degree of satisfaction functions for a universally temporally qualified goal.

of its components. Three reasonable alternatives would be

$$\begin{aligned} f(x,y) &= \min(x,y) \\ f(x,y) &= x \cdot y \\ f(x,y) &= \begin{cases} 1 & \text{if } x = 1 \text{ and } y = 1 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

The first will always return a value at least as great as the second and the second at least as great as the third. The first implies that the goal is satisfied overall to the extent of its "weakest" component, the second implies that the two factors are utility independent ([Keeney and Raiffa, 1976, Section 5.2]), and the third implies that only total satisfaction is satisfactory. See the discussion in [Dubois and Prade, 1984] for more discussion.

Recall now that we are seeking a utility function for goal i that is a function of a chronicle, whereas the degree of satisfaction function depends on a time point as well. Consider, for example, the existential goal "have all the rocks at the depot by noon," and suppose that the truck makes two trips to the depot. The first time it arrives at 11:30 and brings with it half of the rocks. The second time it arrives at 12:30 with the rest of the rocks. Should degree of satisfaction be measured with respect to the first time point, the second time point, or some other? The first will probably have a higher **DST** value than the second, because it occurs before the deadline, whereas the second has a higher **DSA** value because at that point all the rocks are at the depot. A reasonable approach in the context of comparing alternative plans is to take the maximum **DS** value, so we define, for existential goals,

$$\mathbf{U}_i(\mathsf{c}) = \mathbf{US}_i(\max_{\mathsf{t}} \mathbf{DS}_i(\mathsf{t},\mathsf{c}))$$

where $\mathbf{US}_i$ is the utility of satisfaction function, discussed below.

The situation for universal temporal qualifiers is somewhat more difficult, since the relationship between satisfaction at a point and satisfaction over the interval can be a complex one. Consider the goal "keep my heart rate

between 160 and 200 beats per minute between 1:00 and 1:30." Figure 4 shows plausible atemporal and temporal satisfaction functions, respectively. The atemporal function is straightforward: the goal is totally satisfied if the heart rate at a time t is within the range, but tends to taper off as the rate goes outside.

The temporal function now can measure how far outside the stated interval boundaries the goal condition persists. If the boundaries are strict ones (that is if there is no utility benefit associated with exercising longer), then $\mathbf{DSA}_i(\mathsf{t},\mathsf{c})$ would assign value 1 to all time points inside the goal's interval $< \mathsf{t}_1, \mathsf{t}_2 >$ and value 0 to all points outside. If, however, there is some benefit to extending the interval, the function can assign 1 to values in the intervals, then lower degrees of satisfaction to time points occurring prior to t_1 or subsequent to t_2. We can then take utility to be the maximum total satisfaction, as follows:

$$\mathbf{U}_i(\mathsf{c}) = \mathbf{US}_i(\max_{\mathsf{t}'<\mathsf{t}_1,\ \mathsf{t}_2<\mathsf{t}''} \sum_{t=\mathsf{t}'}^{\mathsf{t}''} \frac{\mathbf{DS}_i(t,\mathsf{c})}{\mathsf{t}''-\mathsf{t}'})$$

where goal i's temporal qualifier is $<\mathsf{t}_1,\mathsf{t}_2>$.

As a final example consider the goal "stir the sauce continuously between 9 and 9:15." Stopping even for an instant in that interval will ruin the sauce, but stirring outside the interval is irrelevant. We might associate with this goal a **DSA** function that assigns satisfaction 1 if "stirring" is true at time t in chronicle c, and 0 otherwise. The temporal satisfaction function would likewise assign 1 if its argument were in the interval, and 0 otherwise. Suppose now that there is one point in the interval at which stirring ceases. According to the above measure, we would sum the atemporal satisfaction function over the interval <9,9:15>, but the result would essentially be 1, since stirring was true over virtually the entire interval. In this case we want the utility function to be the product:

$$\begin{aligned}\mathbf{U}_i(\mathsf{c}) &= \mathbf{US}_i(\max_{\mathsf{t}'<\mathsf{t}_1,\mathsf{t}_2<\mathsf{t}''} \prod_{t=\mathsf{t}'}^{\mathsf{t}''} \frac{\mathbf{DS}_i(t,\mathsf{c})}{\mathsf{t}''-\mathsf{t}'}) \\ &= \mathbf{US}_i(\prod_{t=\mathsf{t}_1}^{\mathsf{t}_2} \frac{\mathbf{DS}_i(t,\mathsf{c})}{\mathsf{t}''-\mathsf{t}'}).\end{aligned}$$

3.3.2 Utility of satisfaction

The function $\mathbf{US}_i$ maps a degree of satisfaction number into a utility number, which will then be combined with utilities for other goals in computing the plan's overall utility. Why not use the degree of satisfaction value directly as utility? First, utility as a function of goals will typically be task dependent whereas degree of satisfaction is largely task independent. Second, at some point one has to consider the tradeoff between satisfying the various goals. What should a marginal improvement in g_1's satisfaction be worth in terms of a decline in g_2's degree of satisfaction? The answer to this question may be different depending on the exact circumstances, and this difference will be reflected in the way the goals are weighted relative to one another, and this weighting is accomplished by assigning them different **US** functions.

The **US** functions also provide a way to express the tradeoff between goal satisfaction and resource consumption, the latter being represented in the utility function by the "residual utility" introduced below.

3.4 The horizon problem revisited

The introduction mentioned that goals can be used to determine when to terminate plan elaboration and that this is one of the main advantages of using goals in planning. When all our goals are all-or-nothing goals, it is clear that we can stop planning when we have reached the end time of the goal furthest in the future.[3] What can be said when we are dealing with soft goals—deadlines the agent can violate and still get a utility benefit, or intervals for which achieving the condition outside the interval bounds is rewarded?

In these cases one has to establish the planning horizon for each goal dynamically. For any goal we can compute the maximum utility we could possibly realize if the horizon were extended indefinitely into the future, and this number will decline as the horizon is extended beyond the deadline. If for some A_1 and some goal g_i we can demonstrate that receiving this maximum utility award would not cause it to be preferred to some other alternative, we needn't extend the planning horizon any further. [Hanks, 1990c] uses this technique to limit probabilistic inference in plan projection.

3.5 Utility functions for multiple goals

So far we have only discussed utility for individual goals. An agent will typically be attempting to satisfy multiple goals simultaneously. We can assume that global utility is linear additive in the $\mathbf{U}_i$ functions, which is to say that

$$\mathbf{U}(\mathsf{c}) = \sum_{i=1}^{n} \mathbf{U}_i(\mathsf{c}) + \mathbf{U}_\mathsf{R}(\mathsf{c})$$

where the $\mathbf{U}_i$ functions are utility functions for the goals and the function $\mathbf{U}_\mathsf{R}$ is a "residual" utility to be defined below. While there are no explicit weighting factors attached to the $\mathbf{U}_i$ and the $\mathbf{U}_\mathsf{R}$ functions, the utility function is nonetheless a weighted average—the relative weights appear in the $\mathbf{US}_i$ functions.

We should also point out that this functional form does not imply that any of the $\mathbf{U}_i$ functions are linear additive in any *attribute* in c, a common and arguably unrealistic assumption of most decision-theoretic analysis. We are just assuming that the satisfaction of, and utility associated with goals g_i and g_j can be computed independently. In other words, they are independent in the goal hierarchy—neither is a subgoal of each other, and they are not both being performed in service of some higher-level goal. The g_i represent only the agent's top-level goals.

The function $\mathbf{U}_\mathsf{R}$ represents the "residual" utility associated with the chronicle.[4] It measures how well off the agent is apart from factors taken into account by the explicit goals. As such it measures two important and closely interrelated features of the chronicle:

[3] For a formal discussion of how the temporal relation between actions and effects/goals can be captured in a probability calculus see [Haddawy, 1990].

[4] This function is called the "salvage value" in the literature on sequential decision-making, but that term is inappropriate for our purposes.

1. the resources consumed in achieving the explicit goals, and
2. how well the agent is prepared to meet *expected future demands on those resources.*

Since the residual utility measures the ability to meet expected resource needs we can associate with it a time point: the latest time point associated with any explicit goal. Thus the goal utility functions $\mathbf{U}_i$ evaluate the chronicle up to the point that the last goal is (possibly) satisfied, and the residual utility function evaluates the chronicle beyond that point in time.

This notion of residual utility puts our formalism in sharp contrast with other approaches to decision-theoretic planning ([Horvitz *et al.*, 1989], [Boddy and Dean, 1989], [Etzioni, 1989], for example), which associate with each goal not only a *utility* function but also a *cost* function measuring its resource usage. We argue that the cost, or value, of a resource may depend crucially on the expected future demands for that resource. Whether a truck's fuel tank is full at the end of the day is irrelevant, for example, if it is filled with fuel every night at the depot.[5] Therefore associating a fixed cost with fuel will tend to make a planner favor fuel-conserving trips for no good reason. On the other hand, if a long trip through the desert is anticipated in the morning, conserving fuel may be absolutely essential—far out of proportion to its replacement cost.

We should note a couple of things about the residual utility function. First is that it re-introduces the nice property of a planning horizon. One must project up until the most distant goal horizon, but after that point in time the residual utility function summarizes future projection. The second point, of course, is that as a result the residual function may be quite hard to assess—this is undoubtedly the hardest assessment problem lurking in our formalism. The general problem of arriving at, and evaluating beyond, a planning horizon is noted in the literature on decision theory, *e.g.* [Keeney and Raiffa, 1976, Chapter 9], but no satisfying systematic approaches exist except for extremely regular problem domains. On the other hand the sort of reasoning required—expectations about recurrent or regular goals and their implications—seems wholly appropriate for AI analysis ([Wilensky, 1978], for example). So while we do not attempt to minimize the scope of the unsolved problem, we at least hope to have divided it up correctly.

4 Goal-oriented Utility and the Probability of Goal Satisfaction

In Section 2 a goal was characterized as describing a partition of the state space such that within each region of the partition, utility is within given bounds. Goals distinguish regions of relatively high utility from regions of relatively low utility. The utility of a plan can then be characterized simply in terms of its probability of achieving a particular goal. In Section 3 we described how utility functions could be defined in terms of partial goal satisfaction. We now show how the probability of goal satisfaction is related to goal-derived utility.

4.1 Probability of complete satisfaction and complete dissatisfaction

A goal determines a utility function over the set of all possible chronicles. This utility function can be described in terms of the region over which the goal is completely satisfied, the region over which it is not at all satisfied, and the intermediate region. For example, consider the existentially temporally qualified goal "get all the rocks to the depot by noon." All chronicles in which all the rocks are at the depot before noon will have the same high utility value, relative to this goal. There will typically be some time point after which satisfying an existential goal will have no utility benefit. Say in this case that it's 5:00pm. Then all chronicles in which no rocks are at the depot before 5:00pm will have the same low utility. All other chronicles will have some intermediate utility with respect to this goal. So at this point the chronicles obey the conditions set down in Section 2.2, in which the goal describes three regions of chronicle space. The region of high utility is just described by the goal itself:

$$\exists t : now \leq t \leq noon \quad \text{all-rocks-at-depot}$$

The region of low utility is described by the sentence

$$\forall t : now \leq t \leq 5{:}00 \quad \text{no-rocks-at-depot}$$

And the region of intermediate utility is described by the sentence formed by conjoining the negation of each of these.

Now consider the universal goal "keep my heart rate between 160 and 200 beats per minute from 1:00 until 1:30." All chronicles in which my heart rate is within range over all of the specified interval will have an equally high utility value. Chronicles in which my heart rate is within range only at time points far from the interval of interest, and chronicles in which the rate is always well out of the desired range will have no utility benefit with respect to this goal. All other chronicles will be assigned some intermediate utility value. So chronicle space is again divided into three regions. Suppose that the time horizon is 12:00 to 2:00 and the heart rate horizon is 80 bpm to 220 bpm. The region of high utility is again just described by the goal itself:

$$\forall t : 1{:}00 \leq t \leq 1{:}30 \quad \text{heart-rate-in-range(160,200)}$$

The region of low utility is described by the sentence

$$\forall t : 12{:}00 \leq t \leq 2{:}00 \quad \text{heart-rate-out-of-range(80,220)}$$

And the region of intermediate utility is described by the sentence formed by conjoining the negation of each of these. Figure 5 shows how the goal divides chronicle space into the three utility regions.

In general, the high utility region of chronicle space for either an existentially or universally temporally qualified goal will just be described by the goal itself. The low utility region will be described by a sentence of the form

$$\forall \mathrm{t} : \mathrm{t}' \leq t \leq \mathrm{t}'' \; \psi,$$

where $\mathrm{t}' < \mathrm{t}_1$, $\mathrm{t}'' > \mathrm{t}_2$, and $\psi \rightarrow \neg\phi$ (ϕ being the atemporal goal condition).

[5] A better example might be a rental truck whose contract specifies that the truck be returned *empty* at the end of the rental period.

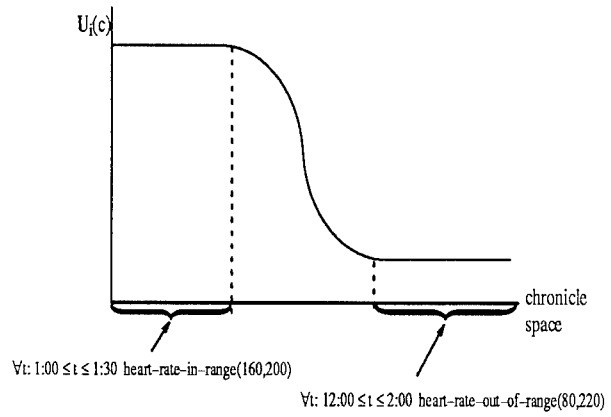


Figure 5: Utility function for a universally temporally qualified goal.

4.2 Satisfying multiple goals

Section 2 discussed the relation between maximizing the probability of a single goal and maximizing expected utility. How can these results be applied to multiple goals? Under the assumption of Section 3.5, that global utility is linear additive in the utility functions for the various goals, the situation is relatively simple. We derive results for the case in which all the utility functions for the goals are simple step functions, which will be the case if both the atemporal and temporal degree of satisfaction functions always return either 0 or 1, for example. The results are easily generalized to the more complex cases discussed in Section 2. Assume for the time being that the residual utility is zero. Incorporating non-zero residual utility will be discussed in the next section. Suppose we have two goals so that global utility is the sum of the utility for each of the goals

$$\mathbf{U}(c) = \mathbf{U}_1(c) + \mathbf{U}_2(c).$$

Suppose further that we have two plans A_1 and A_2. We show that if plan A_1 maximizes the probability of achieving each of the goals individually then it maximizes global expected utility. If the two regions of high utility are G_1 and G_2 then

$$\mathbf{P}(G_1|A_1) > \mathbf{P}(G_1|A_2)$$

$$\mathbf{P}(G_2|A_1) > \mathbf{P}(G_2|A_2)$$

implies that

$$\mathbf{EU}_1(A_1) > \mathbf{EU}_1(A_2)$$

$$\mathbf{EU}_2(A_1) > \mathbf{EU}_2(A_2)$$

and adding the two

$$\mathbf{EU}(A_1) > \mathbf{EU}(A_2).$$

Similar results hold for the noisy step, continuous, and noisy continuous forms of utility functions discussed in Section 2.

4.3 Residual utility as noise

Residual utility can be treated as noise in the utility function. Since plans must be compared according to their probability of satisfying each goal separately, the residual utility must be factored as noise into the utility function for each goal. Let $\mathbf{U}_{RH}$ be the highest residual utility value and $\mathbf{U}_{RL}$ be the lowest over all possible chronicles. Suppose we have a goal with a step utility function with constant high value UG and constant low value $\overline{UG}$. This can now be analyzed as a step function with noise where the high and low utility values are

$$UG_H = UG + \mathbf{U}_{RH}$$
$$UG_L = UG + \mathbf{U}_{RL}$$
$$\overline{UG}_H = \overline{UG} + \mathbf{U}_{RH}$$
$$\overline{UG}_L = \overline{UG} + \mathbf{U}_{RL}.$$

So inequality (2) becomes

$$p_1 > \frac{(UG - \overline{UG})p_2 + (\mathbf{U}_{RH} - \mathbf{U}_{RL})}{UG - \overline{UG}}$$

or

$$p_1 > p_2 + \frac{\mathbf{U}_{RH} - \mathbf{U}_{RL}}{UG - \overline{UG}} \quad (5)$$

So the smaller the range of possible residual utility values relative to the range of the utility of completely achieving and completely failing to achieve the goal, the more useful the goal is in comparing candidate plans. And as long as

$$\frac{\mathbf{U}_{RH} - \mathbf{U}_{RL}}{UG - \overline{UG}} < 1$$

we can get some mileage by comparing plans in terms of their probability of completely achieving the goal and of completely failing to achieve the goal. Similar results hold for continuous utility functions with noise.

5 Related Work

Our work has focused on exploring the relationships between symbolic goals and numeric utilities. We have examined the problem of building utility functions from symbolic goal descriptions, particularly when those goal descriptions make explicit mention of time constraints. We have also analyzed the relationship between the "traditional" planning problem (find a sequence of operators that will, or will probably, achieve the goals) and the corresponding decision problem (find a sequence of operators that maximizes utility).

Most of the work exploring the intersection of planning and decision theory has ignored the problem of building a utility model (taking a preference structure or utility function as given). This is the case with [Horvitz *et al.*, 1989] and with [Wellman, 1988]. Another approach, that of [Feldman and Sproull, 1975] and [Boddy and Dean, 1989], is to define the decision problem in a narrow enough domain (*e.g.* robot navigation on a grid) so that the preference measure becomes obvious (*e.g.* Euclidian distance from a goal coordinate).

A third approach maintains the explicit representation of goals in the system. The framework in [Hansson

et al., 1990] contains only symbolic goals and no notion of tradeoffs or partial satisfaction. Probabilities measure the likelihood that a partial solution will eventually lead to one that will (certainly) satisfy the goal. Their utility model is therefore the simple step function we identified, in 2, with the problem of planning to maximize the probability of goal satisfaction.

The framework in [Etzioni, 1989] also preserves the notion of symbolic goals, but under very stringent conditions of independence (that methods for achieving distinct goals will not interact) that are usually violated in prototypical blocks-world planning scenarios. He assumes a degree-of-satisfaction function, like ours, but does not explore the temporal/atemporal distinction. Goal-related utility is then the product of degree of satisfaction and a weighting factor, and global utility is the sum of the goal-related utilities. His effort is not oriented primarily toward exploring the utility model, and as such he does not discuss the issues involved with partial goal satisfaction, residual utility, and the relation between decision-theoretic and symbolic planning.

The notion of partially satisfied goals and their role in the decision-making process appears prominently in the literature on fuzzy mathematics and decision analysis. In particular our notion of a degree-of-satisfaction function bears close resemblance to a fuzzy-set membership function. The seminal paper in this area is [Bellman and Zadeh, 1980]; also see the papers in [Zimmerman *et al.*, 1984], of which the most relevant to this paper is [Dubois and Prade, 1984]. They discuss the role of *aggregation operators* in the decision-making process. In the language of fuzzy-set theory a goal may be expressed as a fuzzy set, a plan's membership function with respect to that set indicates the extent to which the plan satisfies that goal. An aggregation operator combines membership functions for individual goals into an aggregate membership function which is an indicator of global success—this is called the *decision set*. A decision maker then selects an alternative that is "strongly" a member of the decision set. Dubois and Prade categorize and analyze various aggregation functions.

So our analysis is similar to the efforts in fuzzy decision making in that it emphasizes the representation problems associated with expressing partial satisfaction of goals. Fuzzy sets may be a more appropriate representation than degree of satisfaction when the latter (a numeric function) cannot reasonably be assessed. If we can only assess vague satisfaction measures like "reasonably well satisfied," "utter failure," and "complete success," the fuzzy-set methodology provides a way to incorporate these measures into a precise analysis. As such it is essentially complementary to our analysis.

6 Conclusion

Classical planning techniques and decision-theoretic analysis can play complementary roles in decision making. The former provides a computational theory of how to generate plans, given a set of symbolic goals; the latter provides a normative theory for comparing alternative plans, given utility and probability valuations.

This paper explored the question of how to build a utility model for a domain given a set of symbolic goals of the sort used by planners. Our conclusions about the relationship between goals and utilities are as follows: classical symbolic (boolean-valued) goals are insufficiently rich to represent many aspects of a reasonable planning domain. We thus extended the notion of a goal to allow partial satisfaction. We characterized a goal as consisting of atemporal and temporal constraints and defined functions describing the value of satisfying either constraint partially. The agent's utility is then measured in terms of the extent to which it satisfied its goals as well as how efficiently it did so (as measured by the residual utility function).

Under certain circumstances, namely that the potential residual utility is small relative to the utility associated with the goals, we can demonstrate a strong correspondence between maximizing utility and maximizing the probability of satisfying goals. Planning to maximize goal probability is a special case of utility maximization.

Our representational framework attempts to capture the best features of both symbolic planning and numeric utility optimization: the (symbolic) goals, provide guidelines for the planner in its task of generating alternatives. Utility functions associated with the goals along with the residual utility function provide a principled way of comparing those alternatives.

Future work should proceed in three areas:

- Assessment. The problem of assessing utility functions, especially the goals' utility of satisfaction functions and the residual utility function, still remains. The difficult task is to generate, for each new planning problem, utility functions that accurately reflect the agent's current and expected future objectives and resource needs.
- Computation. We have provided a representation framework, but not a computational theory. It remains to be seen whether the decision-theoretic choice paradigm can be efficiently applied, though preliminary work, [Hanks, 1990c, Wellman, 1988], is encouraging, neither of these programs attempted to build the utility model at run time.
- Validation. We made several assumptions about forms of various components of the utility model, for example that temporal and atemporal degree of satisfaction were utility independent, and that utilities for separate goals were linear additive. We will need to validate these assumptions by applying the framework to complex planning problems.

References

[Bellman and Zadeh, 1980] R.E. Bellman and L.A. Zadeh. Decision-making in a fuzzy environment. *Management Science*, 17:B141–B164, 1980.

[Boddy and Dean, 1989] Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings IJCAI*. AAAI, August 1989.

[Dean and Boddy, 1988] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings AAAI*, pages 49–54, 1988.

[Dubois and Prade, 1984] Didier Dubois and Henri Prade. Criteria aggregation and ranking of alternatives in the framework of fuzzy set theory. In H.J Zimmerman, L.A. Zadeh, and B.R Gaines, editors, *Fuzzy Sets and Decision Analysis*, pages 209–240. North Holland, 1984.

[Etzioni, 1989] Oren Etzioni. Tractable decision-analytic control. Technical Report CMU–CS–89–119, School of Computer Science, Carnegie Mellon University, February 1989.

[Feldman and Sproull, 1975] J.R. Feldman and R.F. Sproull. Decision theory and artificial intelligence II: The hungry monkey. *Cognitive Science*, 1:158–192, 1975.

[Haddawy and Frisch, 1987] P. Haddawy and A.M. Frisch. Convergent deduction for probabilistic logic. In *Proceedings of the Third Workshop on Uncertainty in Artificial Intelligence*, pages 278–286, Seattle, Washington, July 1987.

[Haddawy, 1990] P. Haddawy. Time, chance, and action. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, Boston, July 1990.

[Hanks, 1990a] Steven Hanks. Controlling inference in planning systems: Who, what, when, why, and how. Technical Report 90–04–01, University of Washington, Department of Computer Science, April 1990.

[Hanks, 1990b] Steven Hanks. Practical temporal projection. In *Proceedings AAAI*, 1990.

[Hanks, 1990c] Steven Hanks. Projecting plans for uncertain worlds. Technical Report 756, Yale University, Department of Computer Science, January 1990.

[Hansson *et al.*, 1990] Othar Hansson, Andrew Mayer, and Stuart Russell. Decision-theoretic planning in bps. In *Working Notes—Planning in Uncertain, Unpredictable, or Changing Environments*, page 39, 1990. Stanford Spring Symposium Series.

[Hogarth, 1975] R.M. Hogarth. Cognitive processes and the assessment of subjective probability distributions. *Journal of the American Statistical Association*, 70:271–294, 1975.

[Horvitz *et al.*, 1989] Eric J. Horvitz, Gregory F. Cooper, and David E. Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings IJCAI*, pages 1121–1127, 1989.

[Horvitz, 1988] Eric J. Horvitz. Reasoning under varying and uncertain resource constraints. In *Proceedings AAAI*, pages 111–116, 1988.

[Keeney and Raiffa, 1976] Ralph L. Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. John Wiley & Sons, 1976.

[McDermott, 1982] Drew McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6:101–155, 1982.

[Savage, 1971] L.J. Savage. Elicitation of personal probabilities and expectations. *Journal of the American Statistical Association*, 66:783–801, 1971.

[Wellman, 1988] Michael P. Wellman. Formulation of tradeoffs in planning under uncertainty. Technical Report MIT/LCS/TR–427, MIT Laboratory for Computer Science, August 1988.

[Wilensky, 1978] Robert Wilensky. Why John maried Mary: Understanding stories involving recurring goals. *Cognitive Science*, 2:235–266, 1978.

[Zimmerman *et al.*, 1984] H.J Zimmerman, L.A. Zadeh, and B.R Gaines, editors. *Fuzzy Sets and Decision Analysis*. North Holland, 1984. TIMS Studies in the Management Sciences, Volume 20.

Issues and Architectures for Planning and Execution

Steve Hanks
Dept. of Comp. Sci. & Engr. FR-35
University of Washington
Seattle WA 98195
hanks@cs.washington.edu

R. James Firby
Jet Propulsion Laboratory, MS 301-440
4800 Oak Grove Drive
Pasadena CA 91109
firby@robotics.jpl.nasa.gov

Abstract

Planning in realistic domains forces us to confront two main issues: uncertainty and urgency.

Uncertainty arises because the planner is neither omnipotent, omniscient, nor alone in the world. As such it will typically lack perfect information about current and future states of the world or about the exact effects of various events (including its own actions).

Urgency is a more practical matter: planning takes time, and passing time can lead to foregone opportunities or worse. These two factors combine to produce a situation in which the agent cannot plan completely in advance: uncertainty will prevent it from being able to infer a best course of action, and urgency will ensure that even if it did so the opportunity to pursue such a plan will be lost.

Both these points argue for some sort of run-time decision making, or "reactive planning" as it has come to be called. Reactive planners typically make decisions at run time based on a limited amount of information (*e.g.* only that which is currently available from sensors) and on the basis of a minimal amount of inference.

Neither uncertainty nor urgency obviate the need for more deliberative decision making, however. Planning under uncertainty carries with it its own set of issues, having to do with representing uncertainty in the domain, efficiently generating planning options, and taking time pressure into account in the process.

Realistic planning therefore generates new problems in three areas: in execution, in deliberation, and in coordinating the two processes. This paper will discuss these issues in the context of a system we have are continuing to develop.

1 Introduction: Acting *versus* Deliberating

An agent situated in a complex and dynamic world is continually faced with the difficult task of trying to figure out what to do next. It is difficult even to formulate the problem precisely, in that the consequences, benefits, and costs of a course of action may be difficult to ascertain. Preferences reflecting the agent's goals, desires, and needs, are even harder to fathom, and tend to change over time.

The representation problems—modeling a complex and dynamic world, and capturing an agent's beliefs, goals, needs, and desires concerning that world—touch on many unresolved issues in the field today. The control problem—how to manage this information in such a way that the agent acts effectively and efficiently—is equally troublesome.

The control problem consists of balancing two reasonable approaches to operating in the world: the first is to make as many decisions as possible as far ahead of time as possible, the second is to defer making decisions as long as possible, and thus to act at the last possible moment. The "look before you leap" and "cross that bridge when you get to it," which highlight the distinction between *deliberating* and *acting*.

Arguing for the former position is the fact that one tends to have more options the further ahead one thinks, thus forethought can tend to improve one's lot. Furthermore, commiting to act in a particular way tends to improve one's state of information about future states of the world, leading to more informed choices and better decisions.

On the other hand, one's information about the immediate future is typically much better than information about the distant future, arguing that better decisions come as a result of waiting as long as possible. Furthermore, whenever one builds a plan one makes assumptions about future states of the world, and significant changes to the world may well occur between the time a decision is made and the time the resulting plan is to be executed, rendering the plan ineffective and the planning effort wasted. Finally, detailed prediction and prior commitment to detailed plans of action may simply be beyond the cognitive capabilities of the agent (and might generate only a minimal benefit anyway).

Clearly neither policy, think ahead or act at the last moment, should be carried out to the exclusion of the other, and clearly one difficult problem facing the agent is whether it should try to generate and commit to some plan of action to achieve a goal, or whether it should postpone commitment, perhaps until new information

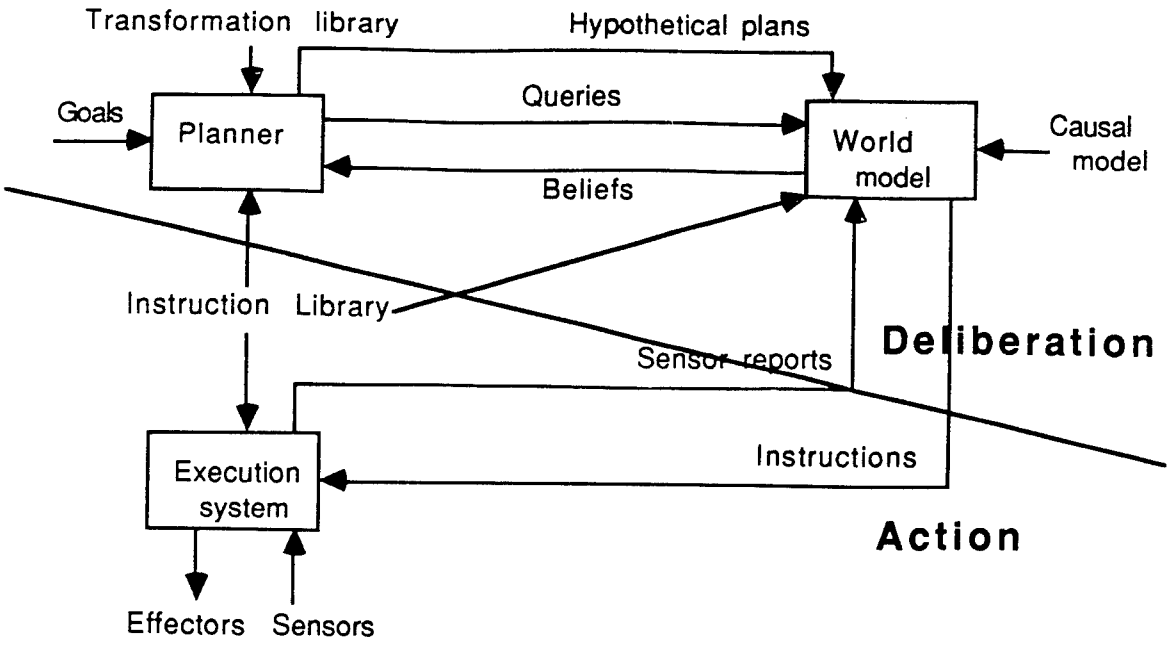


Figure 1: An architecture combining deliberation and action

3. Watch the environment for urgent situations that may affect the success of the current plan and deal with them appropriately.

The rest of this section elaborates on these responsibilities and discusses a way they can be implemented using RAPs as described in [Firby, 1989].

2.1 Generating Actions for a Plan Step

There are two major problems involved in generating actions for an abstract plan step at execution time: it must be done without projection or comparison of alternative action sequences, and different sequences of action may be needed to carry out the same step in different situations. For example, a plan might contain a step like "go to the store." Clearly such a step must be broken down into many simpler actions before it can be carried out in the world and, just as clearly, those actions depend on whether the store is across town or across the street. Furthermore, if the selected actions don't actually place the agent at the store it does not make sense to continue with the next plan step.

We use a plan expansion library to deal with the problem of choosing appropriate actions under stringent time constraints. The library consists of a hierarchical collection of predefined methods for each abstract plan step that the execution system might be called on to perform. Use of a library makes the generation of actions for a plan step quick and easy since an appropriate method (*i.e.* set of actions) can simply be looked up when needed. In essence, the plan library represents the agent's knowledge of how to get things done and each entry serves both as an abstract planning operator, or "primitive", out of which plans can be assembled, and as a record of actions that can be taken to realize that primitive at run time.

The Reactive Action Package, or RAP, is the basic plan library representation unit in our system. A RAP is a declarative structure that links a goal (*i.e.* plan step), a test for confirming the achievement of that goal, a collection of methods to use to achieve the goal in different situations, and a test of applicability for for each method to define the situations in which it might be used. Both the goal completion and method applicability tests take the form of queries to the current world model.

The execution system carries out plan steps using the following algorithm. First, a plan step is selected for execution (see Section 2.3) and if it represents an atomic action it is executed directly, otherwise its corresponding RAP is looked up in the library. Next, the step's check for completion is used as a query to the world model and, if satisfied, the step is considered complete and the next step can be run. However, if the step has not yet been satisfied, its method-applicability tests are checked against the world model and one of the methods with satisfied tests is selected nondeterministically. Finally, the substeps of the chosen method are incorporated into the current plan in place of the step being executed, and that step is suspended until the chosen method is complete. When all substeps in the method have been executed, the step is reactivated and its completion test is checked again. If all went well the step's applicability condition will now be satisfied and execution can proceed to the next step in the plan. If not, method selection is carried out again and another method is attempted.

An extremely important point is that the tests for confirming the achievement of a goal and selecting between alternative methods make reference to the current state of the world model, which means that all sensing operations required to acquire or update the information referenced by these queries must be included in the methods for achieving the goal. There is no other way to ensure that the right data appears in the world model at the right time. This point seems obvious when one realizes that the choice of which action to take must occur at run time, the choice depends critically on the state of knowledge at that time, and the sensing operations to

comes to light.

In the AI literature this dichotomy has surfaced as the question of whether an agent, at any moment, should *plan* or should *act*. The former involves mental activity only, *e.g.* generating new plans or refining one's predictions about the future. The latter involves effecting changes to the external world (or perhaps sensing the world) but the nature of those operations depends only on information immediately at hand.

Two arguments are often advanced in favor of an agent that primarily or even exclusively acts (or perhaps "reacts" is a better term): that the *urgent* nature of the world means that the penalty for inactivity is high. The image often suggested is that of a robot agent getting run over by a truck while trying to decide how to get to the other side of a road. The second argument is that the *uncertain* nature of the world means that the agent will typically lack information crucial to the decision-making process until the time comes to act. Here the image is of a robot unable to plan to get a drink of water (oil?) because it is unable to infer the exact position of the drinking glass ahead of time.

Of course advocates of the deliberative approach suggest equally compelling images: robots painting themselves into corners, running out of gas, or failing to anticipate unfortunate chemical reactions, all of which could have been avoided with a little forethought.

The point is that (1) intelligent agency requires both capabilities, and (2) as a practical matter different issues tend to arise in the attempt to implement the two modes of behavior. For this reason we propose an architecture based on the dichotomy between action and deliberation,[1] a diagram of which appears in Figure 1.

This paper attempts to set out the issues associated with implementing action- and deliberation-oriented systems. It also describe the interaction between systems of the two sorts, since an additional point follows from the discussion above: (3) that integrating representations for, and coordinating the behaviors of, the two modes of behavior is a crucial problem in designing an intelligent agent.

The remainder of this paper devotes itself to representation and control issues in action and execution, in planning and deliberation, and in coordinating the two.

2 Issues in Execution

The execution system is charged with the responsibility of taking action to carry out a plan. Traditionally, the notion of a plan has been a sequence of *atomic* actions that can be implemented directly in hardware and includes no conditionals, loops, or sensing. When a plan is that detailed the execution system can simply execute each planned action in sequence. In a realistic world, however, the deliberation system is faced with uncertainties and time constraints that make construction of a complete sequence of atomic actions impossible. Instead, a plan will always contain steps at widely different levels of abstraction, and the execution system must generate atomic actions on its own for both the most specific and the most general of those steps.

When plan steps are too vague to execute directly, they must be treated as *goals* to achieve rather than simply as actions to execute. This shift in the semantics of a plan step frees the deliberation system to reason abstractly and to use the execution system to take care of any detailed environmental interactions that are overlooked. It also frees the execution system to adapt to the specific circumstances it encounters at run time by choosing atomic actions as needed rather than in advance. The deliberation system can simply plan to "take a glass out of the cupboard" and the execution system can figure out how to move the arm to achieve that goal *after* the cupboard has been openned and a suitable glass identified. However, by making the execution system responsible for choosing actions at run time, we run the risk of re-introducing the entire planning problem under the guise of execution.

Ensuring a timely response to urgent situations requires that any projection done by the execution system must be tightly controlled. The only way to guarantee such control is to be able to curtail the projection process at any time and make further decisions using only the information gathered to that point. Since urgency may force arbitrarily little projection, we have chosen to simplify the projection control problem by not allowing the execution system to do any projection at all. The execution system has access to the current state of the world model, current active sensor values, and whatever expected future states of the world have already been derived by the deliberation system, but it does not generate and compare alternative courses of action.

Incomplete projection and inherent uncertainty in the world model force the execution system to confront the fact that it cannot know everything that might affect its choice of actions for a plan step. Therefore, action choices must be predicated on partial information and inappropriate actions will sometimes be chosen by mistake. Similarly, the world will sometimes change in unpredictable or dangerous ways that prevent some part of a plan from achieving its desired result. To remain robust in such situations, the execution system must be able to check that the actions executed in service of a plan step actually achieve their intended effects and, if they do not, the system must either adapt and try again or else admit failure. Furthermore, the system must watch for and deal with urgent situations not anticipated in the plan.

Our execution system therefore has the following responsibilities:

1. Generate atomic actions to achieve an abstract plan step using no projection and an imperfect world model.
2. Confirm the results of those actions to ensure that the intent of the plan step is actually achieved.

[1] We use the term "deliberation" rather than "planning" because the latter suggests the "classical planning" paradigm of generating provably-correct plans in full detail, which is not at all what we are advocating here. As a first cut, deliberation refers to a class of operations carried out with respect to the agent's model of the world whereas action refers to operations carried out with respect to the external world.

acquire that knowledge are actions too. If all actions are to be treated the same way (and they should be), sensing operations must occur in the same plans as the actions that require the information they yield.

2.2 Dealing with Plan Step Failures

Choosing a method for a plan step given a RAP from the plan library is a straightforward task. Sometimes, however, a RAP will contain no applicable method, or an atomic action will fail because its preconditions are not met in the external world even though they are true in the world model. In these situations the action or plan step involved fails because it cannot achieve its intended effect. When a failure occurs, the execution system reconsiders the plan step that chose the method with the failed action. Often the world will have changed since that method was selected, and in that case one of the other methods for the step can be used instead. Even if the situation hasn't changed, it may be appropriate simply to try the same method again. For example, a grasp operation may fail because the arm was bumped at a critical moment. Nothing in the world will have changed when the grasp operation is reconsidered, and it just needs to be tried again.

The execution system retries a plan step, whether an explicit failure has occured or not, until its completion test is satisfied or none of its methods are applicable in the current situation. The tenacity of this implicit looping behavior gives the execution system a great deal of robustness in the face of uncertainty and incorrect plan choice. A method can be tried, and if it doesn't work, it can be retried or another can be attempted in its place. There is no substitute for this ability to try methods over again when an agent's world model is incomplete and uncertain, because mistakes are inevitable and must be dealt with routinely. No execution system faced with real sensors and effectors can expect all actions to work properly the first time they are tried.

Unfortunately, while repeatedly retrying to achieve a plan step makes the execution system robust, it raises another problem: when to stop. If grasping the glass continues to fail because the arm keeps getting bumped, it is fruitless to keep trying the same operation over and over again; some factor must be involved that is not being taken into account by the current method. This problem is addressed simply in our execution system. A futile loop is detected whenever a plan step tries the same method twice in exactly the same situation (as represented in the world model) without success. That plan step then fails in exactly the same way as if it had no applicable method, and its enclosing plan step is given a chance to select another method. This is not a completely satisfactory solution, however, and the problem of dealing with futile action/environment loops remains an important problem for all intelligent agent architectures to address.

2.3 Choosing What to Execute

A plan consists of a partially ordered network of goals at varying levels of detail and choosing which plan step to refine and execute next is a difficult scheduling problem. In general, there may be many different goals that can be worked on next, either because the deliberation system has not yet committed to a specific ordering, or because the plan contains several steps that can be carried out equally well in any order (or even simultaneously).

The best way for the execution system to choose among possible next plan steps without using any lookahead is still an open research issue, but a simple algorithm is described in [Firby, 1989]: the deliberation system assigns each plan step a priority based on its importance or expected utility. The execution system then looks at all possible next plan steps and chooses the one with the highest priority, breaking ties in favor of nearer deadlines.[2] The selected step is either executed directly or one of its methods is chosen and incorporated into the current plan at a priority modified by notations in the method. The plan is now examined again and another step is selected. If nothing untoward happens, this algorithm has the effect of incorporating actions for a step into the plan and then selecting and executing those actions one by one.[3] However, the system can shift its attention to a high-priority step should one arise.

2.4 Monitoring Urgent Situations

The ability of the execution system to shift attention to higher priority plan steps can be used to recognize and deal with urgent problems and opportunities. Plan steps can be augmented with a test for activity and remain dormant until that test is satisfied in the world model. Using this device, RAP methods can be written to include high priority substeps to deal with expected difficulties, or recognize possible opportunities. By gating those substeps with appropriate activity tests, they become active at a high priority and interrupt the current course of action precisely when they are necessary. The only *caveat* is that a plan must explicitly include steps to handle all problems or opportunities that might arise during its execution. This restriction can be mitigated somewhat by including default steps in parallel with all plans to deal with common everyday problems like running low on gas, or meeting enemy troops. Strategies for using this method to monitor and protect situations in the world are discussed at length in [Firby, 1989].

The ability of the execution system to cope with abstract goals can also be exploited to allow quick reaction to things like loud noises or flashes of light. Assuming that the sensing system is designed to detect such basic and threatening events asynchronously, we can automatically generate a new goal at a high priority to deal with an event when it occurs. This goal can then be

[2] This is actually a simplification. The execution system also tries to focus its attention on one goal at a time by prefering to execute all of the actions in one method before switching to actions in another method. See [Firby, 1989] for more details.

[3] The execution method of expanding a plan step into more and more detail until atomic actions are reached bears a resemblance to many "classical" planning techniques. One difference is that the expansion is depth first based on the current situation and backtracking results from trying and failing in the real world.

deliberated upon if there is time, passed directly to the execution system, or incorporated into the current plan and effectively passed to both systems simultaneously. When it reaches the execution system it is treated like any other plan step and if it has a high priority it will interrupt whatever is being done. A single uniform notion of plan steps as goals is therefore used both for ordinary and exceptional situations.

3 Issues in Deliberation

We noted above that deliberation is the process of reasoning within the confines of the agent's world model. In Figure 1 the deliberation subsystem consists of the planner along with the *model manager*—the module responsible for maintaining the agent's model of the world. Deliberative tasks are therefore three:

1. to propose, repair, refine, or otherwise improve the agent's plans (future commitments to act)
2. to make predictions about future states of the world based on what is currently known
3. to react to new information and assess how it affects the agent's beliefs, plans, and so on.

The three are tightly interconnected: deciding on a course of action requires predicting future states, and future states of the world in turn depend on what the agent will do. We can, however, study the process of prediction without regard to *how* current commitments were arrived at, we can study the process of generating and refining plans without regard to *how* the necessary predictions are made, and we can study the process of monitoring and belief revision without regard to *what* will be done in light of significant changes to the model. We therefore discuss, in turn, the issues associated with in the world, and generating commitments to act.

3.1 Probabilistic temporal projection

The general problem of temporal projection is how to predict whether some proposition φ will be true at some time point t, using evidence occurring temporally prior to t. Three sorts of evidence get used in the computation:

1. reports from the sensors
2. symbolic causal rules representing the agent's model of how things change in the world (both in response to the agent's own actions and to other events)
3. background information about the "usual" states of proposition, the occurrence of events, and so on.

We will take the causal rules to be statements of the form "if event E occurs while some fact P is true, then Q will become true at the next instant in time." (See, *e.g.*, [McDermott, 1982].)

Uncertainty can come from a number of sources:

1. one can doubt whether the sensor always correctly reports on φ's state
2. one can be unsure as to whether a relevant event E actually occurred at some point in time
3. one can lack confidence in the rules: perhaps the rules mentioning φ aren't *really* necessary and sufficient predictors of φ's state changes.

We have adopted a probabilistic approach to the problem and thus compute the quantity $P(\varphi_t)$ with background information taking the form of prior probabilities. We take into account all forms of uncertainty listed above: faulty sensors, unpredicted events, and incomplete or incorrect causal models.

Important computational problems arise in implementing this approach, in that a tremendous amount of evidence must be brought to bear in computing the probability. Sensory observations of φ can extend arbitrarily far back into the past, as can the relevant causal rules (since they are implicitly quantified over all time points). Most of this evidence, however, will not affect the φ's probability significantly.

Evidence loses its impact—its power to change a probability estimate—for two reasons:

1. To the extent that the evidence is "unreliable" (a faulty sensor report or a causal rule whose precondition does not hold), it should not affect the probability.
2. The more temporally distant a piece of evidence is (the longer it occurred before t), the more likely it is that some *other* event occurred in the meantime, changing φ's state and rendering the earlier evidence irrelevant.

In [Hanks, 1990c] we make precise the notions of reliability, temporal distance, and impact.

Although computing φ's exact probability requires that we consider a potentially infinite amount of information, we might expect that under the right circumstances—sensors that are reasonably reliable and changes that occur reasonably infrequently—we can compute a good approximation of the probability using only a few pieces of evidence. The question is how good need a "good" approximation be?

This information is provided by the application in the form of a *probability threshold* τ. The threshold may be generated as part of the planning process—we may decide, for example, that plan P_1 is preferable to plan P_2 if the probability of some φ exceeds some value τ. For example, I may plan to drive to work instead of riding my bicycle if the chance of rain is greater than 60%. In that case we don't care about an exact answer to the question "what is the probability of rain," but only to what side of the threshold (0.6) the exact answer lies. A "good" approximation is one that reports correctly with respect to the threshold. We present in [Hanks, 1990c] a heuristic algorithm for limiting the search for evidence, the limit being computed on the basis of how close the current estimate is to the threshold.

Probabilistic assessment is triggered by a "probabilistic query," which is a question of the form "to what side of threshold τ does $P(\varphi_t)$ fall?" Information about probabilistic assessment is returned in a data structure called a *belief*, which says something like "the probability of proposition φ at time t is {above,below} threshold τ," and also apprises the application of its current estimate for $P(\varphi_t)$. Beliefs also represent a commitment by the model manager to notify the application if new information changes the system's estimate with respect to the threshold (see Section 3.4 below).

3.2 Plan projection

Plan projection is the process of answering the question "given what I now know about the world, if I were to execute plan P at some future time t, what might I expect to happen?" or more particularly, "will the intended effects of my plan actually be realized?"

In some sense the probabilistic temporal reasoning algorithm provides a method for plan projection: a proposed plan is a sequence of events, an action is an event that can trigger causal rules, and the projection question becomes a probabilistic query, just as above.

The method is inadequate, however, for reasons both formal and practical. First of all, the probability calculus involves several assumptions reasonable for infrequent, unplanned events, but unreasonable for a series of events comprising a plan.[4] Furthermore the representation of actions as causal rules (and thus having effects restricted to binary-valued random variables) is too restrictive: it does not allow for easy representation of parameters like a truck's fuel consumption, contents of its cargo bay, and so on.

As a practical matter, an action (event) in a plan typically has many effects. Moving the truck takes time, consumes fuel, wears down the tires and battery, causes its cargo to move, may cause the truck to become dirty, and so on. One would have to represent each such effect as a separate rule, and the program would have to repeatedly compute the rules' preconditions.

More serious, however, is the fact that the information returned by a probabilistic query is sparse at best. Suppose a planner put together a course of action, posted it to the world model, then posed a query about the likelihood of success. Suppose the probability of success was low, what then does the planner do? The problem is that the *belief* returned in response to a temporal query does not allow the planner to diagnose the likely failure and repair the plan. The planner needs to know both *that* and *why* plan failure may occur.

To that end we introduce the notion that an action is a mapping from *state descriptions* into *outcome sets*. A state description is a formula, but we require that the descriptions comprising an action's domain partition the set of possible worlds. Outcome sets are likewise sets of formulas, but the set of legal formulas has been extended to represent things like real- and set-valued propositions. We are thus in the position where given a world state we know exactly what effects the action will have, but we may not know which world state will hold at execution time.[5]

Our view shifts somewhat: queries cause us to compute the probability that a set of propositions will be true in the (single) future world, whereas projection assigns a probability distribution over *sets* of future worlds, and within each a proposition is either certainly true or certainly false. The probability of a goal formula being true is the probability that the real world will end up being one of those future worlds in which the goal is (certainly) true. The result of a projection is a *scenario tree*: a tree of possible worlds representing alternative outcomes for the plan. Each path through the tree (called a "chronicle") indicates one way the plan's execution may proceed; we can associate a probability with each chronicle, thus supply the probability that the plan will succeed. The scenario tree branches every time an action's effects depend on a prevailing state of the world, and that state cannot be determined with certainty (at plan or projection time).

As a practical matter we cannot generate the entire scenario tree, unless, of course, we have perfect knowledge about what the world will be like at execution time. We therefore need to keep some of the paths implicit. Keeping most of the tree implicit saves space and time, but in doing so we lose the ability to make precise characterizations about the plan's effects. Suppose, for example, that driving the truck over a muddy road may cause it to become dirty, and that we are projecting a plan that involves a trip over road R. We may or may not want to represent the two alternatives "truck arrives clean" and "truck arrives dirty" explicitly. If we do so we double the number of chronicles in the scenario tree, but if we don't we can only infer the disjunction "truck arrives either clean or dirty."

Balancing the need for parsimony against the need to articulate important distinctions in a plan's outcome is the problem confronted by the projector. The projector proceeds under the assumption that *no* distinctions are important unless it is provided with information to the contrary. This policy tends to result in a "fully implicit" tree. But at some point during the projection or subsequent planning process, "questions" get raised about the world. These questions can come directly from the planner ("what is the probability that the plan will succeed") or they can arise in the projection process itself when the projector has to compute the probability that a particular chronicle will be realized. These questions, which are exactly the probabilistic queries we discussed in the previous section, tell the projector what aspects of the world are important, and thus what parts of the scenario tree should be made explicit. The projector examines the scenario tree and tries to make explicit exactly those portions of the tree that will allow an unambiguous answer to the query at hand. [Hanks, 1990b] is a summary of the action and scenario-tree representation and the projection algorithm.

3.3 Integrated projection and the world model

Note from the discussion above that the process of probabilistic temporal reasoning and projection are intimately connected: queries initiate projection (scenario-tree articulation), which gives rise to more queries, and so on. We can thus view the planner's world model as a network of *belief* data structures, each of which is based

[4] Basically the assumption is that events will occur infrequently relative to the length of time that preconditions to their rules remain true. See [Hanks, 1990c, Chapter 3].

[5] This is an oversimplification: our action representation allows for nondeterministic effects by allowing an action's state-description formula to contain formulas of the form "chance p" which are always taken to hold with probability p. Therefore the probability that this state description will be realized, and thus what effects the action may have, cannot be known ahead of time.

on evidence consisting of sensory observations, causal rules, prior probabilities, and hypothetical commitments to act. Generating these beliefs—answering queries—involves both projection and probabilistic assessment, but the planner is never aware of this distinction.

3.4 Monitoring and revising the world model

The deliberation system is responsible for maintaining the integrity of the world model in light of new evidence. New evidence presents itself in one of two ways: observations are received from the sensors and posted to the world model, and new planning commitments are received from the planner and likewise posted.

In either case the projection/assessment algorithm has assumed that its current (at the time of assessment) set of observations and plan commitments are the *only* such events, and it has furthermore posted *monitors* to the database to look for interesting new events, each monitor being associated with a *belief*. A monitor looks for particular patterns over particular intervals of time (the form of both depend on the individual assessment). When new information triggers the monitor it notifies its belief, which incorporates the new evidence into its assessment. If the new information causes its information to change with respect to the belief's threshold, it goes on to notify everything that depends on the belief (which might be other beliefs, or might be application-supplied functions that would cause a plan commitment to be questioned).

3.5 Generating planning options

So far the world model has accepted potential plans as given; eventually we have to confront the problem of how they are generated in the first place. We have just begun work on this topic, so our ideas are still tentative, but it may be worthwhile to point out how the problem appears within our framework and the direction the work is taking.

The approach we are experimenting with involves the synthesis of *decision-theoretic choice* with *transformational planning*. The sequence goes something like this:

1. The agent is given a set of (symbolic) goals, which suggest initial candidate plan alternatives. These alternatives will tend to be quite vague at first.
2. Through the projection process the agent tries to establish, on decision-theoretic grounds,[6] that one of these alternatives is preferable to the others. This attempt will almost certainly fail at first because the vague nature of the alternatives will not allow precise predictions about the future.
3. The result of this attempt—the *scenario* structures returned by the projections—will, however, point out significant gaps in the agent's state of information about the alternatives, and may also point out where an alternative is likely to fail. This information will suggest plan transformations, which will take the form either of changes to an alternative, or perhaps an indication of where the alternative needs to be further elaborated.
4. Returning to Step 2, the agent once again tries to establish a dominating alternative.
5. The process continues either until one alternative dominates the others, or until no further transformations apply, or until time pressure forces the planning process to terminate (see below). Since the agent will always have *estimates* of the expected utilities associated with its current alternatives, it can at any point choose the alternative with the highest estimate though it can't be sure that subsequent analysis might reveal that it made a poor choice.

The existing interface provided by the world-model manager is well suited to this sort of analysis: probabilisitic queries are just the sort of information the planner will need to do the decision-theoretic analysis. Many questions remain, however. The first is how to pose the problem in the language of decision theory. The formulation requires both a *probability model* and a *utility model* of the domain. Much of the work on probabilistic projection is oriented toward providing the probability model; the utility model is a topic of current research [Haddawy and Hanks, 1990]. Next the nature of the plan transformations needs to be explored. Work like [Linden and Owre, 1987] and [Simmons, 1988] explores the general technique of iterated transformation and projection; it remains to be seen how these techniques can be integrated into our model of probabilistic projection and decision-theoretic choice.

4 Issues in Coordination

Coordinating the subsystem that interacts with the world and the subsystem that interacts with the world model suggests a new set of issues, again involving both representation and control. The representation issues center around the question of how the world model should represent the agent's behavior, *i.e.* the performance of the execution system, and how information gathered by the execution system should be incorporated into the deliberation system's world model.

4.1 Representation

We can identify three main requirements for a representation that will support both deliberation and execution. The first is that since predicting the future requires a model of the agent's behavior, whatever representation is used by the execution system to guide its behavior must also serve as model of that behavior for the deliberation system. Similarly, since *influencing* the future requires that the deliberation system affect the behavior of the execution system, the deliberation system must be able to communicate to the execution system, using that same representation, suggestions for future actions. Finally, since the deliberation system relies on the execution system to collect information about the world, their models of sensors and sensing activities must be compatible.

[6] Essentially this just means that the agent will make its decision on the basis of tradeoffs between probability of success, reward for success, and penalty for failure.

Recall from Figure 1 that the deliberation and execution system share the RAP library—it represents instructions executable by the execution system, the planner's "plan library," and the projector's "model of action." Two features of the RAP library system that make it amenable for all three uses are its *hierarchy* and its *annotations*. The hierarchy was discussed briefly in Section 2: a RAP has associated with it a proposition (condition it intends to achieve) along with a set of *methods* any of which might effect the desired state under the proper circumstances.

The problem of biasing the execution system arises when the planner/projector arrives at a good course of action and needs to ensure that the execution system actually carries it out. The problems here are more practical than theoretical. The ways a planner would change the behavior of the RAP system are two: by choosing in advance which method to select in executing a particular RAP, and by specifying an order in which RAP system's execution agenda, and the only problem is how to relate the representation of a RAP in the planner/executor's world model to the actual instantiation of that RAP in the agenda.

RAP annotations were not discussed in Section 2, but play an important role in projecting the effects of the execution system's actions. Each RAP is annotated with the effects it will have if executed—conditions it will cause to be true in the world, resources it will consume, and so on. These annotations are exactly what we meant in Section 3.2 by *outcome sets*. Of course not all of a RAP's effects (outcomes) will be relevant under every set of circumstances, but the task of the projection algorithm is to separate the important outcomes and interactions from the irrelevant. The correspondence between RAP annotation and the projector's action model is therefore very tight.

The final representation issue involves the sensor model. The execution system is in charge of the agent's sensors, and the only way the world model gets information about the world is through sensor reports. The projector's model of a sensor report is that of an *observation*, which consists of a proposition, a time point, and a number indicating the agent's estimate of the sensor's reliability. Furthermore, the world-model manager generates *monitors* in the course of forming beliefs, that are charged with the responsibility of looking for relevant observations that might subsequently be added to the world model, and notifying the appropriate beliefs. The RAP system also has a concept of monitors, which can be used to generate sensor operations that check periodically whether a particular condition becomes true in the world [Firby, 1989, Section 4.7]. There is thus a tight connection between monitors in the execution sense and monitors in the deliberation sense: deliberation monitors give rise to execution monitors, and thus to sensing operations. Deliberation monitors further give the execution system an indication of what sensor reports are currently relevant to the agent's world model: only sensor reports for which deliberation monitors are currently active need be passed from the execution system to the deliberation system.

4.2 Control

We mentioned the fundamental control issue at the beginning of this paper: how does an agent decide whether to act on the basis of its current state of information or instead to do further deliberation, information gathering, or both before committing to a course of action.

This decision, called the problem of "decision-theoretic control," has received a lot of attention in the literature lately, for example in [Boddy and Dean, 1989], [Etzioni, 1989], [Horvitz *et al.*, 1989]. The problem is posed as to whether the agent should spend the next unit of time in "physical activity" or in "mental activity," that is in acting or in deliberating. The decision involves balancing the possible benefit of discovering a better course of action during that next time unit against the possible cost of opportunities foregone by not acting. We argue in [Hanks, 1990a] that it may well be impossible to make this decision in a principled way, since doing so requires (1) that we have a good characterization of the improvement in the plan we expect to realize from a unit time spent in deliberation, (2) that we have a good characterization of the opportunity cost associated with delaying action by one time unit, and (3) that the decision about whether to plan or to act must be made in a negligible amount of time.

Our position has been that in many cases the decision on whether to act, plan, or gather more information will be clear: the execution system can be charged with the responsibility of reacting to emergencies like fires, attacking animals, or oncoming cars, and react to them without recourse to the deliberation system at all. Indeed, to get the response time necessary to avoid catastrophe the agent *cannot* refer to any higher-level processes. By the time the decision-theoretic control systems decided that it was crucial to act immediately it would already be too late.

But urgency is not the only reason one wants to stop working on a plan. The other reason, and one ignored by the "reactive planners," is that the planner may lack information necessary to choose one alternative over another. This situation will be noticed during the projection process in one of two ways. The first arises when the choice between two alternatives depends on whether a particular fact will be true or false, but the probability estimate for that fact is equivocal. In that case the planner can delay the choice between those alternatives and schedule an operation to gather the required information if it knows how to do so. By posting a monitor that is looking for that new information the planner can ensure that it will automatically reconsider the choice once new information is received.

The other way deliberation can be suspended is if the projection process gets "bogged down." The projector has associated with each projection a maximum number of branches allowed for a scenario tree. Expansion of that tree is suspended when that number is exceeded, a condition indicating that the agent does not have enough information about the future state of the world to make good predictions about the effects of its actions. Once again, projection can be suspended and resumed when new information refines probabilities, causing branches

to be discarded as improbable.

We conclude, therefore, that the decision to act rather than deliberate arises in a number of situations, most of which do not require a conscious decision by the agent to do one or the other. This is a good thing in our view, since making that decision in a principled way is not going to be possible at runtime.

4.2.1 Simultaneous action and deliberation

A strange assumption implicit in the decision-theoretic-control paradigm is that deliberation and action cannot happen simultaneously. There seems to be no good reason to make this assumption, yet that is the assumption one makes when one speaks of trading a time unit of inference (deliberation) against a time unit of action. If we relax that assumption things get much more difficult, in that the opportunity cost of deliberating becomes harder to compute. Options like "think about the problem while acting so as to keep your options open as long as possible" become available. This kind of behavior is common in everyday life—waiting in a long line for a movie, for example, while simultaneously discussing the possibility of going elsewhere.

When we admit the possibility of simultaneous action and deliberation our attention focuses on *coordinating* the two processes rather than *choosing* between them. When should the deliberation system interrupt the current course of execution because it has discovered a better way to achieve the goal (but perhaps too late)? Similarly, when should the execution system interrupt the deliberation system because it has discovered the plan is going wrong, and what should the deliberation system do at that point? How does one facilitate the sharing of representations (as outlined above) when the two processes are running more or less independently? These are the questions we believe are central to building an agent that acts, and reacts, intelligently in its world. Our current research centers around coordinating the independent and simultaneous operation of the execution and deliberative systems.

5 Related Work

Some of the most influential ideas in the AI literature come from the "classical planners" such as STRIPS [Fikes and Nilsson, 1971], NONLIN [Tate, 1977] and DEVISOR [Vere, 1983]. The basic model behind these systems is to plan everything in advance and *prove* that it will work. There has been a lot of work to enhance these systems in various ways by adding non-monotonic notions of proof and domain-dependent search heuristics [Wilkins, 1988, Dean *et al.*, 1987] but the notion that a plan cannot be executed until proven correct remains unchanged. We claim that this idea is fundamentally flawed because urgency, uncertainty, and exogenous events make it impossible for an agent to prove realistic plans correct in advance. An agent must be able to plan without proof and let the execution system take care of the details. Such a plan won't always work, but if you cannot act in the real world without a provably correct plan, then you cannot act at all.

In response to the impossibility of constructing a provably correct plan before acting, some researchers have been building systems in which explict plans are not required. The basic idea behind these systems is to use either a programmer [Brooks, 1987] [Agre and Chapman, 1987], or an automatic system [Kaelbling, 1988], to transform a description of the agent's goals into a machine to achieve those goals. Within the machine there is no explicit set of goals, world model, or contemplated plan of action. Information flows from the machine's sensors through a decision network to enable or disable actions; plans and goals appear only implicitly in the machine's interaction with the world. Using a stateless decision network ensures fast response to changes in the environment, but it does not allow action choices to be influenced by deliberative processes. When goals are not represented explicitly, they cannot be changed dynamically and there is no way to reason about alternative plans for carrying them out. We claim that the ability to act on changing goals in a reasoned manner is the hallmark of an intelligent agent and must not be neglected in the face of urgency and uncertainty. Our execution system is designed for quick response *informed by* a deliberate plan.

Numerous viewpoints exist on ways that deliberation and action might be combined within a single agent architecture. These views fall into two broad catagories: uniform and layered architectures. Uniform architectures use a single representation and control structure for both action and deliberation while layered architectures use different algorithms and knowledge representations to perform these functions in different layers.

One example of a uniform architecture is the PRS system [Georgeff *et al.*, 1986] which interleaves planning and acting by using the same engine to project the future, propose plans, and initiate actions. However, while PRS allows goals, plans, and world knowledge to be represented in a common language interpreted by a single processor, it does not make any commitment as to exactly how planning and acting should be interleaved. All control decisions are encoded in the PRS language (*i.e.* meta-KAs) and PRS itself is intentionally silent on language content. Thus, PRS is a framework in which agent architectures can be couched but it is not itself an agent architecture. Other researchers have also developed uniform frameworks in which control knowledge can be encoded as domain specific rules [Hayes-Roth, 1990] but, as yet, they too have little to say on the general problem of when to deliberate and when to act.

The *Entropy Reduction Engine*, [Bresina and Drummond, 1990], is an example of a layered agent architecture. This architecture includes three distinct layers:

- the *reactor*, which initiates actions based on a plan-net modified by situated control rules SCRs,
- the *projector*, which constructs new SCRs to direct the reactor based on the plan-net and goal constraints, and
- the *reductor*, which generates goal constraints to direct the projector toward solving the system's overall goals.

Each layer is designed to work independently, but control information flows from goal to constraint to SCR to action. The advantage of such an approach is that each layer constitutes a separate process, and there is no need to decide between deliberating and acting; the reactor is always acting and the reductor and projector are always deliberating.

A problem with the *Entropy Reduction Engine* is that each interface between layers involves a different representation, and information only flows one way—from goals to actions. As a result, there is no way for the activities of the reactor to influence the activities of the layers above. This is particularly important when the world model used during deliberation is not complete (which we argue is inevitable) and actions in the plan net do not have their usual or intended effects. Suppose an SCR fires in a certain situation and causes an action selection that does not change the world in any currently discernable way. Presumably the same SCR will fire again and the reactor will be caught in a futile loop, performing the same action over and over without making progress. We argue that the unreliability of world models and the uncertain effects of actions must be accepted as central aspects of any agent architecture and the execution process must be able to recognize problems as they arise and communicate that knowledge to the deliberation system.

The division between uniform and layered architectures is an interesting one because it reflects a bias as to which problems the implementors wish to address. Uniform architectures such as PRS make the assumption that deliberation and action are so closely intertwined that they cannot fruitfully be separated. Within a uniform architecture, all of the deliberation machinery can naturally and easily be brought to bear on every action decision the system makes. Unfortunately, computational cost and complexity quickly slow such a system to the point where it cannot react in a timely manner. To cope with time pressure the system must begin to make explicit control decisions about when it should deliberate and when it should "just act." The problem for such a system is to decide what it means to "just act" and then decide how the trade-off between action and deliberation should be made. The most common approach is to make control decisions explicit in the plan representation and allow the system to reason about its own reasoning procedures (*e.g.* meta-KAs in PRS and control plans in [Hayes-Roth, 1990]). One must be careful that such schemes do not lead to an infinite regress of meta-reasoning.

Layered architectures like the *Entropy Reduction Engine* and that used on the ALV [Payton, 1986] make the assumption that reaction time is so critical and deliberation is so slow that action must often be taken without resort to any deliberation at all. As a result, deliberation and action are separated into pieces that use different algorithms and often different world models and plan representations. Unfortunately, this division raises a new communications problem: how can the different pieces share each other's knowledge and commitments. The most common approach is to pass ever more detailed action descriptions down the layers toward the hardware and then pass success and failure messages back up. One must be careful that such communication schemes are made rich enough that each layer does not have to *model* the activities of the layer below.

We claim that different reasoning processes are appropriate for acting and deliberation because action is driven by urgency while deliberation needs time to consider alternatives and consequences. Therefore we have adopted a partitioned rather than a uniform architecture and we address the communication problem by using a shared world model and plan representation. Deliberation and action both manipulate plans and the world model but use different algorithms focussed on the different problems that each must address.

6 Conclusions

Intelligent agency requires two distinct capabilities: deliberating to consider alternative futures when time permits, and acting when urgency or lack of knowledge demands it. The division between the two processes is more pragmatic than theoretical but still very real. Uncertainty, urgency, and the complexity of the world require adaptability and reactivity when taking action. The same uncertainty and complexity complicates projection and the comparison of possibilities while deliberating over alternative plans of action. The different aspects of "planning" that are emphasized during deliberation and action require the use of separate systems and algorithms for their implementation.

Our architecture consists of an execution system based on RAPs [Firby, 1989] and a deliberation system based on a probabilistic world model manager and projector [Hanks, 1990c]. A proposed transformational planner generates and refines plans based on information supplied by the model manager. These plans are represented explicitly at all levels of refinement and are available to guide the execution system as it interacts with the world. To this end the execution system is charged with the responsibilities of: (1) making detailed action choices to carry out planned actions in the face of uncertainty, (2) coping with the failure of those actions to achieve their intended results, and (3) watching for and dealing with urgent problems or opportunities. Similarly, the deliberation system is charged with the responsibilities of: (1) proposing and refining plans to guide the execution system, (2) making predictions about future expected states of the world, and (3) coping with any plan changes required when new information arrives or the execution system detects a problem.

Although the algorithms for deliberation and action are different, the two activities are parts of the same process. They share a central world model and a representation of plans of action. The world model is a record of what is currently believed about past, present, and future states of the world. It supplies the execution system with the information needed to choose between methods for a plan step and to confirm that a plan step has been achieved. It supplies the deliberation system with the grounding needed for making inferences and predictions about the future. Sharing the model makes new information acquired by the execution system avail-

able immediately to the deliberation system, and allows new projections of the future made by the deliberation system to inform further choices in the execution system.

A common world model lets the deliberation and execution systems exchange information, but a shared plan representation is what really ties the two systems together. Plans are the description of what the agent intends to do and are used by the deliberation system for projecting the future and by the execution system for guiding action selection. A shared plan representation allows any plan under consideration by the deliberation system to be executed directly, and any plan refinements generated by the execution system to be incorporated directly into ongoing deliberations. A shared representation also allows the deliberation system to use the same plan library when considering possible futures as the execution system uses when selecting actions at runtime. A common plan representation gives the agent a single consistent vision and purpose.

6.1 Some Parting Philosophy

The classical planning paradigm makes two very convenient assumptions:

1. That there exists a set of "primitive actions," which provides a convenient level of abstraction for use in plans — below that level the planner doesn't have to worry about execution at all.
2. That the world is certain, simple, and closed, which leads to an easy criterion for choosing a plan: a plan is good if it provably works.

Neither of these assumptions are entirely bad. There probably *is* some detail the planner should ignore (the exact position of a glass in the kitchen, for example), because: (1) it generally won't know details of this sort before execution time, (2) such details probably won't affect the plan's ultimate success very much, and (3) there are so many details that the planner will be swamped if it tries to take them all into account.

Closed-world assumptions are also fairly harmless if judiciously applied: the agent probably *does* have control over and reasonably complete knowledge about *many* imporant aspects of the world *sometimes*, and making closed-world assumptions makes the process of reasoning about possible futures much more efficient [Hanks, 1990c, Chapter 5].

The problem is that these assumptions have been applied too broadly: primitive actions are taken to be fixed and complete, and the closed-world assumption is applied to all propositions at all times. In fact, the situation is often even worse in that the assumptions are often actually inherent in the representation itself (*e.g.* the situation calculus representation of events, which makes reasoning about asynchronous non-planned actions very difficult, and the STRIPS action representation with its add and delete lists). As such, it becomes impossible for systems using these representations even to reason situations in which the assumptions are violated.

What we've tried to do is build a system in which these assumptions (atomic actions and a closed world) can be applied when appropriate, where the meaning of "appropriate" depends on both the planning problem at hand and the execution environment. The RAP representation provides a reasonable and flexible notion of an atomic action because the execution system ensures that either (1) the details involved in carrying out a RAP goal are taken care of "transparently," or (2) the planner will be notified that some anomaly (planning failure) has been detected.

The projection susbsystem makes the closed world assumption for particular aspects of the world and over particular periods of time, but is also sensitive to the fact that these assumptions might be violated. It is willing to revise its beliefs if subsequent information, discovered at execution time, demands it. The projection system also "knows when it doesn't know enough" to make decisions, and can pass actions to the execution system that will provide it with the necessary information.

The important point is that while we attempt to employ simplifying assumptions appropriately, we know that these assumptions are too strong in general and therefore accept that mistakes and inconsistencies will arise through their application. Much of the planning process must wait until more detailed information becomes available and often an agent's interaction with the world will uncover new information that invalidates previous plans. A real agent architecture must allow a flexible and complex interaction with the world that the traditional notion of planning and execution cannot support. We offer a system in which planning becomes a continuous process of deliberating over evolving partial plans, which are designed to guide an action system, while at the same time gathering information and acting on the world to achieve the agent's goals.

References

[Agre and Chapman, 1987] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings AAAI*, pages 268–272, 1987.

[Boddy and Dean, 1989] Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings IJCAI*. AAAI, August 1989.

[Bresina and Drummond, 1990] John Bresina and Mark Drummond. Integrating planning and reaction. In *Spring Symposium Series*, Stanford University, 1990. AAAI.

[Brooks, 1987] Rodney Brooks. Intelligence without representation. In *Proceedings of the Workshop on the Foundations of Artificial Intelligence*. MIT, June 1987.

[Dean *et al.*, 1987] Thomas Dean, R. James Firby, and David Miller. The FORBIN paper. Technical Report 550, Yale University, Department of Computer Science, July 1987.

[Etzioni, 1989] Oren Etzioni. Tractable decision-analytic control. Technical Report CMU–CS–89–119, School of Computer Science, Carnegie Mellon University, February 1989.

[Fikes and Nilsson, 1971] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208, 1971.

[Firby, 1989] R. James Firby. Adaptive execution in complex dynamic worlds. Technical Report 672, Yale University, Department of Computer Science, January 1989.

[Georgeff *et al.*, 1986] Michael P. Georgeff, Amy L. Lansky, and Marcel J. Schoppers. Reasoning and planning in dynamic domains: An experiment with a mobile robot. Tech Note 380, AI Center, SRI International, 1986.

[Haddawy and Hanks, 1990] Peter Haddawy and Steve Hanks. Issues in decision-theoretic planning: Symbolic goals and numeric utilities, 1990. This volume.

[Hanks, 1990a] Steven Hanks. Controlling inference in planning systems: Who, what, when, why, and how. Technical Report 90-04-01, University of Washington, Department of Computer Science, April 1990.

[Hanks, 1990b] Steven Hanks. Practical temporal projection. In *Proceedings AAAI*, 1990.

[Hanks, 1990c] Steven Hanks. Projecting plans for uncertain worlds. Technical Report 756, Yale University, Department of Computer Science, January 1990.

[Hayes-Roth, 1990] Barbara Hayes-Roth. Dynamic control planning in intelligent agents. In *Spring Symposium Series*, Stanford University, 1990. AAAI.

[Horvitz *et al.*, 1989] Eric J. Horvitz, Gregory F. Cooper, and David E. Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings IJCAI*, pages 1121–1127, 1989.

[Kaelbling, 1988] Leslie Pack Kaelbling. Goals as parallel program specifications. In *Seventh National Conference on Artificial Intelligence*, St. Paul, MN, August 1988. AAAI.

[Linden and Owre, 1987] Theodore A. Linden and Sam Owre. Transformational synthesis applied to alv mission planning. In *Proceedings of the DARPA Knowledge-Based Planning Workshop*, December 1987.

[McDermott, 1982] Drew McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6:101–155, 1982.

[Payton, 1986] D.W. Payton. An architecture for reflexive autonomous vehicle control. In *International Conference on Robotics and Automation*, San Francisco, CA, 1986. IEEE.

[Simmons, 1988] Reid G. Simmons. A theory of debugging plans and interpretations. In *Proceedings AAAI*, pages 94–99, 1988.

[Tate, 1977] Austin Tate. Generating project networks. In *Fifth International Joint Conference on Artificial Intelligence*. IJCAI, 1977.

[Vere, 1983] Steven Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3), 1983.

[Wilkins, 1988] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan-Kaufmann, 1988.

Envelopes as a Vehicle for Improving the Efficiency of Plan Execution*

David M. Hart, Scott D. Anderson, Paul R. Cohen
Experimental Knowledge Systems Laboratory
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

Envelopes are structures which capture expectations of the progress of a plan. By comparing expected progress with actual progress, envelopes can notify the planner when the plan violates those expectations. The planner then has the opportunity to modify the plan to increase its efficiency given the unexpected progress. This paper presents a specific example of the construction and use of an envelope, followed by a discussion of the general utility of envelopes for improving the efficiency of plan execution.

1 Introduction

Most AI planners test the postconditions of an action after its completion to see if it succeeded, but in our domain, actions take so long to execute that advance knowledge of the probable outcome is valuable. Therefore, we monitor actions *during* execution. We represent the *a priori* expectations of action progress, which we compare with the actual state, in structures we call *envelopes*. By comparing actual progress to the expectations about progress stored in envelopes, we can see whether a plan[1] is executing better or worse than we expected.

Inefficiency in plan execution encompasses wasteful use of resources in a plan that is succeeding, ineffective use of resources in failing plans, and even the costs of recovering from a failed plan. Envelopes are chiefly concerned with the efficiency of plan execution, and only indirectly with the planning process. In this paper, we will show an envelope that categorizes the progress of a plan as *better*, *worse*, or *as* expected. If progress is better than expected, we can increase the efficiency by reducing resource expenditure; if worse, we can act to avert plan failure, possibly by adding resources.

The term "envelope" derives from the idea of a "performance envelope" in engineering, describing the performance profile of a mechanism under various conditions. When the performance of a plan in our system goes outside its envelope, the plan may no longer be appropriate for the current environmental conditions. This paper details the construction and use of a particular envelope in a multiagent, real-time problem solving system that fights simulated forest fires. It also discusses the general utility of envelopes for improving the efficiency of plan execution.

2 Monitoring Plan Execution

There is an obvious advantage to knowing how a plan is progressing when the planner can act based on that knowledge; if the plan is failing, the planner can either abort the plan (avoiding throwing good resources after bad) or add resources to the plan so as to avert the failure. In domains where actions are not interruptible or are of such short duration that there is no time to add or subtract resources from an action in progress, there is clearly no utility to monitoring an executing action. But fighting a forest fire can require the efforts of many agents over many days, so that plans execute over long time spans and there is ample time to add and subtract resources. Therefore, in our domain (simulated in a testbed called Phoenix [1]), it pays to monitor a plan during its execution.

Doyle's work [2] addresses monitoring, but in a robotics domain using a STRIPS-style action model with specified preconditions and postconditions. Monitoring verifies the truth of the preconditions and postconditions:

> ...we assume that the successful execution of actions can be verified by instantaneously verifying the action's preconditions before its execution and instantaneously verifying its postconditions after its execution. This approach proves inadequate for some actions. [2]

Doyle goes on to describe cases that require just the sort of continuous monitoring that envelopes are designed for, such as actions that are extended over time and can fail at any point, and actions involving looping, which can be viewed as extended action.[2]

Our work is closest in spirit to that of Sanborn and Hendler [6]. Their simulated robot, which tries to cross a

*This research was sponsored by DARPA-AFOSR contract F49620-89-C-00113; the Office of Naval Research, under a University Research Initiative grant, ONR N00014-86-K-0764; the Office of Naval Research, contract # N00014-88-K-0009; and a grant from the Digital Equipment Corporation.

[1]Because our plans are structures of one or more actions, we will use the terms interchangeably.

[2]The example of looping that he gives is of filling a bucket from a hose, which we believe is more naturally viewed as an extended action.

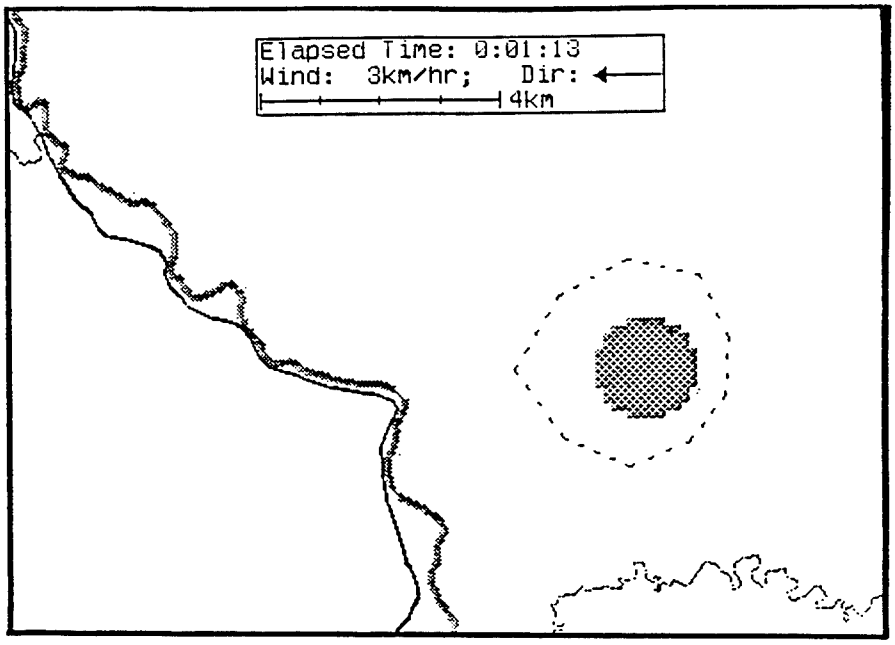


Figure 1: A fire (shaded region) has been set in the Phoenix forest, seen here 1 hour 13 minutes afterwards. The wind is from the East at 3 kph. The polygon of dashed lines marks both the projected shape of the fire after about 17 hours and the intended placement of fireline for the indirect-attack plan. Other lines are rivers and roads.

busy street, must monitor the objects in the world (cars rushing past) and predict whether they will run over it. We view this as an envelope around the plan of crossing the street, attempting to avert the catastrophic failure of the plan, not to mention the robot. This is a clear case of an extended action (though performed as a loop of single steps forward and backward) in which forewarning of failure is critical. The forewarning is achieved by predicting the location of the cars and the robot; we will see the significance of prediction for envelopes below.

Envelopes have been implemented as a general mechanism in Phoenix. Performance falling outside expected bounds is termed a "violation," and violations notify the planner so that its planning knowledge can be brought to bear on the situation. The purpose of envelopes, then, is to provide information to the planner that guides its decision-making during plan excution. While the planner has a number of options, it typically responds to violations as we have mentioned—adding or subtracting resources. Without envelopes, these opportunities to increase efficiency would go unnoticed.

3 Constructing an Envelope

In Phoenix, simulated forest fires are controlled by bulldozers cutting fireline around the fire. In some cases, it is too dangerous for bulldozers to cut a fireline close to the fire, and so we use what is called "indirect attack," in which the bulldozers cut a line some distance away. In indirect attack, a central fireboss coordinates the actions of the bulldozers. For example, in Figure 1, the intended placement of fireline, to be cut by several bulldozers, is the polygonal shape surrounding the fire. The fireboss selects a polygon such that the estimated time required for n bulldozers to cut the line, $BT(n)$, is less than the estimated time remaining until the fire spreads to the polygon, FST; the difference is the amount of slack time in the plan.

Figure 2 illustrates how an envelope for the multiple-bulldozer, indirect-attack plan is constructed. We define "progress space" as the percentage of the fireline which is completed, PFC, versus time (elapsed simulation time, t). The point at the upper right is the estimated time that the fire arrives at the polygon, t_{fa}, and 100% of the fireline is dug. Lines 1 and 2 are defined by the expected rate that some number of bulldozers can cut fireline: line 1 has a slope of $100/BT(n)$, because n bulldozers must cut 100% of the line; line 2 has a slope of $100/BT(n-1)$. t_p is the time that these estimates were made and the envelope was built. t_{ls} is the latest time at which n bulldozers can start digging line and expect to finish the fireline before the fire arrives.

The filled circle labeled CP represents the current position in progress space—(t_{now}, PFC_{now}). The location of CP within the regions of progress space indicates how the plan is progressing: crossing below line 1 suggests that the plan will fail, since the bulldozers would need to cut the fireline at a rate faster than we think they can;[3] crossing above line

[3]The plan will not *necessarily* fail, since the bulldozers might do better than we expect, even though they so far

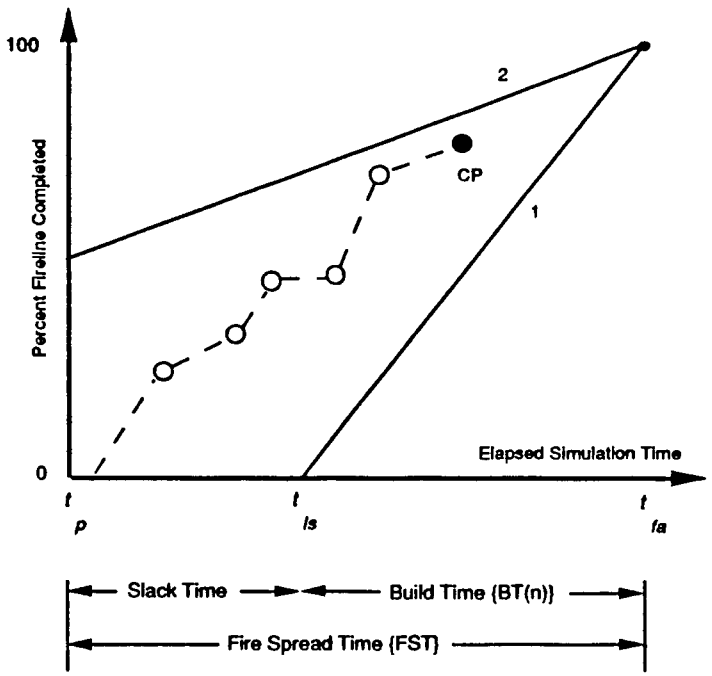


Figure 2: An envelope for the Indirect-Attack Plan

2 suggests that we should save resources by retiring a bulldozer, since $n-1$ of them should be able to cut the remaining line in time. The area between the lines is the envelope—the range of expected performance—and going outside it is a violation of the envelope, signalling to the agent that something should be done.[4]

4 Using Envelopes

The plan library of each Phoenix agent contains skeletal forms of envelopes as well as plans. Part of the definition of a plan denotes what envelopes, if any, should be instantiated to monitor the execution of that plan. Consequently, when a plan is instantiated at run-time, the associated envelopes are also instantiated, and the envelopes initialize themselves from plan variables, such as the $BT(n)$ and t_{fa} variables above. Each envelope provides a *monitor method* whose purpose is to update the current progress (CP) and locate it in progress space. Then, while the plan is executing, a periodic action called "monitor-envelopes" causes the agent to verify that the plan is progressing satisfactorily by running these monitor methods.

The monitor method checks sensory information previously gathered and stored in the plan variables, such as the current positions of the bulldozers and the fire, and determines which region CP lies in. If CP is within the region for acceptable progress, nothing more happens. However, if CP has crossed into a region of unexpected progress (either better or worse), the envelope is violated and the monitor method adds an item to the agent's agenda so that the agent can notice and respond to the violation. In Figure 3, we show an envelope in which the CP falls into the failure region.

While the fireboss could do many things as a result of this violation (for example, the fireboss might buy time, say by dumping fire retardant on the fire or expanding the polygon around the fire), consider the case in which it sends another bulldozer to help dig line. A new envelope must be set up for monitoring this modified plan. The failure boundary for the new envelope will be determined by a line whose slope is $100/BTn+1$. The additional bulldozer is sent, and the fire is successfully contained within the original polygon, which means the modified plan has succeeded, where the original plan would almost certainly have failed.

We have mentioned that multiple envelopes might exist simultaneously; one way that this occurs is when a plan and a step in the plan both have envelopes. For example, in the multiple-bulldozer, indirect-attack plan, each step of digging a side of the polygon has an envelope. This allows the fireboss to apportion the time constraints from the overall plan to the steps in the plan. A violation of a step's envelope may indicate a problem with the whole plan, or may simply mean that other steps will have to do better than expected. Therefore, the step envelopes do not eliminate the need for the plan envelope—the latter integrates the information from step envelopes.

Furthermore, since digging a side of the polygon will in fact be executed by bulldozers and not by the fireboss, the envelope for that step must be an explicit data structure that can be communicated to the bulldozers. We call this an *agent envelope*, since it is monitored by the agent who receives it, allowing the fireboss to turn its attention elsewhere. If a violation occurs, the agent reports back to the fireboss, who assesses the significance of the violation by consulting the plan envelope. Agent envelopes free the fireboss from

have not, or the fire might take longer to reach the fireline than we thought, say if it rains.

[4]Conceptually, the envelope is just that area of progress space, but we also use the term to describe the data structure representing this area and associated code for creating the envelope and updating CP.

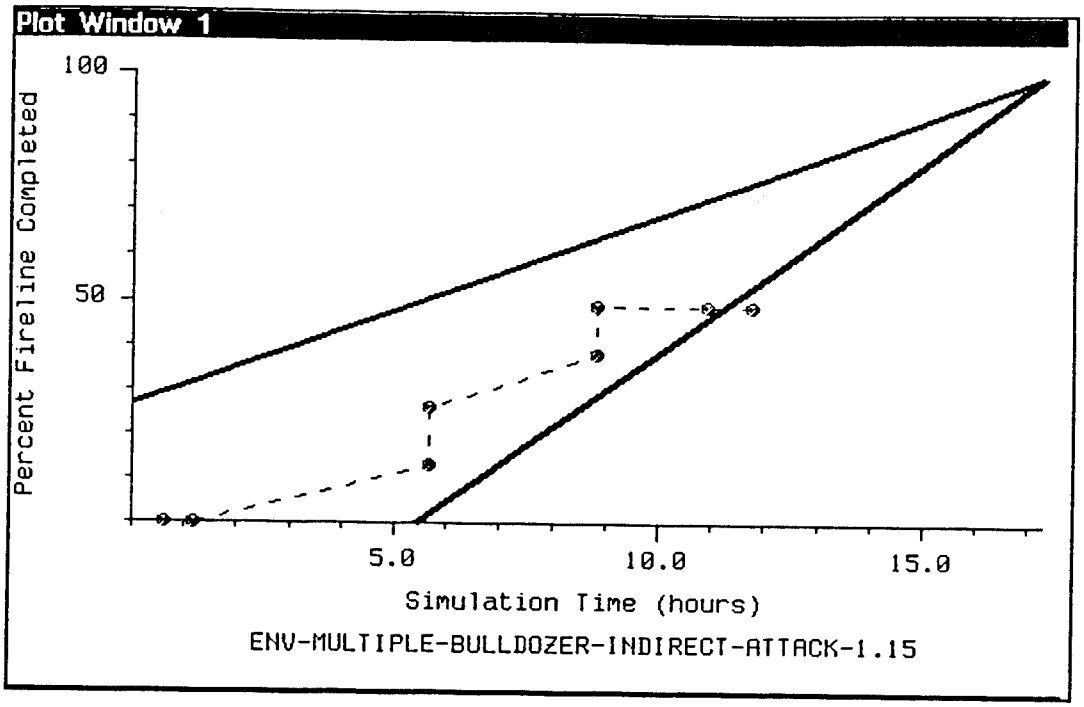


Figure 3: Plot of the Multiple-Bulldozer, Indirect-Attack Envelope in Phoenix, showing a violation at nearly 12 hours into the simulation. The recent progress is flat because the bulldozers have stopped to refuel.

the task of monitoring the progress of component plan steps carried out by other agents—the fireboss assumes the agent's progress is within expectations unless it receives a violation report from the agent. Powell and Cohen [5] discuss the use of envelopes to coordinate activities among echelons in multiagent, operational planning.

Another way that multiple envelopes occur is when data dependencies exist between envelopes. For example, the estimate of when the fire will reach the polygon, t_{fa}, is crucial to the way that progress space is carved into regions. Unfortunately, that estimate is based on inexact information, because the fireboss does not know exactly where the fire is or exactly what the local wind conditions are. Therefore, we put an envelope around our estimate and periodically recalculate it. Should the re-calculation indicate that the t_{fa} estimate is quite wrong, the t_{fa} envelope is violated, and this causes the envelope on the plan to be revised.

5 Utility of Envelopes

It is intuitively obvious that information about the progress of a plan cannot hurt and could well prove invaluable. However, the agent must put effort into gathering this information, and therefore the cost of envelopes must be worth their benefit. To summarize the possible benefits, using envelopes allows an agent to:

- modify a failing plan so as to prevent its failure
- abandoning a plan that is irretrievably failing.
- retire surplus resources from a plan that is going unexpectedly well.
- improve a plan that is going unexpectedly well. For example, in the multiple-bulldozer, indirect-attack plan, it can move the vertices nearer the fire so as to reduce the forest lost.
- reduce communication overhead between cooperating agents via agent envelopes, that is, by allowing them to share expectations and only communicate when those expectations are violated.

Envelopes cost the planner in three different ways: the cost of setting them up, the cost of monitoring them at time intervals, and the cost of responding to violations. Assessing these costs is complicated by the way they can be traded off against each other. For example, a planner could create a "quick and dirty" envelope, using estimates that are of low quality but quick to compute. Employing such an envelope burdens the planner later, since spurious violations are more likely and time spent responding to them will be wasted. On the other hand, a planner can put a lot of time and effort into creating a great envelope, with precise boundaries based on the best information, resulting in regions that categorize the situation quite well. Violations of a high-quality envelope can be better trusted to indicate a problem with the plan. We can create such high-quality envelopes in Phoenix for some activities, but the cost of creating them is high. For example, the speeds with which a bulldozer travels and digs line can be predicted quite accurately using expensive operations that iterate over the points on its route and sum the costs. The benefit is that the envelope reflects very closely the probable time required to dig a segment of fireline. The cost is the expense of calculating this information, a cognitive task that competes with other necessary cognitive activities for available "thinking" time[5]. This tradeoff balances the cost of building an envelope now with the costs of responding to violations later.

[5]For more on Phoenix agents' cognitive structure, see [1]

Another tradeoff is in the monitor methods of the envelopes. It's important not to make these too time-intensive, since the cost will be incurred many times over the course of plan execution. For instance, how old should sensory information be before the monitor method discards it and measures the environment anew? One option is to use quick, inexact procedures in the monitor method and then, if a violation occurs, double-check them with better procedures to verify whether the violation is spurious.

In very time-pressured situations, an agent will probably want to choose quick and dirty ways of doing things (including building, monitoring and responding to envelopes), while in less pressured situations, it will probably want to choose higher quality methods; therefore, the Phoenix agent architecture allows for this choice.

An extreme tradeoff is to eliminate envelopes entirely and deal with the plan failure when it arises, that is, dispense with monitoring the progress of the executing plan and rely on reports from the field that the plan has failed. Clearly, in Phoenix, we will have to repair or replace the current plan when the fire is reported to be escaping from the incomplete polygon. The cost of this failure, besides the time to replan, is the loss of more forest and the additional time and effort of bulldozers to control it. The cost of failure must be compared to the costs of using envelopes, which we've noted will depend on the choices made by the agents. We believe that on average these costs will exceed the overhead cost of using envelopes. The same argument can be made with respect to improving a plan that is succeeding: if we can save a little forest by moving the vertices towards the fire when the polygon is being dug faster than anticipated, does this outweigh the cost of using envelopes?

These tradeoffs imply that envelopes have limited utility for some environments and tasks. If the environment is highly variable, so that the estimates and predictions that are built into the envelope don't last, and so that any envelope will be violated shortly, the overhead costs may swamp any benefits. On deeper reflection, though, since envelopes are used for actions that assume some constancy to the environment, such actions would not be used in these highly variable domains. Predictability is the key issue: prediction, used either for planning or building envelopes, is simply not useful in these unpredictable domains. As we mentioned in Section 2, Sanborn and Hendler's simulated robot depends on the predictability of the cars' paths, even though the domain is highly variable. Our approach differs from theirs because they have tightly connected the predictions to the robot's actions, while we notice a violation and let the planner deliberate as long as it wishes over how to solve the problem. We do this because actions can be quite costly; for example, sending another bulldozer to the fire is time-consuming and commits resources which might be required elsewhere.

6 Current and Future Work

We have integrated envelopes into the Phoenix testbed, and added instrumentation to assess the cognitive load on the various agents. We will test the utility of envelopes by comparing the performance of agents with and without them. In particular, we will run a number of different fires in the simulator, varying the factors contributing to unexpected success or failure of plans, such as weather, mechanical breakdowns, and obstacles. Performance will be assessed based on a combination of cost factors such as forest burned, bulldozer time spent, agents lost, and cognitive overhead incurred. We also intend to experiment with the tradeoffs mentioned above, such as balancing the cost of setting up an envelope with the cost of responding to violations.

Another line of research views envelope violations as an opportunity for learning [3]. Envelopes provide information about plans vulnerable to failure, as well as an opportunity to test ways of repairing plans. These repairs, when successful, can be used to modify the plan library.

We also intend to apply envelopes to the problem of assessing the progress of tasks that involve, not acting in the world, but thinking. Many thinking tasks in Phoenix, such as path planning and predicting fire spread, can be computationally expensive and the time available for them is limited. If we can model these computational tasks so that we can predict their progress well enough to use envelopes, we can control them and increase their efficiency. A brief discussion of the use of envelopes for real-time control appears in [4].

7 Conclusion

We have shown an example of how to build and use an envelope for a plan in a fire-fighting domain. This envelope notices when the plan is failing or succeeding too well. The planner can then adjust the plan to the changing conditions, thereby increasing the efficiency of its execution by minimizing the loss of forest and other costs. We've argued that the predictability of the environment indicates whether envelopes will be useful: if the domain is too predictable, plans cannot fail, so there is no point to monitoring them, and if the domain is too unpredictable, plans would not be of a duration for which envelopes would be useful. For the middle ground—environments that are uncertain but not too unpredictable, which we think are quite common—we argue that the benefits derived from the opportunity to increase plan efficiency will outweigh the costs of creating, monitoring and responding to envelopes.

Acknowledgements

The authors wish to thank Adele Howe, Dorothy Mammen, Jerry Powell, and Paul Silvey for helpful comments on drafts of this paper and also Mike Greenberg and David Westbrook for their skilled programming support.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Air Force Office of Scientific Research under Contract No. F49620-89-C-00113. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

References

[1] Paul R. Cohen, Michael Greenberg, David M. Hart, and A.E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, Fall 1989.

[2] Richard J. Doyle, David J. Atkinson, and Rajkumar S. Doshi. Generating perception requests and expectations to verify the execution of plans. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, Philadelphia, PA, 1986. American Association for Artificial Intelligence.

[3] A.E. Howe. Integrating adaptation with planning to improve behavior in unpredictable environments. In *Proceedings of AAAI Spring Symposium on Planning in Uncertain, Unpredictable or Changing Environments*, Palo Alto, CA, March 1990.

[4] A.E. Howe, David M. Hart, and Paul R. Cohen. Addressing real-time constraints in the design of autonomous agents. To appear in *Real-Time Systems*, 1990.

[5] Gerald M. Powell and Paul R. Cohen. Operational planning and monitoring with envelopes. In *Proceedings of the IEEE Fifth AI Systems in Government Conference*, Washington, DC, 1990.

[6] James C. Sanborn and James A. Hendler. Dynamic reaction: Controlling behavior in dynamic domains. *International Journal of Artificial Intelligence in Engineering*, 3(2), April 1988.

Mission Critical Planning: AI on the MARUTI Real-Time Operating System

James Hendler * and Ashok Agrawala †
Department of Computer Science
University of Maryland
College Park, Md. 20742

Abstract

In this paper we describe preliminary results from an effort to use the MARUTI real-time operating system as the platform for the development of an AI system which can integrate planning and reaction in complex environments. This work is motivated by the needs of the types of mission critical reasoning which must occur during emergencies which arise in complex domains. The primary thrust of this ongoing research is to enable a system to directly react to failures and unexpected events. Critically important is the ability to correctly handle time dependent reasoning (directly modeled on the MARUTI operating system) and reaction, and to integrate such reaction with higher level plans for normal operation or error recovery.

1 Motivation

For systems to handle contingencies that might arise in complex environments, there are three critical capabilities they must demonstrate:

1. *Real-time response.* Failures have a time criticality attached to them. The time between the occurence of a failure and damage (to a mission or a piece of equipment) resulting from that failure is often short, but not instantaneous. Thus, for many potential failures there is a "sampling time" which, if guaranteed, will permit the problem to be detected prior to the negative results. These sampling times can be predetermined, but they must be guaranteed — this is a hard real-time, as opposed to a speed, requirement.
2. *Context sensitivity.* The appropriate actions to be taken upon encountering a failure must be sensitive to the overall "mission status" at the time the failure occurs. For example, in an aircraft control system, if a minor failure occurs the pilot should usually be notified and required to authorize the system to take an appropriate automated repair action. However, if that same failure occurs during combat, the pilot (who is too busy to notice) should not be disturbed, and the repair action should be taken automatically or delayed. Upon completion of the engagement, the pilot should be notified about the actions taken or a request for authorization can then be sought.
3. *Planning or rule-based reasoning coupled with reaction.* When a situation requiring an emergency response occurs, the system must project how repair actions to be taken might effect (i) other repairs or normal operations and (ii) the long range goals of the mission. Thus, in a fuel critical mission a repair which might involve an extra engine burn should not be taken without notifying the pilot that this might cause an additional fuel shortage. To allow this kind of behavior, the system must use traditional AI models of expert system rule reasoning (for a shallow model) or temporal projection (if a deep model of the reasoning is required). However, along with this long-term behavior, the system must make short term (or "reactive") adjustments that will preserve mission integrity. Thus, an action which preserves flight stability must be taken without first spending critical time in reasoning about long term effects. After integrity is restored, the long term consequences can be considered.

One way to achieve such behaviors is to implement AI planning technology, particularly "reactive planning systems" on a guaranteed scheduling, real-time operating system. In this manner we are able to take advantage of the reliability provided by the OS, while using the AI technology to guide the reactions taken by the system. Our work in integrating AI and real-time computing focuses on the joining of two ongoing research efforts: an AI system designed to provide reaction and a real-time operating system developed for guaranteed scheduling in mission critical computing environments. In the remainder of this paper we briefly describe the two systems being merged (section 2 and section 3). We also describe some experiments already performed which have shown the feasibility of implementing the system (section 4).

*Also affiliated with the UM Systems Research Center.

This work was funded in part by NSF IRI-8907890 and ONR grant N00014-88-K-0560.

†This work was supported in part by contract DSAG60-87-C-0066 from the US Army Strategic Defense Command. The views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of Defense poition, policy, or decision, unless os designated by other official documentation.

We conclude with a discussion of some of the critical research areas (in both AI and real-time) mandated by the sorts of applications described in this section.

2 Planning in dynamic domains

The classical approach to building AI planning systems[1] has been relatively unsuccessful in application when attempts have been made to extend AI systems to work in complex domains. These systems have significant problems with inefficiency – generating complete and accurate plans in even simple domains is an exponentially hard problem [2], and in more complex domains, particularly those involving other agents interacting with the planner, it is undecidable [19]. In addition, the basic control structure of typical planners, which assumes that plans can be generated and then separately executed, will often fail in such environments. Change in the world occuring during the running of a plan may render portions of it either temporarily or permanently unachievable. Appropriately responding to such changes in the environment requires a reactive component not available to most current planners.

As a simple example, consider a robot attempting to cross a street with a traffic flow. The robot cannot simply wait until a large enough gap occurs as (a) this may simply never happen, leaving the robot standing on the curb *ad infinitum* or (b) once the gap appears and the robot starts, one of the oncoming cars might change speed, direction, etc.

The difficulty in getting current planning systems to handle these *dynamic* situations is caused by the reliance of many of the current planning techniques on some very strong underlying assumptions:

1. the planner has complete knowledge of the world relevant to its task,
2. change in the world is brought about only by the planner's executing primitive plan steps. This change may be modelled discretely, and the planner is completely aware of all effects of its actions.
3. the planner acts alone in the world; there are no outside forces.

Unfortunately, real-world planning situations rarely conform to these assumptions. Typically occuring domains may include continuous change over time, incomplete specifiability at any point in time, and change due in part to the actions of other agents. Thus, the traditional planning paradigm has been shown to be inadequate in practical situations and much current research focuses on solutions. (A good set of papers on such work can be found in the Proceedings of the DARPA Workshop on Knowledge based Planning, Austin, Texas 1987.)

Our past research has focused on the development of techniques for managing observation and action in dynamic domains, known as *dynamic reaction (DR)* [9] and extending it to interact with a planning system via an abstraction hierarchy which combines monitoring and planning [10, 21]. This model is designed for dynamic worlds, where change is ongoing regardless of an agent's actions – strictly goal-directed methods are inadequate. Given observations of the world, the planning agent must coordinate its actions in order to act in harmony with ongoing events in the world.

In the DR model, an agent performs an activity until either (i) its goals lead it to select some new action, or (ii) some event in the world forces it to react. Since nondeterminism is inherent in dynamic worlds, a nontrivial amount of computation is required to arrive at and keep track of the current state of the world. The world is observed as a matter of course in DR, and relevant observations drive the selection of actions. In this way, the system does not explicitly track conditions for the safe execution of its actions. Projected failures are signalled through ongoing, independent observation provided by asynchronously processed, entities known as *monitors*[2].

The DR model has been used to handle the interactions arising in the "Traffic World", a rapidly changing simulation domain in which objects move rapidly through the simulation under external control. The "reacting agent" must cross this environment without being impacted, but under the control of a higher level directional goal. A full description of this work can be found in [9, 19].

In addition to the need for a reactive component, planning systems must be able to achieve long-term goals and to preserve the integrity of previously achieved goals. To handle reaction, the system must reason using a world model which closely matches the external world (this is, in fact, the same level as the model we have been using in the DR system). To interact with a more traditional, higher-level planner, however, the system needs to abstract away from the actual motions of the vehicles etc. If the planning system is too sensitive to the short term changes occuring in the world it is unable to generate long term plans due to the resulting combinatorial explosion. Coupling of DR with a more traditional planning model is accomplished via an abstraction-based planning architecture.

To deal with this problem we have developed a system which is capable of reasoning with multiple levels of abstraction of the world (a full description and formalization of this model of abstraction can be found in [10]). Keeping these levels consistent with one another, handling the interactions at each level in different time scales (for example the reactor may need to react in milliseconds, while the planner can take minutes), and propagating the perceptual information to the appropriate level are the critical problems which we have been addressing to allow planners to integrate "high-" and "low-level" knowledge.

Consider the following situation: a path planning system is to prepare a route over some map. The system designs a set of points to reach and deadlines to reach them. Such a system need not know the actual location of other objects in the world, it simply needs to know roughly how difficult different regions are to traverse. Once the plan starts executing, however, the sit-

[1] A review of these can be found in [11].

[2] The scheduling of these monitors can be performed in real-time. This is the essence of the work described in Section 4.

uation changes drastically. The reactive controller needs to know what other objects are in the perceptual field, what their heading and directions are, and when they will interact with the current path. The high level planner generates plans like "GET-TO POINT-A DEADLINE: +24min." The reactive controller controls moves like "PROCEED-LEFT NOW!"

In fact, the path planner itself may have no work to do during the actual running of the plan (this is a simplifying assumption used in several domain-dependent planning systems). However, if the world starts to get complicated, this assumption won't hold. Once, due to some reaction, the planner is to miss a deadline, it must replan and decide what route to take. During this replanning, however, it may not remain still – the objects which are causing it to miss the deadline may still be around!

The solution to handling such problems, lies in designing a system which can be reasoning "simultaneously" about different levels of the problem. In the DR model a preprocessed version of low-level perceptual data is supplied directly to the monitors which process it (actually hierarchically) to check a particular condition (the direction of a particular car, the speed of some object, whether any new object has appeared, etc.) When a monitor discovers that some condition holds, it updates a global "state-of-the-world" model (used for providing accurate information if needed by the high level planner) with a symbolic abstraction of what it has seen (for example "Speed CAR1 40-60units"). This information is also reported to higher-level monitors, which are used to control the actions of the system.

The planning component of our system is invoked via higher level (more abstract) reactive agents (higher-level monitors) which compute violations of required conditions. Thus, as the lower level reactors change the direction of movement, this is reported to a higher level entity which is checking that a deadline can or cannot be reached. It too updates the state-of-the-world model, but with higher level information – the information is kept at a level of abstraction useful to the path planner (for example, what previous deadlines have been met, the current location of the object, and the projected time to reaching the destination). The planner is then able to take over, when time permits and do the appropriate replanning. Just as was the original plan, this new plan is "compiled" down to new monitoring tasks, and the system continues.

To make this system work, a scheduler must be used to allow the planning and reacting agents to work together as time permits. Low-level monitoring must occur frequently, but for short periods of time. How much processing is required depends on the number of objects which the reactor must take into account. The higher level monitors and the planner itself require more processing time, but over longer intervals. Thus, as the time taken by pure reaction is reduced, the higher levels get more time. In a "safe" environment, this allows the planner to take almost complete control. Thus, in a highly reactive situation (crossing the street) the system is primarily reactive. In a relatively static world, the system becomes more like a traditional strategic planner. (See figure 1).

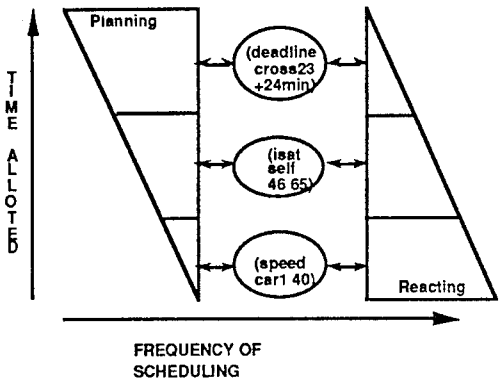


Figure 1: Abstraction and Scheduling

This work was originally implemented to run in a simulated environment which was an extension to the Traffic World modeling the path planning system described above [7, 8]. A more ambitious project, currently underway, simulates the environment for a household robot in a world consisting of a large number of objects, several agents, and a physical simulation consisting of over seventy thousand discrete locations. The success of preliminary experiments in this domain is described in [21]. A drawback of this system, however, is that it uses a simplistic scheduling algorithm which cannot guarantee the scheduling of requests or provide the sorts of support afforded by realistic operating systems.

To overcome this, we have begun work which focuses on the interspersed scheduling of reactive and planning tasks using the MARUTI real-time operating system. In the next section we summarize the work done in the development of the MARUTI system. This is followed by a description of some experiments performed to examine the use of the MARUTI system for implementing the DR-based planning model.

3 The MARUTI Operating System

The MARUTI operating system is designed to support hard real-time applications on a variety of distributed systems while providing fault tolerant operation[14, 5, 15]. Its object oriented design provides a communication mechanism that allows transparent use of the resources of a distributed system. Fault tolerance is provided through a consistent set of mechanisms which support a number of policies. Most importantly, MARUTI supports guaranteed-service scheduling, in which jobs that

are accepted by the system are guaranteed to meet the time constraints of the computation requests under the requested degree of fault tolerance. As a consequence, MARUTI applications can be executed in a predictable and deterministic fashion.

The development of hard real-time applications requires that the analyst estimate the resource requirements for all parts of the computation and then make sure that the resources are available in a timely manner to meet the timing constraints. It tends to be a cumbersome process. As a part of MARUTI system a set of tools have been developed which support the hard real-time applications during various phases of their life cycle.

MARUTI is an object-oriented system whose unit entity is an *object*. While the concepts of objects and encapsulation has been used in many systems, incorporating these concepts in a hard real-time system required numerous extensions. Objects in MARUTI consist of two main parts: a control part (or *joint*), and a set of service access points (SAPs), which are entry points for the services offered by an object. A *joint* is an auxiliary data structure associated with every object. Each joint maintains information about the object (e.g., computation time, protection and security information) and its requirements (service and resource requirements). Timing information, also maintained in the joint, is dynamic and includes all the temporal relations among objects. Such information is kept in a *calendar*, a data structure, ordered by time, which contains the name of the services that will be executed and the timing information for each execution.

An application is depicted by a collection of services gathered in a *computation graph* — a directed graph in which the vertices represent services, and the arcs represent the precedence (or other relationships) between vertices, e.g., children are services requested by parents. A *job* is defined in terms of a computation graph. It is submitted to the system by naming the root of the computation graph. A job may have timing and fault tolerance requirements associated with it. In MARUTI a computation graph is associated with each job.

Objects communicate with one another via *semantic links*. These links are called semantic because they also perform some type and range checking in the values they carry. This concept permits implementation of other semantic checks as a part of the implementation of the semantic link. Objects that reside in different sites need *agents* as their representative on a remote site. An agent is responsible for the remote transmission of messages and also for the external data translation of these messages.

There are two types of jobs in MARUTI , namely *real-time* and *non-real-time*. A real-time job is assumed to have a hard deadline and an earliest start time. For non-real-time jobs, no time constraints are specified and, therefore, jobs are executed on the basis of time and resource availability. MARUTI does not consider priorities for real-time jobs and all accepted jobs are treated equally[23]. (Priorities can be easily incorporated, for example, by implementing a scheme for the revocation of jobs or a multi-priority queue.)

MARUTI views the distributed resources as organized in various *name domains*. A name domain contains a mutually exclusive set of resources. This concept is used in the implementation of fault tolerance using replication. In addition, this division of resources is useful for the distribution, load balancing, fault independence, and feasibility analysis of fault tolerant schemes.

MARUTI is organized in three distinct levels, namely the *kernel*, the *supervisor*, and the *application* levels. The kernel is a collection of core-resident server objects. The kernel is the minimum set of servers needed at execution time and is comprised of resource manipulators. The main task of the supervisor level objects is to prepare the jobs for execution by making reservations through the resource manipulators. The functions of the kernel are: dispatching, loading, time and communication. The services provided at the supervisor level include: allocation, schedule verification, binding, login service, and name service.

The object principle and the use of the joints allow each access to an object to be direct, and the binding philosophy of the operating system supports it uniformly. Access to an executing object is an invocation of a particular service of that object. The joint allows many jobs to invoke a particular service concurrently, while providing full access and timing control.

The resources needed for the execution of the applications are reserved through the resource manipulators at the supervisor level, prior to the the start-time of the application. The communication channels, CPU, memory, disks, and all other necessary resources are made available so that contention is ruled out, and timing guarantees can be issued by the system.

The use of joints, and specifically of calendars, allows verification of schedulability, reservation of guaranteed services, and synchronization. Projection mechanisms support propagation of time constraints between different localities. These projections maintain the required event ordering and assure the satisfaction of the timing constraints. Furthermore, these data structures facilitate the management of mutual exclusion and protection mechanisms.

Communication among objects is achieved in a distributed fashion. Related objects at remote sites are linked through local *agents*. Each local agent is responsible for communication between its locality and its corresponding remote service, as well as for representing the remote site in the communication for schedulability verification and reservation ([1, 12, 13]).

Through the use of *semantic* links exceptions and validity tests are reduced after a link is established. The links are created by the binding and loading processes, and the protection mechanisms are activated and authorizations established prior to run-time. This allows direct access afterwards. Semantic links to remote objects are established through their respective agents.

Jobs in MARUTI are invocations of services in executable objects. The requirement of a reactive system[3]

[3] *Reactive systems* are those that accept new jobs while executing already-accepted guaranteed jobs[6]

justifies supporting *real-time* and *non-real-time* execution disciplines. We note that the non-real-time jobs are assumed to be preemptable so that their processing requirements can be satisfied in the time slots which are available between the executions of real-time jobs.

Fault tolerance is an integral part of the design of MARUTI . The joint of each object may implement fault detection, monitoring, local recovery and reporting[17]. Initially, each joint contains a consistency control mechanism to manage alternatives (redundant objects with state information) or replicas (redundant stateless objects). The resource allocation algorithm supports a user-defined level of fault tolerance, where redundancy can be established temporally (execute again) or physically (parallel execution of alternatives).

Space redundancy supports node and link failures using roll-forward recovery techniques. Critical services may be provided by forum and quorum protocols. Primitives are provided to the user to specify the desired level of fault tolerance by replication or retries, alternate services or different modules, voting mechanisms or majority queries, backward and forward recoveries. This flexibility improves the design of resilient systems.

A capability based approach is used for protection and security [16]. This system is completely predefined prior to execution of the jobs. The necessary information for the capabilities are stored in the joint, and the capability itself is furnished by the user.

Jobs in MARUTI are viewed by the system as computation graphs. Tools are provided to assist in the design and verification of application code. To use the primitives and tools developed, a set of language extensions is required. For that reason, a precompiler is used to convert a MARUTI program into standard programming language constructs. The precompiler also automatically generates the joints. While MARUTI supports many different fault tolerance mechanisms, applications can be written without knowledge of the policy used.

The MARUTI project is an attempt to examine new ways of building verifiable systems which provide a comprehensive set of services in support of the requirements of mission critical systems including hard real-time, distributed operation, and fault tolerance. We started out with a new design and one of the goals of the project was to investigate new and different ways of implementing this system. Our approach has been to start by designing the system in a general way so that new concepts introduced can be evaluated systematically. The design incorporates primitive mechanisms but leaves policies to the application builders. One of the results of this work has been the first comprehensive formulation of the theory of Time-driven systems (as opposed to the interrupt driven design of traditional real-time systems). We believe that time driven design is simpler and more easily verifiable while directly supporting real-time operation. Our experience so far has shown that the complexity of the applications is substantially reduced by this approach.

MARUTI has been designed as a platform for the study of real-time systems. Its characteristics facilitate the development and testing of many different classes of applications. We organized this project to allow us the greatest flexibility in using MARUTI as a research tool. Our first implementation is a prototype on top of *Unix* , running in a distributed environment. This has allowed us to port MARUTI to a number of platforms and upgrade to faster and better machines as they become available. The current prototype has been designed mainly for functional verification and ease of modification[18].

4 Real-time AI

For the planning system described in Section 2 to be applicable to complex problems a large amount of runtime support for real-time computing is necessary. The monitors, viewed as concurrently occuring entities, must be implemented in one of two ways: either a massively parallel MIMD architecture must be provided, a prohibitively expensive option for the simple computations made by the monitors, or a real-time scheduling system must be used to allow the monitors to access shared computational resources (serial or parallel, single or distributed). In the past year, we have been performing experiments focusing on the use of the facilities of MARUTI to support the scheduling needs of the planning system.

The first experiment tried in the integration of MARUTI and the planning system was the development of a simple real-time control demo aimed at exploring the feasibility of implementing the monitoring capabilities (crucial to the planning system) using MARUTI objects. A simple "line tracker" was developed, using the monitors of the AI approach. This example was then implemented using MARUTI code and run on the MARUTI operating system. This task had two goals: first, we wished to find out how difficult it would be to write monitoring code (formerly developed in Lisp) in the augmented C provided by MARUTI, and secondly we wished to make sure that the MARUTI scheduler could handle the requirements of the monitoring tasks.

The goal of the tracker was to follow the position of a rapidly moving line. The position of the line was represented by its X coordinate and it changed with time which was represented as the Y coordinate. Therefore, at time 50 if the position of the line was 12, and the tracker was at 10 it should increase its value for the next iteration. On the other hand, if its value was 15 then it should decrease the value.

The system was implemented using three monitors: Left-monitor, Right-monitor, and Tracker. Tracker would broadcast the current X position to the other monitors which would compare that X to the observed line value (globally available). If the line was to the Left of the current value, Left-monitor would require tracker to decrement the X value. If the line was to the right of the current value, right-monitor would require an increment.

We were pleasantly surprised by the results of this effort. The code was easily developed using the *objects* and *services* provided by MARUTI. Further, the line tracker worked as predicted, with no augmentations needed to the current MARUTI system. The tracker, using asynchronous scheduling of the three monitors and a separate process for moving the line (thus making the tracking

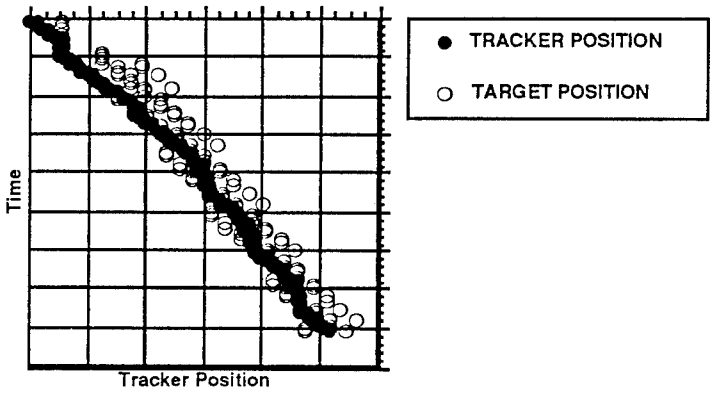


Figure 2: The line tracker: results

task less deterministic) converged on correct X values and kept the tracker within a small bounded interval of the rapidly moving line. Figure 2 shows part of the output of a standard run. The Y dimension represents time, the filled circles represent the position of the tracker at a given point in time, while the empty cirles represent the position of the line. As can be seen, the line is moving rapidly with respect to the speed of the tracker. Still, the tracker is able to stay close to the line and make appropriate adjustments as necessary.

To examine the capabilities of using MARUTI to integrate monitoring with some (simple) rule-based planning (using the abstraction technique described in Section 2), we have implemented a demonstration of the use of the MARUTI -based planning system as a more complex controller. We have developed a simple "pursuer/evader" scenario, in which a "plane," controlled in two dimensions by the planning system, must capture an "evader" which is separately controlled by a user. In addition, a set of simple "rules of engagement" are used by the system to determine behavior. (A schematic view of the system, along with the rules of engagement are shown in figure 3).

In the implemented scenario, the pursuer must decide how to update its course and speed based on a combination of the actions taken by the evader (which are detected via monitoring information extracted from a simulated sensing capability) and a user controlled "mode" over which the pursuer has no control. The system reaches a decision as to whether to "track" the evader (converging on a position near the evader and then setting course and speed to be equivalent to the evader's) or whether to "attack" (setting course and speed so as to converge on the actual location of the evader). The system must, of course, be able to switch modes quickly

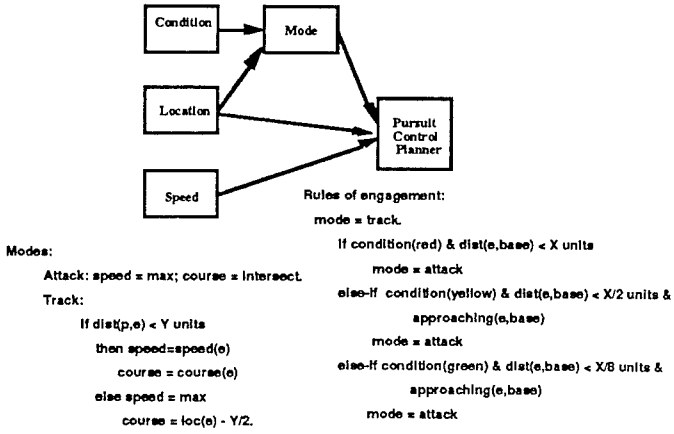


Figure 3: Pursuer/Evader

if the externally generated condition changes (say from "red' to "yellow") or if the evader changes direction. In experiments, the system has reliably been able to pursue the appropriate course of action and to make appropriate decisions.

This demo, although still relatively simple, requires the primary capabilities for mission critical planning (described in Section 1):

1. *Real-time response.* The pursuer must monitor the evader and the world condition both rapidly and often enough to guarantee a response before it can be outmaneuvered.

2. *Context sensitivity.* The appropriate actions to be taken by the pursuer are contingent not only upon the monitored actions of the evader, but also based on the current mode (attack or track) and the externally dictated condition.

3. *Planning or rule-based reasoning coupled with reaction.* The pursuit control planner, although simple, makes decisions as to how to set the speed and direction of the pursuer based on the rules of engagement coupled with the mode. The system must both react to the evader and make appropriate attack/track decisions and the corresponding course adjustments.

We are currently focusing on extending these capabilities, automating the provision of scheduling information to the real-time operating system, generating more flexible real-time scheduling (to meet the sometimes changing demands of the AI system), and demonstrating all of this in a more realistic demonstration of mission critical planning.

5 Current Research Directions

The use of the MARUTI-based AI planning system to perform mission critical emergency response in a more realistic domain requires new capabilities for both the AI and real-time components of the system.

5.1 AI Requirements

The scheduling requirements of the monitoring and planning components of the AI system are currently generated by hand. Thus, although a description of the necessary monitors and their interactions with the planners are developed in the original planning, their scheduling needs are currently computed separately and then added to the code in an appropriate manner. It is clear that for such a system to scale up to considerably larger tasks, the scheduling requirements must also be generated by the overall planning system.

We are currently working on an approach to this problem based on a formula for maximum reaction delay developed by Georgeff and Ingrand [4]. This formula, $\Delta R = p/1 - nt$ is used to represent the maximum amount of time that a system can take in reasoning about changes in the world without being forced to ignore events. We are currently exploring the use of this formula in another way, to determine the critical intervals at which the world needs to be sampled given n events. Thus, a monitor which takes X_m time to execute its primary condition checking must be scheduled at intervals of ΔR with a time of X_m. The X_m's can be precomputed for each class of monitor, while the values of ΔR are computed based on the expected number of certain types of events in the environment, which can be computed at planning time (with a known probability). Thus, these numbers serve as a reasonable heuristic for generating the scheduling deadlines needed by the MARUTI system (i.e. the time and frequency requirements of each of the objects).

This approach to computing the scheduling deadlines also dovetails neatly with the abstraction-based approach described in section 2. In such systems, the change in the world at higher levels of abstraction (such as a projected missing of some deadline in the path planning system), can be expected to happen significantly less frequently than actions at the lowest level (such as the change in the position of moving objects in the world). Thus, the scheduling deadlines generated by this approach will be less frequent for the higher level objects (which also take longer; Note the correspondence with Figure 1).

In addition to this applied work, we are currently working on extensions to a formalization of the abstraction work which focuses on the recognition of information which significantly violates expectations projected at run-time. We have currently formalized the monitoring tasks in a temporal logic which allows the propagation of information between levels using a notion of abstract logics. We are now working towards extending this approach to handle non-monotonic abstract logics and a localized control of inconsistencies arising during monitoring [22].

5.2 Real-Time Requirements

To support the sort of complex reasoning required by the mission critical AI planning demonstration described in this paper, the MARUTI system needs to be extended to handle more flexible scheduling. While a dynamic scheduler that could guarantee all requests would be met would be ideal, such a system cannot be built. Instead, either a guaranteed request scheduler *or* a more dynamic scheduler can currently be implemented. As the guaranteed scheduling is crucial to mission critical computing, we are currently exploring ways to provide more flexibility without compromising this important feature of the system.

One approach we are exploring is the use of non-real-time jobs (those scheduled by MARUTI as time permits) to augment the capabilities of the system. Thus, tasks which are determined to be useful, but not essential, can be scheduled as non-real-time jobs. These tasks include both AI related events (such as minor variations on the plan introduced for optimization) as well as information gathering and record keeping tasks executed by the real-time system to provide feedback for the future scheduling of similar tasks, or to provide a higher level of fault tolerance. (This work is somewhat similar to that of Durfee [3] which has examined the use of a distributed real-time system to support AI applications. Durfee has examined the use of "dummy" events, which are scheduled by the real-time system but which only execute if time allows.)

A more ambitious approach is the use of a context-switching like approach, which allows a set of schedules to be predetermined and then switched in response to information gained at run-time. In MARUTI this facility is provided as scenario facility in which jobs run in a set of predefined scenarios. Such a capability would allow for the AI system to recognize specific situations and literally change the mode of execution to react to them. This approach dovetails neatly with the use of cached plans, a technique currently used in the AI community to improve the efficiency of reaction in planning systems (cf. [20]).

In addition to extensions to MARUTI to support the AI work, the operating system is being improved in numerous other ways. These include:

1. Support of transparent heterogeneous computing,
2. Support of multiprocessors at a computing node,
3. Native kernel development,
4. Extended program development support environment.

6 Conclusions

In this paper we have discussed preliminary work in the implementation of an AI planning/reaction system

which can run on the MARUTI hard real-time operating system. In addition to reviewing both the AI and real-time technologies, we have performed several simple demonstrations showing the feasibility of using these techniques for mission-critical planning systems – those which must guarantee real-time response in reacting to complex situations. We also describe some of the challenges which such guaranteed-response AI systems pose for both the AI and the real-time computing communities. Current efforts focus both on a continuing effort to implement the extensions to both parts of this system, and on the development of more complex scenarios for testing these ideas.

References

[1] Agrawala A. K. and Levi S.-T., "Objects Architecture for Real-Time, Distributed, Fault Tolerant Operating Systems," IEEE Workshop on Real-Time Operating Systems, Cambridge, MA, July 1987.

[2] Chapman, D. (1985) "Nonlinear Planning: a Rigorous Reconstruction", In Proc. of IJCAI-85. pp. 1022-1024.

[3] Durfee, E. "Towards Intelligent Real-time Control Systems," In Hendler, J. (ed.) Working Notes of the AAAI Spring Symposium on Planning in uncertain, unpredictable, or changing environments. Technical Report, Systems Research Center, University of Maryland (forthcoming, 4/90).

[4] Ingrand, F. (i) Personal Communication, 5/89 (ii) Unpublished manuscript, 1/90.

[5] Gudmundsson Ó, Mossé D., Ko Keng-Tai, Agrawala A. K. and Tripathi S.K.
"MARUTI an Environment for Hard Real-Time Applications" CS-TR-2328, Department of Computer Science, University of Maryland, College Park, Maryland, Nov 1989. To appear in Proceedings of the first workshop on Mission Critical Systems. ACM Press

[6] Harel D. and Pnueli A., "On The Development of Reactive Systems," Weitzman Institute of Science, Rehovot, Israel, 1985.

[7] Hendler, J.A."Real Time Planning", *Proceedings AAAI Spring Symposium on Planning and Search*, Stanford, CA. Mar. 1989.

[8] Hendler, J. "Abstraction and Reaction", Proc. AAAI Workshop on Perception, Planning, and Knowledge, Detroit, MI, Aug. 1989

[9] Hendler, J. and Sanborn, J. 'Planning and Reaction in Dynamic Domains" *Proceedings DARPA Workshop on Planning,* Austin, Tx., Dec. 1987.

[10] Hendler, J. and Subramanian, V.S. "A formal model of abstraction for planning," Submitted to *AI Journal*, 4/90.

[11] Hendler, J., Tate, A., and Drummond, M. AI Planning: Systems and Techniques, *AI Magazine* Summer, 1990 (to appear).

[12] Levi S.-T. and Agrawala A. K., "Objects Architecture: A Comprehensive Design Approach for Real-Time, Distributed, Fault-Tolerant, Reactive Operating Systems," CS-TR-1915, Technical Report, Department of Computer Science, University of Maryland, College Park, Maryland, Sept., 1987.

[13] Levi S.-T. and Agrawala A. K., "Temporal Relations and Structures in Real-Time Operating Systems," CS-TR-1954, Technical Report, Department of Computer Science, University of Maryland, College Park, Maryland, Dec., 1987.

[14] Levi S.-T., Tripathi S.K., Carson, S.D., Agrawala, A.K. "The MARUTI Hard Real-Time Operating System" ACM Operating System Review, June 1989 Vol 23 No 3

[15] Levi S.-T., and Agrawala A. K. "Real-Time System Design" McGraw Hill Publishing Co, New York, 1990.

[16] Levy H, "Capability Based Computer Systems" Digital Press, 1984.

[17] Mossé D. and Agrawala A. K, "On Fault Tolerance in Real Time Environments" University of Maryland, March 1990 (under preparation).

[18] Mossé D., Gudmundsson Ó. and Agrawala A. K, "Prototyping Real Time Operating Systems: a Case Study," University of Maryland, March 1990 (under preparation).

[19] Sanborn, J. and Hendler, J. (1988) " Monitoring and Reacting: Planning in Dynamic Domains", *International Journal of AI and Engineering,* Volume 3(*2*), April, 1988. *p. 95*

[20] Schoppers, M. "In defense of reaction plans as caches", *AI Magazine*, 10(4), 1989.

[21] Spector, L. and Hendler, J. (1990) "An Abstraction-Partitioned Model for Reactive Planning" Submitted to the Rocky Mountain Conference on Pragmatics in AI.

[22] Subramanian, V.S. and Hendler, J.A. "An abstraction-based approach to automated reasoning" (Manuscript in preparation).

[23] Yuan X and Agrawala A. K, "A Decompostion Approach to Nonpremptive Real-Time Scheduling" CS-TR-2345 Department of Computer Science, University of Maryland, College Park, Maryland, Nov 1989.

Responding to Environmental Change*

Adele E. Howe, Paul R. Cohen
Experimental Knowledge Systems Laboratory
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

Responding to environmental change is perhaps the most difficult aspect of designing agents to plan and act in complex dynamic environments. In this paper, we analyze the environmental response requirements of such an environment, provided by the Phoenix forest fire fighting simulator, and describe three mechanisms that together address the demands of that environment. The limitations on response imposed by the environment are described in terms of "windows of response opportunity". This framework matches the demands of the different types of environmental change occuring in the environment to the abilities and limitations of the mechanisms intended to address change. As applied in the Phoenix environment, it motivates the design of three different mechanisms that address three different types of change. The three mechanisms, reflexes, lazy skeletal expansion, and responsive adaptation, are described in detail, and their interaction with the environment is illustrated in an example.

1 Introduction

Environmental conditions change. In the most advantageous cases, the change is caused by the efforts of agents working to control aspects of the environment. In the least, the changes are caused by environmental forces, which may or may not be predictable. Whether the environmental change is detrimental to an agent depends upon whether the agent's actions, both thinking and effecting, are *responsive* to those changes.

Responsiveness requires timely appropriate action in response to environmental change. When a truck is driving toward you in your lane, *timely* is immediate and *appropriate* is a simple evasive action. When a hurricane is due to arrive tomorrow, *timely* is within the next day and *appropriate* is a complex combination of actions intended to protect home and family. Different types of environmental changes require different types of responses. An agent must take actions appropriate to and within the time frame of the types of events in its environment. In this paper, we describe how agents in the Phoenix system are responsive to changes in their environment.

*This research was supported by DARPA-AFOSR contract F49620-89-C-00113 and the Office of Naval Research, under a University Research Initiative grant, ONR N00014-86-K-0764. We wish to thank members of the Phoenix group, especially Michael Greenberg, David Hart, David Westbrook, Paul Silvey, Scott Anderson, and Evan Smith, for their contributions to the design and implementation of the Phoenix system. We also wish to thank Carol Broverman and Ross Beveridge for their comments on several drafts of the paper.

2 Responding to Change in Phoenix

Agents in Phoenix work to contain simulated forest fires in Yellowstone National Park[4]. Fires are contained by removing fuel from their paths, causing them to burn out. This process, called building fireline, requires the coordination of several field agents to surround the fire with fireline. One agent, the fireboss, coordinates the activities of semi-autonomous field agents, bulldozers, to build fireline at many points around the fire. Fire spread is influenced by many environmental factors: wind speed, wind direction, terrain cover, elevation gradient, and moisture content. The fire's overall shape and spread is determined by these factors, but so many minute factors are involved in the precise spread of the fire that it is not possible to predict the exact time at which a particular point on the map will catch fire.

2.1 Response requirements for the Phoenix environment

The primary constraint on responsiveness is time. For any environmental change, there is a "window of response opportunity", the time during which the agent can respond. In the example of the truck in the wrong lane, the window is very narrow and once the window has passed, it is simply too late to act.

The window starts when an environmental change is perceived (T_{wb}). It ends when the effects of that change occur (T_{we}). These two points define the window, the time delay between perception of a change, real or impending, and its effect (shown in Figure 1). For certain classes of environmental change, the delay will be short (e.g., the truck); for some, it will be longer (e.g., the hurricane). However, environmental forces are not the only influence on window size. Actions often require some start up time or overhead between their initiation and their effects; for example, the time delay between deciding to put on the car brakes and the car's stop is significant enough to require a safe following distance. So,

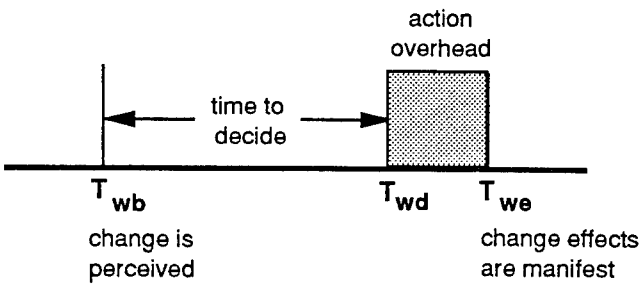


Figure 1: Environmental Change Window of Response

response overhead (T_{wd}) can narrow the window. The window that remains is the amount of time left to decide how to act.

The Phoenix environment includes two qualitatively different kinds of environmental change distinguished by their response windows. Narrow response windows are common when environmental change is unpredictable. For example, when agents work close to the fire, rapid response is necessary if the fire suddenly threatens to engulf them. Wider response windows are common when environmental change is gradual. For example, it takes simulated hours or days to surround even small fires which allows at least hours to make strategic decisions about containment.

Because agents must respond to both types of change, they need mechanisms to address both. A narrow window suggests rapid decision making (to minimize the decision time) and simple action (to minimize the response overhead). If the window is too narrow, it precludes deliberation and complex responses, but encourages reactivity. A wide window affords time to deliberate over the best response. Reactive and deliberative approaches have different fundamental limitations: reactive approaches guarantee response within fixed time bounds and so cannot use additional time even if it is available; deliberative approaches cannot guarantee response within short fixed time, but can exploit additional decision time. The fundamental differences in time usage for the two approaches conflict, which precludes incorporating them both in the same mechanism. Thus, Phoenix agents need two separate mechanisms: a reactive mechanism, which we call *reflexes* and a deliberative mechanism, which we call *lazy skeletal expansion*.

Together, reflexes and lazy skeletal expansion form a two-layer response system. As in Brooks' subsumption architecture [3], each layer provides a particular level of competence. Reflexes address change that occurs faster than lazy skeletal expansion can respond to it; lazy skeletal expansion coordinates actions and avoids detrimental plan interactions.

As discussed, the response window is defined by the time delay between when a change is perceived and when its effects are felt. The response window assumes that the agent *notices* the change at the moment that the window opens and immediately starts formulating a response. However, because agents may have already committed to other actions (e.g., attending to different fires), additional time may pass

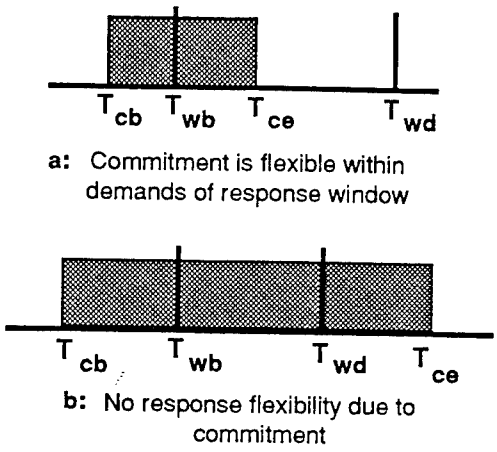


Figure 2: Environmental Change Window Compared to Response Flexibility

between when the environmental response window opens and the agent notices it. Lazy skeletal expansion attempts to minimize the additional time by deferring commitment, as much as possible, to a precise course of action. When necessary commitments have a duration less than the response window, environmental change can be addressed by lazy skeletal expansion. Figure 2, part a, displays this relationship, with T_{cb} indicating the commitment beginning and T_{ce} indicating commitment end; note that the commitment ends before the response window does. When the window has shorter duration than the commitment, then the planner may commit to a course of action that may be rendered impossible by the environment (as shown in Figure 2, part b), thus, resulting in plan failure. Because it isn't always possible to defer commitment, another mechanism is required to adapt the commitment structure (i.e., plans) in response to the detrimental environmental change. We call this mechanism *responsive adaptation* because it adapts plans in progress in response to detrimental environmental change.

All agents, fireboss and field agents, share a common agent architecture that includes these response mechanisms. That architecture consists of four basic components: sensors, effectors, reflexes and a cognitive component. Sensors perceive the state of the environment. Effectors take physical action for the agent in the environment. Reflexes change the settings of effectors to respond within a narrow window. The cognitive component is responsible for tasks related to deliberative response, action coordination and resource management. The cognitive component includes both lazy skeletal expansion and responsive adaptation. In response to environmental conditions, it selects plans from the plan library and adds them to a partially ordered agenda structure, called the *timeline*, for later execution (this process is part of lazy skeletal expansion and will be explained in more detail in Section 2.3). The information and control relationships among

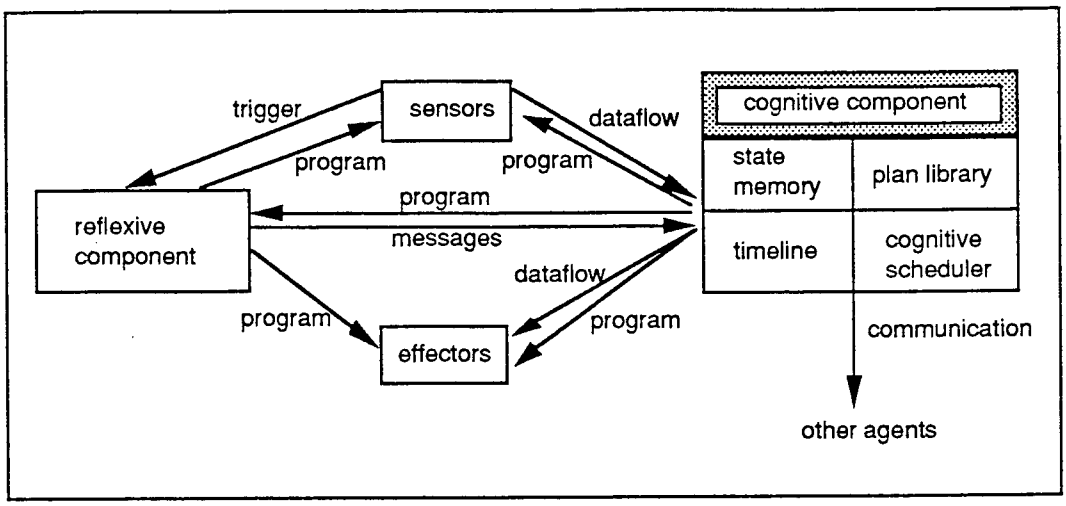


Figure 3: Phoenix Agent Architecture

these components are displayed in Figure 3.

The three mechanisms: reflexes, lazy skeletal expansion and responsive adaptation, account for the range of response that are demanded by the Phoenix environment. Reflexes change what the agent is currently doing. Lazy skeletal expansion changes what the agent is planning to do. Responsive adaptation changes how the agent was planning to do something. The remainder of this section will describe, in detail, the three mechanisms that together ensure responsiveness in Phoenix.

2.2 Reflexes

Reflexes provide time-bounded responses to critical situations. They compensate for the time delay between when a situation occurs and when the cognitive component can notice and deal with it. As such, they constitute an architectural component specialized for rapid, simple response.

Reflexes are associations between sensors and effectors mediated by two simple functions, one for triggering reflex execution based on sensor readings (the trigger function) and one for changing effector settings based on those sensor readings (the response function). Trigger functions are simple functions of the sensor readings, such as whether the value exceeds a threshold or equals some value. Response functions make simple changes to effectors, turning them off or on or making minor parameter adjustments. These functions rely only on currently available sensor readings; reflexes retain no persistent state information other than the values of the operating parameters. These parameters define reflex sensitivity and are set by the cognitive component.

Reflexes are activated in tandem with the sensors and effectors. After a sensor executes, trigger functions, which link the sensor to particular reflexes, are executed to determine whether the associated reflexes should be activated. The trigger function may rely on values from more than one sensor; in which case, it simply checks the most recent readings for the critical sensors. When activated, the reflex executes the response function to change effector settings. For example, when the *sense-road-heading* sensor has a value different than the *sense-agent-heading* sensor, the *follow-road* reflex changes the *heading* parameter of the movement effector to maintain the road heading.

The time required for response by the reflexes is bounded by the activation rate of the sensors. Reflexes are executed immediately after the sensors and require little time to execute. Thus, the agent can respond as quickly as its sensors can notice the environmental change, which reduces the response time from the cycle time of five simulation minutes required by the cognitive component, to seconds of simulation time. This approach can be contrasted with that adopted in PRS[7]. PRS includes all responses (called KAs) in the same framework and relies on the most crucial of them being quickly selected and executed. That is, it relies on fast KAs for reacting to crucial situations. This produces a guaranteed "reactivity delay" of $s + t$ where s is the maximum time to determine the KAs applicability and t is the cycle time (as dictated by the maximum time required to execute a primitive action).

Reflexes respond to environmental changes that have narrow response windows. Consequently, imminent disasters are prevented, but sometimes, at a cost of temporarily stopping progress in a plan. As a result, when reflexes change the settings of an effector, superseding those of the cognitive component, the reflexive component sends a message to the cognitive component warning it of the change. This interaction is described in further detail in Section 2.4.

2.3 Lazy Skeletal Expansion

Lazy skeletal expansion responds to environmental change characterized by wide response windows. Deferring commitment to a precise course of action maximizes the opportunity for change to influence decisions. Skeletal plans provide a structure in which to base action decisions, which expedites action coordination and minimizes interactions.

Deferred commitment is accomplished by interleaving three basic activities: find, expand, and execute. *Find plan* actions are placed on the timeline as part of plans (to defer commitment to a particular plan or action) or in response to exceptional conditions (as noted by messages from the reflexive component or from other agents). These actions use their context within the timeline to search the plan library for skeletal plans appropriate for the context and the cur-

rent state of the world. For example, if the find plan action is to get an agent to the fire, the context includes information about the type of fire fighting plan it is part of, the techniques being used to fight the fire, and the coordination needed with other agents, in addition to the location of the fire. *Expand plan* actions instantiate the plan's network of actions on the timeline. *Execute* actions calculate variable values, manage resources, and control the agent's interactions with the world. As the timeline actions are executed, plans and actions become incrementally added, sensors and effectors are activated, and the agent pursues the plan. Actions become eligible for execution only when the siblings that precede them have already executed; even then, execution may be deferred until information about the state of the world is available. Because plans are combinations of primitive actions and plan expansion actions, this leads to interleaving of action and planning.

Skeletal plans have four parts: applicability conditions, resource requirements, execution methods, and an action network. Applicability conditions describe the conditions under which the action is appropriate. Resource requirements describe the expected time, information (as represented by variables), and physical resource needs of the action. Execution methods are the procedures for executing the action, i.e., what gets called when the action is chosen for execution. An action network is a network of problem solving actions associated by temporal and data dependencies. Figure 4 shows a "generic" Phoenix plan as it might become expanded on the timeline. Vertical lines indicate that the higher action placed the lower action on the timeline. Horizontal lines are temporal relations between actions. The boxes under the action further describe some of its characteristics.

Skeletal planning has previously been applied in domains in which planning and acting are completely separate, cancer therapy advice [13] and experiment design [6]. While the goals and implementation of these other skeletal planning projects are rather different, the structure of the planning is much the same. Lazy skeletal expansion is similar to the planning method employed in PRS[7]. KAs, the representation for procedural knowledge, include an invocation condition, which specifies when the KA is useful, and a body, which describes the sequence of subgoals which constitute the procedure. Thus, the structure of PRS is similar, but the control is distinct. At each execution cycle, PRS checks all KAs for applicability, selecting one for execution. In Phoenix, only actions on the timeline are considered for execution. Thus, PRS provides for more reactive planning, but at the expense of being unable to allocate time and schedule actions beyond an execution cycle. The timeline structure in Phoenix was designed to support real-time scheduling of actions[11], but does complicate plan modification.

2.4 Responsive Adaptation

When lazy skeletal expansion overcommits to a course of action and crosses over an environmental response window, plan failures can occur. Failures occur when the plan either cannot continue or cannot succeed if it does continue. In effect, the environmental change response window conflicts with plan commitment. In Phoenix, this mismatch is a result of: non-local interactions, uncertain or obsolete information, unpredicted changes, and novel situations.

Non-local interactions occur when the agent attempts to respond, in parallel, to different environmental changes and so overcommits its resources. For example, the fireboss treats each fire in the environment as a separate situation, making decisions about containment largely independently; yet, the resources for controlling the fires are fixed and must be shared between the situations. Consequently, decisions about changing resource allocation to a particular fire impact and may thwart resource expectations for plans in progress on a different fire.

Uncertain information causes failure when the agent is forced to commit resources without being certain of the magnitude of the need. For example, fires that appear at the periphery of view may look small to a watchtower, but may actually be conflagrations. Without better information, the fireboss must take an "educated guess" at the real situation and commit field agents to contain it, accepting the possibility that more or fewer agents may actually be needed.

Unpredicted changes naturally produce failure. Fire fighting involves working in constrained situations that are vulnerable to unexpected changes. If the wind changes unexpectedly, a previously safe area in which to build fireline may become dangerous.

Finally, novel situations require commitment to plans that may not actually be best for the situation. When environmental change results in a novel situation, the agent may not know to respond.

The responsive adaptation mechanism responds to environmental changes by adapting plans in progress. Plans provide the structure for controlling action coordination, undesirable plan interactions and resource use. Thus, the expectations included in the plan structure should be preserved, while still addressing the changes in the environment; so, responsive adaptation should change the intended plan by the minimum required for the agent to resume acting. It should make the changes as quickly as possible because computation time is itself a resource and because the response window may have been closed before the response is computed.

Because the process of adapting plans is an action within the context of the plan, it should be accessible to other problem solving mechanisms that direct and constrain the agent's actions, such as resource allocation. Because responsive adaptation is activated when exceptional conditions occur, it should provide broad coverage of possible situations; it is the mechanism of last resort. Georgeff etal in [7] describe examples of bizarre behavior in PRS that results from "mis-applying" actions to novel situations. Without a general replanning capability, an agent repeatedly performs the same inappropriate behavior.

2.4.1 Detecting Failures

The agent detects failures when it cannot successfully conclude an action or plan. Three mechanisms signal failures: execution errors, reflexes, and envelopes. Execution errors occur when an action cannot execute to completion because the state of the world does not match the assumptions, information is not yet available, or, for some problem solving actions, no solution exists. When the agent encounters a dangerous environmental condition, reflexes change effector programming to remove or at least reduce the danger. If this change of programming conflicts with the programming

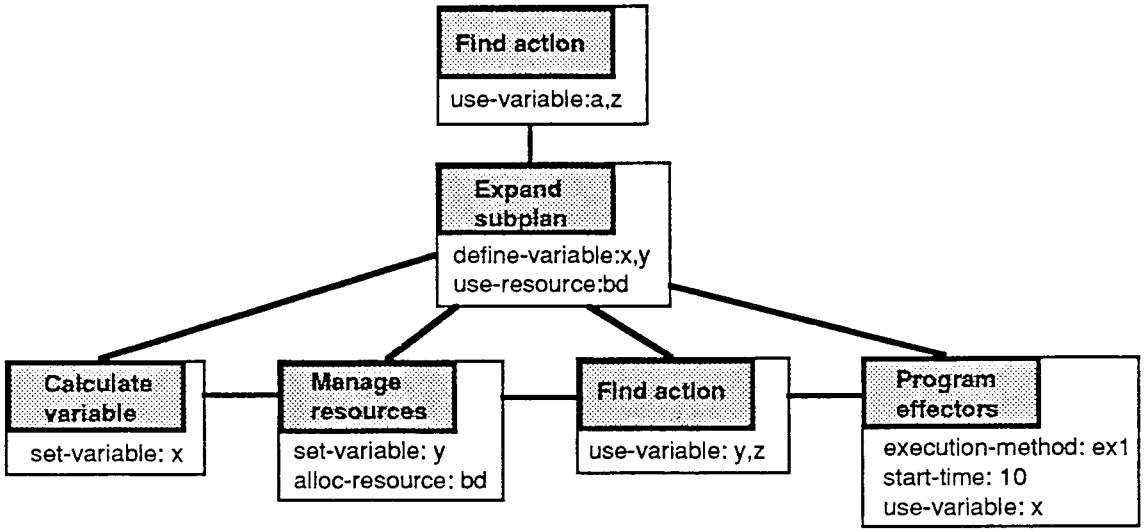


Figure 4: "Generic" Phoenix plan as represented on the timeline with information about inter-dependencies

previously set as part of a plan action, then the reflex signals a failure.

Both execution errors and reflexes signal obvious failure in that the plan is actually prevented from executing. Determining whether an on-going action can *ever* succeed requires active monitoring of the progress of the action. *Envelopes* detect impending failures [8; 12]. They perform sophisticated monitoring of the plan's progress in the world, integrating the efforts of many agents, to determine whether the plan can complete within its environmental and resource limitations. If a plan will be unable to complete successfully under the present conditions, the performance envelope is violated and an impending failure is signaled.

2.4.2 Responding to failure

The detection mechanisms signal failures by adding actions that find recovery plans to the timeline, placed in parallel with the action that initiated them. These actions include readily available information about the failure trigger, the agents involved, and the error type. Recovery actions are structured like other planning actions in that they have a context within other plans; they reference variables and information available in that context; they are scheduled like other actions; and they employ the same planning method–lazy skeletal expansion–for deciding on response. As a type of planning action, adaptation can be smoothly integrated into the planning process. As a timeline action, adaptation has access to the same memory structures and is subject to the same resource management techniques as are other timeline actions.

Responsive adaptation in Phoenix searches the plan library of general recovery plans for one appropriate to the failure. These plans are represented in the same action description language as the domain plans and so are interpreted by the standard execution methods. The plan structure as it is represented on the timeline (and displayed in Figure 4) defines a context or locality for action, indicates dependencies between actions and distinguishes decision points. These structural characteristics provide backtracking points for adapting the plan. Decision points are actions in the plan structure that rely on environmental context to direct their execution. Any action that binds variables or calculates variable values based on context is a decision point. Actions that select other plans for execution use environmental context to determine applicability, and so are certainly decision points. The following recovery plans use the plan structure to identify decision points and dependencies between the decision points and the failure point to support recovery:

- Wait and then re-execute the failed action.
- Reinstantiate the action, updating values for its variables.
- Select an alternative execution method for the failed action. Some execution methods sacrifice accuracy for computation speed; sometimes, the accuracy is necessary.
- Re-calculate values for variables used in the action. Some variables are calculated by actions that were executed earlier in the plan; this action involves re-executing a previous action that set a variable used in the failed action.
- Substitute other variables of the same type as those used in the action.
- Substitute a similar plan step for the failed action that produces approximately the same effect in the environment.
- Allocate additional resources to the plan.
- Re-execute the parent plan selection action, i.e., re-plan.

The recovery plans make mostly simple repairs to the structure of the evolving plan. As such, they can be used in different situations, do not require expensive explanation, and ensure full coverage of all possible failures–the agent must be able to do something. SIPE [14] and IPEM [2] rely on a similar strategy of replanning by general plan repair methods. This strategy sacrifices efficiency for generality and so results in a planner capable of responding to any failure, but perhaps in a less than optimal manner.

Like any plan in Phoenix, recovery plans have applicability conditions to guide their selection. When a failure is

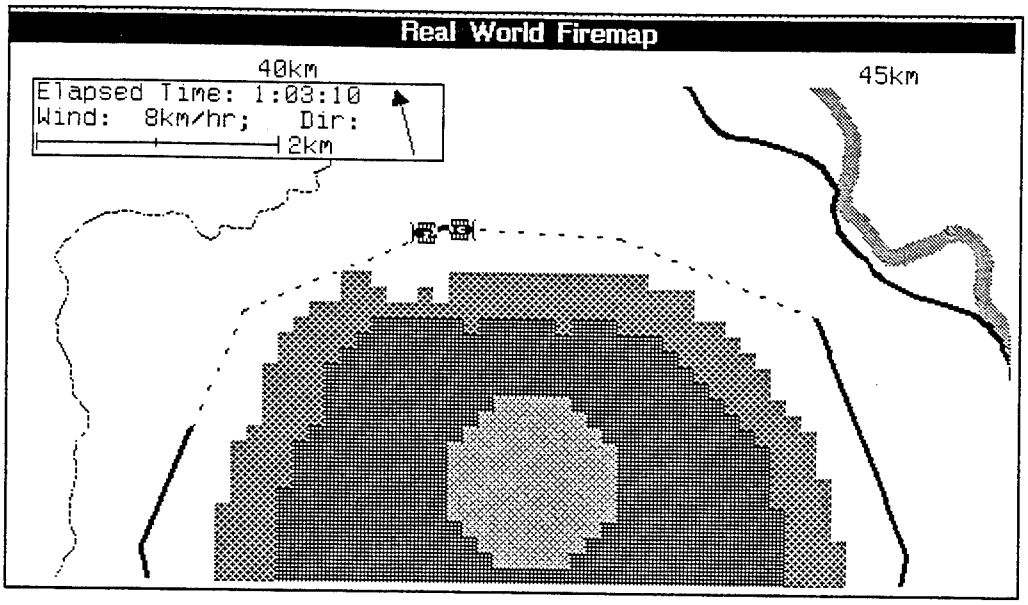


Figure 5: Bulldozer building fireline in indirect attack plan

recognized, the find-plan method searches the recovery plans for the most appropriate and adds it to the timeline at the current location for execution. Because these responses are general, they may not always work. Failure of a recovery plan works much the same as any failure, except that rather than repair the recovery plan, the system will select another method to repair the original failure. Thus, responsive adaptation should always have some response to environmental change that has eluded the other mechanisms.

3 An Example from Phoenix

Fires are fought by building barriers to prevent further spread. Building fireline close to the edge of the fire minimizes the loss of forest, but maximizes the agent's vulnerability to the fire. A more conservative strategy is to fight fires with indirect attack, which involves predicting the likely spread of the fire and building line around it so that the field agents can completely surround the fire before it reaches the fireline.

The indirect attack plan determines where to build fireline to contain the fire at some designated point in the future, allocates the field agents needed to do so, and sends the field agents out to build their first segments of fireline. Relying then on lazy expansion, the plan is expanded further in response to messages from the field agents on their status, e.g., if they've encountered problems, need fuel, or have finished their last assignments. Only when information about field agents' status is known does the planner commit the agents to a course of action and so further expand the plan. In this way, the planner can be responsive to environmental changes that impact what the agents should be doing and exactly how they should be doing it.

Lazy skeletal expansion may keep agents from being assigned to work in a currently dangerous area, but it will not necessarily keep them from danger if the situation changes. If the wind has shifted or the fire spread more rapidly than predicted, the fire may cross the assigned fireline segment. Figure 5 shows a bulldozer (on the left) building a fireline segment (shown as a dotted line) which has been crossed by fire (shaded grey area south of the bulldozers). When sensors detect fire in the bulldozer's path, they trigger an *emergency-stop* reflex. This reflex programs the *movement* effector to stop, thus re-setting the programming installed as part of the plan. The reflex sends a message to the cognitive component which causes a failure signaling action to be added to the timeline, registering the emergency stop.

When the failure action is noted and executed, the planner searches for an appropriate response. Several recovery responses are possible: the plan variable for the fireline segment could be re-calculated, the build line action could be replaced by another type of build line action, or the parent plan could be replaced by another. In this case, the agent chooses to re-calculate the fireline segment variable because it is the cheapest action that is applicable. In Figure 6, the bulldozer has executed this recovery plan and completed building the fireline while still avoiding the fire. If this action had not worked because the fire had engulfed the endpoint of the fireline segment, the bulldozer could replace the original *build line on path* action with an alternative action, *build line parallel to fire* action that would change its behavior to direct attack, or it could have selected a new plan entirely.

4 Understanding Responsiveness in Phoenix

The three responsive mechanisms in Phoenix cover the range of environmental change that an agent will encounter in this environment. These mechanisms work because they are designed to address the demands and exploit the facilitating

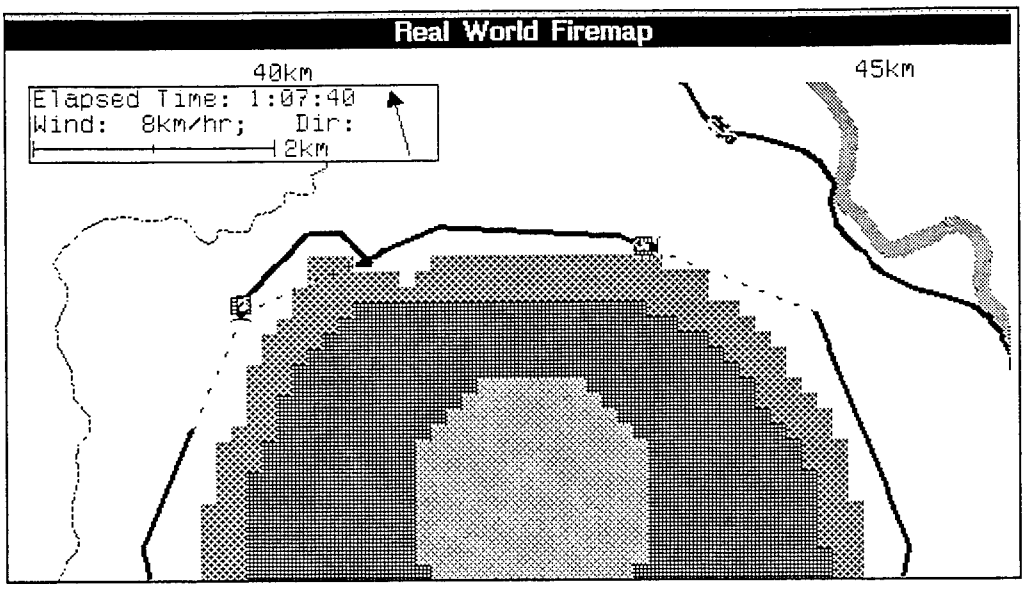


Figure 6: Bulldozer completes building line, albeit not in the originally planned location

characteristics of the environment. Reflexes can rely on simple triggers and make simple responses because rapid catastrophic change is easily recognized and can be averted by simple evasive response. The reactive strategy supported by reflexes has been successful in other systems operating in domains with these demands [1; 5]. Moreover, catastrophic change is relatively rare in the Phoenix environment. Generally, Phoenix provides a forgiving environment. Agents may pursue plans that are not the best and still manage to contain the fire, but at a higher cost. This forgiving nature allows the agent to use skeletal plans in different situations and to rely on general methods for responsive adaptation without risking disaster.

Action in the environment is characterized by stereotypical plans. Most of the basic strategies for fighting forest fires in Phoenix can be represented by a relatively small number of skeletal plans. Additionally, because most actions take place in different geographic locations, interactions are essentially limited to resource contention, which can be accommodated in the plan structure. Thus, we reduced the need for reasoning about plan interaction by using skeletal structures that have "compiled out" that reasoning.

With their responsive mechanisms, agents can respond to any environmental change. Unfortunately, in the current state of development, they cannot always recognize change, and their responses are not always successful. Change is difficult to recognize when its detection depends on something *not* happening or depends on interactions of action and environmental forces. For example, the fireboss must conjecture that a bulldozer agent has become incapacitated when it fails to make contact. The fireboss must predict that a plan will not succeed when the fire starts to expand more quickly than it can be surrounded. Fortunately, research on envelopes [8; 12] will address recognizing detrimental environmental change, as early as possible, in these difficult to detect situations.

Responses are not always successful because the plan library is incomplete. The scope of environmental factors that define situations and the difficulty of anticipating precisely the width of response windows for change in different situations makes it difficult to build a plan library that offers "best" plans for any situations. The agent should learn appropriate plans for itself by experimenting in its environment. A project currently in progress seeks to have the agents do just that [9; 10].

Phoenix provides a rich environment in which to explore notions of responsiveness. It forces agents to confront qualitatively different types of environmental change and respond to them. Understanding the nature of the response demands of the environment is crucial to the design of mechanisms for addressing those demands. The success or failure of agents depends upon the ability of those mechanisms to respond appropriately to their environment.

References

[1] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272, Seattle, Washington, 1987. American Association for Artificial Intelligence.

[2] Jose A. Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Minneapolis, Minnesota, 1988. American Association for Artificial Intelligence.

[3] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1), March 1986.

[4] Paul R. Cohen, Michael Greenberg, David M. Hart, and Adele E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3), Fall 1989.

[5] R. James Firby. An investigation into reactive planning in complex domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 202–206, Seattle, Washington, 1987.

[6] Peter E. Friedland and Yumi Iwasaki. The concept and implementation of skeletal plans. *Journal of Automated Reasoning*, 1:161–208, 1985.

[7] Michael P. Georgeff, Amy Lansky, and Marcel J. Schoppers. Reasoning and planning in dynamic domains: An experiment with a mobile robot. Technical Report Technical Note 380, SRI International, Menlo Park, CA, April 1987.

[8] David M. Hart, Paul R. Cohen, and Scott D. Anderson. Envelopes as a vehicle for improving the efficiency of plan execution. Technical Report 90-21, COINS Dept., University of Massachusetts, 1990.

[9] Adele E. Howe. Adapting planning to complex environments. unpublished PhD Dissertation Proposal, COINS Dept., University of Massachusetts, September 1989.

[10] Adele E. Howe. Integrating adaptation with planning to improve behavior in unpredictable environments. In *Proceedings of AAAI Spring Symposium on Planning in Uncertain, Unpredictable or Changing Environments*, Palo Alto, CA, March 1990.

[11] Adele E. Howe, David M. Hart, and Paul R. Cohen. Addressing real-time constraints in the design of autonomous agents. to appear in *Real-Time Systems*, 1990.

[12] Gerald M. Powell and Paul R. Cohen. Operational planning and monitoring with envelopes. In *Proceedings of the IEEE Fifth AI Systems in Government Conference*, Washington, DC, 1990.

[13] Samson W. Tu, Michael G. Kahn, Mark A. Musen, Jay C. Ferguson, Edward H. Shortliffe, and Lawrence W. Fagan. Episodic skeletal-plan refinement based on temporal data. *Communications of the ACM*, 32(12):1439–1455, December 1989.

[14] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers, Inc., Palo Alto, Ca., 1988.

Planning in Concurrent Domains

Subbarao Kambhampati*
Center for Design Research
and Computer Science Department
rao@cs.stanford.edu

Jay M. Tenenbaum
Center for Integrated Systems
and Computer Science Department
marty@cis.stanford.edu

Stanford University
Stanford, CA 94305-4026

Abstract

In many real-world situations, a planner is part of an integrated problem-solving environment and must operate concurrently with other planners and special purpose inference engines. Unlike the traditional AI planners, planners operating in such concurrent environments have to contend with an evolving problem specification, and should be able to interact and coordinate with the other modules on a continual basis. This in turn poses several critical requirements on the planning methodology. In this paper we identify the ability to incrementally accommodate the changes necessitated by the externally imposed constraints into the existing plans, and the ability to understand and reason about the rationale behind externally imposed constraints at an appropriate level of detail as two crucial requirements for planning in such environments. We then explore directions for extending classical hierarchical planning framework to handle those requirements.

1 Introduction

In many real-world situations, a planner is part of an integrated problem-solving environment and must operate concurrently with other planners and special purpose inference engines (collectively referred to as SDMS in this paper). The SDMS concentrate on different specialized considerations and interact with the planner to post various feasibility and optimality driven constraints on the evolving plan. Consider, for example, the problem of generating process plans in concurrent design environments [Cutkosky and Tenenbaum, 1990, Kambhampati and Tenenbaum, 1990, Kambhampati and Cutkosky, 1991].[1] A goal of concurrent design is to do as much manufacturing planning and analysis as possible during design evolution, rather than waiting for the design to be complete. A typical planner in this environment will have to operate concurrently with SDMS specializing in such considerations as geometric analysis, fixturing, tolerancing etc., all of which influence planning. Another example is the DARPA logistics planning scenario where several disparate specialized planners (such as evacuation planners, mission planners etc.) are required to cooperate with each other to make a global transportation plan.

Planning in these environments a continual rather than one-shot process as the constraints imposed by the SDMS on the plan force the planner to contend with a continually evolving problem specification. The planner needs to respond to the specification changes in a continual rather than batch driven mode (i.e., the planner cannot wait for all the changes to be complete) as the feedback from the planner often guides the behavior of the other modules in the environment. Further more, the planner needs to understand the rationale behind the externally imposed constraints so that it can play an active role of coordinating its actions with the SDMS .

The classical planning paradigm [Chapman, 1987] fails to adequately handle the requirements of these environments as it considers planning as a one-shot task of constructing a partially ordered sequence of actions for achieving a given set of goals. The planner works under the assumption that it is an isolated module with all knowledge relevant to plan generation at its disposal. This paradigm only accounts for the intra-plan interactions, and ignores the interactions between the planner and the other modules of a problem-solving environment.

To deal with the special requirements of planning in concurrent environments, the planning methodology needs to be extended to provide the following capabilities:

- *Incrementality and accommodating external constraints*

  The planner needs to respond to the changes in its specification and externally imposed constraints by updating its plan efficiently and conservatively. During this update process, it needs to be incremental and reuse as much of its existing plan as possible, as starting from scratch every time some constraint changes can be prohibitively expensive. Thus, to function effectively in a concurrent envi-

*We acknowledge the support of Office of Naval Research under contract N00014-88-K-0620.

[1] A process plan specifies the sequence of setup, fixturing and machining operations for manufacturing a given part (typically described as a set of machinable features).

ronment and interact with other modules efficiently, a planner needs to be *incremental* in its operation.

- *Coordination and understanding rationale behind externally imposed constraints*

  To anticipate external interactions and to coordinate its actions with other modules in the environment, a planner in a concurrent environment needs to be able to understand and reason about the rationale behind the externally imposed constraints. To facilitate this, a systematic interface methodology between the planner and the other heterogeneous modules of the environment needs to be developed.

Intuitively, the planner needs to be incremental in order even to survive as a passive module in a concurrent environment, while it needs the ability to coordinate its actions with the SDMS to play an active role in guiding the global problem-solving activity.

To facilitate incremental modification of plans in response to evolving specifications, the planner should be able to analyze the plan at multiple levels of detail and focus modification at the appropriate level. Further, it should use least-commitment strategies during planning, as over-constrained plans do not lend themselves well for flexible reuse [Kambhampati, 1989, Kambhampati, 1990a]. The criterion for accommodating external constraints is for the planner to be able to incorporate the changes imposed by those constraints while reusing as much of the existing plan as possible and preserving the correctness of the overall plan.[2] This requires the planner to reason about the effect of the external constraints on the correctness of the existing plan and make minimal modifications to it to regain correctness.[3] To analyze the effect of external constraint on the correctness of the plan, as well as to modify the plan conservatively, the planner needs to represent and reason about the internal dependencies of its plans in a systematic fashion.

Since, hierarchical nonlinear planning (e.g. NOAH [Sacerdoti, 1977], NONLIN [Tate, 1977], SIPE [Wilkins, 1984]) is the prominent method of abstraction and least commitment in domain independent planning, in this paper we will explore ways of extending it to handle the incrementality requirements of concurrent domains. In section 2, we will develop a precise characterization of the correctness of the plan in terms of a representation of its causal dependency structure called the *validation structure* and introduce a framework of incremental modification based on it. In section 3, we will present techniques to accommodate the external constraints which compute the ramifications of those constraints on the validation structure of the plan, and repair the plan to regain correctness.

[2]This is in contrast to approaches such as [Hayes, 1987], that allow externally imposed constraints to enter the plan, but do not reason about the effect of those constraints on the correctness of the plan.

[3]It should be emphasized here that we are not concerned with the absolute correctness of the plan, but rather its correctness with respect to the planner's own model of the domain (see [Kambhampati, 1990a]).

Enabling the planner to reason about the rationale behind the constraints imposed by the SDMS poses special problems because of the heterogeneous nature of the modules in a concurrent environment. Previous research in distributed planning (e.g. [Durfee and Lesser, 1988, Corkill, 1979]), black board systems (e.g. [Hayes-Roth, 1987]), embedded systems (e.g. [Georgeff, 1990]) and multi-agent planning architectures (e.g. [Lansky, 1988]), mostly addressed the issues of coordinating the actions of homogeneous planners working on different aspects of a single problem where each planner can understand and reason about the rationale behind the constraints imposed by other planners. Thus they are not well-suited to real world concurrent domains with heterogeneous modules. In section 4, we propose a methodology of interfacing the planner and the SDMS that relaxes the strong assumptions made by these approaches. Our approach is to enable the planner to understand the rationale at some appropriate level of abstraction. Accordingly, the external constraints will be accompanied by explanations that constitute "sufficient" (rather than necessary and sufficient) conditions under which the constraint can be guaranteed to be required by the SDM imposing it. We will discuss the issues involved in generating and coordinating with such window of applicability explanations.

2 Incrementality

In this section we discuss the issues involved in making planning "incremental". We will start with a characterization of correctness of plans in the hierarchical planning paradigm in terms of the plan validation structure. This characterization is used to develop a methodology for modifying a plan to regain correctness. The resultant incremental modification framework forms the basis for accommodating various externally imposed constraints into an existing plan (see next section).

2.1 Validation Structure and Plan Correctness

Hierarchical planning can be seen as a process of reduction of abstract tasks into more concrete subtasks with the help of domain specific task reduction schemata, and resolving any harmful interactions by introduction of additional partial ordering relations among tasks or backtracking over previous decisions [Tate, 1977, Sacerdoti, 1977]. Given a planning problem $[\mathcal{I}, \mathcal{G}]$ where $\mathcal{I}$ is the specification of the initial state and $\mathcal{G}$ is the specification of the desired goal state (given as conjunction of literals to be satisfied), we use a structure called hierarchical task network (HTN) to represent the status of the plan at any moment. A HTN is a 3-tuple

$$\langle\ \mathcal{P} : \langle T, O, \mathcal{V} \rangle\ , T^*\ , D\ \rangle$$

where $\mathcal{P}$ is a partially ordered plan such that

- T is the set of tasks of the plan (with two distinguished tasks $t_\mathcal{I}$ and $t_\mathcal{G}$ to denote the input and goal state specifications respectively)
- O defines a partial ordering over T (with elements of the form "$t_i \rightarrow t_j$", signifying that t_i is a predecessor of t_j)

- $\mathcal{V}$ is a set of conditions with specification about the ranges where those conditions should be held true, and the applicability conditions of tasks of $\mathcal{P}$, or goals of the overall plan those conditions are supporting. Individual elements of $\mathcal{V}$ are called *validations*. They are represented by 4-tuples $v : \langle E, t_s, C, t_d \rangle$, with the semantics that the condition E, which is an effect of t_s, should be held true from task t_s to t_d to support the applicability condition C of task t_j.
- T^* is the union of tasks in T and their ancestors
- D defines a set of parent, child relations among the tasks of T^* (if t^p is the parent of a task t, then t was introduced into the HTN because of the reduction of t^p)

We define the correctness of the plan $\mathcal{P}$ in terms of its set of validations in the following way:
A partially ordered plan $\mathcal{P}$ is considered a correct plan for a problem $[\mathcal{I}, \mathcal{G}]$ iff

- For each goal $g \in \mathcal{G}$ of the plan, and each applicability condition of a task $t \in T$, there exists a validation $v \in \mathcal{V}$ supporting that goal or condition. If this condition is not satisfied, the plan is said to contain "*missing validations.*"
- None of the plan validations are violated. That is, $\forall v : \langle E, t_i, C, t_j \rangle \in \mathcal{V}$, (i) $E \in effects(t_i)$ and (ii) $\nexists t \in T$ s.t. $\Diamond(t_i \prec t \prec t_j) \wedge effects(a) \vdash \neg C$ (where, the relation "$\prec$" is the transitive closure of the relation "$\rightarrow$"). If this constraint is not satisfied, then the plan is said to contain "*failing validations.*"

In addition, we introduce a condition of non-redundancy as follows

- For each validation $v : \langle E, t_s, C, t_d \rangle \in \mathcal{V}$, there exists a chain of validations the first of which is supported by the effects of t_d and the last of which supports a goal of the plan.[4] If this constraint is not satisfied, then the plan is said to contain "*unnecessary validations.*"

A plan that is correct by this definition is said to have consistent validation structure. The missing, failing and unnecessary validations defined above are collectively referred to as *inconsistencies* in the plan validation structure.

Notice that this definition of correctness is concerned exclusively about the validations that the planner has established. The plan may be correct by this definition and still be inapplicable from the point of view of some SDM . This is in consonance with the philosophy that the planner in a concurrent environment should primarily be concerned about keeping the plan correct from its perspective. When changes are necessitated by the externally imposed constraints, the planner should preserve correctness with respect to its validation structure while accommodating those changes.

[4]More formally, $\forall v : \langle E, t_s, C, t_d \rangle \in \mathcal{V}$ there exists a sequence $[v^1, v^2 \ldots v^k]$ of validations belonging to $\mathcal{V}$, such that (i) $v^k : \langle E^k, t_s^k, C^k, t_G \rangle$ supports a goal of the plan (i.e, $C^k \in \mathcal{G}$) (ii) $v^{k-1} : \langle E^{k-1}, t_s^{k-1}, C^{k-1}, t_s^k \rangle$ supports an applicability condition of t_s^k, v^{k-2} supports an applicability condition of t_s^{k-1} and so on, with v^{k-1} supported by an effect of t_d.

2.2 Annotating Validation Structure

To facilitate efficient reasoning about the correctness of the plan, and to guide incremental modification, we represent the plan validation structure as annotations on the individual tasks constituting the HTN [Kambhampati, 1990c]. In particular, for each task $t \in T^*$, we define the notions of e-conditions , e-preconditions and p-conditions as follows: The e-conditions of a task represent the set of validations that it or its descendents in the HTN provide to the rest of the plan. If $R(t)$ represents the set of tasks consisting of t and its descendents in the HTN (also called sub-reduction)[5], $e-$conditions(t) is given by the set

$$\{v \mid v : \langle E, t_s, C, t_d \rangle \in \mathcal{V} \wedge t_s \in R(t) \wedge t_d \notin R(t)\}$$

The e-preconditions represent the set of validations that the task or its descendents consume from the rest of the plan. $e-$preconditions(t) is given by the set:

$$\{v \mid v : \langle E, t_s, C, t_d \rangle \in \mathcal{V} \wedge t_s \notin R(t) \wedge t_d \in R(t)\}$$

Finally, the p-conditions represent the validations that should necessarily be preserved by the effects of the task and its descendents to guarantee the correctness of the plan. $p-$conditions(t) is given by the set

$$\left\{ v \;\middle|\; \begin{array}{c} v : \langle E, t_s, C, t_d \rangle \in \mathcal{V} \wedge t_s, t_d \notin R(t) \wedge \\ \exists t_i \in T \text{ s.t. } t_i \in R(t) \wedge \Diamond(t_s \prec t_i \prec t_d) \end{array} \right\}$$

The e-conditions , p-conditions and e-preconditions (referred to collectively as task annotations) encapsulate the role played by each task in the HTN of the plan in ensuring the correctness of the plan.

2.3 The PRIAR Modification Framework

Based on the notion of validation structure, we have developed a framework for flexible modification of plans in hierarchical planning called PRIAR [Kambhampati, 1990c, Kambhampati, 1989]. In PRIAR framework, a plan is modified in response to inconsistencies in its validation structure. The repair actions depend on the type of inconsistency, and the type of validation involved in that inconsistency. They involve removal of redundant parts of the plan, exploitation of any serendipitous effects of the changed situation to shorten the plan, and addition of high level (refit) tasks to re-establish any failing validations. For example, if the inconsistency is a failing validation $v : \langle E, t_s, C, t_d \rangle$, then depending upon whether or not C can be achieved by the planner, PRIAR either adds a task t_r to the HTN re-establish the failing validation, or replaces the reductions that are dependent on C. The annotations on the individual tasks of the HTN (defined in section 2.2) provide a systematic framework for locating the parts of the plan that need to be removed or replaced. Finally, any

[5]note that $R(t) = \{t\}$ if $t \in T$

refit-tasks introduced during the repair process are reduced by the hierarchical planner, which employs various validation structure based control strategies to localize this reduction [Kambhampati and Hendler, 1989, Kambhampati, 1990b].

3 Accommodating External Constraints

In this section, we discuss how the planner accommodates various types of changes necessitated by by externally imposed constraints. Without loss of generality we will assume that the planner has an initially correct plan which it wants to modify to accommodate the changes and while preserving the correctness of the plan. We will also assume at this point that the changes imposed by the external constraints are non-negotiable, in that they would have to be necessarily accommodated by the planner. In the previous section, we have shown that the planners notion of correctness of its plan is intimately tied to the consistency of the plan validation structure. Thus, the general strategy followed for accommodating changes necessitated by externally imposed constraints is to compute the inconsistencies in the validation structure that result from those changes and use the PRIAR framework to modify the plan to remove the detected inconsistencies. In the following sections we discuss how changes in the specification the problem, and ordering relations among the plan steps are handled through this strategy.

3.1 Changes in Input and Goal Specifications

Often during planning the specifications of the planing problem are modified to reflect the changes in the state of the world and the overall goals of the system. From the definition of correctness of the plan in section 2.1, it should be clear that the changes in problem specifications may give rise to missing, failing or unnecessary validations in the plan validation structure. For example if some of the goals of the plan become unnecessary as a result of changes in the specifications, the validations supporting those goals become unnecessary; if the changes necessitate new goals for the plan, that would lead to a missing validation and finally if the changes delete some assertions of the initial state then the validations supported by the effects of t_I will fail. These inconsistencies in the validation structure can be located by simply examining the validations supported by the assertions in the initial state, and goal state. Any detected inconsistencies are then repaired with the help of PRIAR modification operations.

3.2 Changes in ordering relations

Often external constraints translate into addition or deletion of ordering relations among the steps of the plan. Such changes may be a result of feasibility considerations (for example, in process planning, the geometric modeler might rule out some infeasible feature orderings [Hayes, 1987]), or of optimality considerations (eg. SDMS may group several steps of the plan together for efficient execution etc., and such groupings may be inconsistent with the existing ordering relations [Kambhampati and Philpot, 1990]). In the we provide methods for efficiently analyzing the ramifications of addition and deletion of ordering relations on the correctness of the plan.

3.2.1 Deletion of Ordering Relations

Suppose an ordering relation $o_d : t_a \rightarrow t_b$ is to be deleted as a consequence of some externally imposed constraint. This leads to a change in the partial ordering of the plan and some of the tasks which were previously ordered with respect to each other will now become unordered (parallel).

From the definition of correctness in section 2.1, it should be clear that failing validations are the only type of inconsistencies that can result from the deletion of an ordering relation.[6] Since checking each validation $v \in \mathcal{V}$ for failure is expensive, we use the following method to check only those validations which can fail because of the deletion of o_d.

First o_d is removed from O and the "$\prec$" relation on the tasks of the plan is recomputed. The predecessor-successor fields of the tasks of $\mathcal{P}$ before and after the deletion are compared. For each task $t_i \in T$ of $\mathcal{P}$, we collect the tasks of the plan that become unordered with respect to it. Suppose t_i becomes unordered with respect to tasks $\{t_{i_1}, t_{i_2} \ldots t_{i_m}\}$. From the definitions of section 2.2, we can show that when previously ordered tasks become parallel to t_i, the annotations of those tasks will also become p-conditions of t_i.[7] In other words, the new p-conditions of t_i are given by

$$p\text{-conditions}^{old}(t_i) \cup \bigcup_j annotations(t_{i_j})$$

By definition p-conditions(t_i) comprises the validations of the plan that must be preserved by the effects of t_i, Thus, to locate the validations that are violated by the effects of t_i it is sufficient to check the annotations of the tasks $\{t_{i_1}, t_{i_2} \ldots t_{i_m}\}$. If any validations are actually found to be violated, the PRIAR modification methodology is again used to suggest repair actions. Similar analysis is done for each task in the plan that becomes unordered with respect to other tasks because of the deletion of o_d.

3.2.2 Addition of Ordering Relations

Suppose the externally imposed constraint leads to the addition of a new ordering relation $o_n : t_a \rightarrow t_b$. We have

[6] Deletion of ordering relations increases the parallelism in the plan and this may lead to violation of some established validations of the plan.

[7] If a validation $v : \langle E, t_s, C, t_d \rangle \in \mathcal{V}$ belongs to the annotations of a task $t_j \in T$, then, given that $\forall t \in T\ R(t_j) = \{t_j\}$, we have (see section 2.2)

$$\Diamond(t_s \prec t_j \prec t_d) \vee (t_s = t_j) \vee (t_d = t_j)$$

Now suppose t_j became unordered with respect to $t_i \in T$. We have

$$\Diamond(t_j \prec t_i) \wedge \Diamond(t_i \prec t_j)$$

From the two relations above, we can see that if $v : \langle E, t_s, C, t_d \rangle$ belongs to the annotations of t_j then it will also satisfy the relation

$$\Diamond(t_s \prec t_i \prec t_j)$$

which in turn means that v is a p-condition of t_i.

two possibilities in this situation:

Case 1. *The added ordering is consistent with the current orderings.* As long as $t_b \not\prec t_a$ in the current plan, we can add $t_a \rightarrow t_b$ without causing any inconsistencies. From the definition of the correctness $\mathcal{P}$, we can see that a consistent new ordering *will not* affect the correctness of the existing plan. Thus nothing need be done in this case.

Case 2. *The added ordering is* not *consistent with the current ordering.* That is $t_b \prec t_a$ in the current plan. In this case, there are some "cycles" in the ordering O on $\mathcal{P}$. Let

$$t_b \rightarrow t_{i_1} \ldots t_{i_m} \rightarrow t_a \xrightarrow{o_n} t_b$$

be a cycle [8]. Since o_n has to be necessarily accommodated, the only way of regaining correctness is to break such cycles by removing some other previously held ordering relations in this cycle. Removal of an ordering relation may cause inconsistencies in the validation structure and those need to be located and repaired as discussed in section 3.2.1. The analysis of section 3.2.1 can also be used to decide which ordering relation will necessitate least amount of repair work upon deletion, and choose to delete that ordering relation.

4 Coordination through reasoning about the rationale behind externally imposed constraints

The techniques discussed in the previous two sections enable the planner to play a passive role of efficiently accommodating changes necessitated by any externally imposed constraints on its plan. As the specifications and the plan evolve, the external constraints also evolve and it might be possible to relax some of them under the new situation. Rather than waiting for SDMS to take the initiative, the planner should play an active role in guiding the global problem-solving activity anticipating external interactions and coordinating its actions with the SDMS . The simplest approach for this would require the planner to request all the modules about all the constraints imposed by them, when ever *anything* changes. However, this could be quite inefficient especially since the analyses performed by some SDMs may be computationally expensive. In addition, this approach also seriously undermines the autonomy of the planner as it cannot take any decisions without having them approved by all the modules.

To take an active role in the global problem-solving activity, the planner needs to understand and reason about the rationale behind the externally imposed constraints. This requirement is symmetric: when the planner makes any decisions that affect the outside modules, it should be able to associate a rationale for that decision. In other words, a major requirement for a module to work in a concurrent environment is to be able to provide rationale for the decisions that can be understood by other modules in the environment, and to be able to reason with the rationales provided by the outside modules.

[8] cycles can be found by scanning the "$\rightarrow$" and "$\prec$" relations of the tasks of the plan

Though the issue of coordination have been addressed previously in distributed planning and blackboard based approaches to planning, they generally assume that the individual planners are all identical and share a common representation (e.g. [Durfee and Lesser, 1988]) and that there is a single central knowledge base shared among all modules and planners [Hayes-Roth, 1987, Durfee and Lesser, 1988]. Because of these assumptions, coordination is accomplished in those architectures by allowing each planner to directly reason about the constraints imposed by other modules. This is controlled either through a central black-board mechanism [Hayes-Roth, 1987], a hierarchical organization of planners or through the use of localized representations that explicitly circumscribe the effects of the agent's actions [Lansky, 1988].

In concurrent domains, the assumption of homogeneous modules holds only in situations whose where the modules of a concurrent environment constitute a "vertical" decomposition of the domain, where each module takes all the specialized considerations independently. However, in concurrent domains, the environment typically consists of heterogeneous modules which specialize in various sub-tasks of the overall task. For example, in a domain like process planning where there are several specialized considerations (such as geometric, fixturing etc.) that need to be taken into account, if the planner itself were to take them all into account during planning, plan construction could become prohibitively expensive [Simmons and Davis, 1987]. Further more such an architecture will often be inconsistent with the natural structure of the domain. In process planning, even if the planner is designed to take into account all the geometric considerations during planning, duplicating all the geometric knowledge and inference formalisms into the planner would be very wasteful.

Because of this heterogeneity among the modules , it is not feasible to have each module understand the rationale behind the decisions of the other modules, *even* if it were provided to them. For example in process planning, it is infeasible for the planner producing machining sequence to understand all the details of constraints posted by modules specializing in fixturing and geometric reasoning etc.

Suppose an external constraint c_e (say an ordering relation) is imposed on the plan by the geometric modeler to ensure that there will be a clear access path to make a feature $feature_1$. There is no straightforward way for the geometric modeler to communicate to the planner that the the constraint c_e is imposed to ensure clear access path to $feature_1$ since the planner's domain model may not give it the ability to reason about the notion of "clear access path." In the absence of any rationale accompanying c_e, the planner would have to rely on the geometric modeler to correctly update the constraints in the event that constraint is no longer needed. This approach, in as much as it requires frequent geometric checks, is inefficient.

Thus, for computational tractability and functional autonomy of individual modules, planner in a concurrent environment should be able to understand the rationale

behind external constraints at some appropriate level of detail.

4.1 Window of Applicability (WOA) Explanation

Since the individual modules cannot reason about the rationale behind the externally imposed constraints directly, we propose that the rationale be presented as a set of sufficiency conditions called "WOA (Window of applicability)" explanation for the external constraint. These sufficiency conditions do not constitute the complete set of conditions leading to the imposition of the external constraint, but rather only an abstraction of those conditions that the planner can reason about.

Thus, each externally imposed constraint will be associated with a 3-tuple

$$\langle c_e, \text{woa}_{c_e}, \text{sdm}_{c_e} \rangle$$

where c_e is the constraint (changed specification or ordering relation) that the planner needs to accommodate into its plan, sdm_{c_e} is the SDM that is imposing that constraint, and woa_{c_e} is the WOA explanation associated with c_e. The semantics are that

- WOA consists only of predicates that the planner can reason about [9]
- As long as woa_{c_e} is holding, c_e should necessarily be accommodated into its plan
- If the planner finds at any point of time that woa_{c_e} is not holding, then it is possible that c_e is no longer required. The planner can then request the sdm_{c_e} to check if the constraint is still required. [10] (If the sdm_{c_e} were to retract the constraint c_e, then the planner can use the methodology developed in sections 2 and 3 to appropriately modify the plan to accommodate the change.)

Note that this allows the planner to essentially ignore outside modules unless the WOA of the constraint posted by them are affected. This provides functional autonomy to the planner, by obviating the need to poll each and every SDM for every change in the environment.

[9] A proposition f is considered *reachable* if the planner can reason about the ramifications of its actions on the truth of the proposition. The WOA should only consist of reachable propositions. In TWEAK representation, which does not allow any domain axioms, a proposition is reachable iff f codesignates with some proposition in the add or delete lists of some operator template of the planner.

[10] The external constraints may have been voluntarily imposed by the external modules or may have been posted to satisfy some request by the planner. In the process planning example, either the geometric modeler may have imposed c_e because *it* wanted to ensure clear access path, or alternatively the planner requested it to ensure clear access path. In the latter case, the planner may model clear access path as precondition *ClearAccess(feature)* of some task of $\mathcal{P}$ whose truth needs to be computed by an outside module (*cf.* "compute-conditions" of nonlin [Tate, 1977]). When asked to make such a condition true, the geometric modeler may then impose some constraints (such as c_e) on the current plan. For the purposes of our current discussion, this distinction is immaterial.

In the process planning example discussed above, the planner might be given the WOA explanation

$$a \leq \mathit{length}(\mathit{feature}_i) \leq b$$

for the constraint c_e, with the semantics that as long as the length of the feature $\mathit{feature}_i$ is in the range $[a\,b]$, the constraint c_e would be required by the geometric modeler. When the WOA is violated, it does not mean that the c_e is automatically retracted. Instead, the corresponding SDM (in this case geometric modeler) will be requested to repeat the computation to see if c_e is still required in the current situation.

4.2 Classes of WOA Explanations

A critical issue about the WOA explanation concerns their level of detail: as mentioned above, they should be at such a level as to allow the planner to reason with the explanation, and they should constitute *sufficient* conditions for assuming that c_e is still needed. There may be a spectrum of such conservative sufficiency conditions for any given external constraint, with tradeoffs between the informedness of the WOA and the ability of the outside modules to reason with it.

For instance, in the process planing example above, the geometric modeler could return as the WOA of the ordering constraint a representation of the configuration space in which the $\mathit{ClearAccess}(h_1)$ predicate will remain. However, that WOA is likely to be at too low a level of detail to be directly useful to the planner. On the other hand it could return as the WOA a list of quantities whose values it used in arriving at c_e as the constraint to be imposed. In this case, the planner would have to reinvoke the geometric modeler any time any of those quantities change.

While the former WOA is expressive but difficult to reason with, the latter is easy to reason with, but very conservative. The correct level of abstraction of WOA depends ultimately on the degree of similarity between the domain models and the inference strategies of the planner and the SDMS . For example, if all the modules are identical, as in distributed planning, the WOA can include both the necessary and sufficient conditions for c_e. While in complex environments with heterogeneous modules, even a WOA that specifies the quantities on whose values c_e depends, would be of utility.

A promising strategy is to provide WOA explanations that are at multiple levels of abstraction. This will allow the planner to choose the abstraction that it can handle, while still allowing some other SDM to reason with the WOA at a deeper level of detail.

Careful investigation will be needed to find classes of explanations that satisfy the conservative (sufficiency) property of the window of applicability explanations, while at the same time allowing reasoning at a level suitable for other modules.

5 Summary

Planning in many real world situations requires concurrent operation between the planners and other modules of the environment. This poses several requirements that

are not supported by the existing approaches to automated planning. In particular, we identified the ability to incrementally accommodate changes necessitated by the externally imposed constraints into the existing plans, and the ability to understand and reason about the rationale behind externally imposed constraints at an appropriate level of detail as two crucial requirements for planning in such environments. We have then explored ways of extending the hierarchical planning framework to handle these requirements. In particular, we discussed the techniques for making the planning incremental and for accommodating the externally imposed constraints by reasoning about their effect on the correctness of the plan. Next we proposed a methodology for coordination between planners and external modules that depends on attaching a set of sufficiency conditions rather than necessary and sufficient conditions as justifications to various externally imposed constraints. We have discussed the issues of generating and reasoning with such conditions.

Acknowledgements

We would like to thank Mark Cutkosky for many helpful discussions and comments on a previous draft of this paper. Andrew Philpot and Randy Wilson also provided helpful suggestions.

References

[Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3), 1987.

[Corkill, 1979] D.D. Corkill. Hierarchical planning in a distributed environment. In *Proceedings of Sixth IJCAI*, August 1979.

[Cutkosky and Tenenbaum, 1990] M. R. Cutkosky and J. M. Tenenbaum. A methodology and computational framework for concurrent product and process design. *Mechanism and Machine Theory*, 23(5), 1990.

[Durfee and Lesser, 1988] E.H. Durfee and V.R. Lesser. Predictability versus responsiveness: Coordinating problem solvers in dynamic domains. In *Proceedings of Seventh NCAI*, August 1988.

[Georgeff, 1990] M. Georgeff. Decision-making in an embedded reasoning system. In *Proceedings of Eleventh IJCAI*, August 1990.

[Hayes-Roth, 1987] B. Hayes-Roth. Dynamic control planning in adaptive intelligent systems. In *Proceedings of DARPA Knowledge-Based Planning Workshop*, December 1987.

[Hayes, 1987] C. Hayes. Using goal interactions to guide planning. In *Proceedings of 6th National Conference on Artificial Intelligence*, pages 224–228, July 1987.

[Kambhampati and Cutkosky, 1991] S. Kambhampati and M. R. Cutkosky. An approach toward incremental and interactive planning for concurrent product and process design. In *Proceedings of ASME Winter Annual Meeting on Computer Based Aproaches to Concurrent Engineering*, (To appear) 1991.

[Kambhampati and Hendler, 1989] S. Kambhampati and J.A. Hendler. Control of refitting during plan reuse. In *Proceedings of 11th International JointConference on Artificial Intelligence*, pages 943–948, August 1989.

[Kambhampati and Philpot, 1990] S. Kambhampati and A. Philpot. Incremental planning for concurrent product and process design. Technical report, Center for Design Research and Computer Science Department, Stanford University, CA, (In preparation) 1990.

[Kambhampati and Tenenbaum, 1990] S. Kambhampati and J.M. Tenenbaum. Towards a paradigm for planning in interactive domains with multiple specialized modules. In *AAAI-90 Workshop on Automated Planning for Complex Domains*, August 1990.

[Kambhampati, 1989] S. Kambhampati. *Flexible Reuse and Modification in Hierarchical Planning: A Validation Structure Based Approach.* PhD thesis, Center for Automation Research, Department of Computer Science, University of Maryland, College Park, MD 20742, October 1989.

[Kambhampati, 1990a] S. Kambhampati. A classification of plan modification strategies based on their information requirements. In *AAAI-90 Spring Symposium on Case-Based Reasoning*, March 1990.

[Kambhampati, 1990b] S. Kambhampati. Mapping and retrieval during plan reuse: A validation-structure based approach. In *Proceedings of 8th National Conference on Artificial Intelligence*, August 1990.

[Kambhampati, 1990c] S. Kambhampati. A theory of plan modification. In *Proceedings of 8th National Conference on Artificial Intelligence*, August 1990.

[Lansky, 1988] A. Lansky. Localized event based reasoning for multiagent domains. *Computational Intelligence Journal*, 4(4), 1988.

[Sacerdoti, 1977] E.D. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier North-Holland, New York, 1977.

[Simmons and Davis, 1987] R. Simmons and R. Davis. Generate, test and debug: Combining associational rules and causal models. In *Proceedings of 10th International Joint Conference on Artificial Intelligence*, August 1987.

[Tate, 1977] A. Tate. Generating project networks. In *Proceedings of 5th International Joint Conference on Artificial Intelligence*, pages 888–893, 1977.

[Wilkins, 1984] D.E. Wilkins. Domain independent planning: Representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.

Deadline-Coupled Real-Time Planning*

Sarit Kraus
Institute for Advanced Computer Studies and
Department of Computer Science
Univ. of Maryland, College Park, MD 20742

Madhura Nirkhe
Department of Electrical Engineering and
Institute for Advanced Computer Studies
Univ. of Maryland, College Park, MD 20742

Donald Perlis
Department of Computer Science and
Institute for Advanced Computer Studies
Univ. of Maryland, College Park, MD 20742

Abstract

We have undertaken a project in combining declarative and procedural forms of real-time planning for novel deadline situations. In deadline situations the time taken in reasoning toward a plan brings the deadline closer. Thus the planning mechanism should take account of the passage of time during that *same* reasoning. This general consideration is also the subject of other work. However, we are attempting to treat *all* facets of planning as deadline-coupled; the problem then is how to take proper account of the approaching deadline when any such accounting itself simply takes more time and seemingly gets in the way of its own accuracy. We employ the mechanism of step-logics toward solving this problem.

1 Introduction

We have undertaken a project in combining declarative and procedural forms of real-time planning for novel deadline situations. In deadline [1] situations the time taken in reasoning toward a plan brings the deadline closer. Thus the planning mechanism should take account of the passage of time during that *same* reasoning. This general consideration is also the subject of other work.

However, we are attempting to treat *all* facets of planning as deadline-coupled; the problem then is how to take proper account of the approaching deadline when any such accounting itself simply takes more time and seemingly gets in the way of its own accuracy. We employ the mechanism of step-logics toward solving this problem.

Meta-planning is the usual proposal for reasoning about the reasoning process. But that takes time too! Maybe the time taken by meta-planning can be kept very short. But what of highly novel settings in which one cannot a priori assign expected utilities to various conceivable options or refinements? Then the planner may have to decide on utilities and other factors in real time. In these cases it seems unlikely that such meta-planning will always have a modest time cost. In what follows,we present an illustrative example; sketch the structure of our program and show a few important steps of the output.

1.1 An Illustration

To elaborate, we present an illustrative domain, which we call the Nell and Dudley Scenario: [2] Nell is tied to the railroad tracks as a train approaches. Dudley must formulate a plan to save her and carry it out before the train reaches her. If we suppose Dudley has never rescued anyone before, then he cannot rely on having any very useful assessment in advance, as to what is worth trying. He must deliberate (plan) in order to decide this, yet as he does so the train draws nearer to Nell. Thus he must also assess and adjust (meta-plan) his ongoing deliberations vis-a-vis the passage of time. Since the setting is novel Dudley does not have a "canned" procedure $rescue_heroine(H)$ which he can just invoke with $H = Nell$. However, Dudley has acquired some "shelf" procedures for solving simpler subproblems that are encountered in more routine situations. For example, we do not expect Dudley to deliberate on how to run, he knows that he must take a number of paces depending upon the distance. He also knows that running is a means of transporting oneself from one place to another. Similarly, axioms come to his aid regarding which subtasks he must perform to complete the operation of untying Nell once he reaches the rail track. His total effort (plan, meta-plan and action) must stay within the deadline. He must in short, reason in time about his own reasoning in time.

*This is an extended version of our paper [Kraus *et al.*, 1990]. This research was supported in part by NSF grant IRI-8907122, and in part by the U.S. Army Research Office(grant DAAL03-88-K0087).

[1] We do not agree with the claim in [Russell and Wefald, 1989] (page 401) that "the 'deadline' model of time pressures is overly restrictive, since in reality there is almost always a continuous increase in the cost of time." We think that many situations do involve relatively hard deadlines; e.g. getting to the airport in time, not to mention the more dramatic Nell and Dudley case below.

[2] This problem was first mentioned in the context of time-dependent reasoning by McDermott [McDermott, 1978].

1.2 Related Work

Drew McDermott [McDermott, 1982] and Andrew Haas [Haas, 1985] have discussed the Nell and Dudley problem in terms of a surprising difficulty: if Dudley does not properly distinguish his planned actions from actual events then he may formulate a plan to save Nell and then conclude that his plan will save her, hence she is not in danger, hence he does not need the plan after all! This bizarre possibility can indeed arise in a highly limited representational setting, in which plans are not distinguished from actions.

However, although this is treated in our project, it is only a small part of the main thrust of our concern, which is *to find effective representational and inferential tools by which a reasoner can keep track of the passing of time as he makes (and enacts) his plan, thereby allowing him to adjust the plan so that neither the plan-formation that is in progress, nor its simultaneous or subsequent execution, will take him past the deadline.* In terms of our toy scenario, we want to prevent Dudley from spending so much time seeking a theoretically optimal plan to save Nell, that in the meantime the train has run Nell down. Moreover, we want Dudley to do this without much help in the form of expected utilities or other prior computation.

In [Horvitz, 1988], [Horvitz *et al.*, 1989], and [Russell and Wefald, 1989], decision-theoretic approaches are used to optimize the value of computation under uncertain and varying resource limitations. In both works, deadlines and the passage of time while reasoning are taken into consideration in computing the expected computational utility. However, these works do not account for the time taken for meta-planning. Indeed, this is stated in [Russell and Wefald, 1989] (page 402): "Here we will not worry about the cost of meta-reasoning itself; in practice, we have been able to reduce it to an insignificant level".

Dean [Dean, 1984] proposed a computational approach to reasoning about events and their effects occurring over time. Dean, Firby and Miller [Dean *et al.*, 1988] subsequently designed FORBIN, a planning architecture that supports hierarchical planning involving reasoning *about* deadlines, travel time, and resources. Dean and Boddy [Dean and Boddy, 1988b] formulated an algorithmic approach to solution of time-dependent planning problems by introducing "anytime algorithms" which capture the notion that utility is a monotonic function of deliberation time. Here also, the time for computation is not accounted for : "The time required for deliberation scheduling will not be factored into the overall time allowed for deliberation. For the techniques we are concerned with, we will demonstrate that deliberation scheduling is simple, and, hence, if the number of predicted events is relatively small, the time required for deliberation can be considered negligible." [Dean and Boddy, 1988b] (page 50). [Boddy and Dean, 1989] demonstrated deliberation scheduling for a time-dependent planning problem involving tour and path planning for a mobile robot.

We refer the reader to [Hendler *et al.*, 1990] for a general survey of related work on planning. Some particular results on temporal planning follow. [Allen and Koomen, 1983] formulated a world model based on temporal logic which allows the problem solver to gather constraints on the ordering of actions without having to commit to it when a conflict is detected. [Dean, 1987] discusses how a planner can reason about the difficulty of its tasks, and depending on available time, produce reasonable if not optimal solutions. [Lansky, 1986] and [Lansky, 1988] use a first-order temporal logic model to describe complex synchronization properties of parallel multiagent domains. In [Dean and McDermott, 1987] a computational approach to temporal reasoning is presented in which a problem solver is forced to make predictions and projections about the future and plan in the face of uncertainty and incomplete knowledge. Time-maps are introduced here. [Dean and Boddy, 1988a] examine the complexity of temporal reasoning problems involving events whose order is not completely known.

Our approach has many similarities with the research cited above; the main difference is that we are attempting to account for *all* the time taken for planning and acting.

2 The Status of the Project thus Far

2.1 Using Step-logics

Our current project employs the formalism of "step-logics" introduced by Elgot-Drapkin, Miller, and Perlis ([Drapkin *et al.*, 1987], [Elgot-Drapkin and Perlis, 1990], [Elgot-Drapkin, 1988]) where inferences are parametrized by the time taken for their inference, and in which these time parameters themselves can play a role in the specification of the inference rules and axioms. Step-logics offer a natural representation of the evolving process of reasoning itself. A *step* is a fundamental unit roughly characterized by the time it takes Dudley to draw a single inference. *Observations*, which are inputs from the external world, may arise at the beginning of a discrete time-step. When an observation appears, it is considered a belief in the same time-step. Apart from his observations at the beginning of step i, the only information available to Dudley is a snap-shot of his deduction process completed up to and including step $i-1$. During step i Dudley applies all available inference rules in parallel, but only to beliefs at step $i-1$; new beliefs thus generated through applications of inference rules are not available for use in further inference until step $i+1$. For example, consider the following reasoning (shown is an application of modus ponens — Rule 8, Appendix B) from step 8 to step 9.

8: Now(8); $Run(17:30, dudley, here:there)$;
$Run(17:30, dudley, here:there) \rightarrow$
$At(30, dudley, there)$;
...

9: Now(9); $Run(17:30, dudley, here:there)$;
$Run(17:30, dudley, here:there) \rightarrow$
$At(30, dudley, there)$;
At(30, dudley, there).
...

Although this illustrates the use of modus ponens, in fact, when Dudley goes to save Nell, he will never have a belief such as $Run(17 : 30, dudley, here : there)$, at any step prior to 17, since this is a future prediction and therefore treated as a projection within a plan, rather than as a fact. Notice that Dudley knows what time it is, and therefore that knowledge changes at every step. In effect, step-logics are first-order logics suitably modified to include a $Now(i)$ predicate, where the value of i changes at the end of a time-step.

2.2 Structure and Representation of a Partial Plan

We have created a suitable representational language for a simple (e.g., we are not yet addressing interacting plans) version of the Nell and Dudley representative deadline problem.[3] The set of axioms and inference rules can be found in the appendix. ¿From an initial unsolved goal Dudley formulates the first partial plan, and gives it a name (Rule 2, Appendix B). [4] This name appears as a parameter in all his reasoning concerning the particular partial plan, until the goal is achieved or the plan is aborted as unfeasible. Dudley maintains a set of *Facts* which consists of beliefs obtained through direct observations, and the largest subset of Projection set which is consistent with the observations and whose time intervals have been passed by step i. At all times Dudley remembers the hard deadline which he must meet. The partial plan is a temporally ordered list of action triplets. Each ordered triplet consists of an *action*, preceded and followed, respectively, by its associated *condition* and a *result*. A triplet is written within square brackets [...] and an ordered list of triplets is enclosed within curly brackets {...} in our notation. An *action* may be complex or primitive (atomic). A primitive action takes one step to perform. A complex action must be further refined to the level of primitive actions using axioms. The *condition* is a set of wffs that must be true during the course of the action being performed. The *result* is a set of wffs which are expected to be true at the completion of the action.

2.3 Working Estimate of Time

As Dudley develops a partial plan to save Nell, he continuously refines his estimate of the time to carry the plan to completion, making sure it will not overshoot the deadline. This we call the *working estimate of time* (WET for short) [5]. The WET is Dudley's calculation of how long his partial plan (formed as of the previous step) will take to execute. This he adds to the current time and compares the result to the deadline to make sure the plan is not hopeless (Rules 5 & 6, Appendix B). As long as it is not he declares it *Feasible*, and continues refining and/or putting it into execution. Dudley updates the WET when an action with a fixed non-zero interval between its start and finish times is made part of the plan [6]. As the plan reaches completion the WET reaches a realistic estimate for the time necessary for the execution of the residual partial plan, and thereby helps Dudley keep track of the time available for deliberation as well as acting.

2.4 Projection in the Context of a Plan

The set of *Facts*, along with the actions and the results contained in the current partial plan together form what we call the context set of the partial plan. The mechanism by which each predicate from the context set (based on information from the earlier step) is projected [7] into the future is as follows (Rule 10, Appendix B). Some wffs are related to events that have fixed start and finish times and are not expected to persist beyond their finish time. An example is $Run(T_1 : T_2, Y, L_1 : L_2)$. Beyond time T_2 we do not expect *Run* to continue to hold. There are other wffs which can be projected by default infinitely into the future. However, the projected time range of a predicate is trimmed to exclude the time-overlap with other predicates that can not co-exist in its presence. For example, looking at Nell as she is tied to the railroad tracks, Dudley initially projects that she will remain tied there until infinity, and in particular, at the instant of the projected arrival of the train. This causes him to initiate the formulation of the plan. However, in the context of his partial plan, when the plan is refined to include *Release*, the range of persistence of *Tied* must be trimmed due to the appearance of result *Not_tied*, since the two are mutually exclusive. Dudley maintains the clear distinction that *Not_tied* is true only in the context of the plan, and still maintains the belief $Unsolved(Goal(out_of_danger(Ddl, n, l_2)))$ until the plan is completed executed. This provides an easy solution to McDermott's difficulty mentioned earlier, namely the inability to distinguish between plans and actual events. Thus, via his projection mechanism Dudley deduces supposed changes in the world, thereby revising some beliefs and retaining others: the familiar issues of the frame problem. Dudley uses his projection both in planning and acting.

2.5 Use of Projections in Planning

If a condition C_A for a particular action A can be found in the projection (in the context of the partial plan developed thus far), Dudley does not attempt to find an axiom for achieving C_A. He marks C_A as *satisfied* (Rule 11, Appendix B). If C_A is not expected to be

[3] This version has been implemented in PROLOG.

[4] We will also find the name of the plan useful in later versions which will consider multiple plans.

[5] The WET is one of our concessions to procedural methods: we do not require Dudley to figure out how to do arithmetic but rather allow that he already knows. But we do require him to note the passage of time *during* the execution of the procedure.

[6] Currently, Dudley does not have a procedure to estimate the duration of tasks with unspecified time intervals.If we incorporate such a procedure, the time taken to execute it will also be our concern.

[7] Projections (and persistences) have been studied by numerous authors; see eg. [McDermott, 1987], [McDermott, 1982] , [Wilensky, 1983], [Charniak and McDermott, 1985], [Kautz, 1986] and [Kanazawa and Dean, 1989]. Our treatment is along the lines of time-maps of [Dean and McDermott, 1987], [Dean, 1987].

true in the projection, he finds an axiom of the form $B_1, \ldots, B_k \rightarrow C_A$, the triplets corresponding to B's are *chained* to the triplet for A, and C_A is marked *satisfied* (Rule 12, Appendix B). A flexible time margin is available between R_{B_k} (the result of B_k) and C_A. At every step, all conditions marked satisfied are re-examined to see if a changed projection set has rendered them unsatisfied again. An action A whose condition C_A is satisfied using either of the two methods is further decomposed if it is not primitive, by using an axiom of the form $D_1 \wedge \ldots \wedge D_k \rightarrow A$ (Rule 14, Appendix B). A chain of triplets each corresponding to D_i is added in place of the triplet for A. In the current implementation, we regard the $D_1 \wedge \ldots \wedge D_k$ as a *shelf* plan for doing A, and hence do not allow flexible time margins between the end of D_j and start of D_{j+1}, $1 \leq j \leq k$.

2.6 Use of Projections and Observations in Acting

Dudley may start to act on the partially developed plan as soon as it is possible to perform a primitive action, not waiting for the plan to reach completion. At the same time he continues planning [McDermott, 1978],[Georgeff and Lansky, 1988]. His predicted projections and observations are compared; conflicts resolved in favor of the latter (Rule 13, Appendix B). Projections can suffice to satisfy the condition for a primitive action, when the condition is not directly observed, provided the projections do not contradict any other observations. For example, as Dudley takes one pace after another, he does not necessarily observe that he is $At(L_1 + v)$, $At(L_1 + 2v)$ ...; he can act on the basis of his projections unless he gets an observation input during the course of his pacing, informing him that a certain pace was faulty and landed him $At(L_3)$ instead of the desired destination. He can not proceed with the next pace and must revise his plan and WET in such a situation. When A is acted upon, The start_time of the condition C_A for A is bound to Now and other time variables which have a fixed distance from the start_time are also bound appropriately.

We are currently extending our implementation in various ways, to involve perceptual reasoning,[8] explicit representations for extended actions, revising plans when they are seen to be inadequate, and choosing between multiple plans.

3 Some Illustrative Steps

To illustrate our efforts in a bit more detail, we present below portions of the output from our PROLOG program that implements the ideas we have been discussing. Here Nell is a distance of 35 'paces' from Dudley when he first realizes (step 0) that the train will reach her in 50 time units. He begins to form a plan, seen below in step 1 as *Ppl* ('partial plan'), and refines the plan in subsequent steps. *Ddl* is the deadline time (50 in the example) given to Dudley. d is Dudley, and n is Nell. The subscript *obs* indicates that the wff it is attached to is the result of an observation. Subscripted l's indicate locations and subscripted t's indicate times (step numbers). A colon between times (as in $At(0:\infty, d, l_1)$) represents a time interval during which the predicate is asserted to be true (in this case, that Dudley will be at location l_1 from 0 to infinity.

Proj gives Dudley's projections as to what will be true in the future, based on his partial plan and whatever *Facts* he has to work with. The word *save* that appears as argument to *Ppl*, *Proj* and *Feasible* in step 1, is simply a label naming the plan he is forming.

0: **Facts**$(\{At(0,d,l_1)_{obs}, Tied(0,n,l_2)_{obs}\})$, **Deadline**$(50)$, **Goal**$(out_of_danger(Ddl,n,l_2))$

1: **Facts**$(\{At(0,d,l_1), Tied(0,n,l_2)\})$, **Deadline**$(50)$, **Unsolved**$(Goal(out_of_danger(50,n,l_2)))$, **Ppl**$(save, 1, \{out_of_danger(50,n,l_2))\})$, **Proj**$(save, \{At(0:\infty,d,l_1), Tied(0:\infty,n,l_2)\})$, **WET**$(save,0)$, **Feasible**$(save,0)$

In step 1 the *Ppl* simply records that Dudley plans to get Nell out of danger. In his *Proj* he still expects to remain where he is (l_1 for the indefinite future ('∞') since he has not yet realized in this first second that he must move to save Nell. Nor has he realized he must untie Nell, so he also projects that she will remain tied indefinitely.

2: **Facts**$(\{At(0:1,d,l_1), Tied(0:1,n,l_2)\})$, **Deadline**$(50)$, **Unsolved**$(Goal(out_of_danger(50,n,l_2)))$,

$$\textbf{Ppl}(save, 2, \left\{ \begin{bmatrix} Not_tied(t_1,n,l_2) \\ Pull(t_1:t_2,d,n,l_2) \\ Out_of_danger(t_2,n,l_2) \end{bmatrix} \right\},$$

$\{t_2 \leq 50, t_1 = t_2 - 1\})$
Proj$(save, \{At(0:\infty,d,l_1), Tied(0:\infty,n,l_2)\})$, **WET**$(save,0)$, **Feasible**$(save,1)$

In step 2 Dudley has begun refining his plan, namely he determines that if Nell were untied then he could *Pull* her out of danger; this he infers from general world knowledge (axioms, not shown). The times t_1 and t_2 here are indefinite times that must satisfy only the conditions shown, so that the WET is not too long. The column matrix indicates an action (*Pull*) with its enabling condition (*Not_tied*) and result (*Out_of_danger*). We skip the next three steps for the sake of brevity.

[8] This ties back to spatial reasoning, and to aspects of a plan that involve getting more information; for instance Dudley may have to move in order to see whether Nell is tied. This in turn relates to existing work ([Kraus and Perlis, 1989], [Davis, 1988]) on ignorance and perception.

5: **Facts**($\{At(0:4,d,l_1), Tied(0:4,n,l_2)\}$),
Deadline(50),
Unsolved($Goal(out_of_danger(50,n,l_2))$),

$$\textbf{Ppl}(save,5,\left\{\begin{array}{l}\left[\begin{array}{c}At(t_6,d,l_1)\\ Run(t_6:t_7,d,l_1:l_2)\\ At(t_7,d,l_2)\end{array}\right]_1\\ \left[\begin{array}{c}At(t_3,d,l_2)\\ Untie_1(t_3:t_9,d,n,l_2)\\ Succ_u_1(t_9)\end{array}\right]_{2\ldots}\\ \left[\begin{array}{c}At(t_5,d,l_2), Succ_u_2(t_5)\\ Untie_3(t_5:t_4,d,n,l_2)\\ Succ_u_3(t_4), Not_tied(t_4,n,l_2)\end{array}\right]_4\\ \left[\begin{array}{c}Not_tied(t_1,n,l_2)\\ Pull(t_1:t_2,d,n,l_2)\\ Out_of_danger(t_2,n,l_2)\end{array}\right]_5\end{array}\right\},$$

$\{t_2 \le 50, t_1 = t_2 - 1, t_4 \le t_1,$
$t_5 = t_4 - 1, t_3 = t_4 - 3, t_3 = t_9 - 1,$
$t_7 \le t_3, t_8 = t_7 - 1, t_6 < t_7\}$),
Proj($save, \{At(0:t_8,d,l_1), At(t_7:\infty,d,l_2),$
$Tied(0:t_5,n,l_2), Not_tied(t_4:\infty,n,l_2),$
$Out_of_danger(t_2:\infty,n,l_2), Pull(t_1:t_2,d,n,l_2),$
$Release(t_3:t_4,d,n,l_2), Run(t_6:t_7,d,L:l_2)\}$),
WET($save,4$), **Feasible**($save,4$).

In step 5, Dudley has been able to infer (from axioms not shown) that he can refine his plan by running to Nell from l_1 (since he projects' from earlier steps that he will still be at l_1 at step t_6) to l_2 and releasing her (which will take him three untying actions). The numerical subscripts attached to column matrices show the order in which they are to be read ; also the subscripts show some portions have been omitted for ease of presentation. Note that the result *Not_tied* in the fourth matrix matches the enabling condition of the fifth matrix. Notice in *Proj* at last Dudley knows he must move to l_2 (by some as yet indefinite time t_7, where he then supposes he will remain.

6: **Facts**($\{At(0:5,d,l_1), Tied(0:5,n,l_2)\}$),
Deadline(50),
Unsolved($Goal(out_of_danger(50,n,l_2))$),

$$\textbf{Ppl}(save,6,\left\{\begin{array}{l}\left[\begin{array}{c}At(t_6,d,l_1)\\ Pace(t_6:t_{10},d,l_1:l_1+v)\\ At(t_{10},d,l_1+v)\end{array}\right]_{1\ldots}\\ \left[\begin{array}{c}At(t_8,d,l_1+34v)\\ Pace(t_8:t_7,d,l_1+34v:l_2)\\ At(t_7,d,l_2)\end{array}\right]_{35\ldots}\end{array}\right\})$$

$\{t_2 \le 50, t_1 = t_2 - 1, t_4 \le t_1, t_5 = t_4 - 1,$
$t_3 = t_4 - 3, t_3 = t_9 - 1, t_7 \le t_3, t_8 = t_7 - 1,$
$t_6 = t_7 - 35, t_6 = t_{10} - 1\}$),
Proj($save, \{At(0:t_8,d,l_1), At(t_7:\infty,d,l_2),$
$Tied(0:t_5,n,l_2), Not_tied(t_4:\infty,n,l_2),$
$Out_of_danger(t_2:\infty,n,l_2), Pull(t_1:t_2,d,n,l_2),$
$Release(t_3:t_4,d,n,l_2), Run(t_6:t_7,d,l_1:l_2),$
$Succ_u_1(t_9:\infty), Succ_u_2(t_5:\infty),$
$Succ_u_3(t_4:\infty), Untie_1(t_3:t_9,d,n,l_2),$
$Untie_2(t_9:t_5,d,n,l_2), Untie_3(t_5:t_4,d,n,l_2)\}$),
WET($save,39$), **Feasible**($save,5$)

In step 6 Dudley has planned his run (35 paces) and is ready to start enacting his plan. This is seen by comparing step 6 and step 7; in the latter he no longer has the plan to do the first pace (from l_1 to $l_1 + v$ since he has moved this to his 'do' list of actions (not shown) since (in this case) the *Facts* list does not contradict his projected position of l_1. Here v is his velocity (i.e., one pace per second).

7: **Facts**($\{At(0:6,d,l_1), Tied(0:6,n,l_2)\}$),
Deadline(50),
Unsolved($Goal(out_of_danger(50,n,l_2))$),

$$\textbf{Ppl}(save,7,\left\{\begin{array}{l}\left[\begin{array}{c}At(8,d,l_1+v)\\ Pace(8:9,d,l_1+v:l_1+2v)\\ At(9,d,l_1+2v)\end{array}\right]_{1\ldots}\\ \left[\begin{array}{c}At(41,d,l_1+34v)\\ Pace(41:42,d,l_1+34v:l_2)\\ At(42,d,l_2)\end{array}\right]_{34}\\ \left[\begin{array}{c}At(t_3,d,l_2)\\ Untie_1(t_3:t_9,d,n,l_2)\\ Succ_u_1(t_9)\end{array}\right]_{35\ldots}\\ \left[\begin{array}{c}Not_tied(t_1,n,l_2)\\ Pull(t_1:t_2,d,n,l_2)\\ Out_of_danger(t_2,n,l_2)\end{array}\right]_{38}\end{array}\right\}$$

$\{t_2 \le 50, t_1 = t_2 - 1, t_4 \le t_1,$
$t_5 = t_4 - 1, t_3 = t_4 - 3, t_3 = t_9 - 1,$
$t_6 = 7, t_{10} = 8, \ldots, t_7 = 42, t_7 \le t_3\}$),
Proj($save, \{At(0:t_6,d,l_1), At(t_{10},d,l_1+v),$
$\ldots, At(t_7:\infty,d,l_2), Tied(0:t_5,n,l_2),$
$Not_tied(t_4:\infty,n,l_2),$
$Out_of_danger(t_2:\infty,n,l_2),$
$Pull(t_1:t_2,d,n,l_2), Release(t_3:t_4,d,n,l_2),$
$Run(t_6:t_7,d,l_1:l_2), Succ_u_1(t_9:\infty),$
$Succ_u_2(t_5:\infty), Succ_u_3(t_4:\infty),$
$Untie_1(t_3:t_9,d,n,l_2), Untie_2(t_9:t_5,d,n,l_2),$
$Untie_3(t_5:t_4,d,n,l_2), Pace(t_6:t_{10},d,l_1,l_1+v),$
$Pace(t_{10}:t_8,d,l_1+v,l_1+2v),$
$Pace(t_8:t_7,d,l_1+2v,l_2),\}$)
WET($save,38$), **Feasible**($save,6$)

We see in step 7 above that now Dudley believes he will be at $l_1 + v$ by time 8, having taken the first pace toward Nell during the one second between times 7 and 8. His actions continue, until by step 47 he has saved Nell.

4 Conclusion and Future Work

Our efforts thus far are preliminary evidence that a logic-based real-time planner is feasible. Much more needs to be done, especially regarding multiple/competing plans and interacting subplans. Our next effort involves allowing Dudley two possible means of saving Nell, and he must find them and choose between them while also taking this time spent into consideration.

5 Acknowledgement

The authors wish to thank James Hendler for helpful comments.

A AXIOMS

To facilitate reasoning with triplets, some axioms are given in two forms.

Axioms related to moving:

1. $Run(T : T + (L_2 - L_1)/V_Y, Y, L_1 : L_2) \rightarrow$
   $At(T + (L_2 - L_1)/V_Y, Y, L_2)$.
2. $condition(Run(T_1 : T_2, Y, L_1 : L_2), At(T_1, Y, L_1))$.
3. $result(Run(T_1 : T_2, Y, L_1 : L_2), At(T_2, Y, L_2))$.
4. $Pace(T : T + 1, Y, L_1 : L_1 + V_Y) \wedge$
   $Pace(T + 1 : T + 2, Y, L_1 + V_Y : L_1 + 2V_Y) \wedge \ldots$
   $\wedge Pace(T + k - 1 : T + k, Y,$
   $L_1 + (k - 1)V_Y : L_1 + kV_Y)$
   $\rightarrow Run(T : T + k, Y, L_1 : L_1 + kV_Y)$.
5. $condition(Pace(T : T + 1, Y, L : L + V_Y),$
   $At(T, Y, L))$.
6. $result(Pace(T : T + 1, Y, L : L + V_Y),$
   $At(T + 1, Y, L + V_Y))$.
7. $At(T_1 : T_2, Y, L_1) \rightarrow \neg At(T_1 : T_2, Y, L_2)$.

Axioms related to untying and releasing:

1. $Pull(T : T + 1, X, L) \rightarrow Out_of_danger(T + 1, X, L)$.
2. $condition(Pull(T : T + 1, X, L), Not_tied(T, X, L))$.
3. $result(Pull(T : T + 1, X, L),$
   $Out_of_danger(T + 1, X, L))$.
4. $Release(T : T + 3, Y, X, L) \rightarrow Not_tied(T + 3, X, L)$.
5. $condition(Release(T : T + 3, Y, X, L), At(T, Y, L))$.
6. $result(Release(T : T + 3, Y, X, L), Not_tied(T + 3, X, L))$.
7. $Untie_1(T : T + 1, Y, X, L)$
   $\wedge Untie_2(T + 1 : T + 2, Y, X, L)$
   $\wedge Untie_3(T + 2 : T + 3, Y, X, L)$
   $\rightarrow Release(T : T + 3, Y, X, L)$.
8. $condition(Untie_1(T : T + 1, Y, X, L), At(T, Y, L))$.
9. $result(Untie_1(T : T + 1, Y, X, L), Succ_u_1(T + 1))$.
10. $condition(Untie_2(T : T + 1, Y, X, L),$
    $At(T, Y, L) \wedge Succ_u_1(T))$.
11. $result(Untie_2(T : T + 1, Y, X, L), Succ_u_2(T + 1))$.
12. $condition(Untie_3(T : T + 1, Y, X, L),$
    $At(T, Y, L) \wedge Succ_u_2(T))$.
13. $result(Untie_3(T : T + 1, Y, X, L),$
    $Succ_u_3(T + 1) \wedge Not_tied(T + 1, X, L))$.
14. $Tied(T_1 : T_2, X, L) \rightarrow \neg Not_tied(T_1 : T_2, X, L)$.

B INFERENCE RULES

1. Agent looks at the clock

$$\frac{i : \ldots}{i+1 : \ldots, \mathbf{Now}(i+1)}$$

2. Forms the first partial plan

$$\frac{i : \mathbf{Goal}(G)}{i+1 : \mathbf{Ppl}(p, i+1, \{G\}), \mathbf{feasible}(p, i)}$$

3. Finds a triplet whose result matches the goal

$$\frac{i : \mathbf{Ppl}(p, i, \{G\}), \mathbf{result}(A, G), \mathbf{condition}(A, C_A)}{i+1 : \mathbf{Ppl}(p, i+1, \left\{ \begin{bmatrix} C_A \\ A \\ G \end{bmatrix} \right\})}$$

4. Finds a triplet whose action matches the goal

$$\frac{i : \mathbf{Ppl}(p, i, \{G\}), \mathbf{result}(G, R_G), \mathbf{condition}(G, C_G)}{i+1 : \mathbf{Ppl}(p, i+1, \left\{ \begin{bmatrix} C_G \\ G \\ R_G \end{bmatrix} \right\})}$$

5. Computes WET and checks if feasible

$$\frac{i : \mathbf{Ppl}(p, i, \{\ldots\}), \mathbf{Deadline}(Ddl), \mathbf{WET}(i) + i \leq Ddl}{i+1 : \mathbf{Feasible}(p, i)}$$

6. Computes WET and checks if unfeasible

$$\frac{i : \mathbf{Ppl}(p, i, \{\ldots\}), \mathbf{Deadline}(Ddl), \mathbf{WET}(i) + i > Ddl}{i+1 : \neg\mathbf{Feasible}(p, i)}$$

7. Observations become instant beliefs

$$\frac{i : \ldots}{i+1 : \mathbf{Facts}(\ldots, \alpha); \alpha \in \mathbf{OBS}(i+1)}$$

8. Modus Ponens

$$\frac{i : \ldots, \alpha, \alpha \rightarrow \beta}{i+1 : \ldots, \beta}$$

9. Inheritance

$$\frac{i : \ldots, \alpha}{i+1 : \ldots, \alpha}$$

if α is not $\mathbf{Now}(i)$.

10. Projection

$$\frac{i : \mathbf{Context_set}(i), \mathbf{Proj}(i)}{\begin{array}{c} i+1 : \mathbf{Proj}(i+1) = \{X(S : R, \ldots) \mid X(S : F, \ldots) \in \mathit{Context_set}(i), \\ F \leq R, \mathit{Context_set}(i) \not\vdash \neg X(S : R, \ldots); S : R \text{ is the maximum such interval.}\} \end{array}}$$

11. Satisfy a condition for an action by looking at projection

$$\frac{i : \mathbf{Ppl}(p, i, \left\{ \ldots \begin{bmatrix} C_A \\ A \\ R_A \end{bmatrix} \ldots \right\}), C_A \in \mathbf{Proj}(i)}{i+1 : \mathbf{satisfied}(C_A)}$$

12. Satisfy condition using an axiom

$$\frac{i : \mathbf{Ppl}(p, i, \left\{ \ldots \begin{bmatrix} C_A(T : \ldots) \\ A \\ R_A \end{bmatrix} \ldots \right\}), C_A \notin \mathbf{Proj}(i), B_1, \ldots, B_k(T' : T^* \ldots) \rightarrow C_A(T^* : \ldots); T^* \leq T}{i+1 : \mathbf{Ppl}(p, i+1, \left\{ \ldots \begin{bmatrix} C_{B_1} \\ B_1 \\ R_{B_1} \end{bmatrix} \ldots \begin{bmatrix} C_{B_k} \\ B_k \\ R_{B_k} \end{bmatrix} \begin{bmatrix} C_A \\ A \\ R_A \end{bmatrix} \ldots \right\}), \mathbf{satisfied}(C_A)}$$

13. Perform a primitive action

$$\frac{i : \mathbf{Ppl}(p, i, \left\{ \begin{bmatrix} C_A \\ A \\ R_A \end{bmatrix} \ldots \right\}), \mathbf{primitive}(A), C_A \in \mathit{OBS}(i); \text{ or } C_A \in \mathbf{Proj}(i) \text{ and } \mathbf{Facts}(i) \not\vdash \neg C_A}{i+1 : \mathbf{Ppl}(p, i+1, \{\ldots\})}$$

14. Refine a non-primitive action when its condition is satisfied

$$\frac{i : \mathbf{Ppl}(p, i, \left\{ \ldots \begin{bmatrix} C_A \\ A \\ R_A \end{bmatrix} \ldots \right\}), \mathbf{satisfied}(C_A), Q_1 \wedge \ldots \wedge Q_k \rightarrow A}{i+1 : \mathbf{Ppl}(p, i+1, \left\{ \ldots \begin{bmatrix} C_{Q_1} \\ Q_1 \\ R_{Q_1} \end{bmatrix} \ldots \begin{bmatrix} C_{Q_k} \\ Q_k \\ R_{Q_k} \end{bmatrix} \ldots \right\})}$$

15. A formula from the projection becomes a fact

$$\frac{i : \mathbf{Facts}(i), \mathbf{Proj}(i)}{\begin{array}{c} i+1 : \mathbf{Facts}(i+1) = \mathbf{Facts(i)} \cup X(T_1 : T_2, \ldots), \\ \text{if } X \in \mathbf{Proj}(i), T_2 < i, \text{and } \mathbf{Facts(i)} \not\vdash \neg X(T_1 : T_2, \ldots) \end{array}}$$

References

[Allen and Koomen, 1983] Allen, J. and Koomen, J. 1983. Planning using a temporal world model. In *Proceedings IJCAI-83*, pages 741–747.

[Boddy and Dean, 1989] Boddy, M. and Dean, T. 1989. Solving time-dependent planning problems. In *Proceedings of IJCAI-89*, pages 979–984, Detroit, Michigan.

[Charniak and McDermott, 1985] Charniak, E. and McDermott, D. 1985. *Introduction to artificial intelligence.* Addison-Wesley, Reading, Mass.

[Davis, 1988] Davis, D. E. 1988. Inferring ignorance from the locality of visual perception. In *Proceedings, AAAI88.*

[Dean and Boddy, 1988a] Dean, T. and Boddy, M. 1988. Reasoning about partially ordered events. *Artificial Intelligence*, 36(3):375–399.

[Dean and Boddy, 1988b] Dean, T. and Boddy, M. 1988. An analysis of time-dependent planning. In *Proceedings, AAAI88*, pages 49–54.

[Dean and McDermott, 1987] Dean, T. and McDermott, D. 1987. Temporal data base management. *Artificial Intelligence*, 32(1):1–55.

[Dean *et al.*, 1988] Dean, T., Firby, R. J., and Miller, D. 1988. Hierarchical planning involving deadlines, travel time and resources. *Computational Intelligence*, 4:381–389.

[Dean, 1984] Dean, T. 1984. Planning and temporal reasoning under uncertainty. In *IEEE Workshop on Principles of Knowledge based Systems*, Denver, Colorado.

[Dean, 1987] Dean, T. 1987. Intractability and time-dependent planning. In *Reasoning about Actions and Plans*, pages 245–266. Morgan-Kaufmann, Los Altos, CA.

[Drapkin *et al.*, 1987] Drapkin, J., Miller, M., and Perlis, D. 1987. Life on a desert island. In *Proc. Workshop on The Frame Problem in Artificial Intelligence*, pages 349–357. American Association for Artificial Intelligence.

[Elgot-Drapkin and Perlis, 1990] Elgot-Drapkin, J. and Perlis, D. 1990. Reasoning situated in time: basic concepts. *Journal of Experimental and Theoretical Artificial Intelligence.*

[Elgot-Drapkin, 1988] Elgot-Drapkin, J. 1988. *Step-Logic: Reasoning situated in time.* PhD thesis, Univ. of Maryland.

[Georgeff and Lansky, 1988] Georgeff, M. and Lansky, A. 1988. Reactive reasoning and planning. In *Proceedings AAAI88*, pages 677–682.

[Haas, 1985] Haas, A. 1985. Possible events, actual events, and robots. *Computational Intelligence*, 1.

[Hendler *et al.*, 1990] Hendler, J., Tate, A., and Drummond, M. 1990. Systems and techniques: AI planning. *AI magazine*, 11(2):61–77.

[Horvitz *et al.*, 1989] Horvitz, E., Cooper, G., and Heckerma, D. 1989. Reflection and action under scare resources: Theoretical principles and empirical study. In *Proceedings of IJCAI-89*, pages 1121–1127, Detroit, Michigan.

[Horvitz, 1988] Horvitz, E. J. 1988. Reasoning under varying and uncertain resource constraints. In *Proceeding, AAAI88*, pages 111–116.

[Kanazawa and Dean, 1989] Kanazawa, K. and Dean, T. 1989. A model for projection and action. In *Proceedings of IJCAI-89*, pages 985–990.

[Kautz, 1986] Kautz, H. 1986. The logic of persistence. In *Proceedings, AAAI86*, pages 401–405.

[Kraus and Perlis, 1989] Kraus, S. and Perlis, D. 1989. Assessing others' knowledge and ignorance. In *Proc. of the 4th International Symposium on Methodologies for Intelligent Systems*, pages 220–225.

[Kraus *et al.*, 1990] Kraus, S., Nirkhe, M., and Perlis, P. 1990. Planning and acting in deadline situations. To be presented in the AAAI-90 Workshop on Planning in Complex Domains.

[Lansky, 1986] Lansky, A. 1986. A representation of parallel activity based on events, structure, and causality. In *Reasoning about Actions and Plans*, pages 123–159. Morgan-Kaufmann, Los Altos, CA.

[Lansky, 1988] Lansky, A. 1988. Localized event-based reasoning for multiagent domains. *Computational Intelligence*, 4:319–340.

[McDermott, 1978] McDermott, D. 1978. Planning and acting. *Cognitive Science*, 2:71–109.

[McDermott, 1982] McDermott, D. 1982. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6.

[McDermott, 1987] McDermott, D. 1987. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33:379–412.

[Russell and Wefald, 1989] Russell, S. and Wefald, E. 1989. Principles of metareasoning. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning.* Morgan-Kaufman.

[Wilensky, 1983] Wilensky, R. 1983. *Planning and understanding.* Addison Wesley, Reading, Mass.

Toward an Experimental Science of Planning

Pat Langley and **Mark Drummond***
AI Research Branch, Mail Stop 244–17
NASA Ames Research Center
Moffett Field, CA 94035 USA

Abstract

In this paper we outline an experimental method for the study of planning. We argue that experimentation should occupy a central role in planning research, identify some dependent measures of planning behavior, and note some independent variables that can influence this behavior. We also discuss some issues of experimental design and different stages that may occur in the development of an experimental science of planning.

1. Experimentation in Planning Research

Many sciences, such as physics and chemistry, attempt to integrate theory and experiment. For instance, theoretical physicists make predictions that are tested by experimental physicists, and when prediction and observation differ, the theory must be revised. Such cooperation between theoretician and experimentalist is a sign of a field's maturity, and it should be encouraged whenever possible.

At first glance, AI work on planning may appear inherently different from the natural sciences. Because researchers study artifacts over which they have complete control, one might think there is no need for experimentation and that formal analysis should suffice. But this view ignores the fact that all theories rely on assumptions that may or may not hold when applied to actual algorithms or real-world domains. Testing theoretical predictions through experiments lets one gather evidence in favor of correct assumptions, and it can point toward modifications when assumptions prove faulty. Long-term progress in planning will depend on such interaction between the theoretical and experimental paradigms.

Also, the complexity of most planning methods makes it difficult to move beyond worst-case analyses, suggesting experimentation as the only practical approach to obtaining average-case results. Thus, the field promises to have a significant empirical component for the foreseeable future. And unlike some empirical sciences, such as astronomy and sociology, planning is fortunate enough to have control over a wide range of factors, making experimentation easy and profitable.

In any science, the goal of experimentation is to better understand a class of behaviors and the conditions under which they occur. Ideally, this will lead to empirical laws that can aid the process of theory formation. In our field, the central behavior is planning, and the conditions involve the algorithm employed and the environment in which planning occurs. An implemented planning algorithm is necessary but not sufficient; one should also attempt to specify both when it operates well and the reasons for its behavior. Experimentation can provide evidence on both these issues.

As normally defined, an *experiment* is a study in which one systematically varies one or more *independent* variables and examines their effect on some *dependent* variables. Thus, a planning experiment involves more than running a planning algorithm on a single problem; it involves a number of runs carried out under different conditions. In each case, one must measure some aspect of planning behavior for comparison across the different conditions. Below we consider some dependent and independent variables that are relevant to planning research. We then turn to broader issues in designing experiments and in developing an experimental science of planning.[1]

2. Dependent Measures of Behavior

To evaluate any planning system, one needs some measures of its behavior. In most experiments, these are the dependent variables that one would like to predict. There are two obvious classes of metrics for planning algorithms – the *quality* of the generated plans and the *effort* required to generate them.

There exist many variations on the notion of plan quality. In a classical planning framework, one might

*Also affiliated with Sterling Federal Systems.

[1] For other discussions of experimentation in AI, see Kibler and Langley (1988) and Cohen and Howe (1988).

simply measure the length of the solution path or the total number of actions. More sophisticated dependent variables involve the time taken to execute a plan, the energy required, or the use of other resources. Alternatively, one can examine the robustness of a plan, as would be characterized by its ability to respond well under changing or uncertain conditions.

However, in many domains, finding any plan at all requires significant search, making it important to measure the time or effort spent in generating a plan. Measures of this sort have predominated in recent experimental studies of learning in planning domains (e.g., Minton, 1990; Iba, 1989). The simplest measure involves the total CPU time, but this metric can depend on both machines and implementations. More appropriate measures include the number of nodes considered in a search tree (Minton, 1990; Mooney, 1989), the number of unifications required (Allen & Langley, 1990), and the number of subgoals generated during the planning process (Jones, 1989). Of course, such internal measures are less interesting for intelligent agents that interact with an external environment; in such cases, measures of overall external time for planning and execution become relevant, despite possible differences in hardware.

Most measures of plan quality and planning effort implicitly assume that the planner will find a solution to every problem, but this is unrealistic in resource-limited situations. In such cases, the agent may be unable to solve certain problems, and it is important to take this into account when reporting experimental results. One response involves explicitly incorporating this result into the quality measure by giving unsuccessful attempts a very low score. Incorporating these cases into measures of effort is more difficult. As Segre, Elkan, and Russell (1990) have noted, averaging failed problems into effort scores can bias results in favor of one system over another. Alternatively, one can simply report the percentage of solved problems, treating this as a separate dependent measure.

3. Comparative Studies of Planning

Informal comparisons among planning algorithms abound in the AI literature, but there are relatively few systematic experiments that examine the behavior of different algorithms on the same problems. However, such comparative studies have an important role to play in developing a well-founded discipline.

3.1 Gross Comparisons of Planning Methods

The simplest form of planning experiment involves comparing the behavior of entirely different algorithms on the same problem or problems. In this case, the independent variable is the particular planning system being used and the dependent variable is one or more of the measures described above. For instance, Sacerdoti compared the behavior of a simple means-ends planner to that of a planner incorporating means-ends analysis and abstraction. More recently, Ruby and Datta (1990) have reported more extensive experiments, comparing these two approaches in terms of nodes searched and length of solution path. One can also imagine experimental comparisons between preplanning and reactive systems, between search-based and case-based methods, and between specific algorithms within the same basic paradigm.

In such comparative studies, it is important to place the systems' behavior in context. To this end, one can usually compare their performance to that of a 'straw man' that uses a simple-minded strategy (e.g., a traditional nonlinear planner) on the same set of problems. If one of the 'advanced' algorithms actually carries out more search or generates lower-quality solutions than this naive approach, this is a cause for concern. Lower bounds of this sort help calibrate the quality of system behavior.

3.2 Parametric Studies of Planning

Gross comparisons between different planning methods have the aura of a competition, in which one method wins and the others lose. However, a science of planning should aim not for simple-minded conclusions but for increased understanding. To this end, researchers should attempt to identify the *reasons* for success or failure on a problem or class of problems, attempting to generalize beyond a specific system and experiment.

This goal requires finer-grained studies of planning algorithms and their behavior. For instance, many systems contain a set of user-specified parameters, and in such cases one can experimentally determine the effect of the parameter settings on system behavior. A number of parameters suggest themselves:

- in preplanning systems, the maximum amount of resources devoted to generating a plan (e.g., limits on time, memory, or search);
- in reactive systems, the frequency at which the agent samples its environment;
- in combined systems, the ratio of deliberation to execution (Maes, in press; Sutton, 1990); and
- in knowledge-intensive systems, the bias toward modifying stored plans versus dynamically constructing new plans.

Ideally, behavior will be 'acceptable' within a wide range of parameter values, with the system's behavior varying slowly as a function of the settings. Hopefully, the same range of values will work across a variety of domains.

A related issue concerns the evaluation function or control scheme that a planning system uses to direct search. If the function contains parameters, then one can examine their relative importance through simple

parametric studies. However, one can also replace the entire control scheme with different ones in an attempt to find improved search methods. For instance, in a case-based system one might compare an existing similarity criterion for indexing knowledge with other approaches, such as Bayesian methods.

3.3 Lesion Studies of Planning Components

Some planning systems contain a number of independent components, and one can study the usefulness of each by removing it from the system. In such a 'lesion' experiment,[2] one runs the system with and without a given component, measuring the difference in performance. If a component does not aid the overall planning process, then it can be removed without undesirable consequences. Some obvious candidates for lesioning include:

- mechanisms for abstraction planning;
- methods for hierarchical planning;
- heuristics for identifying when to replan; and
- techniques for handling special forms of goals.

The above components focus on processes, but one can also imagine lesioning *knowledge* from a system. For example, some planning systems (Wilkins, 1988) incorporate constraints that may narrow the search or improve solution quality, but the influence of these constraints on behavior is an empirical question. Similarly, case-based planning systems draw upon a library of plans (Hammond, 1989) or plan components (Jones, 1989) in the construction of new plans, and one can determine the change in behavior as one adds or removes cases from memory.

One special case of lesion studies focuses on learning, and much of the recent experimental work on planning falls into this area. In this paradigm, one runs a planning system with and without a learning component, then examines differences in performance between the two variants. Allen and Langley (1990), Iba (1989), Minton (1990), Ruby and Kibler (1989), and Shavlik (1990) report evidence that a variety of learning components can improve the behavior of planning systems after sufficient experience in a given domain.

In some cases, researchers have also found negative results; both Iba (1989) and Minton (1990) have shown that naive learning methods can actually degrade planning performance in terms of search required to find solutions. However, rather than abandoning the use of learning methods, both used their results to identify the source of degradation and went on to develop learning methods that improve performance. This work provides an excellent role model for those interested in the experimental study of planning. Kibler and Langley (1988) discuss additional issues that arise in experiments with learning planners, as do Segre et al. (1990).

[2]This approach is common in neuroscience, where researchers excise a specific region of the brain to determine its effect on behavior.

4. Varying the Planning Domain

Seldom will one system always appear superior to another, and this leads naturally to the idea of identifying the conditions under which one approach has better performance than another. To study the effect of the environment on a planning system, one must vary the domain in which it operates. Natural domains, such as path planning for an autonomous vehicle or manipulator planning for an industrial robot arm, are the most obvious because they show real-world relevance. Also, successful runs on a number of different natural domains provide evidence of generality.

The simplest approach to this issue involves designing a set of 'benchmark problems'. To be scientifically useful, each benchmark problem should highlight certain problem attributes to help isolate planners' particular abilities. In addition, a realistic set of benchmark problems can help the scientific community explain its results in terms that can make a difference to those concerned with practical applications. These two goals – fostering scientific comparison and engineering development – place rather different constraints on a set of benchmark problems.

For the purposes of scientific comparison, one must be able to independently vary different task attributes. To achieve this, some benchmark problems should involve *artificial* domains. For situations that involve planning and execution, relevant attributes relate to the initial state specification, the goals, and the domain dynamics. For instance, one might consider the following sorts of task attributes:

- the length of the 'optimal' solution path (e.g., the number of actions in a block-stacking task);
- the effective branching factor (e.g., the number of actions considered for each plan step);
- the complexity of the environment (e.g., the number of obstacles in a navigation task);
- the amount of goal interaction in a planning task;
- the reliability of the domain (e.g., the probability that effectors will have the desired effect); and
- the rate of environmental change not due to the agent's actions.

However the list of task attributes is constructed, the set of representative problems should provide a complete coverage of the task attribute space. Complete coverage will let researchers choose problems from the set that highlight the system capabilities they seek to measure.

The set of task attributes and benchmark tasks should evolve concurrently.

Artificial domains are gaining acceptance with the planning community (e.g., Pollack & Ringuette, in press), since they let researchers systematically study planning behavior across a wide range of situations.[3] Another advantage of artificial domains is that they specify a variety of domain characteristics. In many cases, this lets one determine plans having *optimal* quality, thus establishing upper bounds on a planner's output. One can then compare the plans generated by actual algorithms against these upper bounds. If plan quality approaches this bound, one can also decide whether additional components or extra computation are worth minor improvements in this regard.

For engineering development and technology transfer purposes, tasks that include 'practical' difficulties will be more useful. Domains involving physical output devices such as robot arms and physical input devices such as limit switches will prove more useful in terms of validating particular systems. It is important to include problems in the evolving set of benchmarks that support such engineering evaluation, but discussion of such issues is beyond the scope of this paper.

5. Issues in Experimental Design

Basic experimental method suggests that researchers vary one independent term at a time while holding others constant. However, one can repeat this technique many times to achieve 'factorial' designs that measure dependent variables under all combinations of independent values. Full factorial designs are impractical when many independent variables are involved, but reduced experimental designs are also possible.

The advantage of combinatorial designs is that they let one go beyond the effects of isolated factors and detect *interactions* between independent variables. For instance, one might find that planning method A behaves better than method B in environment X, whereas B fares better than A in environment Y. Alternatively, one might find that two components of a planning method lead to synergy, or that the joint presence of two domain characteristics make planning especially difficult. We anticipate that many of the most interesting results in planning will have this form. The detection of such interactions does more than establish the conditions under which alternative methods should be used; it can also suggest hybrid algorithms.

Another issue in experimental design involves the use of sampling and statistical tests. In the natural sciences, one can never control all possible variables. As a result, researchers must collect multiple observations for each cell in their experimental design, average the resulting values, and use statistical techniques to ensure that conclusions about differences between cells are justified by the data. Although in principle one can control all the factors that influence a planning system, for practical reasons this will seldom be possible, and planning researchers should consider using them as well.

For instance, seldom can one test a planning system on all possible problems from a given domain. Thus, it makes sense to select a random sample, run the system on all problems in this set, and report the mean and variance on dependent measures of interest. In some situations, the effects of the independent variables will be large enough that formal significance tests are not necessary. In other cases, the variances may be sufficiently high that statistics should be invoked. And though exploratory studies are useful, researchers often design experiments with some hypotheses in mind, and whenever possible they should explicitly state and test these hypotheses. In all cases, the experimenter should use caution and common sense in designing his or her experiments and in interpreting the results.

6. An Imaginary Experimental Study

An imaginary example may clarify the nature of planning experiments. Suppose Dr. Calvin has developed a new planning algorithm, OUTSTRIPS, in response to limitations of earlier systems, say an inability to scale to complex problems. In this case, the hypothesis is that the new method will 'outstrip' other systems as task complexity increases. This suggests two independent variables – the algorithm employed and the problem difficulty.

At this point, Dr. Calvin must settle on some measures of difficulty. Rather than using the number of actions in optimal solutions, she favors a more sophisticated metric that incorporates the idea of goal interaction.[4] She also decides to study the systems' behaviors in multiple domains, say an idealized manipulation task like the blocks world and an idealized navigation task. Similar results in multiple domains will lend credence to her findings, so she includes this as a third independent variable.

Calvin must also identify the dependent measures she plans to use, and the explicit hypotheses she hopes to test. Naturally, she is interested in solution quality, which she will measure as the number of actions in the final plan, but she is even more interested in planning effort. Calvin has implemented OUTSTRIPS on her new positronic hardware, but she must run the comparison algorithms (including a straw man) on archaic silicon machines. Since all the systems involved in the study define their search spaces in a similar manner, she decides

[3] One can also view resource limitations (e.g., time or energy) as independent variables that affect task difficulty. Experimental studies of 'anytime' algorithms (Dean & Boddy, 1988) might examine the effect of planning time on quality of the resulting plans.

[4] Jones (1989) provides an initial approach to measuring goal interaction for means-ends systems.

to use the number of expanded nodes as her measure of effort.

In carrying out her experiment, the researcher must randomly select from problems at each level of difficulty, since the number of possible problems increases rapidly with difficulty. However, Calvin is careful to use the same test problems for each system. For each problem, she measures the various systems' search and plan quality, recording the mean and variance for each system-difficulty combination. She follows this procedure in each of the planning domains selected for study.

In this case, let us suppose that, for each domain, OUTSTRIPS requires more search than its competitors on simple tasks, but that it expands considerably fewer nodes on difficult problems, with the gap widening as the difficulty increases. These results constitute evidence in favor of the original hypothesis that OUTSTRIPS scales better than other methods. However, Calvin also notes that her system's plan quality is slightly worse than that for the more expensive algorithms. As expected, she also notes that all systems perform better than the straw man, except on the simplest problems.

In response to these findings, Calvin designs a lesion study in an attempt to identify the particular constraints used by OUTSTRIPS that lead to its superiority. To this end, she repeats the above experiment with lesioned versions of her algorithm, finding that some constraints greatly reduce planning effort, but that one of them is partly responsible for decrements in plan quality. As a result, Calvin has not only arrived at a deeper understanding of her system's success (and how its constraints might be transferred to other systems); she has also determined that deletion of one component actually produces a superior system with respect to plan quality. Of course, this is not the end of the story, for additional experiments by other researchers may identify conditions under which OUTSTRIPS fares poorly, suggesting ideas for even better algorithms.

7. Toward an Experimental Science

Different goals are appropriate for different stages of a developing experimental science. Although planning work remains in the early steps of this evolution, it is worthwhile considering the states that may arise on the path toward a mature scientific discipline.

In the initial stages, researchers should be satisfied with qualitative regularities that show one method as better than another under certain conditions, or that show one environmental factor as more devastating to a certain algorithm than another. Experimental evaluations should become the norm for published papers, with researchers comparing new algorithms against well-tested systems that act as 'straw men'. Parametric and lesion studies should examine the contributions of specific components, leading to improved algorithms that build on limitations identified earlier. Comparative studies that examine different algorithms on the same domains should proliferate, not to show one method superior to another, but to suggest directions for improvement. Online libraries of representative domains should encourage such comparisons.

Later stages of planning research should move beyond qualitative conclusions, using experimental studies to direct the search for quantitative laws that can actually predict performance on unobserved situations. In the longer term, results of this sort should lead to theoretical analyses that explain such effects at a deeper level, using average-case methods rather than worst-case assumptions. For instance, Segre et al. (1990) outline a simple mathematical model of search in planning, which they propose to use in analyzing experimental results. Other researchers should follow this lead, aiming for robust theories of planning algorithms that predict behavior in novel experimental situations. Failed predictions should lead in turn to revised theories, in the same fashion that experiment and prediction interact in the natural sciences.

In summary, the planning field has already started its development toward an experimental science, and future advances should produce improved dependent measures, better independent variables, more useful experimental designs, and ultimately an integration of theory and experiment. However, even the earliest qualitative stages of an empirical science can strongly influence the direction of research, identifying promising methods and revealing important roadblocks. Research on planning is just entering this first stage, but we believe the field will progress rapidly once it has started along the path of careful experimental evaluation.

Of course, the potential benefits of experimentation do not mean that empiricists should report gratuitous experiments any more than theoreticians should publish vacuous proofs. Whether they lead to positive or negative results, experiments are worthwhile only to the extent that they illuminate the nature of planning mechanisms and the reasons for their success or failure. Although experimental studies are not the only path to understanding, we feel they constitute one of planning's brightest hopes for rapid scientific progress.

Acknowledgements

We would like to thank John Allen for useful comments on an earlier draft. Parts of this paper are similar to an earlier manuscript on research in machine learning, co-authored with Dennis Kibler, who has greatly influenced our ideas on experimentation.

References

Allen, J., & Langley, P. (1990). *The acquisition, organization, and use of plan memory* (Technical Report). Moffett Field, CA: NASA Ames Research Center, AI Research Branch.

Cohen, P. R., & Howe, A. E. (1988). How evaluation guides AI research. *AI Magazine*, *9*, 35–43.

Dean, T., & Boddy, M. (1988). An analysis of timedependent planning. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 49–54). St. Paul, MN: Morgan Kaufmann.

Hammond, K. J. (1989). Case-based planning: Viewing planning as a memory task. In B. Chandrasekaran (Ed.), *Perspectives in artificial intelligence*. Boston: Academic Press.

Iba, G. A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning*, *3*, 285–317.

Jones, R. (1989). *A model of retrieval in problem solving*. Doctoral dissertation, Department of Information & Computer Science, University of California, Irvine.

Kibler, D., & Langley, P. (1988). Machine learning as an experimental science. *Proceedings of the Third European Working Session on Learning* (pp. 81–92). Glasgow: Pittman.

Maes, P. (in press). How to do the right thing. *Connection Science*.

Minton, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, *42*, 363–391.

Mooney, R. (1989). The effect of rule use on the utility of explanation-based learning. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 725–730). Detroit, MI: Morgan Kaufmann.

Pollack, M. E., & Ringuette, M. (in press). Introducing the Tileworld: Experimentally evaluating agent architectures. *Proceedings of the Eighth National Conference on Artificial Intelligence*. Cambridge, MA: AAAI Press.

Ruby, D., & Datta, P. (1990). *Reacting to interactions in abstract plans*. Unpublished manuscript, Department of Information & Computer Science, University of California, Irvine.

Ruby, D., & Kibler, D. (1989). Learning subgoal sequences for planning. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 609–614). Detroit, MI: Morgan Kaufmann.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, *5*, 115–135.

Segre, A., Elkan, C., & Russell, A. (1990). *On valid and invalid methodologies for experimental evaluations of EBL* (Technical Report 90–1126). Ithaca, NY: Cornell University, Department of Computer Science.

Shavlik, J. W. (1990). Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, *5*, 39–70.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Proceedings of the Seventh International Conference on Machine Learning* (pp. 216–224). Austin, TX: Morgan Kaufmann.

Wilkins, D. E. (1988). *Practical planning: Extending the classical AI planning paradigm*. San Mateo, CA: Morgan Kaufmann.

Localized Search for Controlling Automated Reasoning

Amy L. Lansky

Sterling Federal Systems
NASA Ames AI Research Branch
MS 244-17, Moffett Field, CA 94035

Abstract

This paper describes the localized search mechanism of the GEMPLAN multiagent planner. Both a formal complexity analysis and empirical results are provided, demonstrating the benefits of localized search. Localized search utilizes an explicit domain decomposition to infer constraint localization semantics and, as a result, to enable the decomposition of the planning search space into a set of spaces, one for each domain region. Each search tree is concerned with the construction of a region plan that satisfies regional constraints. Shifts between search trees are guided by potential regional interactions as defined by the domain's structure. The search algorithm must also ensure that all search trees are consistent, which is especially difficult in the case of regional overlap. Benefits of localization include a smaller and cheaper overall search space and heuristic guidance in controlling search. Such benefits are critical if current planning technologies are to be scaled up to large, complex domains. Indeed, the use of domain localization and localized search are broadly applicable techniques that can be used by many kinds of domain reasoning systems, not just planners.

1 Introduction

By now, the algorithmic complexity of domain-independent planning has become well known [2]. Many planning researchers have given up completely on preplanning frameworks for more reactive action generation strategies. Yet, there are many domains for which purely reactive approaches are inadequate. Imagine, for example, a factory shop floor in which people coordinate their activities simply by dynamically "reacting" to one another. The shop floor would soon become a mess. Some amount of preplanning is necessary for domains that require complex coordination of activities, especially when adherence to coordination constraints is critical. Such domains are numerous and include NASA mission planning[1], building-construction planning, factory planning, and planning of defense-related activities. Given the inescapable need for reasoning about large complex plans, the planning community faces two related obstacles: (1) the inherent costliness of plan construction algorithms and (2) the problem of scaling planning systems up to large domains. Indeed, these obstacles pose problems for any form of planning, whether it is performed before or during execution.

The focus of this paper is the use of *locality* — the inherent structural properties of a domain — to control the explosive cost of planning and other forms of reasoning. The use of localized reasoning, while quite intuitive and natural, has not been a fundamental aspect of most AI systems. A localized domain description is one that is explicitly decomposed into a set of regions. Each region may be viewed as a subset of domain activity with an associated set of "constraints" (properties, goals, or other requirements that the planner must fulfill) that are applicable only to the activities within the region. We refer to this delineation of constraint applicability as *constraint localization. Localized planning* consists of searching a set of smaller, regional planning search spaces rather than a large, "global" space. A GEMPLAN search space may be visualized as a plan-construction search tree, where each tree node is associated with a plan and each arc is associated with a constraint algorithm that transforms the preceding plan into a new plan (with new actions, relationships, etc.) that satisfies the constraint.

The GEMPLAN domain representation allows for the specification of any kind of domain decomposition, including the use of regions that overlap, are disjoint, are organized hierarchically, or form any combination thereof. Criteria for decomposition are usually suggested

[0]This research has been made possible in part by the National Science Foundation, under Grant IRI-8715972.

[1]Throughout the rest of this paper, the term "planning" will be used rather than "planning and scheduling." However, most of the discussion is equally applicable to the more specialized area of scheduling.

by the innate characteristics of a domain, such as its physical structure, its behavioral entities (agents), and its functional elements. Indeed, it is often useful to utilize a decomposition that reflects several criteria simultaneously. Consider, for example, a building-construction domain. Viewed globally, the domain may be described by a set of constraints, some of which describe the actual structure and requirements for a specific building, some of which encode the requirements and capabilities of contractors and physical resources, and some which describe the inherent limitations imposed by domain physics. Clearly, many of these constraints apply only to a subset of the full set of construction activities to be planned. One way to naturally decompose the domain is according to the physical structure of the building – e.g., to utilize separate regions to model each floor or room and its associated constraints. Other regions could model the individual contractors and their constraints. Figure 1 depicts a possible decomposition for a small construction domain.

The primary goal of domain localization is to cluster activities into regions so that constraints are applied as narrowly as possible. The actual decomposition chosen will be used to infer the exact scope of applicability of domain constraints – i.e., each region's constraints apply only to the activities within that region. As we will demonstrate, different localization decompositions will incur different planning costs. While most of our empirical tests to date have utilized user-provided decompositions, we are currently developing an algorithm for automatically learning a good decomposition for a particular domain as well as more general decomposition strategies.

The use of localized reasoning has several benefits. From a representational point of view, locality provides a solution to some aspects of the frame problem; constraint localization may be viewed as a frame rule which limits the effect of actions and properties upon one another [3,5,6]. Most important, however, locality provides a rationale for partitioning a potentially explosive global planning space into a set of smaller, localized planning spaces. This has three interrelated benefits: (1) both the size and cost of the union of a set of localized planning spaces is usually smaller than that of a global, non-localized space; (2) expensive planning algorithms need be applied to much smaller regional plans; and (3) since a localized domain description provides information about how constraints and activities interact, it serves as a heuristic for constraint application. In particular, only relevant (regional) constraints are applied to regional plans and movement between regional search trees occurs only when regions interact. All of these factors clearly facilitate scaling up to large domains. Other planning researchers have also looked at related methods of problem decomposition in order to reduce search complexity [1,4], but these have focussed primarily on goal reduction and operator reformulation rather than search space decomposition.

It should be pointed out that domain localization is applicable to *any* kind of domain reasoning that can be effectively partitioned. Nearly all domains have some inherent structure that can be exploited. For example, localized reasoning could be used by single-agent planners, reactive systems, schedulers, truth maintenance systems, learning systems, image understanding systems[2] — indeed, many reasoning algorithms already utilize heuristics that are provided by domain structure. Our localized search algorithm can thus be applied to many kinds of reasoning systems. However, we do stress multiagent domains here for two reasons: (1) the complexity of coordinating multiagent domains makes localization even more necessary; (2) multiagent domains are typically easy to decompose.

Of course, the benefits of localized search do have a price. From a practical point of view, domains can almost never be partitioned into simple hierarchies or disjoint regions. Domains of any complexity will have regions that "overlap" – that is, some subregions will be shared by more than one encompassing region. For example, in Figure 1, regions **wallA**, **e-control**, and **p-control** each belong to more than one region. This complicates the localized search algorithm because changes within a shared region must be reflected within the search trees of all its ancestors. That is, localized search must pay attention to the problem of *interaction and consistency* among search trees.

In addition, constraint localization will yield large gains only if a domain can be effectively decomposed. If many constraints naturally belong to a region that includes a great deal of domain activity, search will remain quite expensive. To gain real efficiency benefits, regions which may seem intuitively "global" should be composed to include only a minimal amount of activity. For example, in Figure 1, the general contractor's constraints apply only to his/her own activities in **gc-control**, those in **e-control** (electrician control activities), and those in **p-control** (plumber control activities), not to *all* activities in **electrician** and **plumber**. Experience thus far with GEMPLAN (and commonsense intuition about the structure and function of large organizations) indicates that effective localization is natural to obtain.

[2] A Paris-based firm, Framentec, is building a localized image-understanding/plan-recognition system based on the GEMPLAN formalism.

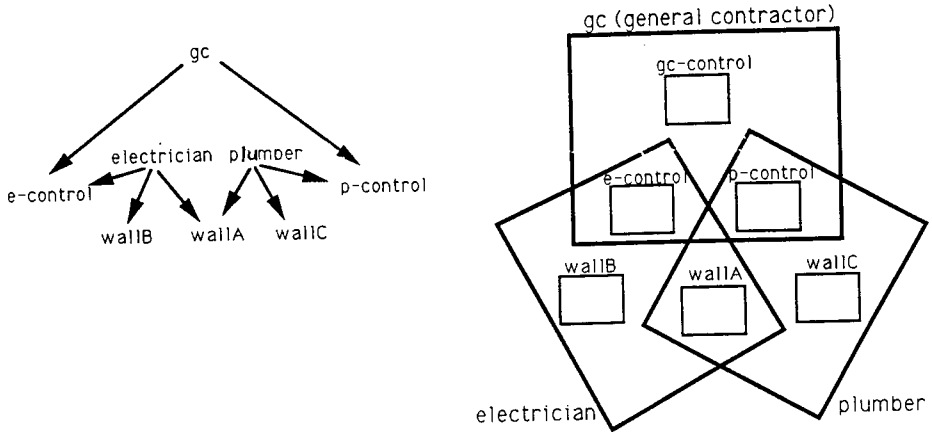


Figure 1: A Localized Construction Domain Description

2 GEMPLAN Overview

GEMPLAN is a planner designed explicitly for multiagent domains that require complex coordination. The current GEMPLAN system is implemented in Prolog on a Sun workstation and has been applied to several test domains: multiagent blocks-world problems, the Tower of Hanoi, and a construction domain. The system includes an execution facility, and has the ability to apply constraints before or during execution. It may thus be viewed as a combined pre-planner/dynamic-planner. While the existing system is primarily designed for pre-planning, we will soon begin implementation of a next-generation GEMPLAN system that spans the pre-planning/dynamic-planning spectrum in a seamless fashion. Our current target applications include large construction domains and data-analysis planning for NASA's Earth Observing System (EOS). GEMPLAN differs from standard hierarchical planners [11,12] in several ways:

▷ GEMPLAN has the ability to satisfy a broad range of domain "constraint forms," not simply the attainment and maintenance of state conditions (the traditional notion of "planning"). The system includes a set of general-purpose constraint satisfaction algorithms for partially-ordered plans, which may be further augmented by user-supplied constraint-satisfaction methods. The default constraint algorithms are fully general – they allow for the addition of and reasoning about any possible temporal relationship between actions, including simultaneity. The current constraint repertoire includes:

- condition attainment and maintenance (based on the modal truth criterion [2]) – i.e., the traditional "planning algorithm." Actions may be defined to have conditional effects. The algorithm also includes full protection capabilities.
- action decomposition (i.e., action hierarchies). GEMPLAN allows for reasoning about actions at mixed levels of detail, rather than confining itself to reasoning "one level at a time," as do some hierarchical planners [12]. Indeed, rather than being inextricably bound to the planner's search structure (hiearchical or otherwise), action decomposition is just another kind of "constraint" to be satisfied by the system, and may be applied at any appropriate time, including at run-time.
- a variety of temporal and causal constraints, including run-time priority constraints.
- attainment of desired patterns of behavior expressed as regular expressions.

It is these constraint satisfaction algorithms that perform the task of plan construction and coordination, by introducing actions, action interrelationships, and variable bindings.

▷ GEMPLAN partitions the global search space into localized search spaces.

▷ GEMPLAN has a highly flexible, tailorable search mechanism. In particular, constraint satisfaction can be guided by user-supplied heuristics and by the changing planning and/or execution context, as it develops.

More details on GEMPLAN appear elsewhere [5,6,7,8,9]. The rest of this paper will focus on GEMPLAN's localized search mechanism. Specific instances of plan construction via constraint satisfaction will be demonstrated in Section 3.

2.1 Search Space Decomposition

As described earlier, a GEMPLAN domain specification is decomposed with the goal of localizing the applicability of constraints as much as possible. For example, the

construction domain depicted in Figure 1 has been partitioned into regions corresponding to the activities of the electrician, plumber, and general contractor. These regions have been further decomposed to include subregions that contain the activities of the electrician and plumber at various walls as well as contractor "control" activities (these might include communication actions or high-level actions that have not yet been expanded into activities at particular walls). Each wall region would be associated with constraints and definitions that are relevant to the actions taking place at that wall. For example, in the case of wallA, these may include constraints relating to coordination of plumber and electrician activities. Each control region might be associated with personal communication and planning constraints for that contractor. The electrician, plumber, and gc region constraints, which apply to all their subregions, might describe more global requirements pertinent to their respective activities. For example, note how the gc constraints apply to all the control regions. These might describe how the general contractor's requests influence the control activities of each subcontractor.

Rather than searching a single global search space, GEMPLAN creates a regional search space for each region. Each search space is concerned with building a plan for its region that satisfies all regional constraints. The planner may thus be viewed as a set of "mini-planners," tied together as dictated by the structural relationships between regions.

2.2 Regions

Let us assume that a domain is specified as a set of regions $R1, ..., Rn$. Each region R is defined by a *region description*:

$$< actions(R), subregions(R), constraints(R), tree(R) > .$$

The set $actions(R)$ consists of the types of actions that may occur directly within R (but not within a subregion of R). The set $subregions(R)$ consists of subregions belonging to R. For each such subregion Ri, we use the notation $Ri \subset R$. The set $constraints(R)$ includes constraints that pertain to activities within R and its subregions. Finally, each region is associated with a plan-construction search tree $tree(R)$.

2.3 Region Search Trees

Figure 2 depicts portions of GEMPLAN planning search trees for the electrician and wallB regions. Each tree reflects search through a space of "plan modification" operations – i.e., it is a plan-construction search space (rather than a domain-state search space). Each tree node is associated with the region plan constructed up to that point in the search, and each tree arc is associated with a plan modification or "fix" that transforms a region plan into a new region plan. Upon reaching a node during planning search, the planner must choose a particular regional constraint to check next. (Thus, an implicit branching factor in the tree is the set of all relevant constraints at each node.) If the chosen constraint is not satisfied by the plan associated with that node, constraint satisfaction algorithms or "fixes" must be applied (there may be several fix algorithms for each constraint, as well as many possible solutions or "fixes" per fix algorithm), resulting in a set of new region plans at the next level down in the tree. A GEMPLAN fix typically adds new actions, relations, and variable bindings to a region plan, and may also generate new subregions. Note that fixes may add actions and relations *anywhere* within the plan it is working on – the precise temporal position is determined by the nature of the constraint and fix.[3]

GEMPLAN uses, by default, a depth-first search strategy for searching its trees, trying constraints and fixes in the order supplied in the domain specification. However, since search should optimally be driven by domain-dependent information and the structure of the plan itself, GEMPLAN allows for flexible user-tuning of tree search. The order in which constraints and fixes are applied can be made context dependent. GEMPLAN also includes a facility that can determine precisely which actions affect which constraints *within* a region. This facility enables *only* relevant constraints to be applied at each step, thereby exceeding the kind of "frame" information already provided by constraint-localization semantics. This coupling of localized search, where only relevant constraints are checked, with further user-tailoring of the search, forms an extremely flexible mechanism of "relevancy-driven-search" – namely, search driven by the most relevant constraints at any particular time in the reasoning process.

2.4 Plan Representation

As stated above, each search tree node is associated with a *region plan*. Each region plan consists of a *local region plan* and a set of *subplans* (the region plans of its subregions). For example, if $R1 \subset R$ and $R2 \subset R$, the region plan for R will include a local region plan for R and region plans for $R1$ and $R2$. GEMPLAN associates all plan information with the smallest region that encom-

[3] For example, unlike some planners (typically, those those perform state-space search), actions need not be added to the plan in an order that is in any way related to the order in which the actions are executed. The fix algorithms may thus be viewed generically as plan modifiers that grow and refine plans in flexible ways.

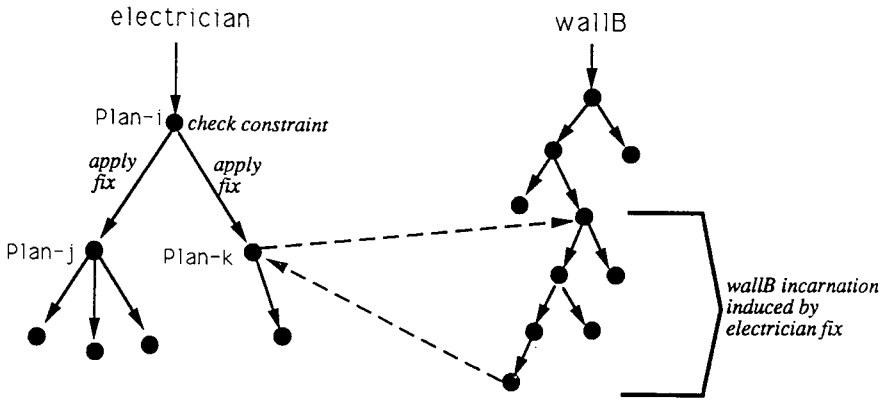


Figure 2: GEMPLAN Search Trees

passes that information. The region plans of $R1$ and $R2$ will thus include all actions, temporal and causal relations, variable bindings, and other plan information that deal exclusively with $R1$ and $R2$, respectively. The local region plan of R will then include plan information that deals specifically with activities in R or that pertains to relationships among R, $R1$, and $R2$ (and therefore cannot be associated strictly with $R1$ or $R2$).

2.5 Guiding Search Among Regional Trees

Search within *tree*(R) is concerned with (1) assuring that all of R's constraints are satisfied by R's region plan and (2) making sure that R's subregions' trees are searched to find a satisfactory plan for their region plans. Referring back to Figure 1, it is the role of *tree*(**electrician**) to make sure that **electrician**'s constraints are satisfied and that *tree*(**wallB**), *tree*(**wallA**), and *tree*(**e-control**) are all visited when their constraints may be affected.

How does control transfer among regional trees? This is done in response to information transmitted to the search mechanism by a fix. Suppose we are in *tree*(R). After applying a fix for one of R's constraints to R's region plan, the fix must return a subset of R's subregions, $R1, ..., Rm$, that may have been affected by the fix. The GEMPLAN search algorithm will then inhibit further search within *tree*(R) until *tree*($R1$)...*tree*(Rm) are all satisfactorily searched. As depicted in Figure 2, if **electrician** affects the subplan for region **wallB** via the introduction of new actions there, search within *tree*(**electrician**) cannot safely proceed until **wallB**'s tree is searched and its constraints are rechecked and satisfied. Notice how shifts between parent and child regions induce a partitioning on the child's search tree. We call these search fragments *incarnations* – search within the child is "reincarnated" each time its constraints are potentially violated due to a fix in its parent's search tree. Each incarnation is thus a subtree initiated by a parent region. In our example, *tree*(**wallB**) may be reincarnated several times due to fixes for **electrician** constraints. Each time *tree*(**wallB**) is revisited, **wallB**'s constraints must be rechecked and satisfied. One restriction on GEMPLAN's search control mechanism is that all search strategies (e.g., breadth-first, dependency-directed, etc.) must be applied within the confines of an individual incarnation. This greatly simplifies the problem of search consistency.

As the reader may have noticed, not all regions are subregions of some enclosing region. In the domain of Figure 1, this is true of **gc**, **electrician**, and **plumber**. To simplify search, GEMPLAN requires that all tree search ultimately flows from some designated "global" regional tree.[4] Although **gc**, **electrician**, and **plumber** do not logically belong to another region as far as constraint applicability, we do need to make sure that some region at least takes "responsibility" for invoking their search trees. Thus, we include the additional relation $\subset_r$ to denote this relationship, and require that each region except some designated "global" region have a "parent" that assumes search responsibility for it. In our example, we shall choose **gc** as the "global" region, with **electrician** $\subset_r$ **gc** and **plumber** $\subset_r$ **gc**. Although *tree*(**gc**) must make sure that *tree*(**electrician**) and *tree*(**plumber**) are visited appropriately, **gc**'s constraints apply only to its region plan, which includes only the subregion plans of **gc-control**, **e-control**, and **p-control**.[5]

[4] This does not preclude the possibility of parallel search of independent subtrees. Our research plans include experimentation with parallel search in GEMPLAN.

[5] Readers of previous papers on GEMPLAN will recall that the GEMPLAN description language includes several types of regions and modes of access between regions (elements, groups, ports, etc.). For the purposes of this paper and for the sake of generality, we simplified the GEMPLAN structural model to include only the relations $\subset$ and $\subset_r$. The semantics of elements, groups, and ports can all be captured in terms of $\subset$ and $\subset_r$.

2.6 Dealing With Regional Overlap

One of the challenges of localized search is keeping all regional search trees consistent with each other. This would be fairly straightforward if domain structure were strictly hierarchical. However, since we allow for regional overlap, some effort is required to keep trees consistent. For example, if a fix in *tree*(electrician) affects region wallA's plan, it is not enough to simply recheck wallA's constraints and return to *tree*(electrician). Region plumber's representation of wallA's subplan must also be updated within *tree*(plumber), and search must also occur within *tree*(plumber) to recheck its constraints. We call this process of maintaining consistency *completion*. Because GEMPLAN allows for quite complex localization structures, the search algorithm must be very careful to perform completion fully and correctly. GEMPLAN must update all affected data structures (in particular, parent tree data) each time it completes searching an incarnation of a shared region. It must also make sure that all affected parent region trees are ultimately reincarnated and that region constraints are rechecked.

3 Example

More complete descriptions of GEMPLAN's search algorithms are provided elsewhere [9,10]. In this section, we attempt to clarify the preceding discussion with an example from the construction domain of Figure 1. Let us assume that the electrician, plumber, wallA, and wallB regions are associated with the following (informally described) constraints:[6]

```
ELECTRICIAN CONSTRAINTS:
(1) action(install-socket(wallA,locA1))
(2) action(install-socket(wallB,locB1))
(3) action(install-socket(wallB,locB2))
(4) decompose(install-socket(W,L),
              {W.electprep(L) => W.insertsocket(L)})

PLUMBER CONSTRAINTS:
(1) action(install-pipe(wallA,locA1))
(2) decompose(install-pipe(W,L),
              {W.plumbprep(L) => W.insertpipe(L)})

WALLA CONSTRAINTS:
(1) (forall L)
    [(forall prep:{electprep(L),plumbprep(L)})
     pattern((prep)*=>)]
(2) fcfs([[electprep,insertsocket],
          [plumbprep,insertpipe]])

WALLB CONSTRAINTS:
(1) all-matching-precede(electprep,insertsocket)
```

[6] Tokens starting with a capital letter denote variables.

The first three electrician constraints require that actions exist in the final plan that install sockets in particular walls and locations. Such `action` constraints simply result in the addition of actions to the plan. The fourth `decompose` constraint requires that each `install-socket(W,L)` action be decomposed into an `electprep` action followed by an `insertsocket` action at wall `W`, location `L`. Note that an action of form `X.Y` denotes an action `Y` occurring at location `X`. The plumber constraints are similar. In this case, only one pipe is to be installed at wallA.[7]

The two wallA constraints pertain to the coordination of the electrician and plumber actions at that wall. The first constraint states that, at wallA, all `electprep` and `plumbprep` actions at the same location follow a certain pattern – they must be totally ordered by the temporal relation =>. The second constraint additionally requires that the electrician and plumber have access to wallA on a first-come-first-serve basis. The constraint description consists of a set of constraint pairs and has the following semantics: any required execution ordering of the first actions in each pair (in this case, required orderings between "prep" actions) will determine the ultimate ordering of the second actions in each pair (in this case, the ordering of `insertsocket` and `insertpipe` actions). Since a total ordering is forced on all "prep" actions at the same location, this will force electricians and plumbers to insert their devices in common locations on a first-come-first-serve basis. Finally, wallB requires that all `electprep` actions precede all `insertsocket` actions. This assures that all electrical wall-prep at wallB will be completed before any electrical components are inserted. At wallA, in contrast, prep and insertion actions may be intermingled, as long as they conform to the two ordering constraints of wallA.

Given these constraints, we will now run through a planning scenario. We will assume that all constraints are imposed strictly in advance of execution. Our discussion will describe the train of reasoning GEMPLAN might go through to create the construction plan depicted in Figure 3.

Reasoning begins at the "global" region gc, which in this case has no constraints of its own, but is responsible for invocation of the electrician and plumber search trees. Let us assume that electrician is invoked

[7] Since this simple scenario does not contain constraints that force the electrician activities (nor plumber activities) to be totally ordered, let us assume, for the sake of realism, that electrician models a set of electricians (and similarly for plumber). In the GEMPLAN construction domain application discussed in Section 5, multiple contractors were indeed used. The planner creates a suitable construction plan given any number of available contractors, performing contractor allocation as planning proceeds.

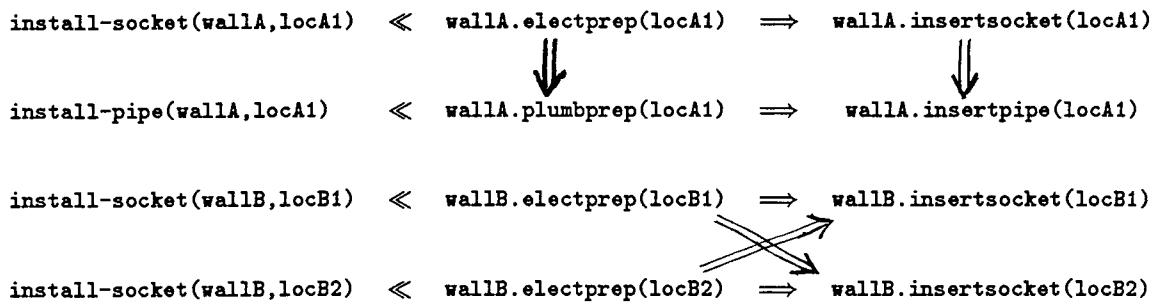


Figure 3: A Construction Plan

first. Constraints 1, 2, and 3, are satisfied by adding the specified `install-socket` actions to the electrician plan. Constraint 4 then decomposes these three actions into the appropriate `electprep` and `insertsocket` actions at wallA and `wallB`. This causes changes in the wallA and wallB subplans of electrician. Before search continues within *tree*(electrician), search within *tree*(wallA) and *tree*(wallB) must occur. Let us assume that wallA is searched first. Both wallA constraints are checked, but both are satisfied. The newly completed incarnation of wallA therefore does not add any new information to the subplan for wallA associated with electrician, but the process of *completion* causes the new version of the wallA plan (that includes the changes made by electrician) to be inserted appropriately into *tree*(plumber).

Then *tree*(wallB) is searched. The wallB constraint causes the relations `electprep(locB1) => insertsocket(locB2)` and `electprep(locB2) => insertsocket(locB1)` to be added. Search then returns to electrician, and the electrician's subplan for wallB is appropriately updated. Note that wallB is not a region of overlap, so no other completion operation need occur.

At this point, all electrician constraints are satisfied. Search then bounces back to gc, which invokes search in *tree*(plumber). The plumber constraints cause the addition of the `install-pipe` action and its decomposition into the appropriate subactions at wallA. After fixing the second plumber constraint, search must occur for the affected wallA region. This causes the actions `electprep(locA1)` and `plumbprep(locA1)` to be forced into some total order (in Figure 3, `electprep(locA1) => plumbprep(locA1)` was chosen) and then, as a result of the second wallA constraint, a similar ordering is imposed on `insertsocket(locA1)` and `insertpipe(locA1)`. The now satisfied wallA plan is appropriately inserted into both *tree*(electrician) and *tree*(plumber) (due to the completion process). All plumber constraints are now satisfied and search bounces back to gc. The constraints within electrician are then rechecked (due to the changes at wallA), but they are still satisfied. Search then terminates successfully.

4 Complexity Analysis

It is clear that no general definitive complexity result can be given for localized search – the size and complexity of the planning search trees for a particular problem will depend on the structure of the domain, the constraints associated within each region, the complexity of their satisfaction algorithms, the domain search heuristics, and the peculiarities of the specific problem itself. In order to provide some theoretical estimate of the benefits of localized search, however, we provide an idealized analysis of search for a domain with a very simple locality structure. We provide best- and worst-case search costs, assuming that constraint algorithms are either all constant, linear, quadratic, or exponential in cost (obviously, most domains will have a mixture of these). Although our analysis is quite idealized, it correlates with the empirical results of Section 5. The reader should also note that, for most of our empirical tests, search has been very close to best-case – i.e., our tests have exhibited very little backtracking. In general, average-case behavior can be expected to be close to best-case behavior if good domain search heuristics are employed.

To formally and empirically assess the benefits of localized search, we must compare it with completely non-localized search. For our formal complexity analysis, we utilize the non-localized and localized domain configurations depicted in Figure 4. For both domains we assume a total of n_c constraints, that each constraint has n_f possible fixes, and that the total number of actions in the final plan is s. The cost of checking any constraint on a plan of size j is $c(j)$ and the cost of fixing a plan of size j is $f(j)$. For the localized case, we assume that the domain has been localized to form a configuration of m subregions $R_1 ... R_m$ and a region G. The actions in the final plan are divided equally among the R_l regions, so

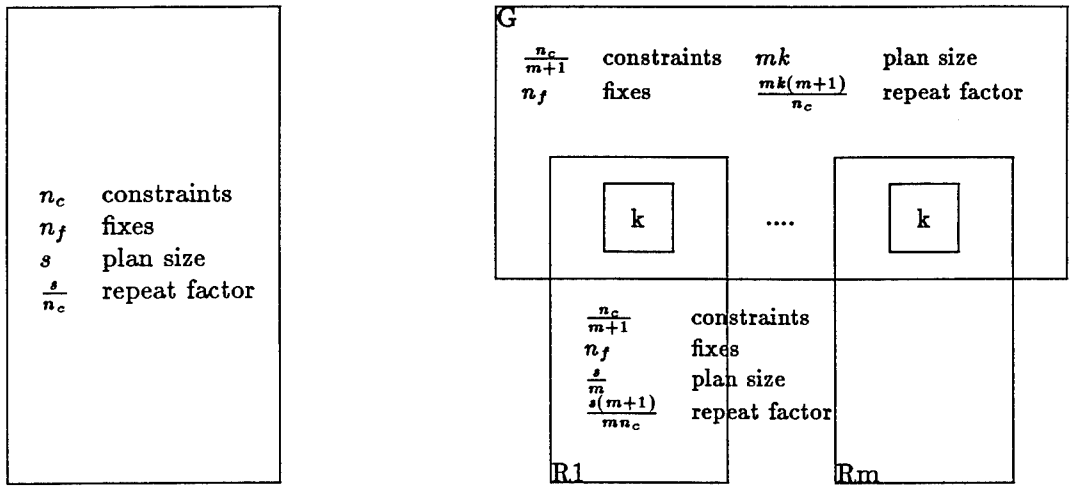


Figure 4: Non-localized and Localized Domains

that each builds a plan of size $\frac{s}{m}$. Each region R_l also contains a subregion consisting of k actions that overlaps with region G. Thus, G's region plan consists of mk actions. The n_c constraints of the original problem are evenly distributed among $G, R_1, ...R_m$ so that each region is associated with $\frac{n_c}{m+1}$ constraints.

Let us now consider the cost of a generic region search tree. Let us assume that, for a region i, there are n_{c_i} constraints, that each constraint has n_f fixes, and that the final size of the region plan is s_i. Because a constraint fix may always, in principle, violate previous constraints that may have been satisfied, constraints may need to be repeatedly checked and fixed. The search thus tends to take the form of a round-robin checking of constraints. We call the number of times the search must cycle through the constraints the search *"repeat factor."* Assuming that our sample region has a repeat factor of r_i, its tree depth is $r_i n_{c_i}$, with average depth to adding an action being $\frac{r_i n_{c_i}}{s_i}$. (Thus, we assume that at most one action is added per fix. In most realistic domains, many actions are often added per fix.)

To calculate search cost, we assume an implicit search space that alternately branches due to choice of a constraint (the costs $c(j)$ accumulated due to constraint testing) and choice of a fix (the costs $f(j)$ accumulated due to constraint fixing). By "best-case search" we mean depth-first search without backtracking – i.e., the cost of one path from the root to the leaves of the search space. The cost of best-case search for region i is

$$\sum_{0 \le j \le s_i} \frac{r_i n_{c_i}}{s_i}(c(j) + f(j)).$$

In contrast, worst-case search cost measures the cost of searching the entire space. For our sample region i this cost is

$$\sum_{1 \le j \le r_i n_{c_i}} n_{c_i}^j n_f^{j-1} c((j-1)\ div \frac{r_i n_{c_i}}{s_i}) + n_{c_i}^j n_f^j f((j-1)\ div \frac{r_i n_{c_i}}{s_i}).$$

We shall now compare the complexity of these formulae for the non-localized and localized cases. For each case, we must assume a repeat factor for each region. In general, this will be a function of the size s_i of the region plan and the number of constraints for that region n_{c_i}. For this analysis, we will set the repeat factor r_i to be $\frac{s_i}{n_{c_i}}$ – that is, we assume that exactly one action is added per fix, that the size of the plan is larger than the number of constraints, and that the depth of the tree is equal to the number of actions in the plan. In most of our test situations, however, the repeat factor tends to be less than this number, with more actions added per fix and, of course, some subset of actions being added by overlapping regions. Moreover, less rechecking needs to be done due to tuning of constraint application. On the other hand, some amount of additional rechecking tends to occur due to the completion process. Thus, our assumption of a repeat factor of $\frac{s_i}{n_{c_i}}$ may be only slightly pessimistic. Given this formulation, the repeat factor for the completely non-localized case will be $\frac{s}{n_c}$. For the lo-

calized case, we have a repeat factor of $\frac{mk}{\frac{n_c}{m+1}} = \frac{mk(m+1)}{n_c}$ for region G and a repeat factor of $\frac{s}{m}/\frac{n_c}{m+1} = \frac{s(m+1)}{mn_c}$ for each region R_l.

The complexities of all cases are summarized in Table 1. We provide best- and worst-case search results, assuming that the complexity of $c(i)$ and $f(i)$ are both constant, linear, quadratic, or exponential. In some cases we supply only the leading term. For all of the localized search cases, we must add to the total cost of the search trees an additional completion cost C. For this idealized analysis, we assume that completion occurs each time an action is added within a region of overlap R. The cost of each completion operation will be a function of the number of additional regions that include R (this will not include the region actually adding the action to R) and the size of the plan data structure for each of those regions (since completion involves the replacement of certain pieces of this data structure). In GEMPLAN, the size of this data structure is a function of the number of regions in the plan – in this case $m+1$. So for this problem, we will assume a completion cost $C = mk(m+1)$ or $O(m^2k)$.

As can be seen in the table, localized search is, in general, always better than global search – in most cases significantly better. The only real exceptions are in the case of constant-complexity best-case search or when the cost of completion overshadows the cost of the search itself. The amount by which localized search wins over global search is proportional to the amount by which s dominates both $\frac{s}{m}$ (the size of each subregion R_l) and mk (the size of G). Thus, increased decomposition is always worthwhile, except for the cost of increased amounts of overlap (which is reflected in the size of mk and the cost of completion C). The overall gains of localized search increase as the complexity and size of the search space increases. As we will show in the next section, our empirical tests on a construction domain have shown universal performance improvement with localized search, with speedups of greater than 50% using a good decomposition.

5 Empirical Results

All of our empirical experiences with GEMPLAN certainly bear out the efficacy of localized search. Our largest application so far is for a building-construction domain. This domain includes multiple instances of each type of contractor as well as multiple walls and footings to which these contractors must be allocated. The problem thus manifests both resource allocation and temporal coordination of access to building components. The application was used to test a variety of localization configurations, including some that were fairly complex, involving both a great deal of hierarchy and overlap.

Table 2 and Table 3 provide timing results for the construction domain (on a SPARC workstation). The "number of regions" column gives the total number of regions that have at least one constraint and one action in the final plan. The "overlap size" column gives a sum of size measures for each region of overlap. For each such region, its "size" is the number of actions in the region multiplied by the number of times it occurs within another region. For instance, in the domain of Figure 1, e-control, p-control, and wallA each occur twice within a parent region. If each region has a total of 2 actions within its plan, the domain's total overlap size would be 12. The overlap size column gives a good idea of how expensive the completion process is. The "largest region" column gives a pair of numbers <*number of constraints,number of actions*> for the region with the largest number of constraints (which, in this case, is usually also the region with the largest number of actions). This measure gives an idea of how big the largest search space in the domain is – i.e., the region space in which the most search will be conducted.

The two tables provide results for the creation of of a 49-action construction plan and a 97-action construction plan. Both used the same basic domain decomposition, with the 97-action plan simply having more walls, contractors, etc. Within each table, the first test case is for a non-localized version of the domain – all constraints are applied globally to the entire plan. The localized(1) test configuration is highly decomposed but also has significant amounts of overlap between regions. The localized(2) case has less localization and much less overlap. Case localized(3) has an intermediate level of both localization and overlap, and attains the best results in both cases. Interestingly, these results jibe with our formal analytical results; increased localization provides increased benefit, except for the added expense caused by with regional overlap. However, notice that, in the 97-action case, localized(1) is faster than localized(2). This shows how, as plan size increases, the cost of dealing with overlap is overshadowed by the shear size of the planning space itself.

complexity of c(i) and f(i)	*Non-Localized (best-case)*	*Localized (best-case)*	*Non-Localized (worst-case)*	*Localized (worst-case)*
constant (b)	$2bs$	$2b(s+mk)+C$	$b(n_c n_f)^d$	$b(m(\frac{n_c n_f}{m+1})^{\frac{s}{m}} + (\frac{n_c n_f}{m+1})^{mk}) + C$
linear (ib)	bs^2	$b(\frac{s^2}{m} + (mk)^2) + C$	$bs(n_c n_f)^s$	$b(s(\frac{n_c n_f}{m+1})^{\frac{s}{m}} + (\frac{n_c n_f}{m+1})^{mk}) + C$
quadratic (i^2)	$\frac{2}{3}s^3$	$\frac{2}{3}(\frac{s^3}{m^2} + (mk)^3) + C$	$d^2(n_c n_f)^s$	$\frac{s^2}{m}(\frac{n_c n_f}{m+1})^{\frac{s}{m}} + (mk)^2(\frac{n_c n_f}{m+1})^{mk} + C$
exponential (b^i)	$2b^s$	$2(mb^{\frac{s}{m}} + b^{mk}) + C$	$(bn_c n_f)^s$	$m(\frac{bn_c n_f}{m+1})^{\frac{s}{m}} + (\frac{bn_c n_f}{m+1})^{mk} + C$

Table 1: Non-Localized and Localized Search Complexity

test case (49 actions)	*number of regions*	*overlap size*	*largest region*	*CPU Seconds*
non-localized	1	0	<40,49>	113.81
localized(1)	24	134	<4,16>	85.78
localized(2)	16	32	<15,28>	79.23
localized(3)	19	76	<8,17>	62.95

Table 2: 49 Action Construction Plan

test case (97 actions)	*number of regions*	*overlap size*	*largest region*	*CPU Seconds*
non-localized	1	0	<52,97>	905.98
localized(1)	37	236	<7,34>	524.97
localized(2)	28	32	<24,58>	725.43
localized(3)	31	102	<8,24>	442.43

Table 3: 97 Action Construction Plan

6 Conclusion

This paper has described a general-purpose technique for localized search, as well as complexity results and empirical test results that illustrate how localized reasoning can provide substantial gains in performance. I strongly believe that the principal of domain localization can be used by a wide variety of reasoning mechanisms. The idea is quite intuitive and natural, but has, surprisingly, not been a fundamental aspect of most AI systems. Its application to planning is vital if such systems are to meet the requirements of large, complex domains.

Acknowledgments

Lode Missiaen helped to design and implement GEMPLAN's localized search algorithm. I would also like to thank Anna Karlin for her assistance with the complexity results. John Bresina, Megan Eskey, Mark Drummond, Monte Zweben, Lode Missiaen, and Guy Boy also provided astute advice towards improving the quality of this paper.

References

[1] Bresina, J., Marsella, S. and C. Schmidt. "Predicting Subproblem Interactions," Technical Report LCSR-TR-92, LCSR, Rutgers University (February 1987).

[2] Chapman, D. "Planning for Conjunctive Goals," Masters Thesis, Technical Report MIT-AI-TR-802, MIT Laboratory for Artificial Intelligence, Cambridge, Massachusetts (1985).

[3] Hayes, P.J. 1973. The Frame Problem and Related Problems in Artificial Intelligence. *In* Artificial Intelligence and Human Thinking. *Edited by* A. Elithorn and D. Jones. Jossey-Bass, Inc. and Elsevier Scientific Publishing Company, pp. 45-59.

[4] Korf, R.E. "Planning as Search: A Quantitative Approach," *Artificial Intelligence*, Volume 33, Number 1, pp. 65-88 (1987).

[5] Lansky, A.L. "Localized Representation and Planning," in *Proceedings of the 1989 Stanford Spring Symposium, Workshop on Planning and Search* (March 1989).

[6] Lansky, A.L. "Localized Event-Based Reasoning for Multiagent Domains," *Computational Intelligence Journal, Special Issue on Planning*, Volume 4, Number 4 (1988).

[7] Lansky, A.L. "A Representation of Parallel Activity Based on Events, Structure, and Causality," in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon*, M. Georgeff and A. Lansky (editors), Morgan Kaufmann Publishers, Los Altos, California, pp. 123-160 (1987).

[8] Lansky, A.L. and D.S. Fogelsong, 1987. "Localized Representation and Planning Methods for Parallel Domains," in *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, Seattle, Washington, pp. 240-245 (1987).

[9] Lansky, A.L. and L. Missiaen, "Localized Search in GEMPLAN," NASA Technical Report FIA-90-04-03-2, NASA Ames Research Center, Moffett Field, CA 94035 (1990).

[10] Missiaen, L. "Localized Search," Technical Note 476, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave. Menlo Park, California 94025 (November 1989).

[11] Tate, A. "Project Planning Using a Hierarchical Nonlinear Planner," Department of Artificial Intelligence Report 25, University of Edinburgh (1976).

[12] Wilkins, D.E. 1984. Domain-independent Planning: Representation and Plan Generation. Artificial Intelligence, Volume 22, Number 3, pp. 269-301.

Transformational Synthesis: An Approach to Large-Scale Planning Applications

Theodore A. Linden
Advanced Decision Systems
1500 Plymouth St.
Mountain View, CA 94043
linden@ads.com

ABSTRACT*

General-purpose planning techniques quickly become computationally intractable as one tries to increase the scale of the problem from simple examples to real, useful applications. Practical solutions lie not so much in enhancing general-purpose techniques but in finding ways to exploit the structure of specific problem spaces without compromising the goals of flexibility, extensibility, and generality that one is trying to achieve with planning technology. Transformational synthesis is a rule-base approach to constructive problem solving that combines the strengths of both planning and programming technology and supports the integration of multiple planning methods. Planning methods are represented as transformations that evolve a declarative representation of partially developed plans. Successful integration requires a common plan representation and shared approaches for plan evaluation and for searching among alternative plans. Recent applications have included a mission planner for multiple Autonomous Land Vehicles (ALV), daily scheduling of tactical air fighter resources, and interactive planning and control for submarines. These planners mix many of the classical planning techniques—using each where it is most appropriate, and different transformations range from generic to domain-specific. For example, the ALV mission planner has the domain knowledge and planning heuristics to plan multiple-vehicle reconnaissance missions at the three highest levels of abstraction. Most knowledge about the environment and all knowledge about the effects of the vehicles' operations is uncertain and is represented in terms of probability distributions. These planners exhibit much of the flexibility and extensibility of general-purpose planners; yet, they have the run-time performance of special-purpose planners.

* This work was partially supported by the Defense Advanced Research Projects Agency (DARPA) and the U.S. Army Missile Command under contract DAAH01-90-0080 and partially supported by IR&D funding from Advanced Decision Systems.

1. Introduction

Recent research on planning technology has been searching for new paradigms that will enable the field to transcend the practical limitations of classical planning techniques. In the process, many of the assumptions involved in the classical planning methods of the 1970s are being questioned—indeed, the very meaning and utility of automated planning has been in question [Swartout 88].

The chief problem with classical planning techniques is that they quickly become computationally intractable when they are applied in applications that go beyond simple examples.[1] Furthermore, when planning is used in real-world applications, the planning must be done with very imperfect knowledge about world states, the planning needs to be interleaved with execution, and a rapid replanning capability is needed. While classical planning techniques can be extended to deal with uncertainty, with replanning, and with other real-world issues, such extensions only aggravate an already intractable problem with run-time performance.

In our approach to planning, we accept the idea that planning involves the creation and maintenance of a declarative data structure called a plan. Other agents (either automated or human) interpret the plan and use it as a guide for execution and control. We do not assume that the plan is the only input to the agent executing the plan or that this agent is a slave to the plan; in fact, one aspect of our approach toward large-scale planning and control systems is to give as much autonomy as possible to the execution agents and defer making decisions that are more appropriately made by the agents that execute the plans.

Past planning research has focused on generic, domain-independent planning techniques. While the motivation for this focus is clear, practical solutions to the computational intractability of planning are to be found by focusing on better ways of exploiting the structure of specific problem

[1] It has long been clear that there are more than just engineering problems to be faced when we attempt to scale up planning algorithms to handle larger applications. See [Chapman 85] about the theoretical limits of planning algorithms such as those used in STRIPS and other planners. The Forbin project [Dean, Miller, & Firby 87] is a recent example confirming that in practice a general planner quickly encounters intractible run-time performance problems.

spaces without seriously compromising the flexibility, extensibility, and generality of the planning system.

Exploiting Domain-specific Knowledge. Classical planning systems assumed that it is feasible to have a simple separation between general, domain-independent planning techniques and domain-specific knowledge. In fact, introspection on human planning does not show any clean separation into domain-independent planning techniques and domain-specific knowledge. Humans who are expert planners in one domain require a long time to become expert in another domain, and their learning in the new domain involves learning procedural as well as declarative knowledge about the domain. One of the contributions of the transformational viewpoint is that it offers more options for combining general domain-independent knowledge with domain-specific knowledge. By encapsulating domain-specific procedural knowledge within transformation rules, we achieve the efficiency of domain-specific procedural shortcuts while preserving most of the modularity, generality, and extensibility that is desired in planning system.

Planning vs. Programming. Classical *planning* technology does not take advantage of the structure of the specific problem in order to control the computational complexity of the problem. At the other extreme, if classical *programming* technology were applied to the same planning application, one would begin with a requirements definition that encourages the use of problem-specific information *throughout* the design of the application. This allows domain-specific details to permeate the entire application design with the result that the application's flexibility, extensibility, and generality are seriously compromised.

The transformational viewpoint exploits the strengths of both planning and programming technology in order to achieve practical, high performance planning systems that still have most of the flexibility, extensibility, and generality that one is trying to achieve with a planning system. Many of the recent efforts to build high performance robotic control systems can be criticized as being simply control programs. But if the robot is to be "intelligent," then it should at least be

1) very flexible in adapting to a wide variety of run time situations,
2) extensible in the sense that it can be instructed to behave in new and different ways without major reprogramming,[2] and
3) general in the sense that some of its knowledge and behaviors are applicable to more than one application domain.

Section 2 of this paper is an overview of transformational synthesis, and Section 3 describes how the idea has evolved out of work on reasoning about formal languages and is now being applied in a series of large planning applications at ADS. In Section 4 we discuss the contributions that this approach makes to planning technology. Section 5 describes how to use this approach when designing large planning applications. Sections 6 and 7 summarize previously published results from the ALV mission planner which is the largest completed application of this approach to planning, and Section 8 contains conclusions about the flexibility, extensibility, and generality that can be achieved with this approach.

[2]One would also like the robot to be able to learn from its own experiences; however, our current goal is limited to making the software be extensible in the sense that a programmer or a user can easily add to the functionality of the robot without having to understand details of its existing functionality

2. Transformational Synthesis Overview

Transformational synthesis is a paradigm for constructing programs, plans, or other complex conceptual objects by evolving them through small, independent changes until a desirable result is achieved. The flow of control for making these changes is largely data directed; i.e., each transformation may be invoked whenever a component of the evolving result matches the pattern specified in the transformation's preconditions. Transformational synthesis is a generalization of an approach to constructive problem solving developed during research on programming technology; and it has proven to be useful in developing other AI software that needs to be unusually flexible and extensible.

When transformational synthesis is used to automate *software development*, a program is evolved from its specifications by a series of small, independent changes or transformations. When transformational synthesis is applied to *planning*, the goals and constraints are represented as an incomplete plan; then planning techniques—implemented as transformations—evolve the plan into a form that can be executed effectively. Many of the transformations package together the results of knowledge engineering and of domain specific reasoning that has been done at design time. This reasoning does not need to be derived again from more primitive reasoning steps while the planner is executing.

As an approach for building practical, real-world planning applications, transformational synthesis is not in competition with specific planning paradigms like case-based planning or constraint-directed planning. Rather, transformational synthesis takes the view that plans are declarative objects that can be generated, refined, evaluated, and modified by many different planning methods. When following this approach, the two key steps are to design a knowledge base and transformations on that knowledge base such that:

- There is a declarative representation of the plan maintained in a knowledge base that is capable of representing the intermediate planning states that occur as the plan is being developed. A critical problem is to find plan representations that are expressive enough to capture all of the information that needs to be shared among planning methods. In practice, this leads to plan representations that have multiple viewpoints and capture knowledge about the goal structures, abstractions, dependencies, resources, alternatives, partial decisions, and uncertainties that need to be

reasoned about at intermediate stages of a plan's evolution.

- The transformations that evolve the plan may be entirely domain independent, they may exploit specific structures of the problem space, or they may be domain-specific and result from knowledge engineering activities that capture the planning shortcuts used by human planning experts. Other transformations apply algorithms or generic planning techniques. Each transformation has preconditions that limit its application to the planning subtasks for which it is effective.

2.1. Comparison with the Blackboard Model

When viewed in these general terms, transformational synthesis is quite similar to blackboard-based approaches to incremental planning where the declarative representations correspond to the blackboard and the transformations correspond to knowledge sources. Transformational synthesis shares with blackboards and rule-based approaches the ability to integrate multiple planning paradigms, to separate planning functionality from control and optimization decisions, and to enhance the modularity, generality, flexibility, and extensibility of the planning software. Historically, blackboard and transformational approaches have been applied to different classes of applications, and these applications have led the two research communities to focus on different aspects of a common problem. Unfortunately, the lessons learned from research in these two communities have often not been transferred effectively to the other community.

2.2. Summary of Transformational Synthesis Contributions

The transformational viewpoint, which is based on many years of research on software technology, brings with it many practical insights for integrating diverse planning techniques and scaling up to large applications. As detailed in the later section on Transformational Synthesis Contributions, these contributions can be grouped into four categories.

1) **Unifying Formalism for Integration.** Transformational synthesis has evolved out of research of software technology which provides a unifying formal theory that can support reasoning about the correctness and termination of transformational processes. This basis in software technology also includes a wide variety of optimization techniques which will all be needed as we try to scale up planning technology to handle large, real-world planning and scheduling problems.

2) **Plan Reuse, Replanning, and Contingency Planning.** Alternative approaches to plan reuse, replanning, and analysis of contingency plans are supported within the transformational viewpoint including work on: a) automatic capture of plan dependencies from the instantiated preconditions of the transformations that are used to generate the plan and b) replanning by replay of transformational derivation histories.

3) **Integration around a Common Plan Representation.** This supports the integration of multiple planning paradigms and the effective use of domain-specific planning knowledge.

4) **Interactive Planning by Co-operating Users.** By using problem-specific abstractions in the representation of the plan and by developing planning processes that imitate the steps that human planners use in manual planning, the evolving plan can be intelligible to users who interactively assist in developing and checking the plan as it evolves. The abstract plan also becomes a context for communication among co-operating human planners.

3. History of Planning by Transformational Synthesis

The transformational approach has evolved out of research on reasoning about formal languages. It has been applied extensively in the form of transformational implementations that compile very high level programming languages [Partsch & Steinbruggen 83, Fickas 85, Balzer 85, Smith et al. 85, Rich & Walters 85, Agresti 86, Lowry & Duran 89] and in work on simplifying mathematical expressions [Silver 86]. Transformations appear as key concepts in recent commercial programming languages like REFINE and Mathematica.

We use the term transformational *synthesis* when this basic transformational approach is applied in constructive problem solving. These constructive problem solving tasks extend the basic transformational paradigm in three ways:

- The transformations implement a variety of different constructive problem solving steps. The transformations used in transformational compilers do mostly simplification, definition expansion, refinement, and optimization. When the transformations implement constructive problem solving steps, arguments about termination and convergence of the transformations become more complex. For example, in the ALV planning application discussed later in this paper, the transformations implement goal elaboration, goal regression, refinement of abstract operations, plan evaluation, and plan critics. The design of the application needs to support an argument that invocation of the transformations will terminate and that when no transformation is applicable to the plan state, a correct plan will have been derived (or an explicit failure will have occurred).
- The data structures that are being transformed are not as simple as mathematical expressions and formally-defined programming languages. In general problem solving applications, the data structures being transformed involve quite complex representations of a problem space that may include all the structures of an object-oriented knowledge base including inheritance of properties and default values, relations treated as objects, and declaratively represented constraints.

- A planning system needs to explore plans that eventually prove to be unsuccessful or less successful than some alternative. Transformational compilers typically do not search among alternative possible transformational derivations; however, for most planning applications we apply transformational synthesis in the context of an overall search strategy that generates and evaluates alternative plans.

Recent examples of the extended use of the transformational approach include the KIDS system for algorithm design [Smith 90], a project management assistant, and a VLSI circuit design system. Here we focus on our recent applications to planning and scheduling applications.

Transformational synthesis was first applied to *planning* during ADS' work on DARPA's **Autonomous Land Vehicle** (ALV) program. Transformational synthesis was used to developed a real-world multi-vehicle mission planner designed to interface with the on-board vehicle control functionality of future ALVs [Linden & Owre 87, Linden 89, Linden & Markosian 89, Linden 90]. The plan is synthesized by transforming the goals into plans where the goal structure, abstraction levels, plan alternatives, uncertainty representations, and probabilistic plan valuations are included in the evolving plan representation.

In current work on RADC's **Advanced Planning System** (APS), ADS is using transformational synthesis to implement constraint-directed reasoning. APS is being implemented as an operational prototype that will partially automate the daily generation of Air Tasking Orders for Tactical Air Force Command Centers. The APS implementation includes extensive constraint propagation and meta-level reasoning that chooses which transformation is best applied next. Each scheduling decision (implemented as a transformation) includes substantial look-ahead to evaluate the probable effects that the decision would have on other scheduling decisions. This look-ahead is supported by the propagation of statistical measures to identify critical resources in a way that is parallel to recent work by Mark Fox and others at CMU [Fox et al 89, Fox & Sycara 89]. For this application we expect that, with a good ordering of the planning and scheduling decisions, it will be possible to generate good schedules without backtracking. (Backtracking is usually not acceptable when multiple users are co-operating to generate the plan interactively.) Plan critics will be able to undo specific previous decisions and are included in the system design.

Another feature of APS is mixed-initiative planning where any planning step can be done either by a human directly or by the automated planner. This has led to a strong emphasis on abstraction in the representation of the intermediate states that occur as a planning session progresses. Since the human planners participate in the planning process, they have to be able to interact with the plan as it is evolving during a planning session. Mixed-initiative planning requires high level modeling of the concepts that the humans actually use while they do their planning.

ADS is also using transformational synthesis on DARPA's **Submarine Operational Automation System** (SOAS) program. A key problem in this application is the integration of strategic and reactive planning. In the initial implementation, reactive plans are represented using Firby's Reactive Action Packages (RAPs) [Firby 89]. The first phase of this effort produced a reimplementation of RAPs and included transformations that generate a simple RAP at run time. A second phase is now underway to produce a more complete prototype planning system with transformations being used to generate reactive plans.

In another application of transformational synthesis, we are designing a planning system for the domain of transportation planning. The focus of this research is on representing and reusing reactive plans and programs. It addresses two common problems in AI planning applications: the integration of look-ahead planning with reactive execution, and the reuse and extension of previously developed plans, planning methods, and programs. As we extend the prototype planner to integrate solutions for a series of increasingly realistic and challenging transportation planning problems, we will measure the degree to which our planner reuses previous plans and the degree of program extensibility that we achieve as we extend the planner to new problems and new domains.

In all of these planning applications, the key to success is in the representation of the problem space and of the intermediate states of the plan together with the goal structure, abstractions, dependencies, resources, alternatives, partial decisions, and uncertainties that need to be reasoned about as the plan is evolved. Similar representation techniques were used in ADS' work on **AirLand Battle Management** (ALBM) program where ADS has recently completed a three year project to demonstrate that it is feasible to provide effective automated assistance to staff planners at the Army Corps level [Stachnick & Abram 88]. The planning process employed in ALBM closely parallels the manual Army planning process.

Each of these applications is quite different in terms of the specific planning techniques that are most appropriate for the application. The ALV and ALBM applications involve plans with complex goals, multiple abstraction levels, and reasoning about sequences of actions to be performed by multiple agents. A simplifying feature of these applications is that control structures for the planning process can be decided at design time based on knowledge engineering that extracts the planning processes actually followed by expert human planners in that domain. The planning process control has to be highly conditional so that it can adapt to the particular planning situation; however, the adaptability that is appropriate does not require explicit meta-level reasoning at run time.

In APS the structure of the plans being developed is much simpler (the complexity arises from resource allocation and scheduling constraints, not from variability in the structure of the plans), and much of the domain knowledge is naturally expressed as constraints. Manual planning processes in the APS domain do adapt to the constraints that are most critical; and, in automating these

processes, we have focused heavily on meta-level reasoning that decides on the order in which scheduling decisions can best be made.

On the SOAS project, much of the focus is on the selection and execution of reaction plans. Much of the domain knowledge can naturally be expressed as reactive plans ("in this situation, a commander would do such and such"), so many of the reactive plans are naturally built in as part of the system's knowledge base. Some of the reactive plans need to be generated automatically as the system is executing. Route plans are the chief example because they are difficult to express as conjunctions of rules that associate actions with situations, and RAPs that express route plans for the submarine are best developed in the run time situation.

The overall lesson from these applications is that the most effective way to express the available domain knowledge is different for different kinds of applications. Sometimes it is most effective to embed the domain knowledge directly in plan fragments, sometimes it is best to incorporate it in the processes that generate plans, and sometimes domain knowledge is best expressed as constraints that are used at a meta-level to control the planning processes. A given application may involve domain knowledge expressed in each of these ways, and the transformational viewpoint makes it easy to include domain knowledge in any or all of these forms.

4. Transformational Synthesis Contributions

In this section we summarize the advantages of adopting the transformational viewpoint. Many of these advantages can and have been pursued in other planning research; however, the transformational viewpoint gives a sound theoretical basis for developing verified plans while using computationally tractable planning methods. The main idea is to develop and verify transformations that encapsulate large chunks of planning knowledge. From a formal viewpoint, verified transformations are like parameterized lemmas that can be applied at will in a powerful theorem proving system. They allow much of the reasoning required for plan generation and replanning to be done once and then reused as needed during planning processes. By embedding appropriate planning knowledge in the transformations, we expect that run time costs for plan generation and replanning will scale up to real-world planning and scheduling applications. By using transformations that can be verified to preserve correctness with respect to formal goals and constraints, correctness is preserved as an invariant property of plans as they are generated.

4.1. Unifying Formalism for Integration.

By drawing on a long history of research on formalizing programming processes, transformational synthesis is supported by a unifying formalism for reasoning about planning processes and for exploring the tradeoffs between efficiency and generality.

Correctness, Convergence, and Termination. Case-based approaches to planning use transformations that tweak or modify previous plans to fit the current situation. Success has been demonstrated on simple problems; but, as we scale up to modify plans for large, real-world applications, we will encounter replanning processes that do not terminate or that are unstable in that alternative execution sequences or minor changes in the situation cause the tweaking to converge to distinctly different plans. We need a sound theoretical basis for reasoning about incremental plan modifications, and I believe this can be achieved by taking the transformational viewpoint where there is already a rapidly developing theoretical foundation. In addition to other theoretical results about transformational approaches, by interpreting transformations as statements in the UNITY program specification language, the logic for reasoning about UNITY programs [Chandy & Misra 88] can be applied to reason about the correctness, convergence, and termination of planning processes.

Integration of Procedural Knowledge. Since transformations are integrated as a construct within a programming language, it is relatively easy to make tradeoffs between procedural, transformational, and purely declarative representations of information. Transformations, which may encapsulate procedural information or may be purely declarative statements about preconditions and postconditions, have advantages that are intermediate between procedural and declarative representations. It is also possible to evolve an application toward a high-performance, large-scale system by designing it initially using mostly declarative representations and transformations, and then, as the appropriate problem solving strategies become better understood, migrating the implementation toward more efficient but less extensible procedural representations.

Compile-time optimizations. Within the transformational approach there has been research on explicit meta-level reasoning to choose the most appropriate transformations (e.g., [Silver 86]); however, experience with transformational approaches indicates that it is better to bind many of these control decisions at design time or at compilation time because run-time reasoning about control can itself become a performance problem. Run-time meta-level reasoning about control is sometimes needed, and when it is needed it may improve performance dramatically; however it is only one of many important techniques that are needed for high performance, large-scale planning and scheduling systems. Others include compilation of abstract data structures and of control reasoning.

Infusion of Software Technology. When transformational synthesis is used both to build plans and to compile the planning system software, the plans and programs are represented in the same knowledge base, and the tools applicable to compiling and optimizing programs are also available for compiling and optimizing the plans. Some transformations that are commonly used for software optimization transfer to the planning domain; for example, an analogue of a transformation that does loop jamming on a program appears to be useful in plan optimization. By

building on the transformational technology, other results from software research will also transfer more easily into the planning community; for example, some research on case-based planning appears to be missing the lesson from programming research that when there are strong dependencies within a plan, modification of the plan may be more difficult and time consuming than regenerating the plan by replaying the relevant portions of its development history.

4.2. Plan Reuse, Replanning, and Contingency Planning.

There are three alternative approaches to automated plan reuse, replanning, and contingency planning (in order of increasing flexibility):

1) selection and instantiation of parameterized plans,
2) incremental modification of previous plans using an explicit record of plan dependencies, and
3) replanning by replay of previous transformational derivation histories.

The first of these approaches is easily supported with any approach to planning, a transformational viewpoint is natural for the other two approaches..

Capturing Plan Dependencies. For incremental plan modification, transformations provide a systematic way of capturing the assumptions and dependencies behind components of the plan. These assumptions and dependencies, which are also the conditions that may need to be monitored during plan execution, appear in the instantiated preconditions of the transformations that generated the plan. When multiple planning paradigms are integrated in the context of a transformational approach, this gives a uniform and systematic way of capturing and recording the plan dependencies as the plan is constructed. Research on this topic will need to distinguish essential preconditions from others. It will be important to determine whether this way of capturing the assumptions and dependencies leads to representations that are manageable for real-world applications. There is an argument that for most resource allocation and scheduling problems, the external assumptions deal mostly with resource availability and the internal dependencies between different entities to be scheduled are not so complex that they will be unmanageable when captured and represented explicitly.

Replanning by Replay of Previous Planning Processes. Research on replanning in the planning community has focused on instantiating parameterized plans and then tweaking or modifying previous plans. Research on programming and design, however, has found that it is frequently better to reuse the program derivation history rather than to reuse a fully detailed program. Replay of transformational derivations is a topic of current research on problem solving [Carbonell 86], programming [Goldberg 89], algorithm development [Smith, 90], and design [Mostow, 89]. We expect that for planning problems, the replay of transformational derivation histories will offer a more general alternative to replanning. In particular, when developing contingency plans for an alternative situation, the alternative situation is likely to cause changes that are quite pervasive through much of the plan. Rather than "tweaking" the plan from the mainline situation, we hypothesize that it will frequently be faster and more effective to replay the derivation history of that plan (or of some other plan). If users participated in developing the main line plan, they may only need to ratify that their planning decisions can be reused in the contingent situation. Experimental results are needed to evaluate this trade off between plan reuse and replan of the derivation history.

4.3. Integration around a Declarative Plan Representation

By capturing in a declarative representation the intermediate state information that is needed by different planning paradigms, one can integrate many separately written planning components.

Integration of Multiple Planning Paradigms. Different planning paradigms are appropriate for different subproblems within a complex planning application. Decision-theoretic planning is appropriate for reasoning about alternative courses of action when there are many uncertainties. Constraint-direct reasoning is often best for detailed resource allocation and scheduling subproblems. Case-based planning can avoid the regeneration of plans similar to previous plans. Situated planners adjust the planning process to real-world time and information constraints. Generative planning is needed to deal with unique and unexpected situations where other planning methods fail. Real-world planning systems need to use the appropriate planning methods for each subproblem. Attempts to stretch a planning paradigm to handle subproblems for which it is not the best paradigm leads to complex, inefficient planning systems.

Effective use of domain-specific planning knowledge. By keeping the general-purpose and domain-specific knowledge in separate transformation rules, the domain-independent portions of a planning application can still be transferred between applications (assuming compatible representations for the evolving plans). Thus, within the transformational synthesis viewpoint it is practical to explore many uses of domain-specific information not only to represent facts about the problem domain but also in all of the following roles which are critical in reducing computational costs:

1) *Identifying useful abstractions.* Effective abstractions can reduce an exponential search problem to one that can be solved in linear time [Korf 87] [Lowry & Duran 89]. Some of these abstractions can be developed through knowledge engineering activities at design time, others need to be generated dynamically at run time [Lowry 90]. Abstractions appear in many different forms and lead to multiple hierarchical structures within an application. In addition to abstraction by ignoring preconditions, there is abstraction by aggregation of resources or other domain objects, there are "genus-species" and "part of" hierarchies, and there are abstraction levels where terms for higher level goals and

operations are defined within their own semantic theories and have implementation relationships to lower level objects.

2) *Dividing the problem into independent subtasks.* It is important to take advantage of domain-specific insights that allow the problem to be divided into subproblems that can be planned almost independently.

3) *Declarative representations of constraints and dependencies.* Many development environments include trigger or demon mechanisms that can implement constraint checking and propagation; however, this is a procedural representation of the domain constraints and leads to very messy planning systems that are not general and extensible. Automatic compilation of demons from declarative constraints combines the advantages of declarative representations with the efficiency of a more procedural representation.

4) *Implementing large inferencing steps.* Domain-specific information can be encapsulated in transformations and then reused to take large steps in reasoning during plan generation or replanning. The planner needs to execute these large inferences efficiently; it does not need to derive them from more general principles each time a plan is generated. These large inferences correspond to lemmas in an inferencing system.[3]

5) *Focusing the control flow within the inferencing.* Experience from automatic theorem proving and from real-world AI applications shows that extensive domain-specific information is needed to focus and control the inferencing strategy.[4]

4.4. Interactive Planning by Co-operating Users.

By emphasizing declarative representations for all of the information about the current state of the plan including its goal structure, abstractions, dependencies, alternatives, partial decisions, and uncertainties, the transformational viewpoint makes it easier to support user visibility into the current state of the plan and to allow users to interactively assist in the plan development. The plan representation also becomes a context for communication between multiple users who are working co-operatively and for communication between users at multiple sites and multiple levels of a hierarchical command structure.

User Visibility into the Developing Plan: Assuming that the plan representation includes the abstractions actually used by human planners in the domain, and assuming that it includes the goal structure, dependencies, alternatives, and other information that humans think about as they plan, it is possible to let users participate actively in the planning process. There is still a user interface problem to make the plan representation intelligible to users, but once the user can understand the intermediate states of the plan as it is being evolved, then users can make some of the planning decisions directly and can direct the transformational development process. The KIDS system [Smith, 90] is a large, working example where a user interactively directs a transformational development process.

Context for Communication among Multiple Human Planners: Once users can understand the state of a fully represented plan as it is being developed or modified, then that plan representation provides an effective context for communication among the users. Users who have specialized areas of expertise can make planning decisions when the plan evolves to the point where the preconditions for that decision are valid and the decision becomes appropriate. These users may operate at different levels of abstraction or at different levels of a hierarchical command structure.

A final advantage of the transformational viewpoint is that tools to support it are more mature and stable than current blackboard tool environments. REFINE has now been available as a supported commercial product for four years, and it is in its third release. It is being used in a wide variety of programming applications using both the transformational paradigm and other programming paradigms. The REFINE knowledge base is tuned for high performance, and in the KIDS system, it routinely manages knowledge bases with over 130,000 object instances.

5. Integration of Domain-Specific Planning Knowledge

This section contains general advice about how to use transformational synthesis in a planning application. Since transformational synthesis is an approach to design and not a design, many details about how it is applied depend on the application.

Integrating Solutions to Independent Subtasks. If a large scale planning application has a computationally tractable solution, then it is possible to break the problem into many almost independent subproblems. This decomposition of the problem into subproblems can take advantage of many different kinds of hierarchies:

- *Subtasking.* Plan representations should support the decomposition of tasks into subtasks.
- *Abstraction Levels.* The plan representation should take advantage of all of the abstractions that are applicable in the domain. Abstract operations are especially

[3]Case-based reasoning is one way for a planner to make a very large step in inferring the right plan by using historical information from either the same domain or from analogous situations in other domains. In many cases, one can achieve even higher performance by incorporating in a transformation the generic lesson learned from previous cases. Then the run-time planning system does not have to repeat the reasoning that generalizes from cases to planning rules.

[4]For a discussion of why general purpose search guiding and pruning techniques have failed to have more than a minor effect on curbing the combinatorial explosion in theorem proving, see [Bundy 83], especially Chapter 7 "Criticisms of Uniform Proof Procedures," pp. 82-95.

important, and many transformations are devoted to refining abstract operations to primitive operations.

- *Reflection.* Systems that do planning and control have a natural hierarchical organization based on levels of reflection—with levels devoted to acting, planning, meta-planning, etc. This reflection hierarchy for planning systems is largely orthogonal to subtask hierarchies and abstraction hierarchies.

One needs to integrate solutions to the different subproblems in a way that does not compromise the flexibility, extensibility, and generality of the system. Most approaches toward building integrated systems require that the interfaces between the different components of the system be designed early in the system's life cycle. Unfortunately, for planning systems it is almost impossible to identify all of the independent subproblems during the early stages of system design. If one is forced into early decisions about the planning system's components and the interfaces between them, the future flexibility and extensibility of the system is seriously compromised.

When the software is designed as a set of transformations of a declarative representation, procedural knowledge can be encapsulated within the transformations. If the software designers adhere to the goal of keeping the transformations independent so they interact only through the declarative representations, then it is relatively easy to add new transformations and to modify existing ones. Thus, new additional planning methods and heuristics can be added throughout the development and operational use of the application.

While transformational synthesis provides a framework for modularizing planning systems, there are still several elements of the system that need to be shared by all the transformations; and these elements need to be designed carefully.

1) The knowledge base representation, especially the representation of the evolving plan.
2) A way of testing and evaluating evolving plans.
3) Search mechanisms.
4) The control discipline that determines when transformations are invoked.

Representing Evolving Plans. When using transformational synthesis, much early design work is devoted to designing the knowledge base representations. It is important to take advantage of domain-specific abstractions in the choice of these representations. It is usually much harder to represent the partial and incomplete plans that occur at intermediate stages of planning than it is to represent the final plan.[5]

A knowledge base representation for the partial plans can be quite complex. The plan representation for ALBM represents goals, subgoals, tasks, subtasks, resources, plan alternatives, and constraints—and relationships between all of these objects.

Plan Evaluation. Testing and evaluation of plans is separable from the transformations that evolve the plan. Since projection of a plan's effect usually involves a lot of uncertainty, some form of reasoning with uncertainty is likely to be needed in the evaluation process. A specific approach that was used in the ALV mission planner is discussed in a later section.

Search. In a planning application, one usually cannot design the transformations so they make linear progress toward a good plan. Thus there needs to be an overall search strategy within which the transformations operate. In our applications thus far, we have been able to use heuristics that keep the overall search space quite narrow—imitating the human planner's approach of only exploring the most promising alternatives.[6] When the search can be kept narrow, it is possible to maintain the alternative plans within the knowledge base representation and the transformations can be used to evolve these alternative plans in parallel. Interim evaluations of the alternative plans are used to focus the transformations on the alternatives that are more likely to succeed.

Control. Transformations, like any rule-based approach, allow control decisions to be separated from functionality. Ideally, one wants the functionality of the planner to be independent of the order in which the transformations are tested and applied; then the run-time performance of the system can be optimized by being smarter about the order in which preconditions are tested. The optimizations can be done by meta-level planning; however, it is often as effective if they are done late in the development cycle by the application's designers once the planning system is functioning and the appropriate problem solving strategies are understood.

Once the semantics of the transformations have been used to optimize the control flow within the planning application, any extension of the planner's functionality may require that the optimization decisions be redone.

6. The ALV Mission Planning Problem

This section summarizes the scope of the planning problems that were handled in the ALV multi-vehicle mission planner. This ALV planner forms the largest completed application of transformation synthesis to a real-world planning application. Results from this work are documented in previous publications [Linden & Owre 87, Linden 89,

[5]Most programming languages are examples of representations that are good at expressing the final results of a development process but are seriously inadequate at expressing what has been decided at intermediate stages of the development leading up to that result.

[6]When broader search algorithms are appropriate for specific subproblems, we have applied a search algorithm within the postconditions of a single transformation so that the transformation's effect contains the results of this search algorithm applied to a narrowly focused subproblem.

Linden & Markosian 89, Linden 90]. One of the problems with evaluating approaches to large planning applications is that it takes years to try out ideas on large applications. Implementation work APS and SOAS is still underway.

The Multi-vehicle ALV Mission Planner generates plans for reconnaissance missions by a group of autonomous land vehicles. Army personnel defined the reconnaissance mission scenario as appropriate for future ALV demonstrations. Thus, transformational synthesis was applied to meet the needs of this given planning scenario; the planning scenario was not tailored to the capabilities of our approach.

The problem is to generate plans for a set of autonomous vehicles that are to carry out reconnaissance missions in areas well in advance of friendly lines. The vehicles are to travel to appropriate observation points where they are to hide and observe designated reconnaissance areas.

The current planner selects the observation points that are to be used, decides which vehicle will go to each observation point, and plans travel routes for each vehicle. All of these planning tasks are interdependent; for example, the choice of the preferred observation points depends on the vehicles' starting locations and their possible routes to the alternative observation points. The plan should minimize risk while traveling to and hiding at the observation points, and it needs to deal with constraints on fuel and time of arrival. In addition, the vehicle's knowledge of its own position degenerates as it travels, and it needs to re-establish its approximate position periodically by passing near known landmarks. The planner does have information about potential observation points and routes—derived from a digital map—but all of this information is uncertain. A plan to have a vehicle move to an observation point at a specified time must deal with this uncertainty.

Abstraction Levels. Our initial planner has the domain knowledge and heuristics to develop plans for these missions at the top three levels of abstraction; namely, the levels dealing with:

1) goals and evaluation criteria,
2) abstract plans that include selection among available observation points and assignment of ALVs to observation points, and
3) route plans down to the level of intermediate waypoints.

The planner is designed to be extended down to lower levels of abstraction and thus integrate with the lower-level planning and perception capabilities that were being developed on the ALV program. Research had already been done on many of these lower-level capabilities—especially road following, route planning [Linden et al. 86], and contingency planning [Linden and Glicksman 87].

The plans that are generated for these reconnaissance missions do not involve extensive interactions between the vehicles—this feature of the planning scenario is due to expected vehicle limitations and we believe it is not a limitation of the transformational synthesis approach. Our design for the APS system does plan for mutual support and other interactions between the different aircraft.

Mission Planning Challenges. The ALV mission planning problem at the top three levels of abstraction is hard enough to seriously challenge past planning technology—especially since we were looking for a general solution that extends easily to other ALV applications and to the lower levels of abstraction. We were faced with the following challenges:

- *Continuous state variables.* Most of the information about the state of each ALV changes continuously; e.g., fuel, position, and position error.
- *Uncertainty.* Essentially all knowledge of the environment is uncertain. Projections of future positions, fuel usage, and arrival times are all uncertain. Planned routes and observation points may turn out to be unusable.
- *Time.* It is easy to plan routes that minimize travel time, but that is not as relevant as trying to arrive at the observation point by some specified time—even when all the information about travel times is uncertain. Sometimes the departure time is so constrained that minimizing time is important, but often the planner must pick the best departure time.
- *Goals and evaluation criteria.* The planner needs to understand how to make trade-offs between risk, mission accomplishment, arrival time, and other factors.
- *Constraints.* Reducing fuel usage, travel time, or position error is important only when there is danger that some threshold will be exceeded.

Our planner deals with all of these challenges—except that in handling uncertainty it does not currently develop contingency plans that foresee the dangers that might arise from blocked routes or unusable observation points. Contingency planning to deal with blocked routes—including avoidance of routes that may require costly detours—was demonstrated in a previous project [Linden & Glicksman 87], but that capability has not yet been integrated with this mission planner.

Additional complications that the planner handles are:

- *Night travel.* Traveling at night involves less risk than daytime travel, so it may be advantageous for a vehicle to travel to a forward position at night and wait there until needed—although waiting behind enemy lines also involves risk.
- *Emergency mode.* The vehicles have the option of traveling in an emergency mode that reduces travel time at the expense of additional risk.

One important feature of this mission planning problem is that one cannot decide at system design time what factors are going to be most important when planning a given mission. Minimizing risk will frequently be important, but for some missions it will be more critical to reduce travel time, conserve fuel, plan routes that pass landmarks, or deal with a combination of these factors. Purely algorithmic solutions break down when there are many dimensions to be dealt with

(enough so straightforward optimization is intractable) and one cannot design the system to solve the problem in a few dimensions that are *determined at system design time* and then extend that solution into the other dimensions. A key aspect of our planner is that we delay until plan generation time the decisions about which dimensions are most important in solving today's particular mission planning problem.

7. Operator Semantics, Uncertainty, and Plan Valuations

In order to project the results of executing a plan, each operator needs a semantic definition that defines its effects. Since the effects of our operators are uncertain, we make extensive use of probability distributions in defining the semantics of the operators so that we can use well established probability theory to compose the effects of sequences of operations and do other internal computations. Of course, what is known about the operators is not a precise probability distribution about its effect but rather a vague statement like "a vehicle traveling from A to B will use about 10 gallons of fuel, give or take a gallon." After this is translated into a probability distribution, the results of the computations have to be interpreted based on the precision of the input data, but the idea of specifying the number of significant digits in a result is a very old and standard scientific method.

A complete description of our probabilistic representations of the semantics of operators is beyond the scope of this paper. We have, however, found that it provides a uniform way of handling several important problems. It allows us to characterize uncertainty in both the domain knowledge and in the effects of the operators. Furthermore, abstract operators have more uncertainty than more primitive operators, in fact, the effects of an abstract operator should be more uncertain than the cumulative effects of a sequence of primitive operators that refine the abstract operator. Thus one of the effects of refining an abstract plan is that the variance in its effects is reduced. It is useful to be able to reason explicitly about this change in variance. Also, as the effects of a plan are projected further into the future, those effects become less predictable. This is modeled effectively by the increasing variance as probability distributions are composed.

Once we have a representation for the semantics of our operators (both abstract and primitive), we can project the effects of executing a plan and then evaluate that plan in terms of the probability that the mission will be accomplished and that the vehicle will not be destroyed. In our current implementation, we do the projection backwards from the goal and then evaluate the plan by comparing its required preconditions with the available start conditions. This is a form of backwards symbolic execution and is equivalent to a forward projection of effects. Except for our use of probability distributions in the calculation, this has strong similarities with symbolic execution studied with respect to the formal semantics of programming languages.

8. Conclusions about Flexibility, Extensibility, and Generality

Transformational synthesis exploits the strengths of both planning and programming technology in order to achieve practical, high performance planning systems that still have most of the flexibility, extensibility, and generality that one is trying to achieve with a planning system. A goal of the transformational synthesis approach is to drive into declarative representations as much of knowledge about the problem space and problem solving methods as possible—while still being practical for large-scale applications.

In contrast to many other planning paradigms, transformational synthesis emphasizes incrementally maintaining plan correctness at intermediate stages of plan development. Furthermore, much of the reasoning that is required to assure these properties of a plan can be done once at transformation design time and embedded in validated transformations. Since the validated transformations encapsulate much of the reasoning that would otherwise have to be done at plan generation time, we anticipate transformational planning will scale up to handle large, real-world planning and scheduling problems.

One goal of our work is to demonstrate an extensible planner that can handle large-scale applications and grow with the application. It will take time to demonstrate extensibility for large applications, but we believe the ALV planner would easily extend to generate complete plans at lower levels of abstraction and for a much wider variety of missions. It is usually easy to extend the planner by adding transformations that handle additional subproblems and lower levels of abstraction. The limiting factor is the plan representation; extensions that require changes or major extensions in the underlying knowledge representation are difficult—which is why it remains important to design these representations carefully.

A good planner should be general enough so that it can generate plans in situations that were not fully anticipated by the planning application designers. While the ALV planner has domain-specific methods for dealing with many planning problems, we expect that it will be able to handle combinations of problems where the particular combination had not been foreseen. It incorporates searching, subtasking, and abstraction levels as basic generic techniques. Furthermore, when designing specific transformations, we have tried to use general techniques that will be applicable for more than just the current problem on which we were focusing. We believe that this generality will pay off as the planner is extended to cover other mission types.

Additional flexibility and generality can be achieved by developing more flexible plan representations. Most of the kinds of relationships that are needed between plan nodes such as subtasking, abstraction levels, sequential ordering, temporal constraints, dependencies, and justifications appear to be independent of the particular application, and we believe that much leverage can be obtained by developing general plan representations applicable to a wide class of planning applications.

References

[Agresti 86] William W. Agresti, "New Paradigms for Software Development, IEEE Computer Society Tutorial, IEEE Catalog Number EH0245-1, Washington, DC,1986.

[Balzer 85] Robert Balzer, "A 15 Year Perspective on Automatic Programming," IEEE Trans. of Software Engineering, Vol. SE-11, No. 11, Nov., 1985, pp. 1257-1268.

[Bundy 83] Alan Bundy, *The Computer Modeling of Mathematical Reasoning*, Academic Press, London, 1983.

[Carbonell 86] J. Carbonell, "Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition," in R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, Morgan Kaufmann, Los Altos, CA, 1986, pp. 371-392..

[Chandy & Misra 88] K. Mani Chandy and Jayadev Misra, *Parallel Program Design, A Foundation*, Addison-Wesley Publ. Co. Reading, MA, 1988

[Chapman 85] David Chapman, "Planning for Conjunctive Goals," MIT, AI Lab Memo AI-8902, 1985.

[Dean, Firby, & Miller 87] The Forbin Paper, Yale Univ. Computer Science Dept., YaleU/CSD/RR #550, July, 1987.

[Fickas 85] Stephen F. Fickas, "Automating the Transformational Development of Software," IEEE Trans. of Software Engineering, Vol. SE-11, No. 11, Nov., 1985, pp. 1268-1277.

[Firby 89] R. James Firby, "Adaptive Execution in Complex Dynamic Worlds," Ph.D. Thesis, Yale University, May 1989.

[Fox et al. 89] Mark S. Fox, Norman Sadeh, & Can Baykan, "Constrained Heuristic Search," AAAI 89, pp. 309-315.

[Fox & Sycara 90] Mark S. Fox and Katia P. Sycara, "Knowledge-based Logistics Planning and Its Application in Manufacturing and Strategic Planning," RADC-TR-89-215, Jan 1990.

[Goldberg 89] "Reusing Software Developments," 4th Annual RADC KBSA Conference, RADC, Sept. 1989.

[Korf 87] Richard E. Korf, "Planning as Search: A Quantitative Approach," AI Journal, 33,1, 1987, pp. 67-88.

[Linden et al. 86] Theodore A. Linden, James P. Marsh, and Doreen L. Dove, Architecture and Early Experience with Planning for the ALV, IEEE Inter. Conf. on Robotics and Automation, San Francisco, CA, April 7-10, 1986, pp. 2035-2042.

[Linden & Glicksman 87] Theodore A. Linden and Jay Glicksman , "Contingency Planning for an Autonomous Land Vehicle," Proc. IJCAI-87, Morgan Kaufman Publ., Vol. 10.

[Linden & Owre 87] Theodore A. Linden and Sam Owre, "Transformational Synthesis Applied to ALV Mission Planning," Proc. of the DARPA Knowledge-Based Planning Workshop, Austin, TX, Dec. 8-10, 1987, pp. 21-1 to 21-11.

[Linden 89] Theodore A. Linden, "Planning by Transformational Synthesis," *IEEE Expert*, 4,2 Summer, 1989, pp. 46-55.

[Linden & Markosian 89] Theodore A. Linden & Lawrence Z. Markosian, "Transformational Synthesis Using Refine," *AI Tools and Techniques*, ed. M. Richer, Ablex Press, 1989, pp. 261-286.

[Linden 90] Theodore A. Linden, "Transformational Synthesis: A Paradigm for Building Large-Scale Planning Applications," Planning Systems for Autonomous Mobile Robots, ed. D. P. Miller and D. J. Atkinson, Prentice Hall, to appear 1990.

[Lowry & Duran 89] Michael R. Lowry and Raul Duran "Knowledge-based Software Engineering," The Handbook of Artificial Intelligence, Vol. 4, ed. by A. Barr, P. Cohen, and E Feigenbaum, Addison-Wesley Publ. Co. 1989

[Lowry 90] Michael R. Lowry, "Abstracting Domains with Hidden State", to appear in Proc. of Workshop on Automatically Generating Abstractions and Approximations, AAAI-90

[Mostow 89] Jack Mostow, Design by Derivational Analogy, Artificial Intelligence, Vol. 40, No. 1-3, Sept. 1989.

[Partsch & Steinbruggen 83] H. Partsch and R. Steinbruggen, "Program Transformation Systems," ACM Computing Surveys, Vol 15, No. 3, Sept. 1983, pp. 199-236.

[Reasoning 87] Reasoning Systems, "REFINE Programmer's Reference Manual," Reasoning Systems, 1987.

[Rich & Walters 86] Charles Rich and Richard C. Walters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann Publ., Los Altos, CA, 1986.

[Silver 86] B. Silver, *Meta-level Inference: Representing and Learning Control Informatin in Artificial Intelligence*, North Holland, 1986.

[Smith et al. 85] Douglas R. Smith, Gordon B. Kotik, & Stephen J. Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute," IEEE Trans. of Software Engineering, Vol. SE-11, No. 11, Nov., 1985, pp. 1278-1295.

[Smith 90] Smith, D.R., KIDS: A Semi-Automated Program Development System, to appear in IEEE Transactions on Software Engineering special issue on Formal Methods, September 1990.

[Stachnick & Abram 88] G.L. Stachnick and J.M. Abram, "Army Maneuver Planning: A Procedural Reasoning Approach," Command and Control Research Symbposium, June, 1988.

[Swartout 88] William R. Swartout "Summary Report on the DARPA Santa Cruz Workshop on Planning," AI Magazine, Summer 1988.

Combining Reactive and Strategic Planning through Decomposition Abstraction*

Nathaniel G. Martin and James F. Allen
Computer Science Department
University of Rochester
Rochester, NY 14627

Abstract

Recent proposals for reactive execution modules provide difficulties for traditional strategic planners. We discuss a technique which uses statistics gathered from the execution of plans to guide the appropriate description of the plan. The plan is elaborated until the strategic planner is sufficiently confident that this plan will indeed achieve its goals based on the previous behavior of the executor. This plan is then given to the execution module.

1 Introduction

Traditionally, planning has been defined as the process of generating a sequence of actions to be executed mechanically. In this paradigm, an intelligent "planning module" generates a sequence of actions that are carried out by a simple "execution module." Such planners, called "strategic planners," have been successful in limited domains. In more realistic domains strategic planners are less successful. The high cost of generating plans makes strategic planning problematic in highly dynamic domains.

Reactive systems have been proposed to overcome the difficulties inherent in strategic planners. Reactive systems increase the power of the simple execution module in the hopes that it will allow a simplification of the planning module. The increased capabilities of the execution module improves the situation in routine cases. In commonly occurring cases the planner can rely on the execution module to perform correctly. The planner can generate a high-level description of the steps to be taken towards achieving a goal and assume that the execution module will fill in the required details. The generation of such high level plans should be less computationally complex than the generation of detailed plans. Increasing the power of the execution module allows it to deal with contingencies that are unknown or unknowable by the planning module.

Unfortunately, generating plans for such a powerful system introduces new complications. Unless the planner can decide what problems it needs to reason about and what problems the reactive execution system can deal with, planning for the richer execution language may be no more tractable than the traditional approach.

Moreover, if execution modules are sufficiently complex that they cannot guarantee the success of the activities they undertake, the task of generating sequences of actions that have a desired result will be difficult. The strategic planner must now recognize that the actions it decides to execute may fail, and that it cannot rely on the results of those actions.

We address both of these problems using a technique based on decomposition abstraction. This technique provides a method of deciding what aspects for the planning problem the reactive execution module is capable of handling on its own based on the prior performance of that execution module. It uses statistics on the execution module's prior performance to constrain the probability that the execution module can accomplish a particular task. If it can, the planner need not reason about that task. If it cannot, the planner must discover the likely causes of failure and specify a plan that avoids them.

2 The Two-Level Model

We assume a model consisting of two semi-independent systems that operate in parallel and interact with each other through a limited communication channel. One of these systems can be thought of as a strategic planner similar to traditional planning systems such as NOAH [Sacerdoti, 1975], STRIPS [Fikes and Nilsson, 1971] and their descendants [Chapman, 1987, Wilkins, 1988]. It is not important for this paper, or the general framework, how the planner operates as long as it supports action decomposition as defined in hierarchical planning systems. The other system is the reactive control system. This system consists of a set of sensory/motor procedures for executing various actions, and is similar to various reactive systems that have been proposed (e.g. [Brooks, 1985, Firby, 1987, Georgeff and Lansky, 1987, Kaelbling, 1988, Chapman, 1990]). At first glance, the two-level architecture is similar to that of the Shakey system [Nilsson, 1984], where the abstract plan reasoner is the STRIPS system, and the control system is the interpreter for the ILA and LLA procedures that actually controlled the robot. There are, however, two important differences.

First, both systems in the two level model execute con-

*This work was partially supported by the Office of Naval Research under contract N00014-82-K-0193 and by ONR/DARPA contract N00014-80-C-D197.

currently. The control system always has a set of goals guiding the agent's activity in real time. Many of these goals will have been generated by the abstract plan reasoner, but there may also be "built-in" goals, such as avoiding danger, that the reactive system always maintains. The abstract planner rarely reasons about these goals, so the agent's actual behavior is only partially determined by the plans the strategic planner creates.

The second and more important difference is there is no fixed level of action primitives. In the Shakey system, the ILAs were all fairly small scale motor actions. In our proposed system, the control level may be able to execute quite complex actions that have been learned through extensive experience. For example, most piano players can play a C scale without considering playing each note or the individual finger movements. Such a pianist could reason about the decomposition of this action, but, besides wasting computation time, the resulting plan will probably execute less efficiently as it ignores the practiced motor routines for the complex action. As an even more extreme example, consider the skills required for driving a car. Novice drivers spend considerable time learning the motor skills necessary for driving, while more experienced drivers might not even know how to do all the individual actions in isolation. Yet clearly the novice is not the better driver. The challenge is to allow for such complex learned "reactive" behavior and yet still have the agent responsive to its planned abstract goals.

As can be seen from the above discussion, the abstract plan reasoner must decide whether or not to decompose actions. Given a goal to play a C scale, should the plan reasoner execute the action **PlayCScale** as a unit (i.e. send the goal as is to the control component), or should it decompose this complex action into its individual sub-parts, **Play**(C), **Play**(D), and so on, and execute them? The answer depends on the plan reasoner's experience. A novice should decompose the action into its sub-parts because each sub-part is simpler to execute. The skilled pianist, however, is better off executing the motor routine learned for playing scales. Note that at the abstract plan level, both the novice and the expert might have the same definitions of the action and its decomposition. The differences arise from their ability to execute these actions.

The abstract plan reasoner maintains statistics on the reactive system's success as it attempts various actions, and computes estimates of success for similar actions from these statistics. The key point is that in deciding whether or not to decompose an action, the agent compares its estimate of success for executing the action with its estimate of success for executing the component sub-actions. The agent can then decide, based on its previous experience, whether to decompose the action or not.

3 The Formalism

We will develop only enough of the representation to make the points in this paper. We use a logic with reified events based on interval temporal logic [Allen *et al.*, To appear 1990, Allen, 1983]. The events of interest are those that consist of an agent attempting to perform an action. With respect to the two-level model, we say that an agent attempts an action if it instructs the reactive system to perform the action. The reactive system then executes a routine. For example, the predicate PlayCScale(a,e) is true when event e consists of the agent a instructing its reactive level to perform the PlayCScale action. The assertion that John attempted to play a C scale at time T_1 would be written as:

$$\exists(e)[\text{PlayCScale } (John, e) \wedge Time(e) = T_1].$$

In other words, there is an event in which the action of John playing a C scale occurs, and the time of this event was the interval T. The occurrence of the action indicates nothing further. For example, John playing the C scale may not have resulted in any sound because his amplifier was turned off.

We collect statistics about the effect of actions using a probability theory based on Kyburg [1974, 1983b] and similar to that used by Loui [1987], Bacchus [1988] and Weber [1989]. Suppose we have a set of events in which an action occurred, E_A, and subset of these events in which that action had the desired result, E_R. The success rate of that action would be the ratio of the number of times the action was attempted to the number of times the action had the desired result. This is written $\%(E_R \mid E_A)$. By associating the sets characterized by a predicate with that predicate $\%(CScalePlayed(Time(e)) \mid PlayCScale(John, e))$ could be the proportion of times John successfully plays a C scale when he attempts it.

As Kyburg [1983a] points out, access to these ratios is usually unavailable. At no particular time can one ascertain how often John will successfully play a C scale; the best one can do is calculate the constraints John's past performance places on all performances both past and future. Confidence intervals [Neyman, 1960] are a well known technique for capturing such constraints. Our technique involves treating these confidence intervals as interval valued probabilities. For example, suppose Mary has been successful 960 of the 1000 times she has attempted a C scale. One can be 95% confidence that the probability she will successfully play a C scale on any particular trial lies in the interval [.95,.97]. We capture this information in two functions *Experience*() and $\widehat{Prob}_{95}()$. Mary's experience playing C scales is expressed as follows:

$$Experience(\text{CScalePlayed } (Time(e)), \\ \text{PlayCScale } (Mary, e)) = (960, 1000).$$

Furthermore, the %95 confidence interval induced by this experience will be expressed:

$$\widehat{Prob}_{.95}(\text{CScalePlayed } (Time(e)) \mid \\ \text{PlayCScale } (Mary, e)) = [0.95, 0.97].$$

Hanks [1990] and Haddawy [1990] also develop systems for reasoning about actions in time probabilistically. Both Hanks and Haddawy use point valued probabilities. Hanks calculates point valued probabilities by choosing the maximum entropy distribution consistent with the observed data. Haddawy generates probabilities through a real valued measure function over the

possible worlds. Though point valued probabilities are well ordered, and therefore easy to compare, the partial order provided by intervals gives the system the ability to represent a kind of meta-knowledge. The width of the interval gives the system the ability to represent how accurate its knowledge is. Comparison is more complicated when using intervals, however, because they only induce a partial order. When comparing two interval probabilities, we say that [a,b] is greater than [c,d] if a is greater that d. If [a,b] and [c,d] overlap we will say they are incomparable.

The effects of an action are those propositions for which statistics are collected when the action is attempted. Thus CScalePlayed($Time(e)$) is a possible effect of the PlayCScale action. Preconditions are similar; they are propositions for which conditional statistics are collected. Thus, in general, information about preconditions, $\mathrm{P}_1(x) \ldots \mathrm{P}_n(x)$, and effects, $\mathrm{E}_1(x) \ldots \mathrm{E}_m(x)$ of an action A are related by statements of the form:

$$\widehat{Prob}_{.95}(A \bigwedge_{i=1}^{n} \mathrm{P}_i(x) \bigwedge_{i=1}^{m} \mathrm{E}_i(x) \mid A \bigwedge_{i=1}^{n} \mathrm{P}_i(x)) = [a, b]$$

For simplicity, we ignore preconditions here, but see Martin and Allen [1990].

A *decomposition* is an axiom stating that achieving a set of sub-goals under certain constraints leads to an effect of an action. These axioms can be used to reason that, in addition to executing the action, the effects of an action can be achieved by executing actions which achieve the sub-goals. The planner may reason about the probability of successfully executing the actions that achieve the sub-goals.

For example, a decomposition of playing a C scale is playing each of the notes in quick succession. Following Kautz [1987], decomposition axioms consist of direct components (DC) and constraints (κ). The direct components are predicates that may be satisfied by executing sub-actions, and the constraints describe the circumstances under which these sub-actions will result in the action being accomplished. For example, a decomposition axiom for the action, **PlayCScale**, might be:

$\forall(a, t_1, t_2, t_3, \ldots, t_9)$
[Played $(a, C, t_1) \wedge$ Played $(a, D, t_2) \wedge \ldots \wedge$
Meets $(t_1, t_2) \wedge$ Meets $(t_2, t_3) \ldots$
$\Rightarrow$ CScalePlayed $(a, t_9) \wedge$
Begins $(t_1, t_9) \wedge$
Ends (t_8, t_9)]

This axiom says if the agent manages to produce all the notes in quick succession, then the agent has played a C scale, and temporal extent of this playing stretched from the beginning of the first note to the end of the last note. In this decomposition the Played(a, *note*, e_i) predicates comprise DC; the temporal constraints comprise κ.

Throughout this paper, constraints are usually over simplified for clarity, and the axioms are summarized using a more conventional STRIPS-style operators. An operator has three parts, the preconditions, P, the effects, E, and the body, B. The preconditions are constraints on the applicability of the operator expressed as a formula; the effects are the results of applying the operator for which statistics are gathered; and the body is a set of decompositions. As an example, consider the operator, **Act**:

Act($\vec{x}$)
P: $P(\vec{x}, t_1)$
E: $E(\vec{x}, t_2)$
B: $B_1(\vec{x}), B_2(\vec{x}) \ldots B_n(\vec{x})$.

This operator is well founded if:

$$\exists(e, \vec{x}, t_1, t_2) \quad [P(\vec{x}, t_1) \wedge \text{Act } (\vec{x}, e) \quad \text{Meets } (t_1, t_3), \wedge Time(e) = t_3]$$

That is, the operator is well founded if there is an assignment to the variables so that the preconditions held immediately before the actions and the action was attempted. The conditional probabilities associated with the operator require an event of the type described by the operator exists. If none exists, the probability is not defined because the denominator of the formula for calculating conditional probabilities will be zero.

The body of an operator is a list of lists of predicates which are considered to be sub-goals for the planner to achieve. Each of these lists of sub-goals represents a decomposition of the action. For example, one of the of the lists of sub-goals of **Act** above, say B_1, might be $(Q_1(\vec{x}, t_0); Q_2(\vec{x}, t_1))$. Such a list of subgoals in the body of operator represents the decomposition:

$\forall(t_0, t_1, t_2, t_3)$
[$P(\vec{x}, t_0) \wedge Q_1(\vec{x}, t_1) \wedge Q_1(\vec{x}, t_1) \wedge$
Meets $(t_0, t_1) \wedge$ Before (t_1, t_2)
$\Rightarrow$ $E(\vec{x}, t_2) \wedge$
Begins (t_1, t_3)
Ends (t_2, t_3)].

That is, the body of an action states that achieving the sub-goals in order achieves the effects of the operator.

Consider again the example from the last section. Suppose Mary is a pianist while John has only played once or twice and that these facts are captured by following statistics:

$$\textit{Experience}(\text{CScalePlayed } (Time(e)), \text{PlayCScale } (Mary, e)) = (960, 1000)$$
$$\textit{Experience}(\text{CScalePlayed } (Time(e)), \text{PlayCScale } (John, e)) = (1, 3).$$

This experience will produce the following approximate probabilities:

$$\widehat{Prob}_{.95}(\text{CScalePlayed } (Time(e)) \mid \text{PlayCScale } (Mary, e)) = [0.95, 0.97]$$
$$\widehat{Prob}_{.95}(\text{CScalePlayed } (Time(e)) \mid \text{PlayCScale } (John, e)) = [0.06, 0.79].$$

(i.e. Mary nearly always successfully plays a scale, whereas it is highly uncertain whether John can play a scale or not).

Suppose further that both John and Mary have the same competence in playing a single note, that

$$\widehat{Prob}_{.95}(\text{Played } (N, Time(e)) \mid \text{Play } (John, N, e))$$
$$= \widehat{Prob}_{.95}(\text{Played } (N, Time(e)) \mid \text{Play } (Mary, N, e))$$
$$= [.98, .99]$$

The question of whether each agent should decompose the **PlayCScale** action now reduces to the question of whether the estimated success of executing **PlayCScale** directly is higher or lower than the estimated success of executing the eight individual notes. For Mary, the estimate of successfully playing the scale directly is [0.95, 0.98], whereas the estimate for successfully playing each of the eight individual notes would be the multiplication of the estimate for each individual note, namely [0.98, 0.99], which produces the estimate [0.85, 0.92]. Clearly Mary should not decompose the action as the former estimate is invariably better that the latter. For John, however, decomposing is better as it produces the estimate of [0.85, 0.92], while directly executing the action is estimated as [0.06, 0.79].

We now describe the operation of a hierarchical strategic planner that works concurrently with a reactive system and that has no fixed set of primitive actions as described above. At each step, the planner looks at the set of operators that achieve its current goal. One of these operators is selected, and the planner decomposes the operator into a sequence of smaller steps by finding operators which achieve the direct components. Constraints on the sub-goals insure these actions do not interfere with each other. For simplicity, we do not discuss selecting the appropriate operator. For details about this process see [Martin and Allen, 1990].

4 The TRAINS Domain

To make the development more concrete, the second example is expressed in the TRAINS domain under development at Rochester. The TRAINS domain is a fairly complex simulated rail transport system. Rather than having a single control system, the planner interacts with multiple control systems, each one playing the role of a train engineer. For the purposes of this paper, however, we will use only a single train engineer. The train engineer's behavior is defined by a set of reactive procedures that execute to accomplish goals. When it is given a goal, it selects a procedure that accomplishes this goal according to some simple heuristic and executes the procedure.

The example shows how reactive and strategic systems can work together to solve problems; it is not a realistic problem in routing trains. Such an example would involve complex temporal reasoning about schedules and deadlines that would obscure the problem of interfacing the strategic and reactive parts of a planner. This example is helpful, however, because one can put oneself in the place of either the reactive system (i.e. the engineer) or the strategic planner. The same issues arise in robotics domains but intuitions are harder to develop because the reactive system must perform actions about which people rarely introspect. As in robotics, the planners only access to information about the TRAINS world comes from the engineer's reports. In a robotics domain, visual routines play the part the engineer's reports play in the TRAINS domain.

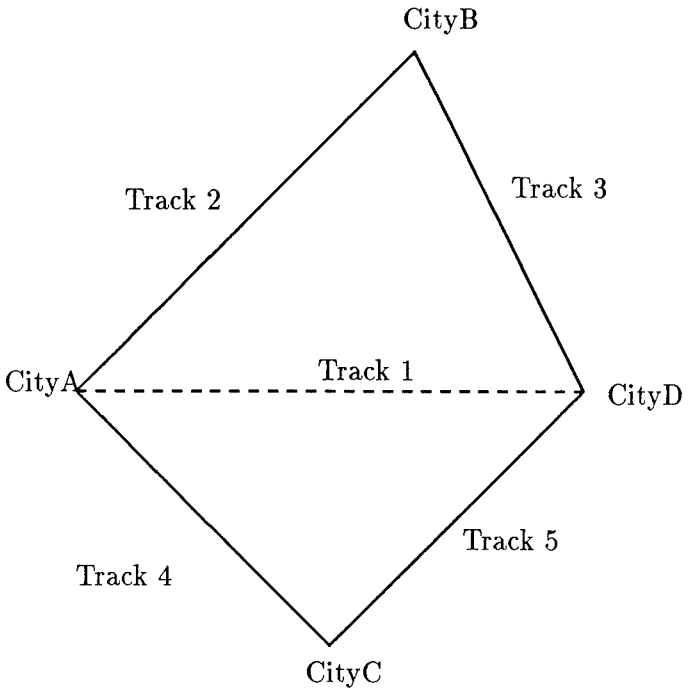


Figure 1: Layout for TRAINS example

The example situation is shown in figure 1. The planner must direct an engine at CityA to the factory at CityD. Let us assume that the direct route between CityA and CityD is poor quality track, and, when the weather is bad, the train runs a serious risk of derailing on that route. Furthermore, switches in CityB are often broken, causing difficulties with this route.

The engineer has access to the same map the planner does, and, using that map, the engineer can choose routes given a destination. The sample statistics arise from the following scenario. Though neither the engineer nor the planner know the true probabilities, the route through CityB and the direct route leads to success half the time. The route through CityC is better with success 80% of the times it is tried. The route through CityC fails in inclement weather which the engineer can forecast, but which the planner cannot. The engineer chooses routes according to the following heuristic of which the planner is unaware. The engineer always chooses to go through CityB if that option is available, otherwise it will chose to go through CityC unless the weather is bad and it has a choice. If the weather is bad and it can choose it takes the direct route.

The planner's decisions are based on it's experience with the engineer. If the engineer usually chooses a good route, the planner tries to leave as much leeway as possible in the plans it specifies. If, on the other hand, the engineer is often mistaken, it will specify plans as rigidly

as possible.

5 Planning in the TRAINS domain

In the initial state for this example, *Train1* is in *CityA*. The planner's goal is to make In(*Train1*, CityD) true. Only the **Move** operator has the desired effect, so the planner instantiates it as shown below:

Move(*Train1*, *CityA*, *CityD*)
P: In(*Train1*, *CityA*, t_0)
E: In(*Train1*, *CityD*, t_1)
B: (At(*CityD*, t_1))
(At(*CityB*, t_2); At(*CityD*, t_1))
(At(*CityC*, t_2); At(*CityD*, t_1))

There are three different decompositions of this operator, each one corresponding to a different route.

The planner can fill out each decomposition by choosing operators representing actions that will achieve the sub-goal. It does this by looking through the available operators for the one with the highest probability of achieving that sub-goal. Once the planner has filled in the decomposition with operators, it must consider two issues: first, should it decompose the operator, and second, if it decomposes, which decomposition is best? It may send any one of the decompositions, or it may send a disjunction of decompositions. As we shall see below, sending a disjunction of decompositions allows the planner to take advantage of the reactive system's decision making abilities.

The other operator used in this example is:

- **Traverse**(*train*, *city1*, *city2*)
  P: Exiting(*train*, *city1*, t_0)
  E: At(*train*, *city2*, t_3)
  B: ()

This operator has no decompositions so the planner's only option is to specify execution of the **Traverse** action.

Consider the system planning to get Train1 from CityA to CityD. The decompositions can all be accomplished by using various instantiations of the **Traverse** operator for each subgoal. For brevity, we name the decompositions in which the sub-goals have been replaced by actions:

Plan-A $\equiv$ (**Traverse**(*Train1*, *CityA*, *CityD*))
Plan-B $\equiv$ (**Traverse**(*Train1*, *CityA*, *CityB*); **Traverse**(*Train1*, *CityB*, *CityD*))
Plan-C $\equiv$ (**Traverse**(*Train1*, *CityA*, *CityC*); **Traverse**(*Train1*, *CityC*, *CityD*))

The planner's decision procedure is to decompose a plan only when it has negative experience with the compound operator. In particular, if the approximate probabilities are incomparable, it will not decompose. Incomparability indicates that the planner has insufficient information to make a choice so it defers the decision to the reactive system hoping that the reactive system will have better information. Though the reactive system performs less complex reasoning than does the strategic planner, the reactive system may have better information as it has access to the real world. For example, the engineer is better able to determine if there is an obstruction in the tracks directly ahead of the train.

Suppose we have the following experience based on the scenario described above:

$$\mathit{Experience}(\text{Plan-A }) = (100, 200)$$
$$\mathit{Experience}(\text{Plan-B }) = (100, 200)$$
$$\mathit{Experience}(\text{Plan-C }) = (160, 200)$$
$$\mathit{Experience}(\textbf{Move}) = (50, 200)$$

This would give rise to the following approximate probabilities:

$$\widehat{Prob}_{.95}(\text{Plan-A }) = [0.43, 0.57]$$
$$\widehat{Prob}_{.95}(\text{Plan-B }) = [0.43, 0.57]$$
$$\widehat{Prob}_{.95}(\text{Plan-C }) = [0.74, 0.85]$$
$$\widehat{Prob}_{.95}(\textbf{Move}) = [0.20, 0.31]$$

Using the decision procedure described earlier, the planner decides to decompose the **Move** into Plan-C and send that to the reactive system.

But if the reactive system has its own decision making capabilities, the planner can do better. Because the reactive system can forecast the weather, it knows which of Plan-A and Plan-C has a better probability of success in any particular situation. Therefore, sending the reactive system the command to Plan-A $\vee$ Plan-B produces better results. We can capture this behavior by introducing the notion of complex plans.

We call Plan-A, Plan-B, and Plan-C basic plans and disjunctions of basic plans complex plans. The planner may maintains information about all of the probabilities to model of the abilities of the engineer.

The planner may combine probabilities. In deciding whether to decompose playing a C scale we used the laws of probabilities. To do so, we had to assume that the underlying probabilities were independent. There, such an assumption was not unreasonable, because the planner insures that the actions will not interfere with each other. Independence cannot be assumed in the new scenario, however, because the planner will never execute more than one of the plans; it will always choose a particular one and execute that one.

Another way of combining probabilities is to combine the actual experience and calculate a confidence interval for this combined experience. This way of updating probabilities is accurate only if the reactive system executes at random. This, however, is a bad assumption if the planner wants to make use of the reactive system's abilities. In particular, if the reactive system has some decision making ability, it will not be randomly selecting a plan from the choices the planner sends it. If the reactive system makes good choices, it should be better than random selection and experience with complex plan

should be better than the simple combination of the basic plans. Thus the planner must collect actual statistics for the complex plans as well as the basic ones.

Suppose we have the following experience for the complex plans:

$$\begin{aligned} &\mathit{Experience}(\text{Plan-A} \vee \text{Plan-B}) = (48, 200) \\ &\mathit{Experience}(\text{Plan-A} \vee \text{Plan-C}) = (182, 200) \\ &\mathit{Experience}(\text{Plan-B} \vee \text{Plan-C}) = (52, 200) \\ &\mathit{Experience}(\text{Plan-A} \vee \text{Plan-B} \vee \text{Plan-C})) \\ &\qquad = (50, 200) \end{aligned}$$

This experience arises due to the difference between the planner's and the engineer's knowledge. Whereas experience with the basic plans reflects the planner knowledge, knowledge of the engineer's decision making abilities is reflected in experience with the complex plans. Experience with the disjunction of all three of the possible decompositions, Plan-A $\vee$ Plan-B $\vee$ Plan-C , might be different from executing the action directly if the engineer had means of achieving the goals of the action of which the planner does not know. Here the engineer has no private techniques for achieving goals, so the probabilities are the same.

The engineer does poorly whenever it is given the choice of going through CityB because its heuristic is to choose that route whenever it can, and that route is bad. On the other hand, it does significantly better than any of the basic plans when given the choice of going through CityC or taking the direct route. The improvement over random choice comes from the engineers ability to forecast the weather. When the weather is bad, it takes the direct route with a 50% chance of success rather than a 100% change of failure.

The planner should recognize that the engineer has special abilities and send it plans that allow it to exercise these abilities. Using the decision procedure outlined above, this is indeed what happens. The planner's experience gives rise the the following approximate probabilities:

$$\begin{aligned} &\widehat{Prob}_{.95}(\text{Plan-A} \vee \text{Plan-B}) = [0.19, 0.30] \\ &\widehat{Prob}_{.95}(\text{Plan-A} \vee \text{Plan-C}) = [0.86, 0.94] \\ &\widehat{Prob}_{.95}(\text{Plan-B} \vee \text{Plan-C}) = [0.20, 0.32] \\ &\widehat{Prob}_{.95}(\text{Plan-A} \vee \text{Plan-B} \vee \text{Plan-C}) = [0.20, 0.31] \end{aligned}$$

As can be seen, the plan Plan-B $\vee$ Plan-C dominates the probabilities of both the complex and the basic plans. Thus, given this information, the planner decides to decompose the **Move** action, and to give the engineer the choice between decompositions Plan-B and Plan-C.

A difficulty with this scheme for maintaining information about the effects of its plans is the exponential number of complex plans. This is not a problem in the representation because there will be a small number of the possible complex plans for which data has been collected. One of the advantages of approximate probabilities is the ability to represent that no information is available. Those complex plans for which no information is available have an approximate probability of [0, 1]. The exponential number of complex plans does pose a problem to the planner in gathering information, however. If it spreads its trials evenly over all the possible complex plans, it will gain expertise only slowly. If, on the other hand, it concentrates on only a few, it risks missing the best plan.

6 Conclusion

We present a two level architecture in which a strategic planner sends commands to a reactive system. The strategic planner chooses an appropriate level of detail at which to communicate using statistics gathered from the reactive system's previous performance. From this previous performance probabilities are approximated by calculating a confidence interval for the true probability. This approximate probability is then used to guide the choice of an appropriate level of detail at which to communicate with the reactive agent. The strategic planner chooses a level of detail that has been most successful in the past.

Interval valued probabilities provide a useful tool in combining reactive execution modules with strategic planners. In addition to the strength of belief the position of the interval gives, the width of the interval gives valuable information about the planners knowledge about the effects of its actions. Using this information, the strategic planner can give guidance to the reactive system only when it knows it has better information than the reactive system does and allow the reactive system leeway when it is uncertain.

The system's weakness is its reliance on the its operators and decompositions. In more realistic domains, the planner will need to deal with preconditions and reason about the probabilities of that are not composed of independent actions. To do this the system will need to reason about preconditions, and the probability of these preconditions being violated. Such reasoning is likely to be quite complex, so search heuristics must be developed. Initial work in this area is reported in [Martin and Allen, 1990]. Tyro, a planner that makes use of approximate probabilities and decomposition abstraction is being developed at Rochester.

References

[Allen *et al.*, To appear 1990] James F. Allen, Henry A. Kautz, Richard N. Pelavin, and Josh D. Tennenberg. *Formal Models of Reasoning About Plans.* Morgan Kaufman Publishing Co., San Mateo, CA, To appear 1990.

[Allen, 1983] James F. Allen. Maintaining knowledge about temporal intervals. *Communication of the ACM*, 26(11):832–843, 1983.

[Bacchus, 1988] Fahiem Bacchus. *Representing and Reasoning with Probabilistic Knowledge.* PhD thesis, University of Alberta, Fall 1988.

[Brooks, 1985] Rodney A. Brooks. A robust layered control system for a mobile robot. Technical Report 864, MIT AI-Lab, Cambridge, MA, September 1985.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[Chapman, 1990] David Chapman. *Vision, Instruction, and Action.* PhD thesis, MIT, Cambridge, MA, 1990.

[Dean and Boddy, 1990] Thomas Dean and Mark Boddy. A temporal probability logic for reasoning about actions. In *Proceedings of the Sixth Annual Conference on Uncertainty in AI*, pages 49–54, 1990.

[Fikes and Nilsson, 1971] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–205, 1971.

[Firby, 1987] R. James Firby. An investigation into reactive planning in complex domains. *AAAI*, 5:202–206, 1987.

[Georgeff and Lansky, 1987] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. *AAAI*, 5:677–682, 1987.

[Hanks, 1990] Steven John Hanks. *Projecting Plans for Uncertain Worlds.* PhD thesis, Yale University, New Haven, CT, 1990.

[Kaelbling, 1988] Leslie Pack Kaelbling. Goals as parallel program specifications. In *AAAI*, volume 1, pages 60–65, 1988.

[Kautz, 1987] Henry Kautz. *A Formal Theory of Plan Recognition.* PhD thesis, University of Rochester, Rochester, NY 14627, 1987.

[Kyburg, 1974] Henry E. Kyburg, Jr. *The Logical Foundations of Statistical Inference.* Reidel, 1974.

[Kyburg, 1983a] Henry E. Kyburg, Jr. *Epistemology and Inference.* University of Minnisota press, Minneapolis, MN, 1983.

[Kyburg, 1983b] Henry E. Kyburg, Jr. The reference class. *Philosophy of Science*, 50:374–397, 1983.

[Loui, 1987] Ronald P. Loui. *Theory and Computation of Uncertain Inference and Decision.* PhD thesis, University of Rochester Computer Science Department, September 1987.

[Martin and Allen, 1990] Nathaniel G. Martin and James F. Allen. Abstraction in planning: A probabilistic approach. Presented at AAAI-90 Workshop on Automatic Generation of Approximations and Abstractions, 1990.

[Neyman, 1960] J. Neyman. *A First Course in Probability and Statistics.* Hold, Rinehaart and Winston, New York, 1960.

[Nilsson, 1984] Nils J. Nilsson. Shakey the robot. Technical Report 323, SRI International, 1984.

[Sacerdoti, 1975] Earl D. Sacerdoti. A structure for plans and behavior. Technical Report 109, SRI International, Menlo Park, California, August 1975.

[Weber, 1989] Jay C. Weber. *A Formal Theory of Plan Recognition.* PhD thesis, University of Rochester, Rochester, NY 14627, 1989.

[Wilkins, 1988] David E. Wilkins. *Practical Planning.* Morgan Kaufmann Publishers, San Mateo, CA, 1988.

Cooperative Planning and Decentralized Negotiation in Multi-Fireboss Phoenix

Theresa Moehlman and **Victor Lesser** *
University of Massachussetts

Abstract

Multi-fireboss Phoenix provides a real time environment to study cooperative planning and decentralized negotiation. Spatially distributed agents (firebosses), having only local views, negotiate to plan a globally acceptable resource configuration. Negotiation is viewed as a distributed search through plans requiring various resource allocations, and hence, leading to different resource configurations. The goal of the distributed search is to find a resource configuration that minimizes the total loss. To realize the negotiation, a three phase framework has been created. We present an example scenario and initial implementation results to concretize the negotiation framework.

1 Introduction

In a centralized or hierarchical environment, problem solving decisions are typically made by a single agent. However, in truly distributed environments, the imposition of such centralized roles seems both inefficient and unnatural. Thus, it seems appropriate for agents to interact as peers. When problems arise in these distributed environments that affect more than one agent, all involved agents negotiate to reach a mutually acceptable solution. Negotiation is decentralized when there is no central mediator or global database and an agent must act knowing only its local state and information received from other agents. Our work entails studying decentralized negotiation in multi-fireboss Phoenix.

Multi-fireboss Phoenix is an extension of the Phoenix fire fighting system. Problem solving in this domain refers to bringing about the actions needed to assess and contain simulated fires (see [Cohen *et al.*, 1989] for a detailed account of the environment). In Phoenix, a single centralized agent controls problem solving whereas in multi-fireboss Phoenix, problem solving is managed by several spatially distributed firebosses or intelligent agents. Section 2 introduces the fire fighting domain and discusses cooperative planning in the multi-fireboss version.

*This work was partly supported by the Defense Advanced Research Projects Agency (DARPA), monitored by the Office of Naval Research under contract N00014-89-J-1877 and by the Office of Naval Research under a University Research Initiative grant, number N00014-86-K-0764.

Spatially distributed agents, having only local views, need some method of negotiation in order to plan a globally acceptable resource distribution. Each resource distribution has an associated cost and a globally acceptable distribution is one that minimizes cost. Hence, the purpose of negotiation is to construct and evaluate possible resource distributions in search of a low cost solution. In section 3, we discuss negotiation for resource allocation in multi-fireboss Phoenix.

Negotiation is viewed as a distributed search through possible resource configurations. The search is structured into three phases representing three problem solving steps. The agents need to 1) look for solutions within a given search cost level, 2) decide if they have exhausted the likely possibilities of finding a solution in a search level, and 3) if necessary, determine how to change the current search level into a new, higher cost level. Section 4 outlines our approach to negotiation. In section 5, an example scenario from the Phoenix domain is presented. Section 6 discusses related work on negotiation. Finally, we conclude with preliminary implementation results and a plan of future work.

2 Cooperative Planning in Phoenix

Phoenix simulates forest fire fighting in Yellowstone National Park. It consists of a "real world" which simulates fires and environmental conditions, agents who control problem solving, and resources that agents use to assess and contain fires. The objective of problem solving in the system is to limit the amount of land burned and to protect high priority land from destruction. Phoenix is a real time environment where problem solving is ongoing (new fires can occur at any time). Hence, agents must be able to respond quickly and adaptively to changes in the environment.

In multi-fireboss Phoenix, each agent is responsible for fires occurring in a predefined geographical area. An agent owns a certain number of watchtowers and bulldozers. Watchtowers are the "eyes" of an agent; they provide information on fires in their area of sight. Bulldozers are the primary fire fighting resource of an agent; they contain fires by building fireline around the perimeter of a fire. In response to a fire occurring in the "real world", an agent receives fire assessment information from its watchtowers, constructs a fire attack plan to contain the fire, and sends bulldozers to build the fireline specified in the attack plan.

Figure 1 shows an example fire attack. The watchtower near the top of the figure has reported the fire (shown in the center of the figure) to its agent. The dotted circular

region around the fire shows the attack plan constructed by the agent; it represents the fireline that will be built around the fire. A bulldozer, sent by the agent to implement the attack plan, is shown building fireline. When the bulldozer completes building the fireline, the fire will be contained because its fuel source will be cut off. In reality, fire can jump fireline, however, this aspect of fire growth has not yet been incorporated into Phoenix.

Agents in multi-fireboss Phoenix must allocate bulldozers with concern for all fires occurring in the system (i.e. they cooperatively plan for the current fire situation). From only local views, the agents try to assign bulldozers to fires in such a way as to globally minimize the damage of the fires. A fire fighting effort involves creating a schedule of bulldozer allocations in connection with a fire attack plan. If an agent does not have enough available bulldozers to implement the plan, it negotiates with its neighboring agents to secure the needed bulldozers for the attack.

Figure 2 shows a simple distributed resource allocation problem in the fire fighting domain. To fight the new fire, Fire-3, Agent-2 needs to secure two bulldozers immediately. Bulldozer B-4 is available and Agent-2 assigns B-4 to the attack on Fire-3. Agent-2 now needs to find one more bulldozer. The fire fighting attack on Fire-1 has just begun and Agent-2 can not release a bulldozer from that effort. If Agent-2 had a global view, it could determine that the attack on Fire-2 is almost complete and it could take a bulldozer from that attack. However, there is no global view and the agents must negotiate in order to find this solution.

The example shown is very simple and it has an obvious solution. More complex problems arise when there are more fires to fight then bulldozers available to immediately fight them. The agents must then create bulldozer allocation schedules for the fire attacks. Some fires attacks may be delayed and bulldozers can be assigned to them at a later time. Moreover, a fire fighting attack can be divided into stages where bulldozers may be added or removed from the attack at various times. Suggesting and evaluating alternative bulldozer allocation schedules is the content of agent negotiation.

3 Negotiation for Distributed Resource Allocation

Phoenix provides a real time domain where problem solving is ongoing. The objective of negotiation is to find a fairly good solution relatively quickly. Even if an exhaustive search is feasible, in a real time domain, the cost of finding an optimal solution may outweigh the savings of an optimal solution over a fairly good one. Furthermore, the ongoing nature of problem solving is also a factor in our negotiation framework. In response to changes in the current situation, agents amend existing solutions rather than start from scratch every time the current situation changes.

Agents are cooperative in distributed Phoenix - they work together to fulfill the global goal of minimizing total land loss. Hence, an agent is willing to incur more local loss than absolutely necessary in order to minimize the total loss of the system. Furthermore, negotiation is viewed as an incremental process. Agents seek solutions of minimal loss first. As the negotiation continues, the agents realize that in order to find a solution they must incur more and more loss.

3.1 Defining the Problem

We begin our study of decentralized negotiation with a two agent model. At any given time in the system, the two agents are faced with a configuration of burning fires. They must develop a resource distribution so that all fires can be contained. A fire is classified into a priority level, ranging from low to high, which estimates the amount of loss that the fire could cause. When a new fire occurs, an agent computes an initial fire projection which gives a standard base for negotiation; it indicates the lowest priority class in which a fire can be contained if the computed fireline segments are built by a certain time.

Each agent maintains a list of goals where a goal corresponds to a fire for which the agent is responsible. Each fire has an associated fire fighting projection which specifies a bulldozer allocation schedule for realizing the plan of the attack projection. Figure 3 shows an example goal. This goal corresponds to Fire-10, a medium priority fire, being fought with the attack plan of Projection-15. Bulldozers B1 and B2 are currently building fireline and B2 will leave the attack after 10 more fireline segments are built, estimated to be at time T2. After that time, B1 will complete the attack.

In this domain, two degrees of *overconstrained* resource situations are distinguished. When the agents do not have enough bulldozers to immediately implement all fire fighting attacks, they are said to be in an *ento-overconstrained* resource situation. To resolve this situation they try to rearrange the bulldozer allocation schedules and fire attack plans without raising the priority level of any goal. If the agents must raise a goal's priority in order to find a resource distribution that qualifies as a solution, they are said to be involved in an *extro-overconstrained* resource situation. In this situation, the agents know that they must delay at least one goal in order to find a solution.

So, what is the problem that the agents need to solve? At any given time, each agent has developed a resource allocation schedule for each fire occurring in its area of responsibility. Then a new fire occurs in one agent's area and there is not enough globally available bulldozers to immediately start fighting that new fire. An ento-overconstrained resource situation is recognized. The agents attempt to modify their resource allocation schedules so that the agent responsible for the new fire has a bulldozer allocation schedule for fighting that fire. If the agents can not find a bulldozer allocation that will successfully contain the new fire without raising the priority level of a goal, they are in an extro-overconstrained situation. The agents negotiate to determine which goals to delay to the next priority level. They then must develop delayed bulldozer allocation schedules for those goals.

3.2 The Elements of Negotiation

At a global level, the search space for a given situation consists of a particular set of fires (the fire situation). As shown in figure 4, alternative resource distributions occur under fire-priority configurations. Fire-priority configurations assign a priority to each fire and hence, approximate the total loss. The agents first search alternative distributions under the priority configuration representing the minimal loss. As negotiation continues, the agents realize that they must incur more loss

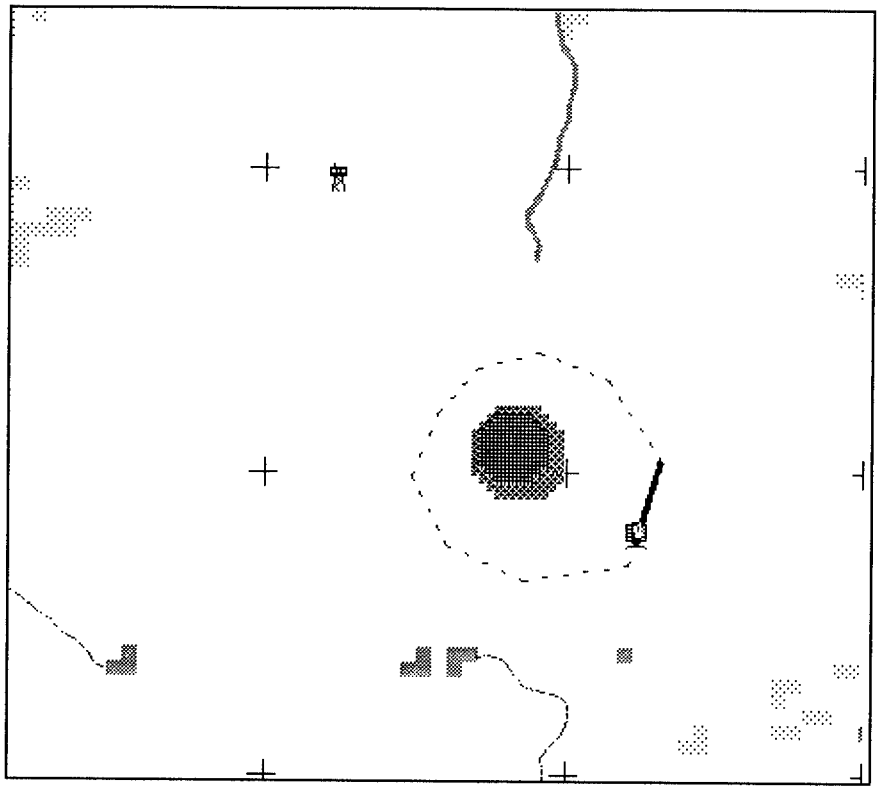


Figure 1: Fire Fighting in Phoenix

to find a solution for the fire situation and thus, must search alternatives under higher loss priority configurations.

Constraints in this domain are centered around four factors: priority, start time, bulldozer number, and bulldozer assignment time. The agents attempt to contain fires within the lowest priority classes possible. In order to contain a fire within a priority class, the computed fireline segments must be built within a certain time. The initial projection computed on a new fire specifies the minimum number of bulldozers needed to contain the fire if the attack starts immediately and if the bulldozers are allocated for the complete attack time. Thus, it provides a starting point for negotiation.

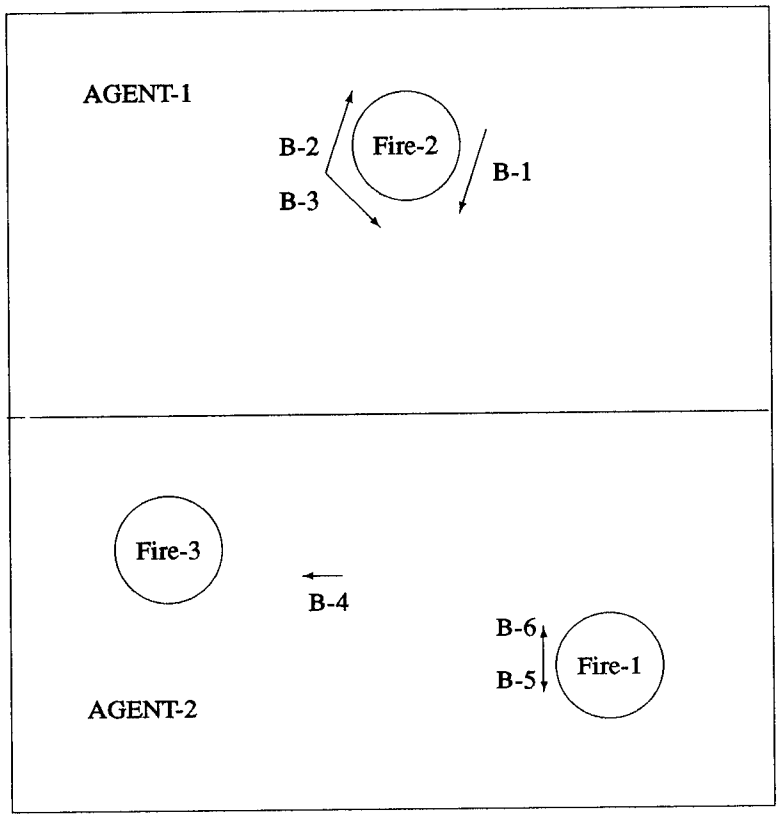
Negotiation involves relaxing constraints imposed by the projection on the new fire and the projections in use on goals. Agents first try to relax the constraints of start time, bulldozer number, and bulldozer assignment time. Relaxing these constraints provides a way to solve an ento-overconstrained resource situation. The following operators are used to find bulldozers for the new fire attack:

- The start time of a fire fighting attack may be *delayed.* If enough resources can be brought to the fire within a certain time, perhaps more than the initial projection specified, the fire can be contained within its initial priority class.
- An agent may be able to *remove* bulldozers on an already started fire fighting attack and still complete the attack specified in the projection.
- An agent may start a fire attack with *less (more)* bulldozers than the projection specified and *add (release)* bulldozers at later times. If a bulldozer schedule can be created that meets the build time constraints of the projection, the fire can be contained in its priority class.

The agents try to find a bulldozer allocation schedule that will successfully contain the new fire within its initial priority class. By using the above operators, the agents search to release bulldozers immediately, temporarily, or in the future thereby creating possible bulldozer allocation schedules for the new fire. If they can construct a bulldozer allocation schedule that achieves the build time constraints of the new fire's projection, they have solved the ento-overconstrained resource situation. If the agents can not create a suitable bulldozer schedule, an extro-overconstrained resource situation is recognized. In this situation, the agents have to agree on which goals to delay. These goals are allowed to reach a higher priority class.

In attempting to minimize global loss, the agents try to delay the fewest fire attacks as well as the lowest priority goals. The current strategy for choosing goals to delay attempts to achieve a minimal loss solution based solely on priority class. More complex strategies can include other factors such as bulldozer locations relative to fire locations, different growth rates of fires (growth rates vary according to type of terrain), and utilization of land features (for example, a fire fighting attack may utilize a lake so that less fireline has to be built).

Once the agents have decided which goals to delay, they then go back into negotiation to find bulldozer allocation schedules for the delayed goals. If the new fire is not delayed, the agents will have found a bulldozer allocation schedule for it when they chose the goals to delay. The search process to find a bulldozer allocation schedule for the delayed goals may

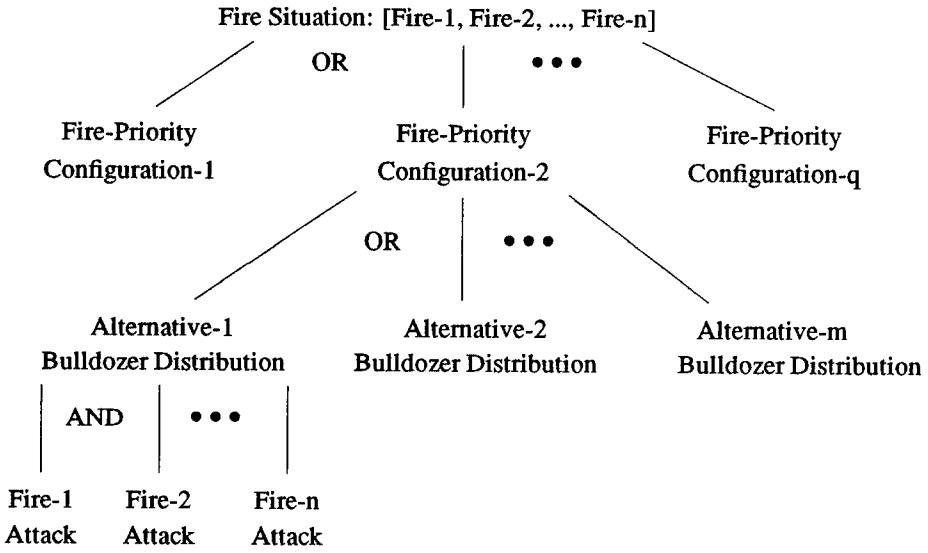


Agents negotiate in order to find the globally obvious resource distribution:
Take a bulldozer off Fire-2 to join bulldozer B-4 on the fire attack for Fire-3.

Figure 2: Distributed Resource Allocation

GOAL-10: Fire, Priority: Fire-10, Medium
Attack-Projection: Projection-15
Fighting-Projection: Entry-1: Bulldozers: (B1,B2)
Start Time, End Time: T1, T2
No-of-segments: 10
Entry-2: Bulldozers: (B1)
Start Time, End Time: T2, T3
No-of-segments: 10

Figure 3: An Example Goal



The goal is to find the alternative bulldozer allocation for the fire situation that:
Qualifies as a solution and occurs under the lowest loss fire-priority configuration possible.

Figure 4: The Global Search Space

lead back into delaying more goals and hence, changing the priority configuration again. Thus, the negotiation process is incremental.

The methods that our agents use to construct alternative bulldozer distributions are closely related to the negotiation operators used by Sathi and Fox ([Sathi and Fox, 1989]). Relaxation in our domain refers to allowing a fire to burn to a higher priority class than the minimum possible. Reconfiguration in our domain means reorganizing the bulldozer teams so that bulldozers can be released from their current assignments and hence, become available to fight a different fire. Finally, composition is similar to assigning bulldozers to a fire attack at different times and adjusting the fire fighting attack to incorporate the varying bulldozer allocations.

4 Our Basic Approach

Negotiation is structured into three phases corresponding to three main problem solving activities. The agents search within a priority configuration to find a solution for a particular fire using a specific base schedule (base schedules are created from the fire projections). If the search does not lead to a solution, the agents must decide whether to perform another search under the priority configuration with a base schedule that has not be previously used or to create a new priority configuration of higher loss. Hence, the three phases of negotiation are 1) searching to find a bulldozer allocation schedule, 2) deciding how to proceed in the search if an impasse was reached, and 3) creating a new search level of higher loss.

4.1 Phase 1: Negotiation to Find Bulldozers

During phase 1 of negotiation, agents search to find a bulldozer allocation schedule for a single fire. If that fire is a new fire, the agent responsible for it computes an initial projection which provides the ideal bulldozer allocation schedule: the minimum number of bulldozers needed to start immediately and stay for the complete attack time. Negotiation is entered when the agent does not have enough idle bulldozers to meet the ideal base schedule. The agent first tries to simply borrow the needed resources by issuing an initial request.

An initial request specifies a start time, priority, and bulldozer number. One agent is asking another: "can you give me *n* bulldozers by time *t* for a *z* priority fire?" The receiving agent replies positive, alternative suggestion, or negative as shown in figure 5. A positive reply indicates that a solution has been found. An alternative suggestion means that the agent can meet only part of the request or it can meet the whole request if the requesting agent agrees to a condition placed on the loan. A negative reply means that the agent did not find a solution or suitable alternative suggestion.

An agent receiving a request tries to fulfill that request without raising the priority of any goal. If the agent can not meet the whole request (it does not have enough idle bulldozers), it begins an ordered search to find an alternative suggestion. First, the agent tries to relax the bulldozer number (it has some idle bulldozers but not enough) or start time (it tries to complete lower priority goals before loaning bulldozers) specified in the request. If the agent does not have any idle bulldozers or lower priority goals, it searches for an alternative suggestion by restricting the bulldozer loan time or by requiring the requesting agent to commit some of its resources at a later time.

The requesting agent evaluates the alternative suggestion. As shown in figure 6, a solution is achieved if the alternative suggestion leads to a bulldozer allocation that achieves the build time constraints of the new fire and if conditions were specified on the loan, the requesting agent can meet those conditions. If alternative suggestion did not lead to a solution, the requesting agent uses the information in the alternative and its own goal situation to construct a base schedule. If the agent received a negative reply, it constructs a base schedule from its own goals alone. Note that a base schedule can be

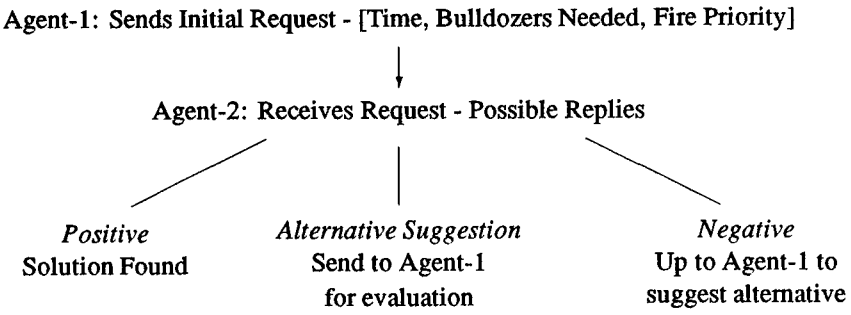


Figure 5: The Beginnings of Negotiation

created in the later phases of negotiation and lead back into phase 1 as shown in figure 6.

The requesting agent then evaluates the constructed schedule using the current projection (either initial or delayed) on the fire. If the schedule is not a solution, the agent tries to determine what will make that schedule a solution. If it can pinpoint a modification, it sends a modified request to the other agent. If it can not find a suitable modification, it realizes that further search on that base schedule is futile. The negotiation continues with phase 2.

Evaluating alternatives and pinpointing problems is specific to the domain. In the fire fighting domain, knowledge of fire behavior enables an agent to look for specific problems with a schedule. For instance, a fire grows exponentially in the direction of the wind. Thus, one function checks if the fire head is contained quickly enough. Another function checks if the problem is that a small amount of fireline is not built in time and computes when a bulldozer must be added to the attack in order to make the schedule a solution.

When an agent finds a modification that will make the base schedule a solution, it sends a modified request to the other agent. A modified request specifies a start time, bulldozer number, and priority as in an initial request. In addition, a modified request can include a reservation field and a comment field. The reservation field is employed when the modified request uses all or part of a previously sent alternative suggestion. The comment field allows an agent to give the other agent information to limit its search. For instance, if the fire head needs to be contained, the replying agent will search for ways to loan a bulldozer immediately, expect the return of that bulldozer relatively quickly, and omit searching for ways to delay the start time.

The agent receiving a modified request checks if it needs to reserve bulldozers or if it can decommit resources specified in a previous alternative suggestion. It then tries to fulfill the request as it does with an initial request though the search space may be smaller if bulldozers are reserved and a comment was included in the request. It then replies positive, alternative suggestion, or negative.

Alternative suggestions and modified requests represent a distributed search to find a solution using a particular base schedule. Modified requests attempt to revise the base schedule or a schedule derived from it to find a solution. An alternative suggestion on a modified request can lead to searching in more depth on a derived schedule or it can lead to searching on a new derived schedule. The path taken depends on the evaluation of the alternative and the requesting agent's goal situation. If no solutions were found using the base schedule, the agents enter into phase 2.

4.2 Phase 2: The Decision to Delay Goals

Deciding to delay goals implies giving up significantly more loss than could be caused by the reconfiguration process of phase 1. Since more loss is contrary to the global objective, the agents first determine if there is another means to achieve a solution under the current fire-priority configuration. Hence, this phase decides whether to try more breadth on the search under a priority class level or to search under a new priority configuration.

To determine if there is a possibility of constructing a base schedule that was not examined and is likely to lead a solution, the agents must characterize the search in phase 1. This process is the least understood of our framework. Two possible characterizations, based on the constraint relaxation of the ideal schedule, are 1) delaying the attack start time and 2) starting the attack with fewer bulldozers than required. A check list is maintained based on these two characterizations. The agent responsible for the fire checks off characterizations during the search in phase 1. Based on the modifications made to the ideal schedule and base schedules, the agent can determine which characterizations were attempted.

If a characterization has not been checked off, the agents search to create a base schedule using that characterization. If delayed start time was not used, the agents search for ways to utilize any idle resources so that more bulldozers will be released sooner then otherwise possible. If the attack was not started immediately, the agents search for ways to release bulldozers immediately (possibly requiring commitments). If they can create a new base schedule, the agents return to phase 1.

If both characterizations have been used, the agents enter phase 3. At this point, they have exhausted their known possibilities of finding a solution for the particular fire given the constraints of the fire priority configuration. Other search characterizations can be used, however, care must be taken so that the agents do not perform a redundant search on variants of the same base schedule.

4.3 Phase 3: Negotiation for Delaying Goals

Phase 3 of negotiation is entered when the agents realize that in order to find a solution to the current fire situation, more land has to be sacrificed than the current minimal amount. Which fires are allowed to burn to the next priority is based on a strategy for achieving minimal loss. The current strategy

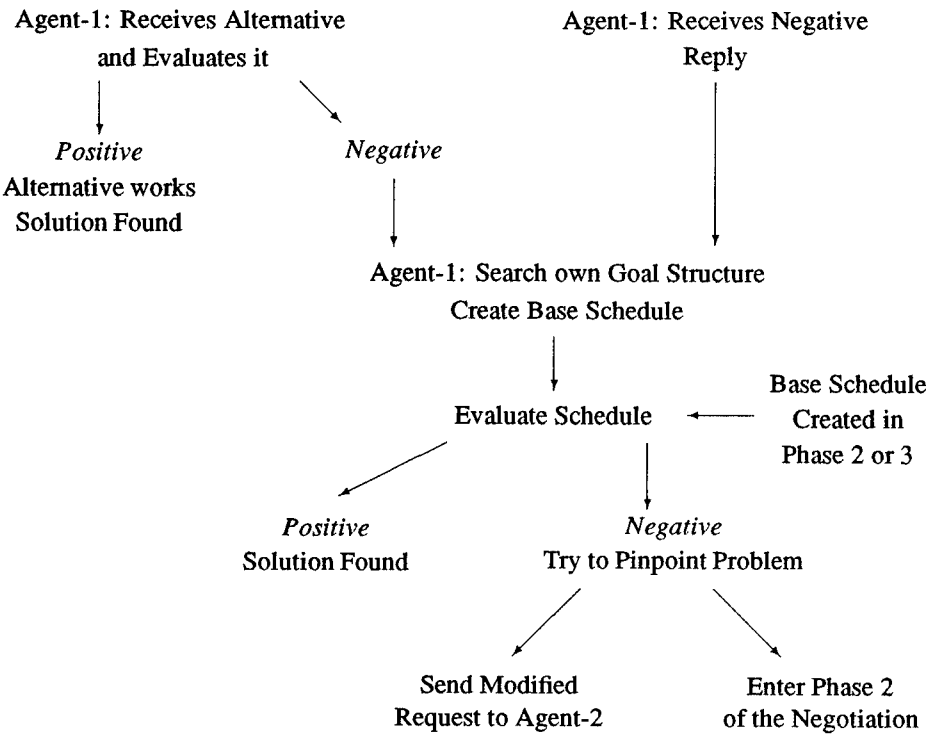


Figure 6: Search within a Priority Class Configuration

tries to raise the fewest number of fire priorities as well as to delay the lower priority fires first. Figure 7 shows the decision process.

Before negotiation can continue, agents must assess the situation. First, they find goals having a lower priority than the new fire. If the new fire is the lowest goal in the system, it will be delayed. A new projection is performed on that fire, letting it reach the next priority. The agent responsible for the fire constructs an initial schedule based on first available bulldozers and re-enters the negotiation process for a single fire using the constructed schedule as the base schedule, the new projection for evaluation of possible solutions, and the first bulldozer assignment time as the attack start time.

If the lowest priority goal in the system is not the new fire, the agents must determine if delaying that goal or any lower priority goal will lead to a solution for the new fire. They construct a schedule for the new fire using the bulldozers from the lower goal to start immediately. If any single lower priority goal will lead to solution for the new fire, it is delayed and the solution to the new fire is implemented. A projection is performed on the delayed fire, letting it reach the next priority level. An initial schedule is created and the negotiation for a delayed bulldozer schedule begins.

If no solution can be found for the new fire by delaying a single lower priority goal, a combination of goal delays is tried. However, there is a heavy bias toward delaying a single fire. Unless very low priority goals can be delayed, the new fire attack will be delayed. The bias is a way to avoid bulldozer thrashing (having bulldozers spend most of their time traveling to fires without accomplishing much useful work). If delaying several lower priority goals is chosen over delaying the new fire, it should be relatively easy to build a delayed resource schedule for those goals.

4.4 Tying it all Together

The unifying theme of the negotiation is examining resource configurations with continuing higher loss levels. The agents first seek bulldozer distributions under the minimal fire-priority configuration. If they can not find a bulldozer distribution under that configuration which qualifies as a solution, they must incur more loss. The agents must then delay goals to create a new priority configuration. They search alternative distributions under that configuration. They may find a solution or they may have to construct a new higher loss priority configuration.

The three phases give structure to the distributed search. They represent three distinct problem solving activities and they provide a way to coordinate the distributed search. Using the negotiation framework, different heuristics may be used to construct alternative resource configurations, characterize the search process, and create a new search level. Thus, the framework provides a means to study various negotiation strategies.

5 Example Scenario

To begin the testing of our approach, initial scenarios were created. These scenarios facilitate understanding what information is needed to be exchanged during negotiation and how the distributed search is conducted. Since the purpose of the section is to illustrate the basic negotiation framework and reasoning process, the following scenario is abstracted from the implementation detail.

Figure 8 shows the starting situation of the scenario. Agent-1 has two goals and no idle bulldozers. Agent-2

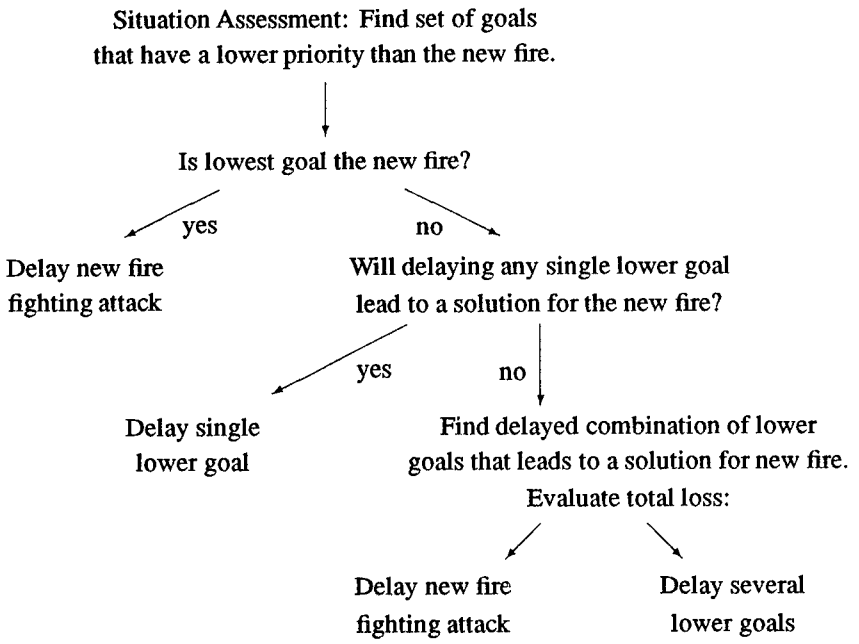


Figure 7: Deciding which Fires to Delay

also has two current goals and no idle bulldozers. Agent-2 has spotted a new fire, fire-8, in its area of responsibility and does an initial projection on it. Fire-8 is a medium priority fire needing a minimum of 2 bulldozers to start immediately.

As shown in figure 9, Agent-2 first tries a simple loan request since it does not have any idle bulldozers. Agent-1 also has no idle bulldozers. However, it does have a lower goal. So, Agent-1 constructs an alternative suggestion where it delays the bulldozer start time so it can complete its lower goal first. It computes when it will have two bulldozers available. Agent-2 receives the alternative from Agent-1. Agent-2 evaluates the alternative and finds that it is not acceptable given the build time constraints of fire-8's projection.

Agent-2 finds a base schedule using its own goals and the alternative suggestion. It can release one bulldozer in 30 minutes from fire-7 and one from fire-6 in 1 hour. Hence, a schedule of 1 bulldozer in 30 minutes and 2 follow-up bulldozers in 1 hour is constructed. This schedule is characterized as delaying the attack start time. The schedule is evaluated; it will not work. Agent-2 pinpoints the problem: in order to contain the fire head, 2 bulldozers need to start within 30 minutes. This modification is characterized as starting with fewer bulldozers since a search will now be conducted to release bulldozers as soon as possible. Agent-2 sends a modified request to Agent-1.

Agent-1 then receives the modified request asking for 1 bulldozer within 30 minutes. It searches for a way to release a bulldozer in 30 minutes. For example, it evaluates taking a bulldozer off fire-3 in 30 minutes and then putting B-3 on fire-3 in 1 hour. If Agent-1 can free up a bulldozer within 30 minutes, the agents have found a solution. However, suppose that Agent-1 can not free a bulldozer in 30 minutes. Agent-1 then sends back a negative reply to Agent-2.

Agent-2 then re-examines its own goals, searching to release a bulldozer within 30 minutes. B6 has already been reserved. Hence, the only goal to search is fire-6. It knows that the earliest time a bulldozer can be released from fire-6 is 1 hour. So, Agent-2 checks if it can temporarily take a bulldozer off fire-6, possibly requiring Agent-1 to send a bulldozer at a later time. Agent-2 may send a modified request to Agent-1 for a bulldozer at a later time so that it could complete fire-6 according to its projection. If this option is not possible or if Agent-1 can not meet a modified request, the agents enter into phase 2 of negotiation.

In phase 2, Agent-2 looks at its characterization check list. Both characterizations have been checked off. Phase 3 of negotiation is entered. At this point, the agents know a fire's priority must be raised. The agents find that fire-4 is the lowest priority class fire in the system as shown in figure 10. One bulldozer will be released from fire-4 and can be assigned to fire-8. Since Agent-2 can release B6 in 30 minutes, fire-8 can be contained within a medium priority class. Thus, the agents decide to delay fighting fire-4 because it represents the lowest loss.

Agent-1 is responsible for fire-4 and must plan a fire attack. Agent-1 computes a projection on fire-4, letting it reach the next priority, medium-low. Agent-1 must get a starting point to construct a base schedule. 1 bulldozer will be released in 1.5 hours from fire-3. It asks Agent-2 the earliest time it can loan a bulldozer. Agent-2 can release a bulldozer in 1 hour from fire-6. Agent-1 then computes a base schedule of 1 bulldozer in 1 hour and 1 bulldozer in 1.5 hours and enters back into phase 1. Note that the attack start time is considered to be 1 hour from the current time. Agent-1 evaluates the schedule. It is not a solution and Agent-1 pinpoints the problem: a small amount of fireline can not be built within the time constraints. It computes 1.5 hours as the latest time that a bulldozer must be brought to the fire. Agent-1 sends a modified request to Agent-2 asking for 1 bulldozer within 1.5 hours. Agent-2 searches its current goals and finds it can give back B3 in 1.5 hours. Hence, it sends a positive reply. Figure 11 shows the final allocation schedules.

Agent-1: Current Goals: [Fire-3, medium, (B1, B2) allocated for complete time]
[Fire-4, low, (B3) allocated for complete time]
Idle Bulldozers: 0

Agent-2: Current Goals: [Fire-6, medium, (B4, B5) allocated for complete time]
[Fire-7, medium-low, (B6) allocated for complete time]
Idle Bulldozers: 0
New Fire: Fire-8

Agent-2's Projection on Fire-8:
Medium priority; Requires 2 bulldozers to start immediately.

Figure 8: Starting Situation of Scenario

Agent-2: Sends Initial Request to Agent-1: (2 bulldozers, medium, now)

Agent-1: Receives Request.
No idle bulldozers but lower priority goal - determine when 2 bulldozers free:
1 bulldozer in 1 hour from fire-4; 1 bulldozer in 1.5 hours from fire-3.
REPLY: Alternative (trying to relax start time constraint):
1 bulldozer in 1 hour, follow-up bulldozer in 1.5 hours.

Agent-2: Receives Alternative Reply. (Ento-overconstrained situation exists)
Evaluates alternative schedule (1 bulldozer in 1 hour, follow-up in 1.5 hours)
Fire-8 will not be contained within medium priority.
Examines own goals: B-6 will be free in 30 minutes from Fire-7.
Can take a bulldozer off Fire-6 in 1 hour.
Best schedule using alternative suggestion:
1 bulldozer in 30 minutes and 2 follow-up bulldozers in 1 hour.
Characterize schedule as delayed start time.
Will schedule work? No - Pinpoint the Problem.
Need 2 bulldozers to start within in 30 minutes in order to contain fire head.
Characterize modification as starting with fewer bulldozers.
REPLY: Modified Request: (1 bulldozer, medium, 30 minutes hence)
Reservation: None; Comment: Trying to contain fire head.

Figure 9: Initial Negotiation of Scenario - Phase 1

Agent-2: Extro-overconstrained Resource Situation - Must delay fire:
Lowest goal: medium-low; Release one bulldozer.
Send information message: Must delay fire -
My lowest loss - medium-low priority and 1 bulldozer released.

Agent-1: Receives Message and examines its goals.
Replies: My lowest loss - low priority and 1 bulldozer released.

Agent-2: Receives Reply - Constructs schedule for Fire-8.
1 bulldozer now with follow-up in 30 minutes (from Fire-7).
Fire-8 can be contained within medium priority.
Send message to Agent-1: Delay your low goal.

Figure 10: The Decision to Delay a Fire - Phase 3

Fire-3: (Medium Priority) - B1 allocated for complete time;
B2 is on attack and leaves in 1.5 hours to go to Fire-4.

Fire-4: (Medium-low Priority) - B4 starts in 1 hour; B2 and B3 start in 1.5 hours.

Fire-6: (Medium Priority) - B4 is on attack and leaves in 1 hour to go to Fire-4;
B5 allocated complete time.

Fire-7: (Medium-low Priority) - B6 completes attack in 30 minutes and then goes to Fire-8.

Fire-8: (Medium Priority) - B3 is on attack and leaves in 1.5 hours to go to Fire-4;
B6 will start in 30 minutes and complete attack.

Figure 11: Conclusion of Scenario

This scenario shows how the distributed search is conducted in a typical situation of the domain. Normally, if the first base schedule constructed on a fire was characterized as starting with fewer bulldozers than required, phase 2 would have led back into negotiation with a base schedule of delayed start time. The idle bulldozers that would have been put on the new fire could be added to other goals to release bulldozers on those goals sooner than would otherwise be possible. However, when the first base schedule delayed start time, the agents usually must enter phase 3 of the negotiation and delay a goal. It is typically difficult to release bulldozers temporarily from a fire fighting attack because there is an emphasis on fighting fires with a minimal number of bulldozers.

6 Related Work

Much of the work on negotiation uses a central mediator. For example, Sycara's work ([Sycara, 1989]) on negotiation uses a central mediator to construct compromises for agents to evaluate. In our work, compromise construction is distributed. Sathi and Fox ([Sathi and Fox, 1989]) also found it necessary to use mediated negotiation when there were more than two agents. However, it can not always be assumed that there is a central mediator or global database. There may be too much information to gather in one place. In addition, a central mediator creates a bottleneck and single point of failure in the system.

Multistage negotiation is similar to our approach ([Conry *et al.*, 1988] and [Kuwabara and Lesser, 1989]). In multistage negotiation there is no central mediator. Agents exchange information to detect conflicts and overconstrained resource situations. In multistage negotiation, agents only give a positive or negative reply to a request whereas we allow another agent to make an alternative suggestion. In addition, we have a strong sense of optimization that is not present in this approach.

7 Conclusion

The basic negotiation framework has arisen out of studies on the fire fighting domain. The framework is suited for real time domains where problem solving is ongoing and no global viewpoint exists. Recently, a simplified version of the framework has been implemented in multi-fireboss Phoenix. We conclude with initial results and a plan of future work.

7.1 Preliminary Results

To begin testing our ideas, we have implemented a complete, though simplified, version of the negotiation framework. In this version, the search in phase 1 is incomplete; not all of the operators are implemented. The implemented operators are releasing bulldozers from attacks (i.e. shortening the bulldozer assignment time), delaying the start time of fire attacks, and creating bulldozer schedules with varying time allocations. Currently, there is no search to temporarily remove bulldozers from attacks. In addition, no conditions are placed on bulldozer loans. In phase 3, only one fire attack can be delayed (i.e. there is no comparison between delaying several lower goals over the new fire attack).

Phoenix simulates "real" time. Fires of medium-low to medium-high priority take about a day to fight. Furthermore, the negotiation during an overconstrained resource situation takes anywhere from 15 minutes to two hours depending on the level of activity in the system. Communication in the system is similar to sending telegrams rather than making telephone calls. Hence, there is a time lag between sending and receiving a message.

The following examples show the negotiation dialogue between the two agents, disfireboss-1 and disfireboss-2. Disfireboss-1 owns bulldozer-1, bulldozer-2, and bulldozer-3. Disfireboss-2 owns bulldozer-4, bulldozer-5, and bulldozer-6. Each agent owns 2 watchtowers. When an agent specifies that a new fire has been spotted, one of its watchtowers has sent assessment information to the agent about that fire. In the traces, schedules are lists of elements of the form (bulldozer number, start time, end time). Evaluation of a schedule compares the amount of fireline that will be built by the schedule to the needed amount of fireline specified in the fire's projection.

The first trace is an example of negotiation when a solution is reached in the first phase. Figure 12 shows the initial activity of the example. Disfireboss-1 spots a medium priority fire, actual-fire.17 and assigns 2 of its bulldozers, bulldozer-3 and bulldozer-1, to the attack. Disfireboss-2 spots a medium-low fire, actual-fire.16, and allocates bulldozer-5 to fight it. Disfireboss-2 then spots another fire, actual-fire.18. This fire is of medium priority with bulldozer-4 and bulldozer-6 allocated to fight it. Hence, Disfireboss-1 has 1 idle bulldozer (bulldozer-2) and Disfireboss-2 has no idle bulldozers.

Figure 13 shows the continuation of the example.

TIME	DISFIREBOSS-1	DISFIREBOSS-2
13:04	New fire - actual-fire.17 spotted Priority: medium	
13:06		New fire - actual-fire.16 spotted Priority: medium-low
13:08	Computing Projection for actual-fire.17 End time: 8/2 14:06 (93982) Needed Bulldozers: 2	
13:09	Can fight actual-fire.17 with my resources	Computing Projection for actual-fire.16 End time: 8/2 14:07 (94045) Needed Bulldozers: 1 Can fight actual-fire.16 with my resources
13:11	Adding agent-goal.13 for actual-fire.17 Bulldozers (bulldozer-3 bulldozer-1) Start time 4257; end-time 93982	Adding agent-goal.12 for actual-fire.16 Bulldozers (bulldozer-5) Start time 4257; end-time 94045
16:40		New fire - actual-fire.18 spotted Priority: medium
16:44		Computing Projection for actual-fire.18 End time: 8/2 17:41 (106913) Needed Bulldozers: 2 Can fight actual-fire.18 with my resources
16:46		Adding agent-goal.14 for actual-fire.18 Bulldozers (bulldozer-4 bulldozer-6) Start time 17175; end-time 106913

Disfireboss-1: Current Goals:
[Actual-fire.17, medium, (Bulldozer-3, Bulldozer-1) allocated for complete time]
Idle Bulldozers: 1

Disfireboss-2: Current Goals:
[Actual-fire.16, medium-low, (Bulldozer-5) allocated for complete time]
[Actual-fire.18, medium, (Bulldozer-4, Bulldozer-6) allocated for complete time]
Idle Bulldozers: 0

Figure 12: Example Trace - Solution in Phase I - Starting Situation

TIME	DISFIREBOSS-1	DISFIREBOSS-2
22:16	New fire - actual-fire.19 spotted Priority: medium-high	
22:22	Computing Projection for actual-fire.19 End time: 8/3 7:38 (157121) Needed Bulldozers: 2 Not enough resources for actual-fire.19 Trying resource loan	
22:23		Received Request for 1 bulldozers; medium-high priority Alternative - lower priority goal Delay Start time to 94045
22:29	Received Alternative from disfireboss-2 Schedule will not work	
22:30	Getting earliest time bulldozers released Bulldozers (bulldozer-1 bulldozer-3) Time available 93982 Evaluating Schedule ((1 37343 124645) (1 94045 124645) (2 93982 157121)) Schedule works - SOLUTION Using the following of my resources ... bulldozer-1: start 93982; end 157121 bulldozer-3: start 93982; end 157121 bulldozer-2 allocated for complete time	
22:31		Received Alternative Accepted Message
22:31		The loan ...
22:31		bulldozer-5: start 94045; end 124645

Figure 13: Example Trace - Solution in Phase I - continued

Disfireboss-1 spots a medium-high priority fire, actual-fire.19. This fire needs 2 bulldozers to start immediately. However, Disfireboss-1 has only 1 idle bulldozer. So, it sends an initial request to Disfireboss-2 for 1 bulldozer. Disfireboss-2 does not have any idle bulldozers, however, it has a lower priority goal. So, it tries to delay the start time of the bulldozer loan. Disfireboss-1 receives the alternative and creates a schedule of one bulldozer (the idle one) allocated for the completed time and another bulldozer starting at time 94045. The schedule is evaluated and found to be inadequate. So, Disfireboss-1 searches its own goals to find when it will have bulldozers available. Its only goal, agent-goal.13 for actual-fire.17, is searched. The two bulldozers from agent-goal.13 will complete the fire attack on actual-fire.17 at time 93982. The current schedule for the new fire is amended to include the allocation of the two bulldozers from agent-goal.13 at time 93982. This new schedule is found to be acceptable and hence, a solution for the new fire attack has been found.

The example shows a typical scenario when a solution is found in phase 1. 4 out of 6 bulldozers are allocated at some time to the new fire, actual-fire.19. Usually when a solution is found in phase 1, several extra bulldozers need to be allocated to the attack in order to make up for delaying the attack start time or starting the attack with fewer bulldozers. In trying to minimize loss, the evaluation only takes into account whether the needed fireline will be built. However, a more sophisticated evaluation would include a risk factor for the future. This type of evaluation may find that having so many bulldozers in one area is too risky and thus, may find the solution of the example too costly to implement.

The second trace shows the negotiation when the agents must delay a fire to find a solution. Figure 14 shows the starting situation. Disfireboss-1 spots a medium priority fire, actual-fire.5 and allocates 2 of its bulldozers (bulldozer-2 and bulldozer-1) to the fire attack. Disfireboss-2 spots a medium priority fire, actual-fire.6 and allocates 2 of its bulldozers (bulldozer-5 and bulldozer-4) to the attack. Then Disfireboss-1 spots another fire. This fire, actual-fire.7, is of medium-low priority and needs 2 bulldozers for the attack plan. Since Disfireboss-1 has only 1 idle bulldozer (bulldozer-3), it sends a request for 1 bulldozer to Disfireboss-2. Disfireboss-2 can honor the request since it has 1 idle bulldozer. Hence, bulldozer-3 and bulldozer-6 are allocated to actual-fire.7. At this point, all bulldozers in the system are being utilized.

The example continues when Disfireboss-2 spots a medium-high priority fire, actual-fire.8. Figure 15 shows the search of phase 1. Disfireboss-2 sends a request of 2 bulldozers to Disfireboss-1. Disfireboss-1 has no idle bulldozers but it does have a lower priority goal (actual-fire.7). So, it tries to delay the start time of the bulldozer loan. However, Disfireboss-2 finds the alternative to be inadequate. Disfireboss-2 then finds that it can release the bulldozers from actual-fire.8 at time 95775. But the amended schedule is not a solution. So, Disfireboss-2 sends a modified request, trying to make the current schedule a solution by adding a bulldozer to the attack at time 77807. However, Disfireboss-1 can not honor the request since a bulldozer can not be released in time from actual-fire.5 and the bulldozers assigned to actual-fire.7

TIME	DISFIREBOSS-1	DISFIREBOSS-2
13:04	New fire - actual-fire.5 spotted Priority: medium	
13:10	Computing Projection for actual-fire.5 End time: 8/2 14:07 (94072) Needed Bulldozers: 2 Can fight actual-fire.5 with my resources	
13:13	Adding agent-goal.5 for actual-fire.5 Bulldozers (bulldozer-2 bulldozer-1) Start time 4404; end-time 94072	
13:35		New fire - actual-fire.6 spotted Priority: medium
13:38		Computing Projection for actual-fire.6 End time: 8/2 14:36 (95775) Needed Bulldozers: 2
13:39		Can fight actual-fire.6 with my resources
13:40		Adding agent-goal.6 for actual-fire.6 Bulldozers (bulldozer-5 bulldozer-4)
13:41		Start time 6044; end-time 95775
14:35	New fire - actual-fire.7 spotted Priority: medium-low	
14:39	Computing Projection for actual-fire.7	
14:40	End time: 8/2 15:37 (99438) Needed Bulldozers: 2 Not enough resources for actual-fire.7 Trying resource loan	
15:27		Received Request for 1 bulldozers; medium-low priority Can honor request
15:29	Received POSITIVE from disfireboss-2	
15:34		Loaning bulldozer bulldozer-6
15:35	Now in charge of bulldozer-6 Adding agent-goal.7 for actual-fire.7 Bulldozers (bulldozer-6 bulldozer-3) Start time 12915; end-time 99438	

Disfireboss-1: Current Goals:
[Actual-fire.5, medium, (Bulldozer-2, Bulldozer-1) allocated for complete time]
[Actual-fire.7, medium-low, (Bulldozer-6 [on loan], Bulldozer-3) allocated for complete time]
Idle Bulldozers: 0

Disfireboss-2: Current Goals:
[Actual-fire.6, medium, (Bulldozer-5, Bulldozer-4) allocated for complete time]
Idle Bulldozers: 0

Figure 14: Example Trace - Delaying a Fire - Starting Situation

TIME	DISFIREBOSS-1	DISFIREBOSS-2
17:40		New fire - actual-fire.8 spotted Priority: medium-high
17:44		Computing Projection for actual-fire.8 End time: 8/3 3:01 (140493) Needed Bulldozers: 2
17:45		Not enough resources for actual-fire.8 Trying resource loan
17:48	Received Request for 2 bulldozers; medium-high priority Alternative - lower priority goal Delay Start time to 94072	
17:49		Received Alternative from disfireboss-1 Schedule will not work
17:50		Getting earliest time bulldozers released Bulldozers (bulldozer-4 bulldozer-5) Time available 95775 Evaluating Schedule ((2 94072 97240) (2 95775 140493)) Schedule inadequate - Planning modified request
17:51		Needed build time 23740.602 to make schedule a solution Modified Request: 1 bulldozers; start-time 77807 Characterized as Delaying the start time
17:55	Received Modified Request	
17:56	Sending reply of negative	
17:57		Received NEGATIVE on modified request
17:58		ENTERING Phase II
17:59		Phase 2 Trying to create base schedule starting with fewer bulldozers Can release 0 bulldozers now
18:01	Received Enter Phase II message - fewer characterization	
18:02	Can release 0 bulldozers now	
18:03		Entering Phase III

Figure 15: Example Trace - Delaying a Fire - Phase 1 and 2

TIME	DISFIREBOSS-1	DISFIREBOSS-2
18:05	Received Enter Phase III message	
18:09		Neighbor lowest priority: medium-low; Bulldozers freed: 2 My lowest priority: medium; Bulldozers freed: 2 Bulldozers I have available 0 Delay neighbor lower goal
18:18	Loaning bulldozer bulldozer-3 Returning bulldozer bulldozer-6	Back in charge of bulldozer-6 Now in charge of bulldozer-3
18:19	Deleting goal agent-goal.7 Updated goal list: (agent-goal.5)	Adding agent-goal.8 for actual-fire.8 Bulldozers (bulldozer-3 bulldozer-6) Start time 22736; end-time 140493
18:26	Computing Projection for actual-fire.7 End time: 8/3 21:31 (207112) Needed Bulldozers: 1	
18:27	Not enough resources for actual-fire.7 Trying resource loan	
18:48		Received Request for 1 bulldozers; medium priority Negative reply to request
19:09	Received DENIAL from disfireboss-2	
19:10	Getting earliest time bulldozers released	
19:11	Bulldozers (bulldozer-1 bulldozer-2) Time available 94072 Evaluating Schedule ((2 94072 207112)) Schedule works - SOLUTION Using the following of my resources ... bulldozer-1: start 94072; end 207112 bulldozer-2: start 94072; end 207112	

Figure 16: Example Trace - Delaying a Fire - conclusion

are already reserved for the new fire attack.

Disfireboss-2 receives the negative reply from Disfireboss-1. Since the bulldozers from its only goal have already been reserved, the agents enter into phase 2. The characterization of starting with fewer bulldozers has not been checked off. Hence, the agents search for a way to release bulldozers immediately. However, neither agent can release a bulldozer, so, phase 3 is entered.

Figure 16 shows the conclusion of the example. The agents have entered phase 3. They find that delaying the lowest goal in the system, agent-goal.7 (actual-fire.7), will enable Disfireboss-2 to contain the new fire, actual-fire.8, within a medium-high priority class. The bulldozers assigned to acutal-fire.7 are re-allocated to the new fire. Disfireboss-1 must then find a schedule for the delayed fire attack on actual-fire.7.

Disfireboss-1 computes a delayed projection on actual-fire.7, allowing it to reach medium priority. Disfireboss-1 finds a solution to the delayed fire attack within its own goals. By using 2 bulldozers instead of 1, the attack start time can be delayed long enough so that bulldozer-1 and bulldozer-2 can finish the fire attack on actual-fire.7.

Note that the first example takes only a quarter of an hour while the second example takes two hours. The time difference relates to the length of negotiation as well as current activity in the system. Clearly, the second example involves much more negotiation than the first example. In the first trace, most of the computations for the fire attacks have been performed before the negotiation starts. In the second example, the computations for the attacks are being performed concurrently with the negotiation. The current projections take into account the negotiation time by assuming a worst case scenario. A more intelligent agent would be aware of the time factors and adjust the projections accordingly.

In the second example, a solution was found after only one fire attack was delayed. The number of iterations through the framework loop is dependent upon the degree to which the situation is overconstrained. In other words, the farther into the future bulldozers are assigned, the more fire attacks that will have to be delayed. The second example has a low degree of overconstrainment; initially, bulldozers have only been assigned to one fire attack.

Before we can start evaluating and comparing negotiation strategies, a more complete search of phase 1 needs to be implemented. It is likely that lower cost solutions will be found once conditional loans have been included. Though the initial version is simplified, it has shown that the framework gives struture to the search process, coordinates communication, and provides a means to study decentralized negotiation.

7.2 Future Work

After we implement a more complete version of the framework, there our several directions are research can take. We can expand the basic framework. For instance, we can include an explicit model of time. Because Phoenix is a real time domain, the agents need to limit the amount of time spent negotiating and understand how the time limitations restrict the search process. In addition, the current characterization of the search process is a simple model. We feel that the characterization process is an important aspect of distributed search and a more complex model should be built.

Alternatively, we could expand the two-agent model into a multi-agent model. The actual negotiation becomes quite complex and agents would, by necessity, have to be more sophisticated. For example, if an agent commits resources to one agent, it must not commit those same resources to another agent. Furthermore, finding alternatives that minimize the global loss may not be quite as clear as in the two agent model.

In addition, the environment can be made more complex and thus, problem solving would be more difficult. For instance, bulldozers can be made to use fuel. Hence, agents would have to include time for bulldozers to get fuel in the allocation schedules. We can also add uncertainty into the environment. For example, watchtowers may not have perfect vision. Finally, environmental conditions can be made more realistic including such things as rain, wind shifts and so on.

Multi-fireboss Phoenix is a rich domain for the study of cooperative planning. Our basic framework allows us to explore different negotiation strategies and evaluation functions. We can work in incrementally more difficult environments and configurations as we learn more about cooperative planning and decentralized negotiation. Thus, Phoenix provides an environment for the long term study of real-time distributed problem solving.

References

[Cohen *et al.*, 1989] Paul R. Cohen, Michael L. Greenberg, David M. Hart, and Adele E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3):34–48, 1989.

[Conry *et al.*, 1988] Susan E. Conry, Robert A. Meyer, and Victor R. Lesser. Multistage negotiation in distributed planning. In Alan H. Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 367–384. Morgan Kaufman, 1988.

[Durfee *et al.*, 1989] Edmund H. Durfee, Victor R. Lesser, and Daniel D. Corkill. Cooperative distributed problem solving. In Avron B. Barr, Paul R. Cohen, and Edward A. Feigenbaum, editors, *The Handbook of Artificial Intelligence*, volume 4, chapter 17, pages 83–147. Addison-Wesley, 1989.

[Kuwabara and Lesser, 1989] Kazuhiro Kuwabara and Victor R. Lesser. Extended protocol for multistage negotiation. In *Ninth Workshop on Distributed Artificial Intelligence*, pages 129–161, Rosario Resort, Eastsound, Washington, September 1989.

[Sathi and Fox, 1989] Arvind Sathi and Mark S. Fox. Constraint-directed negotiation of resource reallocations. In Les Gasser and Michael N. Huhns, editors, *Distributed Artificial Intelligence*, volume 2, chapter 8, pages 163–193. Pitman Publishers and Morgan Kaufmann Publishers, 1989.

[Sycara, 1989] Katia Sycara. Multiagent compromise via negotiation. In Les Gasser and michael N. Huhns, editors, *Distributed Artificial Intelligence*, volume 2, chapter 6, pages 119–137. Pitman Publishers and Morgan Kaufmann Publishers, 1989.

Optimization of Multiple-Goal Plans with Limited Interaction*

Dana S. Nau
University of Maryland†

Qiang Yang
University of Waterloo‡

James Hendler
University of Maryland§

Abstract

Past planning systems have generally focused on control structures capable of working in all domains (domain-independent planning) or on specific heuristics for a particular applied domain (domain-dependent planning). An alternate approach is to abstract the kinds of goal and subgoal interactions that occur in some set of related problem domains, and develop planning techniques capable of performing relatively efficiently in all domains in which no other kinds of interactions occur. In this paper we will demonstrate this approach on a particular formulation of multiple-goal planning problems.

In particular, we demonstrate that for cases where multiple-goal planning can be performed by generating individual separate plans for each goal independently and then optimizing the conjunction, we can define a set of limitations on the allowable interactions between goals that allow efficient planning to occur where the restrictions hold. We further argue that these restrictions are satisfied across a significant class of planning domains. We present algorithms which are efficient for special cases of multiple-goal planning, propose a heuristic search algorithm that performs well in a more general case, and describe a statistical study that demonstrates the efficiency of this search algorithm.

*This work was supported in part by an NSF Presidential Young Investigator award for Dr. Nau with matching funds from Texas Instruments and General Motors Research Laboratories, NSF Equipment grant CDA-8811952 for Dr. Nau, NSF Grant NSFD CDR-88003012 to the University of Maryland Systems Research Center, NSF grant IRI-8907890 for Dr. Nau and Dr. Hendler, ONR grant N00014-88-K-0560 for Dr. Hendler, and NSERC operating grant OGP0089686 for Dr. Yang.

†Computer Science Department, Systems Research Center, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA. Email: nau@cs.umd.edu.

‡Computer Science Department, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada. Email: qyang@watdragon.waterloo.edu.

§Computer Science Department, Systems Research Center, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA. Email: hendler@cs.umd.edu.

1 Introduction

One of the most widely used strategies in problem-solving is to decompose a complex problem into several simpler parts. This is particularly true in planning, where a complicated goal is usually decomposed into two or more subgoals to solve. The reason for this is that decomposition tends to divide the exponent of an exponential problem, thus drastically reducing the total problem-solving effort. Korf [7], for example, has demonstrated that if the subgoals are independent, then solving each one in turn will divide both the base and the exponent of the complexity function by the number of subgoals.

The major limitation of the above approach is that although it treats the goals as independent, this condition does not really hold for most planning problems. Instead, the goals or subgoals may interact or conflict with each other.[1] Unfortunately, it appears impossible to achieve both efficiency and generality in handling goal/subgoal interactions. Domain-independent planners ([12, 14, 9, 16, 2, 18]) attempt to handle interactions which can occur in many possible forms, and thus they sacrifice the gains in efficiency which might possibly be achieved if some of these forms were disallowed. Domain-dependent planners ([1, 6, 8, 4, 5, 10]) can often do better at dealing with goal/subgoal interactions by imposing domain-dependent restrictions on the kinds of interactions that are allowed—but the restrictions they use are often too restrictive for the planners to be applicable to other domains.

In this paper, we propose an approach which falls in between domain-dependent and domain-independent planning: to abstract the kinds of goal and subgoal interactions that occur in some set of related problem domains, and develop planning techniques capable of performing relatively efficiently in all domains in which no

[1]The most famous example of this is the "Sussman anomaly," in which solving one goal undoes the independently derived solution to the other.

other kinds of interactions occur. We will refer to this approach as *limited-interaction planning.*

The restrictions which we impose on the goal interactions allow us to develop relatively efficient techniques for solving multiple-goal planning problems by developing separate plans for the individual goals, combining these plans to produce a naive plan for the conjoined goal, and performing optimizations to perform to yield a better combined plan. For example, consider the following situation (based on [17]):

> John lives one mile from a bakery and one mile from a dairy. The two stores are 1.5 miles apart. John has two goals: to buy bread and to buy milk.

The approach usually taken is to conjoin this into the single goal

```
(GOAL JOHN
      (AND (HAVE BREAD) (HAVE MILK)))
```

Suppose that we have developed separate plans for the two individual goals (drive to the dairy, buy milk, and come home; and drive to the bakery, buy bread, and come home). Taken together, these two plans will solve the conjoined goal; and the next step is to recognize that the "come home" step of the first plan can be merged with the "get there" step of the second, to produce a better plan.

The restrictions required for our approach to be applicable are limiting, but not as severely limiting as the domain-dependent heuristics used by many application-specific planners. Our goal has been to develop clear and precise restrictions which delineate a class of planning problems broad enough to be useful and interesting, but "well-behaved" enough that planning may be done with a reasonable degree of efficiency.

This paper presents one set of restrictions satisfying the above criteria, and argues that these restrictions are satisfied across a significant class of planning domains. It also discusses the complexity of the resultant planning problems, and demonstrates that limited-interaction multiple-goal planning can be performed efficiently under these restrictions.

2 Problem Statement

One example of this limited-interaction approach can be found in multiple goal planning problems. We consider a goal to be a collection of predicates describing some desired state of the world. A plan for that goal is a set of actions, together with a partial ordering on the order in which the actions must be performed,[2] such that if the actions are performed in any order consistent with the ordering constraints, the goal will be achieved. Actions can have costs, and the cost of a plan is the sum of the costs of the actions. We assume that the plans for the individual goals have already been found, and we look at how to combine them into a "global plan".

[2]In addition to the usual kind of partial ordering constraint having the form "action a must be done before action b," we also allow constraints specifying that two actions must be performed at the same time.

Depending on what kinds of interactions occur among the actions in the plans, it might or might not be possible for the plans to be combined. In this paper, we consider only the following kinds of interactions.

1. Let A be a set of actions $\{a_1, a_2, \ldots, a_n\}$. Then there may be a *merged action* $m(A)$ capable of accomplishing the effects of all actions in A, such that $\text{cost}(m(A)) < \sum_{a \in A} \text{cost}(a)$. In this case we say that an *action-merging interaction* occurs, and that the actions in A are *mergeable.*

   One way in which an action-merging interaction can occur is if the actions in A contain various sub-actions which cancel each other out, in which case the action $m(A)$ would correspond to the set of actions in A with these sub-actions removed. If the cost of each action is the sum of the costs of its sub-actions, then the cost of $m(A)$ is clearly less than the sum of the costs of the actions in A.

   Note that even though a set of actions may be mergeable, it may not always be possible to merge that set of actions in a given plan. For example, suppose a and a' are mergeable, but in the plan P, a must precede b and b must precede a'. Then a and a' cannot be merged in P, because this would require b to precede itself.

2. An *action-precedence* interaction is an interaction which requires that an action a in some plan P_i must occur before an action b in some other plan P_j. This can occur, for example, if b removes one of the preconditions necessary for a, and there is no other action which can be inserted after b to restore this precondition.

   Much previous work in planning has dealt with deleted-condition interactions, which are not precisely the same as actino-precedence interactions. However, there is a significant class of problems where action-precedence interactions are the only form of deleted-condition interactions. This class includes certain kinds of scheduling, database query-optimization, and automated manufacturing problems. Examples appear later in this section.

3. Plans for different goals may sometimes contain some of the same actions. The *identical-action* interaction occurs when an action in one plan must be identical to an action in one of the other plans.

4. Sometimes, two different actions must occur at the same time. We call such an interaction a *simultaneous-action interaction.* This is different from the identical-action interaction, because these simultaneous actions are not identical. An example would be two robotic hands working together in order to pick up an object.

The only kinds of interactions which might make it impossible to combine a set of plans into a global plan are the action-precedence, identical-action, and simultaneous-action interactions. The problem of finding out whether or not a set of plans can be combined into a global plan we call the *multiple-goal plan existence problem.*

As an added complication, each goal G_i may have several alternate plans capable of achieving it, and thus there may be several different possible identities for the global plan for G. The least costly plan for G_i is not necessarily part of the least costly global plan, because some more costly plan for G_i may be mergeable in a better way with the plans for the other goals. We define the *multiple-goal plan optimization problem* to be the problem of choosing which plan to use for each goal, and which actions to merge in these plans, so as to produce the least costly global plan for G.

Problems involving optimizing multiple-goal plans occur in a number of problem domains, such as automated manufacturing factory scheduling, and database query optimization. In these domains, multiple goals must be achieved within the context of a set of constraints (deadlines, machining requirements, etc.) The general class of *all* such problems clearly will not fit within the confines of the restrictions specified in this paper (for example, we have not yet extended our approach to deal with scheduling deadlines), but significant and useful classes of problems can be found which satisfy these restrictions. Several examples are given below.

Example 1. Consider again the shopping example given in Section 1, in which John has two goals: `(HAVE BREAD)` and `(HAVE MILK)`. To achieve the `(HAVE BREAD)` goal, a plan could be:

```
(GO HOME BAKERY),
(BUY BREAD),
(GO BAKERY HOME)
```

To achieve the `(HAVE MILK)` goal, a plan could be:

```
(GO HOME DAIRY),
(BUY MILK),
(GO DAIRY HOME).
```

Suppose it takes less time to go between the bakery and the dairy than it does to go home from the bakery and then go from home to the dairy. Then the action `(GO BAKERY HOME)` in the first plan can be merged with the action `(GO HOME DAIRY)` in the second plan, resulting in a cheaper overall plan:

```
(GO HOME BAKERY),
(BUY BREAD),
(GO BAKERY DAIRY),
(BUY MILK),
(GO DAIRY HOME).
```

Example 2. Consider the automated manufacturing problem of drilling holes in a metal block. Several different kinds of hole-creation operations are available (twist-drilling, spade-drilling, gun-drilling, etc.), as well as several different kinds of hole-improvement operations (reaming, boring, grinding, etc.). Each time one switches to a different kind of operation or to a hole of a different diameter, one must put a different cutting tool into the drill. Suppose it is possible to order the operations so that one can work on holes of the same diameter at the same time using the same operation. Then these operations can be merged by omitting the task of changing the cutting tool. This and several other similar manufacturing problems are of practical significance (see [3, 5]) and, in fact, much of the work in this paper derives from our ongoing work in developing a computer system for solving such problems [10, 11].

Suppose hole h_1 can be made by the plan

P_1: spade-drill h_1, then bore h_1;

and hole h_2 can be made by either of the plans

P_2: twist-drill h_2, then bore h_2;
P_2': spade-drill h_2, then bore h_2;

with $\text{cost}(P_2) < \text{cost}(P_2')$. If h_1 and h_2 have different diameters, then the least costly global plan will be to combine P_1 and P_2. However, if they have the same diameter, then a less costly global plan can be found by combining P_1 and P_2', merging the two spade-drilling operations, and merging the two boring operations.

Other examples include the problem of finding a minimum-time schedule for satisfying some set of orders for products to be produced in a job shop, and the problem of multiple-query optimization in database systems([13]). These examples are discussed in more detail in [19].

3 Solving the Problem

3.1 One Plan for Each Goal

Many planning systems stop once they have found a single plan for each goal, without trying to find other plans as well. When there is only one plan for each goal, the multiple-goal plan existence problem is easy to solve. Let S be a set of plans containing one plan for each goal. Unless the interactions prevent the plans in S from being merged into a global plan, one global plan is just the set of individual plans in S, with additional ordering constraints imposed upon the actions in these plans in order to handle the interactions. This *combined* plan is called combine(S), and we have developed an algorithm to produce it in time $O(n^3)$, where n is the total number of actions in the plans (see [19]).

The plan combine(S) is not necessarily optimal—and even when there is only one plan available for each goal, the multiple-goal plan optimization problem is NP-hard (see [19]). But to make the problem easier to solve, we can impose restrictions on how the goals can interact with each other.

Restriction 1. If S is a set of plans, then the set of all actions in S may be partitioned into equivalence classes of actions $E_1, E_2, \ldots, E_p$, such that for every set of actions A, the actions in A are mergeable if and only if they are in the same equivalence class. We call these equivalence classes *mergeability classes.*

Restriction 2. If combine(S) exists, then it defines a partial order over the mergeability classes defined in Restriction 1; i.e., if E_i and E_j are two distinct mergeability classes and if combine(S) requires that some action in E_i occur before some action in E_j, then combine(S) cannot require that some action in

E_j occur before some action in E_i. (This does not rule out the possibility of an action in E_i occurring immediately before another action in E_i; in such a case, the two actions can be merged.)

Restriction 1 is reasonable for a number of problems (for example, it is already satisfied in the Examples 1 and 2). Restriction 2 is more limiting in general, but it still allows a number of important problems to be included. For instance, Restriction 2 is trivially satisfied in Example 1 since there is only one possible merge. In Example 2 it is satisfied in a more interesting way, since there exists a common sense ordering of the machining operations.

We have designed an algorithm for finding a least costly plan, with a worst case time complexity of $O(n^3)$ (see [19]), where n is the total number of actions in the plans.

3.2 More than One Plan for Each Goal

For some multi-goal planning problems, it is reasonable to expect that more than one plan may be found for each goal. (For example, this is done by the SIPS planning system for the manufacturing problem discussed in Example 2 [10]). Finding more than one plan for each goal is more complex computationally than finding just one plan for each goal, but it is useful because it can lead to better global plans.

3.2.1 A Heuristic Algorithm with Multiple Plans per Goal

If more than one plan is available for each G_i, then there may be several different possible identities for the set of plans $S = \{P_1, P_2, \ldots, P_g\}$, and it may be necessary to try several different possibilities for S in order to find one for which combine(S) exists. This means that for this problem, the multiple-goal plan existence and optimization problems are both NP-hard[3]. Here we present a heuristic approach for solving these problems, which uses a best-first branch and bound algorithm to search through the space of all possibilities for S. The details of this algorithm are described in [19].

Suppose that we are given the following: (1) for each goal G_i, a set of plans T_i containing one or more plans for G_i, and (2) a list of the interactions among the actions in all of the plans. In the search, the state space is a tree. Each state is a set of plans; it contains one plan for each of the first i goals for some i. The initial state is the empty set (i.e., $i = 0$). If S is a state containing plans for the goals $G_1, G_2, \ldots, G_i$, then an immediate successor of S is any set $S \cup \{P\}$ such that P is a plan for G_{i+1}. A goal state is any state in which plans have been chosen for all of the goals $G_1, G_2, \ldots, G_g$. The cost of a state S is the cost of the plan obtained by applying to S the merging algorithm for one plan per goal; i.e.,

$$\text{cost}(S) = \text{cost}(\text{merge}(\text{combine}(S))).$$

The search algorithm is a best-first branch-and-bound search using a lower bound function L to order the members of the list of alternatives being considered. If $L(S)$ is a lower bound on the costs of all successors of S that are goal states, then the algorithm is guaranteed to return the optimal solution. We now discuss various possible functions to use for L. To do this, we temporarily assume the following property: that merging plans for two different goals always results in a plan at least as expensive as either of the two original plans. In other words, if P and Q are plans for two distinct goals, then

$$\text{cost}(\text{merge}(\text{combine}(P, Q))) \geq \max(\text{cost}(\text{merge}(P)), \text{cost}(\text{merge}(Q))). \quad (1)$$

In [19], we discuss what happens when this property is not satisfied.

If Eq. (1) is satisfied, then clearly $L_0(S) = \text{cost}(S)$ is a lower bound on the cost of any successor of S (this would correspond to using $h \equiv 0$ in the A* search algorithm). However, a better lower bound can be found as follows. We associate with each state S some sets $H_1(S), H_2(S), \ldots, H_g(S)$, which are computed as follows. For the initial state ($S = \emptyset$), for $j = 1, 2, \ldots, g$,

$$H_j(S) = \{\text{all actions in } P | P \text{ is a plan for } G_j\}. \quad (2)$$

Let S be any state at level $i-1$, and let S' be the state formed from S by including a plan P_i for the goal G_i. Then, for $j = i+1, \ldots, g$,

$$H_j(S') = \{Q' | Q \in H_j(S)\}, \quad (3)$$

where Q' is Q minus each action that falls into the same mergeability class as some action in P_i. Thus each member of $H_j(S')$ is the set of all actions a in some plan for G_j such that a cannot be merged with any action in S'.

If $H_i(S)$ and $H_j(S)$ are as defined in Eqs. (2) and (3), then we define them to be *strongly connected* if for some $s \in H_i(S)$ and $q \in H_j(S)$, s and q contain some actions that are mergeable. $H_i(S)$ and $H_j(S)$ are *connected* if they are strongly connected, or if there is a set $H_k(S)$ such that $H_i(S)$ is connected to $H_k(S)$ and $H_k(S)$ is strongly connected to $H_j(S)$. Connectedness is an equivalence relation, so let $C_1(S), C_2(S), \ldots,$ be the equivalence classes it induces over the set $\{H_j(S') | j = i+1, \ldots, g\}$. We refer to these equivalence classes as *connectedness classes.*

Suppose S' is an intermediate state during the search, at level i of the search tree. For each connectedness class $C_i(S')$, let

$$L_{3i}(S') = \max\{\min\{\text{cost}(Q) \mid Q \in H_j(S')\} \mid H_j(S') \in C_i(S')\}, \quad (4)$$

where the min of an empty set is taken to be 0. The new lower-bound function is defined to be

$$L_3(S') = \text{cost}(\text{merge}(\text{combine}(S'))) + \sum_{i=1}^{r} L_{3i}(S'). \quad (5)$$

It can easily be shown that L_3 is admissible.

Now we consider the computational complexity for evaluating L_3. Suppose that state S' was formed by adding some plan P_i to S. The sets $H_i(S')$ can be obtained according to Eq. (3). Since $H_i(S')$ and $H_j(S')$ are subsets of $H_i(S)$ and $H_j(S)$, respectively,

[3] However, polynomial-time solutions do exist for several special cases [19].

$H_i(S')$ and $H_j(S')$ cannot be connected unless $H_i(S)$ and $H_j(S)$ are connected. Thus the connectedness classes $C_1(S'), C_2(S'), \ldots$, can be computed by starting splitting $C_1(S), C_2(S), \ldots$, into subclasses. Let $h = \max(|H_j|)$, $v = \max(|V|)$, and $u = \max(|C_i|)$, where $V \in H_j, H_j \in C_i, i = 1, \ldots, r$, . Then computing L_3 for state S' takes time $O(r(u^3(hv)^2 + h|P_i|))$, where P_i is the plan newly included into S'.

3.2.2 Experimental Results with Multiple Plans per Goal

In the worst case, the search algorithm takes exponential time. Since the multiple-goal plan optimization problem is NP-hard, this is not surprising. What would be more interesting is how well the search algorithm does on the average. Since it is hard to characterize what the "average case" is, we restricted ourselves to doing empirical performance evaluation on a class of problems that seemed to us to be "reasonable."

We conducted experiments with the algorithm for planning in the automated manufacturing domain. The problem to be solved was to find a least-cost plan for making several holes in a piece of metal stock, as described in Example 2. For this experiment, we randomly generated specifications for 100 machined holes, randomly varying various hole characteristics such as depth, diameter, surface finish, locational tolerance, etc. We used these holes as input to our EFHA process planning system [15], telling it to produce at most 3 plans for each hole. EFHA found plans for 81 of the holes (for the other 19 the machining requirements were so stringent that EFHA could not produce any plans for them). The distributions of the hole characteristics were chosen to give a wide selection of plans, not very many "obviously best" plans, lots of opportunities exist to merge actions in different plans, and the necessity of making tradeoffs in choosing which plans to merge.

The results of the experiments are shown in Table 1. Each entry in the table represents an average result over 450 trials. Each trial was generated by randomly choosing n of the 81 holes (duplicate choices were allowed), invoking the search algorithm on the plans for these holes using the lower bounding function L_3, and recording how many nodes it expanded in the search space. The total cost of each plan was taken to be the sum of the costs of the machining operations in the plan and the costs for changing tools. The average number of nodes in the search space (column 2 of the table), the average number of nodes expanded by the algorithm (column 3 of the table), and the average fraction of the search space expanded by the algorithm (the quotient of columns 2 and 3) closely match the functions $y = 1.26(2.89^n)$, $y = 1.24(1.35^n)$, and $y = 1.03(0.469^n)$, respectively.

We regard the performance of the algorithm as quite good—especially since the test problem was chosen to be significantly more difficult than the kind of problem that would arise in real-world process planning. In real designs, designers would normally specify holes in a much more regular manner than our random choice of holes, making the merging task much easier. For example, when merging real-world process plans, we doubt that there would be many of the mergeability tradeoffs mentioned earlier; and without such tradeoffs, the complexity of the algorithm is polynomial rather than exponential.

Table 1: Experimental results for the search algorithm using L_3, on the process plans for some randomly chosen holes.

Number of holes n	Nodes in the search space	Nodes expanded
1	2	1
2	10	2
3	34	3
4	98	4
5	284	6
6	852	9
7	2372	12
8	6620	16
9	19480	22
10	54679	28
11	153467	38
12	437460	51
13	1268443	61
14	3555297	86
15	9655279	110
16	29600354	170
17	80748443	223
18	250592571	250

4 Conclusion

In this paper, we have proposed an approach which falls in between domain-dependent and domain-independent planning: to abstract the kinds of goal and subgoal interactions that occur in some set of related problem domains, and develop planning techniques capable of performing relatively efficiently in all domains in which no other kinds of interactions occur. This paper has concentrated on a particular example of limited-interaction planning: how to do plan optimization in the presence of a particular set of limitations on goal/subgoal interactions. The results are summarized below.

The multiple-goal plan optimization problem is NP-hard. However, by imposing some restrictions that are reasonable for some problem domains, the problem can be made computationally easy when there is only one plan available for each goal. If each goal has multiple alternate plans, the problem is still NP-hard even with the restrictions, but in this case there is a good heuristic approach for solving the problem.

We regard this work as a first step, which demonstrates the potential improvements to planning that can be found by exploiting restrictions on allowable interactions. In our future work, we would like to explore natural extensions of our approach for creating plans rather than just optimizing existing plans. In addition, it may be possible to develop similar techniques for use in planning or plan optimization, in cases where the interactions sastisfy restrictions other than the specific ones

described in this paper. As we continue our research into more general forms of limited-interaction planning, we are convinced that this approach has potential for significantly improving the performance of planning systems across a number of additional domains.

References

[1] T.C. Baker, J.R. Greenwood "Star: an environment for development and execution of knowledge-based planning applications" *Proceedings DARPA Knowledge-based Planning Workshop*, Dec. 1987.

[2] D. Chapman, "Planning for Conjunctive Goals," *Artificial Intelligence* (32), 1987, 333-377.

[3] T. C. Chang and R. A. Wysk, *An Introduction to Automated Process Planning Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[4] M. R. Cutkoski and J. M. Tenenbaum, "CAD/CAM Integration Through Concurrent Process and Product Design," *Proc. Symposium on Integrated and Intelligent Manufacturing at ASME Winter Annual Meeting*, 1987, pp. 1-10.

[5] C. Hayes, "Using Goal Interactions to Guide Planning," *Proc. AAAI-87*, 1987, 224-228.

[6] Garvey, T. and Wesley, L. "Knowledge-based Helicopter Route Planning" *Proceedings DARPA Knowledge-based Planning Workshop*, Dec. 1987.

[7] Korf, R.E., "Planning as Search: A Quantitative Approach," *Artificial Intelligence* (33), 1987, 65-88.

[8] Linden, T., and Owre, S. "Transformational Synthesis Applied to ALV Mission Planning" *Proceedings DARPA Knowledge-based Planning Workshop*, Dec. 1987.

[9] D. McDermott *Flexibility and Efficiency in a Computer Program for Designing Circuits*, AI Laboratory, Massachusetts Institute of Technology, Technical Report AI-TR-402, 1977.

[10] D. S. Nau, "Automated Process Planning Using Hierarchical Abstraction," Award winner, Texas Instruments 1987 Call for Papers on Industrial Automation, *Texas Instruments Technical Journal*, Winter 1987, 39-46.

[11] D. S. Nau, R. Karinthi, G. Vanecek, and Q. Yang, "Integrating AI and Solid Modeling for Design and Process Planning," *Proc. Second IFIP Working Group 5.2 Workshop on Intelligent CAD*, Cambridge, England, Sept. 1988.

[12] E. D. Sacerdoti, "A Structure of Plans and Behavior," *American Elsevier, New York*, 1977.

[13] T. Sellis, "Multiple-Query Optimization," *ACM Transactions on Database Systems* (13:1), March 1988, 23–52.

[14] A. Tate, "Generating Project Networks," *Proc. IJCAI*, 1977, 888-893.

[15] S. Thompson, "Environment for Hierarchical Abstraction: A User Guide," Tech. Report, Computer Science Department, University of Maryland, College Park, 1989.

[16] S. A. Vere, "Planning in Time: Windows and Durations for Activities and Goals," *IEEE Transactions on Pattern Analysis and Machine Intelligence* (PAMI-5:3), 1983, 246-247.

[17] R. Wilensky, *Planning and Understanding*, Addison-Wesley: Reading, Massachusetts, 1983.

[18] D. Wilkins, "Domain-independent Planning: Representation and Plan Generation," *Artificial Intelligence* (22), 1984.

[19] Q. Yang, D. S. Nau, and J. Hendler, "Exploiting Limited Interactions in Plan Optimization," submitted for publication, 1990. Available as Tech. Report CS-TR-2411, Computer Science Department, University of Maryland, College Park, 1990.

Deferred Planning and Sensor Use*

Duane Olawsky and **Maria Gini**
Computer Science Dept., University of Minnesota
4-192 EE/CSci Building
200 Union Street SE
Minneapolis, MN 55455
olawsky@umn-cs.cs.umn.edu
gini@umn-cs.cs.umn.edu

Abstract

Traditional approaches to task planning assume the planner has access to all of the world information needed to develop a complete, correct plan—a plan which can then be executed in its entirety by a robot. We consider problems where some crucial information is missing at plan time but can be obtained from sensors during execution. We discuss the solution of these problems through deferred planning (i.e., by deferring specific planning steps until more complete information is available and then restarting the planner). We also present early results of a comparative study of strategies for deciding which plan steps to defer.

1 Introduction

Traditional approaches to task planning assume that the planner has access to all of the world information needed to develop a complete, correct plan—a plan which can then be executed in its entirety by a robot. Unfortunately, this information about the world may not always be available at plan time. This is particularly true when we consider autonomous robots that must operate under general goals over extended periods in unpredicatable, and changing environments. When crucial information is missing at plan time, it may be impossible to find a complete plan without obtaining additional information. Fortunately, this information is often available at execution time through the use of sensors. The problem, then, is how to integrate execution time sensory data into the planning process which, in traditional approaches, is completed before execution begins.

The ability to integrate sensory data into the planning process is important. First, it provides greater robustness for autonomous robots. With this capability a robot could complete novel variations of tasks by realizing what information it knows and what it must find out through sensor use. It could then obtain the necessary information and perform the task. In this way the robot could work around its incomplete knowledge, filling in the gaps, to solve what would otherwise be an unsolvable problem. Second, this integration is helpful in robot recovery from execution errors and unexpected events. In these cases, it is likely that significant world information is missing (or at least in doubt). A robot controller could use this ability to collect the required information and then finish its task.

There are two reasons why it is difficult to integrate sensory data into the planning process. The first has already been mentioned—the planning process is traditionally completed before execution (and therefore sensor use) begins. The second difficulty is that the information obtained from sensors can have a dramatic effect on the shape of the plan. To make our discussion more concrete we will use the following tool box domain throughout this paper.

The robot is in a room with n tool boxes $t_1, t_2, \ldots t_n$, each containing wrenches and bolts of various sizes. The robot knows the initial locations of the wrenches and bolts. Bolts are identified by a unique name, and wrenches are identified by size (assume one wrench per size). The robot has been instructed to close and bolt one or more tool boxes with particular bolts. To perform each bolting operation, the robot must use a wrench of a size that matches the bolt. A sensor is available that can classify bolts by the size (e.g., a number from 1 to 10). For simplicity, the bolts sizes are indicated along the same scale as the wrench sizes. We also assume the robot has a tool belt into which it can put an unlimited number of bolts and wrenches.[1]

Figure 1 describes a sample problem instance. There are two tool boxes, s and t. Box t is to be bolted with bolt b_t. Initially, the robot is at box t. There are two wrenches available, one of size 4 and another of size 5. The correct action sequence will vary depending upon which tool box contains the needed wrench, and this in turn depends on the size of b_t. The plan when the wrench is in s will differ from the plan when the wrench is in t.

*This work has been funded by the NSF under grants NSF/DMC-8518735 and NSF/CCR-8715220.

[1] We are not concerned here with the arm-empty conditions used to define the blocks world. Our main goal in defining this domain is to study how sensor use can be interleaved with planning.

Initial State:
((at t)(bolt-not-inserted s)(bolt-not-inserted t)
(open s)(closed t)
(wrench-in-tbox 4 t)(wrench-in-tbox 5 s)
(bolt-in-tbox b_t t))

Goal State:
((bolted t b_t))

Figure 1: Sample Problem.

(Open-Tbox t)
(Get-Bolt b_t)
(Get-Wrench 4)
(Close-Tbox t)
(Insert-Bolt b_t t)
(Bolt t b_t)

Figure 2: Sample plan when b_t has size 4.

Sample action sequences are shown in Figures 2 and 3.

If the planner knows the size of b_t, it can find a complete plan before execution begins. Otherwise, the robot must use its sensors during execution to obtain the bolt size, and this information then determines the further actions that are necessary to achieve the end goals.

2 Why Use Conventional Planning

Planning is desirable in robotics because it attempts to map out future activities of the robot so that the robot avoids undesirable situations during plan execution. Although planning systems are known to suffer from computational complexity, with well crafted heuristics they have proven to be useful even for complex tasks [Wilkins, 1989].

A well recognized problem with planning is the inability of most planners to deal with the inexactness and noise of the real world. Several solutions have been proposed including the following:

- eliminating planning altogether in favor of reactive planning [Brooks, 1986] or situated systems [Agre and Chapman, 1987, Kaelbling, 1988],
- combining reactivity and planning [Georgeff and Lansky, 1987, Drummond, 1989, Hayes-Roth, 1987, Nilsson, 1989],
- preplanning for every contingency [Schoppers, 1987],
- verifying the executability of plans and adding sensing whenever needed to reduce the uncertainty [Brooks, 1982, Doyle, Atkinson and Doshi, 1986],
- interleaving planning with execution [Durfee and Lesser, 1989, Turney and Segre, 1989, Dean and Boddy, 1988, Hsu, 1990, McDermott, 1978].

(Open-Tbox t)
(Get-Bolt b_t)
(Close-Tbox t)
(Insert-Bolt b_t t)
(Goto s)
(Get-Wrench 5)
(Goto t)
(Bolt t b_t)

Figure 3: Sample plan when b_t has size 5.

Reactive systems, which are often proposed as a solution to the problems with conventional planning, suffer from being myopic. They tend to react to local changes, and have a short-term view of the problem they are trying to solve.

We are interested in exploring how to use conventional planning in domains in which the plan-time information is incomplete. This includes exploring strategies to maximize the chances of producing a plan that, despite incomplete knowledge, avoids premature actions.

3 Adapting Conventional Planning Techniques

The next question is how best to use conventional planning techniques to solve the problems we are considering. Is it necessary to extend these techniques in some way, or can we just define new operators at the correct level of abstraction that will allow a conventional planner to handle these problems? We contend that extensions are necessary. To demonstrate this, we attempt to define the required operators and point out the difficulties we encounter.

We must define the operators so that the planner need not be explicitly aware of the fact that sensors are being used. Thus, no sensor processes will be available to the planner. Assume that the size of some bolt B is unknown. We begin by collapsing two separate subgoals of the BOLT process with the properties (Boltsize B ?z) and (Have-Wrench ?z), into a single goal with property (Have-Wrench-for-Bolt B). In this way we hide the size of the bolt and the identity of the matching wrench. Let this new goal be achieved by the process (Get-Wrench-For-Bolt B). It is this process upon which we must focus. What effect does this process have on the world state. At the very least, after executing this process, the robot will have a wrench that it did not have before, and that wrench will no longer be in any tool box. (It is also likely that the robot will be in a different location.) The crucial observation is that the planner cannot know which wrench has been removed from a tool box. The identity of that wrench is determined entirely by execution-time sensory data that is not available to the planner. Thus, from the planner's perspective, (Get-Wrench-For-Bolt B) has nondeterministic effects, and this is problematic in conventional planners. If we allow such nondeterministic effects, the planner will have difficulty solving other goals that require obtaining a wrench since it no longer knows the location of all of the wrenches.

It thus appears necessary to extend conventional planning techniques to deal with the class of problems we are considering. There are three basic ways to do this:

1. Find a complete plan (or set of plans) that will work for all possible values of the relevant sensor reading. That is, plan for all contingencies (This is similar to universal planning [Schoppers, 1987] and to "tree plans" [Nilsson, 1989]).
2. Find a single complete plan based on an assumed value of the sensor reading. This plan will work (without modification) only if the assumption is correct.
3. Defer planning decisions that depend on sensor readings until those readings are available, then continue planning with the new information.

Which of these strategies is appropriate depends on external considerations such as the criticality of mistakes (i.e., Are they reversible? Is reversal costly?), the complexity of the domain, and the acceptability of suspending execution to do more planning.

3.1 The Three Approaches Compared

Planning all paths is often expensive and difficult and should be avoided if possible. If there are 20 different sizes of bolt, the planner might need to find a slightly different plan for each of the 20 possible sensor values. Matters are even worse in the likely event that more than one sensor reading is required. If the size of two different bolts must be determined by sensor readings from 20 possible values, there would be 400 combinations, each of which might correspond to a slightly different plan. The amount of planning grows exponentially in the number of readings that are needed. Although it might be possible to represent these 400 possible plans efficiently through the use of disjunctive nodes in the plan network, this does not really solve the problem. To do complete preplanning, the planner must still analyze the potential interactions (e.g., conflicts) that arise when any of the 400 possible combinations occurs. Despite the expense of this approach, there are still cases where it might be appropriate if it is computationally feasible:

- The same plan will be used many times with potentially different sensor values in each execution. Note that the same initial state must be satisfied in each use of the plan. In this case the cost of the plan is justified by its long-term usefulness.
- Time constraints during execution make it undesirable or impossible to do any execution-time planning (either deferred planning or replanning).
- The criticality of errors in the plan is so high that the cost of extra planning is outweighed by the cost of a mistake.

Unfortunately, even with all the planning effort associated with this approach, most execution-time errors and unexpected events are not anticipated. Unless these problems can be anticipated and handled in the plan, replanning may still be necessary. Due to the size and complexity of a plan in this approach, replanning to correct these problems could be difficult and costly.

Although approach (2) is less expensive, there is always a possibility that the assumptions made were incorrect, and the plan is therefore invalid. When this happens, replanning is necessary. Parts of the original plan will likely be discarded, and as a result, some planning effort is wasted. It is also possible that, due to the assumptions, some action is taken prematurely and must later be undone. If the premature action is irreversible, it might be impossible to solve the problem. Approach (2) is most appropriate when the following are all true:

- It is acceptable to have the robot stop during execution while replanning occurs.
- The criticality of plan errors is low. That is, actions are reversible, or the cost of failure is small (e.g., the robot can throw away an inexpensive part and start over with a new one).
- Some particular value for a sensor reading is more likely than any of the other possible values. In this case the planner has something upon which to base its guess. The odds are more in its favor.

One advantage of this approach over the deferred planning approach discussed below is that, when the planner guesses correctly, no execution-time planning is needed. However, if the planner guesses incorrectly, the time needed for replanning will probably be longer than the time needed to continue planning in a deferred planner since the replanner usually must remove parts of the original plan.

In the same vein, probabilistic reasoning has been proposed to reduce the complexity of planning. For instance, when expectations are available concerning how long propositions are likely to persist, probabilistic predictions can be made [Dean and Kanazawa, 1988]. Drummond [Drummond and Bresina, 1990] proposes an algorithm that maximizes the probability of satisfying a goal. The algorithm achieves a balance, in terms of robustness, between triangle tables [Fikes and Nilsson, 1971] and universal plans [Schoppers, 1987].

With the deferred planning approach, the planner avoids doing a lot of work that will later be discarded. Instead, it completes only those portions of the plan for which it has enough information at plan-time. Since the planner, in its initial phase, does not find a complete plan, there is the possibility that important dependencies and constraints in the plan will be missed. In this case some action might be taken prematurely which must later be undone. As with the replanning approach, if the premature action is irreversible, it might be impossible to solve the problem. Thus, care must be taken to detect these dependencies and constraints as early as possible before the robot has taken too many actions. Deferred planning is appropriate when the following are true:

- It is acceptable to have the robot stop during execution while planning continues.
- The criticality of plan errors is low. That is, actions are reversible, or the cost of failure is small.

It is the deferred planning approach that we are studying. The central problem for this approach is how to avoid premature actions that must be reversed (or even

worse, that cannot be reversed). In Section 3.2 we will describe how we have implemented this approach, and integrated it with an execution simulator. In Section 3.3 we will give an example of how this system works. Section 4 outlines a number of strategies for deciding which plan goals to defer.

3.2 A Deferring Planner

The basis of our system is an agenda-controlled planner called BUMP (Basic University of Minnesota Planner). BUMP uses STRIPS-style operators [Fikes and Nilsson, 1971] to build a plan network consisting of goal nodes and process nodes. At present, BUMP is very basic in that it does not do hierarchical planning [Sacerdoti, 1974], nor does it use special methods to reason about resources [Wilkins, 1988]. It does maintain links from process nodes to goal nodes that record the purposes of each process node in the plan. The other major component is the EXECUTION CONTROLLER (EC). This controller is at the top-level in our system. It invokes BUMP to get solutions (plans) for particular problems, and it then controls the execution (in simulation) of the steps within those plans. It can also invoke the planner on a partially specified plan, asking BUMP to finish it. A system diagram is shown in Figure 4.

To solve the problems with which we are dealing, the BUMP plan must contain requests for sensor readings that obtain the information that the planner is missing. This is accomplished by adding a new type of process node to the planning system. A SENSOR PROCESS NODE constitutes an instruction to the execution controller (and hence the robot) to take a particular sensor reading at a particular point in the execution. We assume that the results of a sensor process can, at the planner's level of abstraction, be described by one or more logical predications.[2] This allows us to represent sensor processes in much the same way as non-sensor processes. That is, they are described by three lists of predications:

Add List — A list of predications describing the properties asserted as a result of the process. At least one of these will be the new information obtained by the sensor. This list can also specify side effects of the sensor process.

Delete List — A list of predications for properties denied as a result of the process. (This would likely include things that are changed in the world as side effects of sensing.)

Precondition List — Properties that must be true in order to use the sensor. This list will be used to generate the set-up actions for the sensor.

Since sensor processes are explicitly represented in the plan in much the same way as all other processes, their side effects as well as their set-up actions can be dealt with by BUMP.

A sensor process is used (like any other process node in a BUMP plan) to achieve one or more of the properties on its Add List. For example, to solve a goal node G for property (Boltsize B ?z), BUMP can insert a sensor process node (SENSE-BOLTSIZE B) into the plan. This sensor process node, when executed, will assert that the bolt B has some particular size as determined by the relevant sensor or sensors. The node could for example assert the property (Boltsize B #4). If some property matching (Boltsize B ?z) is already asserted, either in the initial state or by some process node that can occur before G, then the planner can solve G by performing the appropriate linking operation. No additional sensor process node is needed. Thus, the planner can easily recognize what information it already has available and what information must be obtained from sensors. Furthermore, it performs this reasoning through the same mechanisms that determine whether to use a helpful interaction or an operator to solve a goal. While planning, BUMP uses special dummy constants in place of the values that will come from sensor readings. During the initial planning phase, the plan variable ?z from (Boltsize B ?z) will be bound to one of these constants. Any subsequently attempted plan goals that refer to one of these constants will be deferred until the executor has obtained the reading.

When all goals in the plan network have been either solved or deferred, BUMP returns the plan at its current state of completion. The execution controller then begins executing the partial plan, preferring sensor processes over other parallel processes since the former increase the robot's information about the world. This preference also extends to the set-up actions of sensor processes and to any other process nodes that are constrained to occur before a sensor process. This strategy is intended to obtain the sensory data at the earliest possible point in execution in order to avoid the problems caused by premature actions. Once a plan-requested sensor reading is obtained, BUMP is immediately restarted with the new information which it can use to make additional plan decisions. BUMP returns a new (perhaps still partial) plan to the executor. This cycle continues until all the necessary sensing has been done and BUMP has found a complete plan. The execution controller then executes the remainder of that plan.

3.3 An Example

To clarify this process we present an example. Consider the problem shown in Figure 1. A sample trace for this problem is shown in Figure 5. After an initial planning phase, BUMP halts with one Sense-Boltsize process in the plan and with the corresponding Have-Wrench goal deferred. EC begins the execution of the partial plan. The first two operations are required as preparatory steps for the third operation (Sense-Boltsize B). Since this solution is done in simulation, EC asks the user for a sensor reading. In this case, 4 is entered. BUMP is now restarted with this new information and this time produces a complete plan. The remainder of

[2] How sensor data is converted into such predications is a nontrivial problem that is beyond the scope of this paper. We do however assume that the conversion would be based upon some hierarchical representation of sensor data which allows that data to be represented at multiple levels of abstraction [Henderson and Shilcrat, 1944]. The planner would work at one of the highest levels.

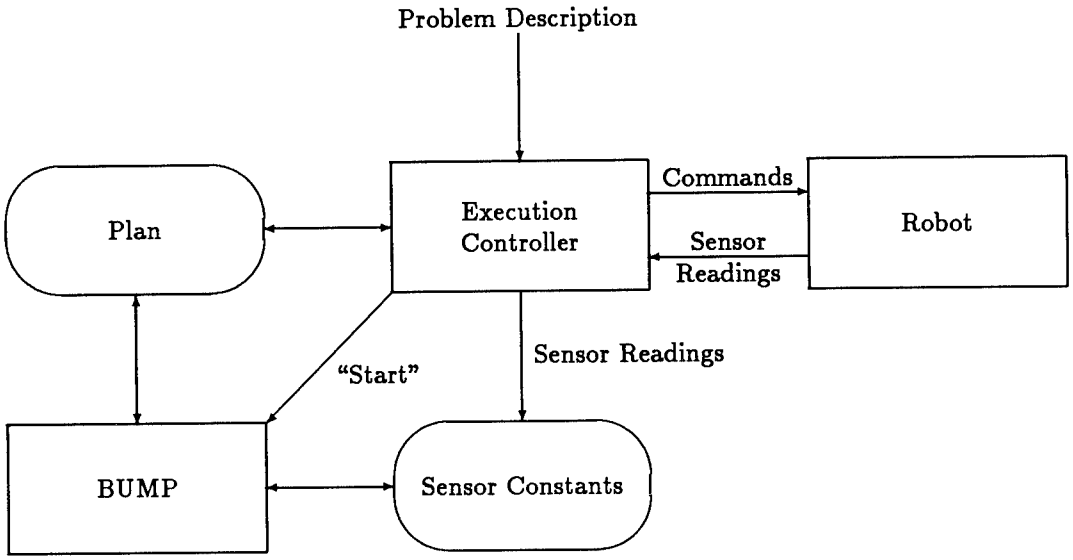


Figure 4: System Architecture.

```
<Initial Planning...>

Executing #<PROCESS19> (Open-Tbox T)
Executing #<PROCESS15> (Get-Bolt Bt)
Executing #<SENSOR-PROCESS11> (Sense-Boltsize Bt)
Enter the size of bolt Bt:  4

<More Planning...>

Executing #<PROCESS33> (Get-Wrench 4)
Executing #<PROCESS28> (Close-Tbox T)
Executing #<PROCESS23> (Insert-Bolt Bt T)
Executing #<PROCESS5> (Bolt T Bt)
```

Figure 5: Sample run for boltsize = 4.

the operations are now executed and the task completed.

4 Deferral Strategies

The primary question in deferred planning is deciding what goals to defer. At the very least we want to defer the goals that are defined in terms of a sensor reading since we do not know the complete goal statement until the reading has been obtained. For example, we cannot formulate a plan or solve the goal (Have-Wrench ?s) until we know the value of ?s, the size of the wrench we must retrieve. We may not know this until we have used sensors to determine the size of some bolt.[3]

Is it advantageous to defer additional goals? That is, should we do as much preplanning as possible, or should we be more conservative? To study this question, we have defined two distinct deferral strategies:

Continue Elsewhere - In this strategy we defer only those goals that are defined in terms of data that must be obtained through a sensor reading. This strategy preplans as much as possible.

Stop and Execute - As soon as BUMP reaches a goal defined in terms of a sensor reading, it stops, deferring all remaining goals until the sensor reading has been obtained. This approach is "maximally conservative".

The Stop and Execute strategy does less preplanning than the Continue Elsewhere strategy. This has the disadvantage that crucial plan dependencies can be missed, and as a result, actions can be taken prematurely. On the other hand the planner will do significantly less planning with incomplete information. This tends to decrease the number of premature actions. Furthermore, Stop and Execute respects the order in which the planner wants to attack goals (which is, of course, independent of the order in which they are achieved during execution), but Continue Elsewhere does not. This is important for BUMP since it orders goals heuristically, and it is likely to be important for other planners as well.

We are currently conducting a study on the performance of these two strategies. We have conducted actual system tests for a set of 32 problems defined for a 2-box version of the tool box world. In this version there are two boxes s and t, and they are to be bolted shut with bolts b_s and b_t, respectively. Initially, the robot is at tool box s. Without loss of generality, we assume b_s has size 4 and b_t has size 5 (we can rename sizes to make this true), but the planner does not know this and must add sensor processes to the plan. The problem space is defined as in Figure 6. Note that since there are two boxes that must be closed, the planner must be careful not to bolt a box containing a wrench that will be needed later.

[3]It should be noted that goals such as (Boltsize B ?s) are not treated in this way. When this goal is first encountered BUMP does not immediately know that a sensor reading is needed. Recall that when the bolt size is already known, a sensor process is not added to the plan. If the bolt size is unknown, a sensor process is added, ?s is bound to a sensor constant, and any further references to ?s will be recognized as a reference to a sensor reading.

Initial State: (Bolt-in-tbox b_x x)
$\wedge \neg$ (Bolt-in-tbox b_y x)
$\wedge$ (Wrench-in-tbox 4 x)
$\wedge$ (Wrench-in-tbox 5 x)
$\wedge$ (At s)

Goal State: (Bolted x b_x)(Bolted y b_y)

Figure 7: 2-Box Study "Stop and Execute" Failure Cases.

Initial State: (Bolt-in-tbox b_s s)
$\wedge \neg$ (Bolt-in-tbox b_t s)
$\wedge$ (Wrench-in-tbox 5 s)
$\wedge$ (At s)
Goal State: (Bolted x b_x)(Bolted y b_y)

Figure 8: 2-Box Study "Continue Elsewhere" Failure Cases.

We consider such a case to be a failure since significant actions are taken prematurely.[4]

As a control we tested BUMP with complete information on the 32 problems. It produced a correct plan with no failures for every problem. The Stop and Execute strategy fails on two of the 32 cases, and Continue Elsewhere fails on four of them. The failure cases are described in Figures 7 and 8. The variables x and y range over the set $\{s, t\}$. The Continue Elsewhere failures occur because BUMP follows the rather natural heuristic of doing everything it can at its initial location before going somewhere else. In cases where b_t is not in s, the initial plan will instruct the robot to bolt s before sensing b_t. When the size of b_t is finally determined, its wrench may have already been bolted inside s. BUMP has ordered these actions prematurely and incorrectly since insufficient information was available at the time.

We have also experimented with a modified Continue Elsewhere strategy called Sense Before Closing. In this strategy, the planner attempts to order all sensor processes before all Close-Tbox processes. (This ordering is not always possible because of other ordering constraints that may already be in the plan.) This strategy performed as well as Stop and Execute (see Figure 9 for the failure cases), however, it is not as general-purpose as the first two strategies. It is applicable only in domains where we want *all* sensor operations to precede all box closings. This would not be the case if the robot were requested to bolt a box, move to another room, and then do more sensing there.

[4]In this case the mistake is easily reversed. If the robot were welding the boxes shut, the recovery would be more difficult.

goal-ordering $\in$ {[(bolted s b_s)(bolted t b_t)], [(bolted t b_t)(bolted s b_s)]}
b_s location $\in$ $\{s,t\}$
b_t location $\in$ $\{s,t\}$
wrench 4 location $\in$ $\{s,t\}$
wrench 5 location $\in$ $\{s,t\}$

Figure 6: 2-Box Study Problem Space.

Initial State: (Bolt-in-tbox b_s s)
$\wedge \neg$ (Bolt-in-tbox b_t s)
$\wedge$ (Wrench-in-tbox 5 s)
$\wedge$ (At s)
Goal State: (Bolted t b_t)(Bolted s b_s)

Figure 9: 2-Box Study "Sense Before Closing" Failure Cases.

5 Discussion

Interleaving of planning and execution has been used extensively. For instance, in the work of [Durfee and Lesser, 1989] the planner uses a blackboard based problem solver to abstract sensory data. This enables the planner to approximate the cost of developing potential partial solutions to achieve long-term goals. Detailed plans are created only for the immediate future using the sketch of the entire plan. By keeping the long-term goals the planner bases its short-term details on a long-term view.

Dean and Boddy [1988] propose a class of algorithms that they call "anytime" algorithms. These algorithms can be interrupted at any point, returning a partial plan. The quality of this plan depends upon the time used to compute it.

We have decided to investigate a more limited class of problems. We are interested in proposing and evaluating strategies to be used when some information is missing at planning time and needs to be obtained with sensors. In our approach planner decisions that depend on sensory information are deferred. As soon as sensory data become available the planning activity is resumed.

Doyle [Doyle, Atkinson and Doshi, 1986] uses sensors to verify the execution of a plan. The sensor requests are generated after the plan has been produced by examining the preconditions and postconditions of each action in the plan. Domain dependent verification operators map assertions to perception requests and expectations. Since perception requests are actions that could have preconditions, the planner is used to modify the original plan to guarantee that the preconditions are established. If the expectations are not satisfied by the perception the plan is repaired using predefined fixes. The entire process is done before executing the plan.

Our work has been inspired, in part, by the recent work of [Turney and Segre, 1989]. The system they present, SEPIA, alternates between improvising and planning. It addresses sensing errors, control errors, and modeling errors. Their example is a traveling salesperson problem with time constraints at every place to be visited. The set of rules suitable for firing contains rule instances whose preconditions and constraints have been met, but whose sensor requests have yet to be evaluated. Since sensing is assumed to be expensive, the system fires the rule instance with the fewest sensor requests first. The cost of a rule is proportional to the number of sensor requests it contains. The planner is interrupted when the cumulative cost exceeds its budget. The quality of the heuristic improvisation strategy has the most significant effect on the quality of the solution (both with the simple improvisation strategy and with SEPIA). This seems to suggest that it is more important to develop good heuristics than to develop a highly sophisticated planner.

Dean [1987] recognized the complexity of solving realistic planning problems and suggested heuristic approaches to decompose a task into independent subtasks that are easier to solve. He suggested using a library of strategies applicable to a set of tasks instead of a library of plans.

The need to plan with incomplete information raises interesting theoretical issues in finding an appropriate balance between the time spent to plan and the time spent to get additional information. Hsu [1990] proposes a method for planning with incomplete information. She shows that if the information available to the planner is not sufficient to produce a plan, then no amount of planning will help find the optimal solution. The idea is to generate a "most general partial plan" without committing to any choice of actions not logically imposed by the information available at that point. An anytime algorithm is then used to chose the appropriate action on the current partial plan when the system has to act. She defines a PERCEPT to be a (possibly partial) description of the world. Percepts are saved to form HISTORIES. A history prescribes or prohibits some actions, allowing the refinement of a partial plan. Finally, a plan is a mapping from histories to actions. Instead of using the most general partial plan she introduces the notion of effective partial plan. Conceptually an effective partial plan is a huge table where each entry contains a perceptual history and a set of actions. This resembles universal plans and is probably impractical unless powerful domain heuristics can be used to prune the search space.

6 Further Work and Conclusions

We are currently extending our strategy study to a 3-box world where the robot must bolt three boxes with three different bolts using three different wrenches.[5] Preliminary results suggest that as the problem becomes more complicated, Continue Elsewhere will begin to outperform Stop and Execute. This is due to the fact that BUMP with the Stop and Execute strategy is unable to plan more than one sensor operation ahead. This is too shortsighted for complex problems. More interestingly, preliminary results also suggest that neither of the general-purpose strategies are very good at avoiding failures, and that more specialized, domain-dependent strategies such as Sense Before Closing may be necessary.

To conclude, we have adapted a conventional planner to do deferred planning. This planner can then be used for problems where there is insufficient information at planning time to develop a complete plan. We have developed several strategies for deciding which plan goals to defer, and we are studying the performance of these strategies. In the 2-box study, the Stop and Execute strategy seems to perform (slightly) better than the other two strategies. The 3-box study is still in progress.

References

[Agre and Chapman, 1987] Philip E. Agre and David Chapman. Pengi: an implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272, Seattle, Washington, July 1987. American Association for Artificial Intelligence.

[Brooks, 1982] Rodney A. Brooks. Symbolic error analysis and robot planning. *International Journal of Robotics Research*, 1(4):29–68, 1982.

[Brooks, 1986] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[Dean, 1987] Thomas Dean. Intractability and time-dependent planning. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, eds. M. Georgeff and A. Lansky. Morgan Kaufmann, San Mateo, California, 1987.

[Dean and Boddy, 1988] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, Minneapolis, Minnesota, August 1988. American Association for Artificial Intelligence.

[Dean and Kanazawa, 1988] Thomas Dean and Keiji Kanazawa. Probabilistic temporal reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 524–528, Minneapolis, Minnesota, August 1988. American Association for Artificial Intelligence.

[Doyle, Atkinson and Doshi, 1986] R. J. Doyle, D. J. Atkinson, and R. S. Doshi. Generating perception requests and expectations to verify the execution of plans. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 81–87, Philadelphia, Pennsylvania, August 1986. American Association for Artificial Intelligence.

[Drummond, 1989] Mark Drummond. Situated control rules. *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, May 1989. Morgan Kaufmann.

[Drummond and Bresina, 1990] Mark Drummond and John Bresina. Anytime synthetic projection: maximizing the probability of goal satisfaction. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, Massachusetts, August 1990. American Association for Artificial Intelligence.

[Durfee and Lesser, 1989] Edmund H. Durfee and Victor R. Lesser. Incremental planning to control a blackboard-based problem solver. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 58–64, Philadelphia, Pennsylvania, August 1986. American Association for Artificial Intelligence.

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189-208, 1971.

[Georgeff and Lansky, 1987] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677–682, Seattle, Washington, July 1987. American Association for Artificial Intelligence.

[Hayes-Roth, 1987] Barbara Hayes-Roth. Dynamic control planning in adaptive intelligent systems. In *Proceedings of the DARPA Knowledge-Based Planning Workshop*, pages 4-1–4-7, Arlington, Virginia, 1987.

[Henderson and Shilcrat, 1944] Tom Henderson and Esther Shilcrat. Logical sensor systems. *Journal of Robotics*, 1(2): 169–193, 1984.

[Hsu, 1990] Jane Yung-jen Hsu. Partial planning with incomplete information. AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments, March 1990.

[Kaelbling, 1988] Leslie P. Kaelbling. Goals as parallel program specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 60–65, Minneapolis, Minnesota, August 1988. American Association for Artificial Intelligence.

[McDermott, 1978] Drew V. McDermott. Planning and acting. *Cognitive Science*, 2:71-109, 1978.

[Nilsson, 1989] Nils Nilsson. Action networks. In *Proceedings of the Rochester Planning Workshop*, pages 21-52, Rochester, New York, October 1988. University of Rochester.

[5] There are, of course, problem instances where two or more bolts have the same size, but these are easier for the planner and therefore less interesting.

[Nilsson, 1989] Nils J. Nilsson. Teleo-reactive agents. Draft Paper, Stanford Computer Science Department, September 1989.

[Sacerdoti, 1974] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Schoppers, 1987] Marcel J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039–1046, Milano, Italy, August 1987. International Joint Committee on Artificial Intelligence.

[Turney and Segre, 1989] Jennifer Turney and Alberto Segre. A framework for learning in planning domains with uncertainty. Technical Report TR 89-1009, Cornell University, May 1989.

[Wilkins, 1989] David E. Wilkins. Can AI planners solve practical problems? Technical Note 468, SRI International, Menlo Park, July 1989.

[Wilkins, 1988] David E. Wilkins. *Practical Planning – Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, San Mateo, California, 1988.

EXPLOITING PLANS AS RESOURCES FOR ACTION

David Payton

Artificial Intelligence Center
Hughes Research Laboratories
3011 Malibu Canyon Road
Malibu, CA 90265

Abstract

When plans are used as programs for controlling the action of autonomous robots, their abstract representation can easily obscure a great deal of the critical knowledge that originally led to the planned course of action. In this paper, we highlight an autonomous vehicle experiment which illustrates how the information barriers created by abstraction can result in undesirable action. We then show how the same task can be performed correctly using plans as a resource for action. As a result of this simple change in outlook, we become able to solve problems requiring opportunistic reaction to unexpected changes in the environment.

1 Introduction

In the endeavor to develop intelligent autonomous robotic agents capable of interacting with a dynamic environment, there has been a growing awareness that traditional planning methods may not be compatible with the demands for real-time performance. Recent efforts to re-evaluate the relationship between plans and action have led to alternative viewpoints in which plans are not primarily responsible for controlling a robot's behavior. Work by Brooks, for example, is aimed at avoiding the use of plans altogether [Br]. In this approach, intelligent action is a manifestation of many simple processes operating concurrently and coordinated through the context of a complex environment. While there is no tangible representation for plans in such a system, plans are implicitly designed into the system through the pre-established interactions between behaviors. Similarly, Agre and Chapman have shown how a system that determines its actions through the constant evaluation of its current situation can perform complex tasks that might otherwise have been thought to require planning [AC1]. Despite their emphasis on the theme that action is obtained by always knowing what to do at any instant, Brooks, Agre, and Chapman do not discard the notion that look-ahead and anticipation of future events are desirable activities. While these activities are normally associated with planning, there is a difference in how the resultant "plans" are represented and used in their systems.

Agre and Chapman, for example, draw a sharp distinction between the concept of plans as communication and the more traditional views of plans as programs [AC2]. The key difference lies in the idea that plans must be constructed as a *resource* to the autonomous agent, not as an explicit set of instructions to be followed [Su]. As a resource, plans must serve as sources of information and advice to agents that are already fairly competent at dealing with the immediate concerns of their environment. In this sense, plans are used optionally, and serve only to enhance system performance. This is a significant departure from the conventional view of plans which puts them in the role of specifying a distinct course of action to systems which are often incapable of doing anything without them.

The differences between these two perspectives on planning are clearly evidenced when information from a map must be used to help guide an autonomous vehicle that must also make extensive use of sensors for detailed maneuvering and obstacle avoidance. In a plan-driven system, map-based plans are typically constructed to describe the optimal path that must be followed in order to arrive at a specified goal location. However, since the vehicle will invariably stray from the ideal path as it avoids sensed obstacles, the plan must be expressed in an abstract form that allows for error. In contrast, when map-based plans are represented for use as resources for action, this abstraction is not necessary. Instead, it is possible to make direct use of all information within the state-space of the map. As a result, information of all possible alternatives may be retained, allowing for flexible opportunistic behavior.

Our own experience with the DARPA Autonomous Land Vehicle (ALV) has led to some valuable insights into some of these issues. In a series of experiments performed by members of the Hughes Artificial Intelligence Center in August and December of 1987, a number of successful tests of autonomous cross-county navigation were performed using a system with integrated map and sensor-based control [Da] [KPR]. Some of the difficulties encountered in these experiments have pointed out certain consequences of the inappropriate use of abstraction that can occur in plan-driven systems. In this paper, we highlight one of these experiments to illustrate how the information barriers created by abstraction can lead to undesirable action. We then show how the same task can be accomplished without abstraction using plans as a resource for action, and we discuss how this approach may be extended for more complex problems.

2 The Misuse of Abstraction

In one of the cross-country experiments performed with the ALV we witnessed a surprising example of how easily plans can be misinterpreted in a plan-driven system. In this experiment, a very simple abstraction of a map-based plan was used to provide guidance to sensor-based obstacle avoidance behaviors. As shown in Figure 1, the basic mission objective was for the vehicle to get from one location to another while maintaining radio contact at all times. The map-based planner generated an appropriate route plan and abstracted a sequence of intermediate sub-goals to represent the critical points along this path. A portion of this sequence is illustrated in Figure 1 as Goals 1, 2, and 3. Note that the route had to veer specifically around one side of a rock outcrop in order to avoid loss of radio contact. To accomplish the mission, the sensor-based behaviors had primary control of the vehicle so that all obstacles could properly be avoided. The behavior decisions, however, were always biased in favor of selecting a direction toward the current map sub-goal whenever possible. As soon as the vehicle got within a specified radius of its current sub-goal, that goal would be discarded and the next sub-goal would be selected. On paper and in simulation, it seemed that this approach would be effective.

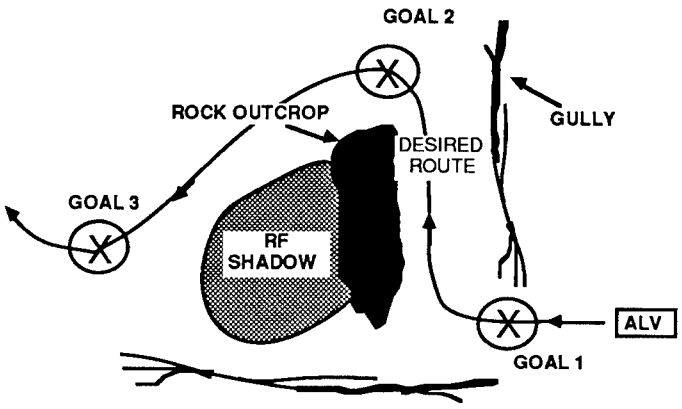


Figure 1. An ALV route plan expressed as a sequence of intermediate goal points.

When we attempted to perform this mission with the ALV, the deficiencies of our method became strikingly clear. During the execution of this route, the vehicle achieved Goal 1 but then, because of local obstacles, was unable to turn appropriately to reach Goal 2. Figure 2 depicts the difference between the desired and actual routes. While this error is clearly apparent from the map data, the sensor-based behaviors had only the abstract route description as their guide, and this gave no indication that there was any problem with their action. Fortunately, contrary to our expectations, radio contact was not lost behind the obstacle. The mission could still be completed successfully if the vehicle were to move onward to Goal 3. Despite this new opportunity, however, the vehicle continued to persist toward Goal 2 because the abstract route description failed to give any indication that the original goal sequence was no longer suitable.

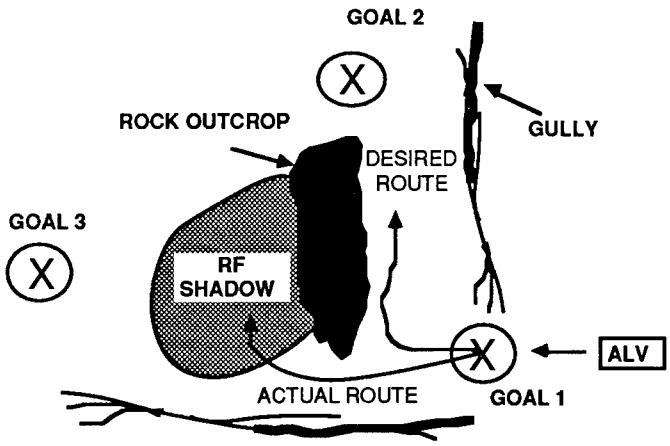


Figure 2. Errant vehicle action while executing its route plan.

This example highlights the system's inability to take opportunistic advantage of unexpected situations when such situations are not properly accounted for in the abstract plan. We know from our understanding of the mission constraints that Goal 2 was merely an intermediate waypoint intended to keep the vehicle away from the rf shadow. Looking at the abstract plan in isolation, however, there is no way of knowing why a particular sub-goal has been established. The Goal 2 location could just as easily have been a critical choke point along the only path to Goal 3. It is only through our understanding of the underlying mission constraints that we can both identify the vehicle's failure to turn right and see the opportunity that arose as a result.

The deficiencies of the abstract route plan may at first appear to be due solely to the simplicity of the representation. Certainly a more sophisticated approach could be employed in which further path constraints are added to help prevent the vehicle from straying from the desired route. Should any significant deviation from the plan be detected, the route might then be re-evaluated. This strategy, however, focuses on preventing the violation of constraints which may in fact have very little bearing on the successful completion of overall mission objectives. Consider, for example, a case in which the vehicle can get near Goal 2, but cannot get close enough to satisfy the criterion of the abstract plan. The system may expend a great deal of time and energy attempting to reach this arbitrary sub-goal when it might otherwise have no difficulty proceeding onward. The problem stems from the fact that the sequence of subgoals is both an overspecification and an underspecification of mission objectives. If the true constraints on vehicle motion relative to a given mission are properly represented, then subgoal locations become immaterial. Therefore, the real deficiency of the abstract route plan lies in the fact that in specifying a pre-determined course of action, it fails to supply the information needed for intelligent decision-making.

3 Avoiding Unnecessary Abstraction

In order to minimize the amount of information lost in forming a plan for action, it is best if all relevant knowledge is organized with respect to a given problem and then,

without any further abstraction, provided in full for use in real-time decision-making. In order for this to be possible, the plan must no longer be viewed as a program for action, but rather, as a resource to help guide the decision-making process. When this viewpoint is adopted, there is no longer a need to translate plans into awkward representations for action. Instead, the original state-space in which the plan is formulated can be retained, enabling the plan to provide advice to sensor-based behaviors whenever the current state of the system can be identified within that state-space. We refer to plans formulated and used in this manner as *internalized plans*, since they embody the complete search and look-ahead performed in planning, without providing an abstracted account of an explicit course of action [Pa].

The difference between the use of internalized plans and conventional abstracted plans is best illustrated in the context of the previous example. In contrast to the abstract route plan, consider a gradient description of a plan to achieve the same objectives. As illustrated in Figure 3, there is no explicit plan shown, yet one can always find the best way to reach the goal simply by following the arrows. Such a representation would not ordinarily be thought to be a plan because it provides no specific course of action. As a resource for guiding action, however, the gradient field representation is extremely useful. No matter where the vehicle is located, and no matter how it strays from what might have been the ideal path, turn decisions can always be biased in favor of following the arrows.

Upon closer examination of Figure 3, we can see not only how the mistake of entering the rf shadow could be avoided, but we see also how the system could be opportunistic should the vehicle happen to enter the shadow and be able to continue onward. First, when the vehicle had to make a choice between going left or right near the bottom of the rock outcrop, the gradient field would strongly bias its decision in favor of going right. If the vehicle got too close to the shadow on the left, the gradient field would actually be telling it to turn around. Further, should the vehicle happen to be forced to go below the rock outcrop and enter the rf shadow, then it would continue to be directed toward the final goal despite the radical deviation from its expected path. This type of behavior is opportunistic in that the vehicle is not constrained to reach any arbitrary pre-established sub-goals, and therefore all action can be directed exclusively toward achieving the mission objectives.

A more dramatic illustration of the difference between a conventional route plan and an internalized plan can be seen in problems requiring the attainment of any of several possible goals. This type of problem is often referred to as the "Post Office Problem" [Ed] because it can be likened to the task of finding the shortest route to the nearest of several post offices in a neighborhood. In the example shown in Figure 4, the mission requires that the vehicle reach either of two distinct goal locations. The resultant gradient field is computed by propagating a search wavefront simultaneously from each of the two goals. As the wavefronts meet at a Voronoi edge, a ridge is created in the gradient field which will cause the vehicle to be guided toward one goal or the other depending on which side of the ridge it happens to be located.

Clearly, it would be difficult for an abstract route plan to capture the essence of choice contained in the gradient field representation. If we were to produce a route plan, we would invariably have to select a route to the closest goal, as shown in Figure 4. Once such a choice is made, however, we have discarded all that is known about the alternate goal even though that goal was nearly as close as the one selected. In contrast, by using the gradient field directly, the choice of goals may be made during the execution of the mission. Without having made an *a priori* selection of goals, the best choice may be made at every instant in time, regardless of how the vehicle might stray while avoiding obstacles.

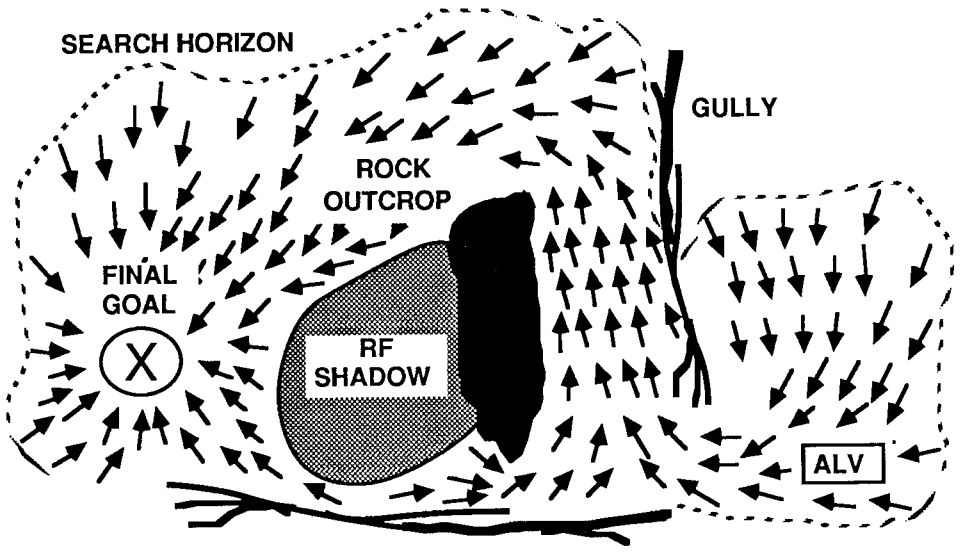


Figure 3. A gradient field representation provides one form of internalized plan.

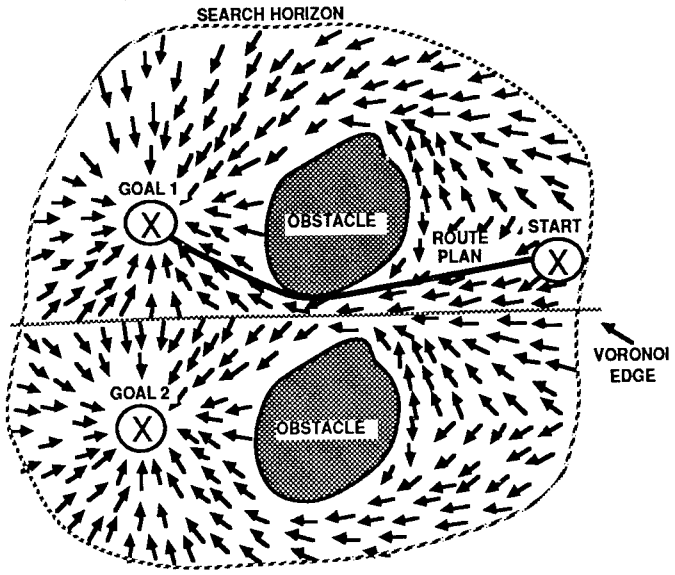


Figure 4. The gradient field provides a useful internalized plan for reachingeither of two goals.

The gradient field is an ideal example of an internalized plan because the map-grid state-space in which the original problem is formulated is the same state-space in which the plan is represented. The gradient field, in fact, is a natural by-product of existing route planning algorithms [MPK]. These algorithms begin by assigning a cost to each grid cell of a digital terrain map. By associating high costs with locations that are undesirable according to mission criteria, a combination of mission constraints can be represented. Whether an A* [Ni], or Dijkstra [Di] search algorithm is employed in the cost grid, the net result of the search is a score for each grid cell, indicating the minimum cost remaining to get from that cell to the goal. From any given grid cell, the best incremental step to get to the goal is the neighboring grid cell which has the lowest score. Ordinarily, when we use these scores to compute a standard route plan, we simply begin at the starting point and locally choose the lowest-score adjacent cell until we finally reach the goal. The record of our steps along the way gives us the minimum cost path to the goal. If we look at these scores in a slightly different way, we see that the best path to the goal from any grid cell may be determined by selecting the direction of the lowest-score adjacent cell. Thus, without any further abstraction, search in the map-grid can provide a useful resource for action.

4 Using Plans as Resources

The method of use of a gradient field is an important factor in establishing it as an internalized plan representation. Since a digital terrain map generally cannot provide adequate resolution to support detailed maneuvering around small obstacles, there is inevitably a need to incorporate the advice provided by the gradient field into real-time decision-making processes which are attending to immediate sensory data. While, ordinarily, a single abstract route plan is generated, some approaches have taken advantage of a gradient field in order to quickly generate new route plans should the constraints of an initial plan be violated [LMD] [CF]. Problems with establishing and monitoring these constraints, however, are still unavoidable. In contrast, use of the gradient field as an internalized plan requires that the real-time decision-making processes continuously attempt to locate the system within the state-space of the plan and bias each decision in favor of the recommended course of action. The absence of an explicit course of action means that no arbitrary plan constraints need be established or monitored. The plan is a resource, providing suggestions for preferred action but never actually controlling the system. If, for any reason, no suggestion is available from the plan, the real-time decision-making processes must proceed in a reasonable manner on their own accord.

Another vector field type of representation, the *artificial potential field*, appears superficially very similar to the gradient field and it also is used for robot navigation and obstacle avoidance [Kr][Kh][Ar]. The basic differences, though, between how these two types of representations are constructed and used sheds further light on what it means for a plan to serve as a resource for action. The computation of potential fields is generally based on a superposition model in which charges are distributed such that repulsive forces are generated near obstacles and attractive forces are generated near goals. Superposition allows the potential field vector at any point to be computed quickly by adding up the contributions from each charge. The resultant field, however, does not represent an optimal path, and may easily contain local minima and traps. In contrast, the gradient field is computed from a more time consuming graph search process. As a result of this search, the gradient field has no local minima and will always yield the set of all optimal paths to the goal.

A more significant distinction between gradient fields and potential fields, however, is in how they are used. Often, when potential field methods are employed for navigation, the potential field is used for direct control of action. All sensory information is compiled into a single representation which is suitable for modeling an appropriate distribution of charges. The local potential field forces are then continuously computed at the location of the vehicle, and these forces are used directly to compute the desired motion. On the other hand, as internalized plans, gradient fields are never used to provide direct control of the vehicle. Instead, they are merely an additional source of information provided to a set of real-time decision-making processes. Since these processes can make use of many disjointed representations of the world in order to control the vehicle, there is never a need for all features of the environment to be abstracted into a single representational framework.

It is helpful to view internalized plans as though they were sources of supplementary sensory input data. From this perspective, it is clear that action is not controlled by plans any more than it is by sensory input. Instead, the system must be viewed as an entity which interacts with its environment, responding to both internal and external information sources. The gradient field plan, for example, can be thought of as a phantom compass that always gives a general idea of the right way to go. Just like other sensors, data from this internal sensor influences action but is never

used to the exclusion of other sensory data. At any given time, however, a single information source can have significant influence over system behavior if need be. Just as an external sensor can be used to ensure that the vehicle never runs into obstacles, an internalized plan can be used to ensure that mission constraints are not violated. Thus, despite the fact that there is no top-down control, the system can adhere to high level mission requirements.

5 Multiple Internalized Plans

A significant advantage of using internalized plans as resources for action is that it is possible to use multiple internalized plans simultaneously. Each plan can contribute an additional piece of advice which can enhance the overall performance of the system. In this way, different plans may be formulated in incompatible state-spaces without the need to merge these state-spaces through abstraction.

We can consider as an example, the combined use of map-based plans with plans based on symbolic mission constraint data. In the case of the rf shadow problem, a constraint to maintain radio contact may be derived from mission knowledge. If this knowledge is used in conjunction with a signal strength sensor, then whenever the vehicle enters an rf shadow, it can immediately back up in order to regain contact. In the absence of such problems, the gradient field produced from map data can constantly provide advice on which way to go. An unexpected loss of radio contact would then be treated much like an encounter with an obstacle. The vehicle would have to make special maneuvers in order to regain contact and ensure that the same mistake would not be repeated. After this, the map-based plan would regain primary influence.

There are also many cases in which it might be desirable to use multiple internalized plans formulated within the same state-space. For example, a gradient field plan could be augmented with information about the amount of fuel and time required to get from each grid-square to the goal. While this information could not directly indicate a course of action, it might allow available fuel and time resources to be monitored constantly and compared with expected needs. If there were barely enough fuel to succeed but plenty of time available, the vehicle might be able to switch to a simple fuel conserving strategy such as reducing its speed. If time and fuel were both in short supply, the gradient field might need to be re-computed, placing more emphasis on conserving fuel and time resources and possibly less emphasis on other factors such as vehicle safety.

Another form of internalized plan exploits the map as a resource for action by probing it directly during execution. As the vehicle is traveling, the portion of the map corresponding to the area just in front of the vehicle is examined to determine what types of features should be detected. This understanding of the local environment can have a direct bearing on how sensor data is interpreted for action. Remember, for example, the problem illustrated earlier in Figure 2. Here, one of the main reasons the vehicle failed to avoid the rf shadow was that its sensors indicated a clear path in this area. This error could be overcome by differentiating between obstacles that are observable and those that are not, and then appropriately discounting sensor readings that are known to be inapplicable. Thus, by treating the map as if it were a sensor, the value of real sensor data can be greatly enhanced.

A great diversity of behavior may also be gained by dynamically combining information from multiple gradient fields. Consider, for example, two independent gradient fields, one which can guide a vehicle along a safe, well hidden route, and another which can lead the vehicle to nearby observation points. We can imagine that the vehicle is guided by the safe gradient field until the time comes for it to make an observation. Then, the gradient field for getting to observation points would become the primary guiding factor. Such a gradient field, formed similar to the field in Figure 4, would lead the vehicle to the nearest of several possible observation points. Once an observation point had been reached and observation data collected, the safe gradient field would again be used for guidance. Using such a combination of internalized plans allows the performance of tasks that would be difficult to accomplish with a symbolic plan. Without an explicit plan for action, it is the interplay between the vehicle and its environment that determines how the mission will ultimately be carried out.

6 Conclusion

Although abstraction is necessary if we are to provide organization and structure to the vast amounts of information available to an intelligent agent, we have seen examples in which the abstraction of plans can obscure their true intent and result in serious failures. In light of these issues we must ask whether forming the abstraction was really necessary or whether it was merely an artifact of an approach in which plans are regarded as programs rather than as resources for action. Using internalized plans, we have shown that with no abstraction of the map-based plan, we can obtain an ideal resource for action.

Just as the grid of a digital terrain map is an abstraction of the Earth's surface, abstraction may be used to create other state-spaces which are suitable to use for planning. In many cases, however, it may be best not to attempt the fusion of information from different sources if an excessive degree of abstraction is required to do so. Instead, state-spaces should be formed to suit the type of information available, and once planning is performed in these state-spaces, no further abstraction of the results should be performed. The unabstracted product of planning search provides a measure of desirability for transitions from one state to the next, and this measure may be used directly as a resource for action.

Although the discussion in this paper has focused primarily on internalized plans based on map data, it is also possible to consider internalized plans based on symbolic data such as found in more general problem-solving domains. There are some significant differences, however, between symbolic data and maps. In maps, state can be defined by position and orientation, and proximity between states is easily estimated by a Euclidian metric. In more complex domains, state may be difficult to define and even more difficult to sense. Proximity of states may be

determined only through knowledge of what state transitions are achieved by various operations. However, when a domain can be divided into a set of recognizable states, and these states can be linked according to their accessibility to one-another, then internalized plans can be produced. Just as with map data, search through an abstract state space can indicate the progression of states required to reach a desired goal. If this knowledge can be used as advice within a system that can move between states on its own accord, then we can generate an internalized plan.

Acknowledgments

Many of the concepts presented in this paper were fostered through frequent discussions with David M. Keirsey, J. Kenneth Rosenblatt, and Charles Dolan. I am extremely grateful for their contributions and insights. The advice of Joseph Mitchell and Rodney Brooks was also very helpful in bringing to light many important areas of related work. Many thanks also to my wife Karen and Jimmy Krozel for their editing assistance.

References

[AC1] Agre, P., and D Chapman, "Pengi: An implementation of a Theory of Activity","*Proc. of the Sixth National Conf. on Artificial Intelligence*, Seattle, Washington, July, 1987, pp. 268-272.

[AC2] Agre,. P., and D. Chapman, "What are plans for?"AI Memo 1050, MIT Artificial Intelligence Laboratory, 1987.

[Ar] Arkin, R., "Motor Schema Based Navigation for a Mobile Robot: An Approach to Programming by Behavior," *IEEE Conf. on Robotics and Automation*, March 1987, pp. 264-271.

[Br] Brooks, R. A., "Intelligence Without Representation," *Preprints of the Workshop in Foundations of Artificial Intelligence*, Endicott House, Dedham, MA, June, 1987.

[CF] Chan, Y.K., and M. Foddy, "Real Time Optimal Flight Path Generation by Storage of Massive Data Bases," IEEE National Aerospace and Electronics Conf. (NAECON), Dayton OH, May 1985.

[Da] Daily, M., J. Harris, D. Keirsey, K. Olin, D. Payton, K.Reiser, J. Rosenblatt, D. Tseng, and V. Wong, "Autonomous Cross-Country Navigation with the ALV," *Proceedings of DARPA Knowledge-Based Planning Workshop*, Austin, Texas, December 1987, (also appearing in *Proceedings of IEEE Conference on Robotics and Automation*, Philadelphia, PA., April, 1988.)

[Di] Dijkstra, E.W., "A Note on Two Problems in Connection with Graph Theory," Numerische Mathematik, Vol 1, 1959, pp. 269-271.

[Ed] Edelsbrunner, H., *Algorithms in Combinatorial Geometry*. Springer-Verlag, Berlin, 1987, pp 298-299.

[KPR] Keirsey, D.M., D.W. Payton, and J.K. Rosenblatt, "Autonomous Navigation in Cross Country Terrain," *Proceedings Image Understanding Workshop*, Boston, MA, April, 1988.

[Kh] Khatib, O. "Real Time Obstacle Avoidance for Manipulators and Mobile Robots," *IEEE Conf. on Robotics and Automation*, March 1985, pp. 500-505.

[Kr] Krogh, B.H., "A Generalized Potential Field Approach to Obstacle Avoidance Control," *Robotics International Robotics Research Conference*, Bethlehem, PA, August 1984.

[LMD] Linden, T.A., J.P. Marsh, and D.L. Dove, "Architecture and Early Experience with Planning for the ALV," *IEEE International Conf. on Robotics and Automation*, April, 1986, pp. 2035-2042.

[MPK] Mitchell, J.S.B., D.W. Payton, and D.M. Keirsey, "Planning and Reasoning for Autonomous Vehicle Control," *International Journal for Intelligent Systems*, Vol. 2, 1987.

[Ni] Nilsson, N.J., *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.

[Pa] Payton, D.W., "Internalized Plans: a representation for action resources," Workshop on Representation and Learning in an Autonomous Agent, Lagos, Portugal, Nov. 1988.

[Su] Suchman, L., "Plans and Situated Actions: the problem of human machine communication," Cambridge University Press, 1987.

Responding to Impasses in Memory-Driven Behavior: A Framework for Planning

Paul S. Rosenbloom & Soowon Lee
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292

Amy Unruh
Knowledge Systems Laboratory
Computer Science Department
Stanford University
701 Welch Rd. (Bldg. C)
Stanford, CA 94305

Abstract

One approach to bringing coherence to the area of planning is to develop a framework that covers the set of requisite planning behaviors, enables comparisons among them by decomposing the behaviors into common primitives, and forms the basis for an integrated planning system. In this article we report on an on-going effort to build such a framework from the combination of a basic memory-driven agent, behavioral impasses, and generic responses to these impasses. This framework is loosely based on the planning strategy embodied within the Soar architecture, and is illustrated with examples from Soar. Though the framework's current level of development still falls far short of an integration of all of planning, progress has been made.

1. Introduction[1]

What is the range of planning behaviors necessary in an intelligent agent, and how do (or should) they arise during performance? Though this is a key question in the design and construction of intelligent agents, we — the AI community — are still rather far from answering it. What we have are partial answers to both parts of this question. For the first part, we have identified a grab bag of planning methods and behaviors — such as linear planning, nonlinear planning, opportunistic planning, wait-and-see planning, hierarchical planning, abstraction planning, goal decomposition, least-commitment planning, constraint posting, case-based reasoning, tweaking, schema-based planning, skeletal planning, reaction planning, reactive planning, backward chaining, operator subgoaling, means-ends planning, simulation, envisionment, projection, lookahead search, state-space search, and temporal planning — and have a few attempts to build planners capable of multiple behaviors, such as NOAH [Sacerdoti, 1977], MOLGEN [Stefik, 1981a; Stefik, 1981b], SIPE [Wilkins, 1984], and TWEAK [Chapman, 1987]). For the second part, we have proposed three general classes of answers. (1) The traditional answer is that these behaviors are preprogrammed into a general planning method that is run prior to performing each task and (possibly) when the application of the plan to the task goes awry. (2) The reactive answer is that the agent is already structured in such a way as to obviate the need for planning at run time, so no planning behaviors occur either just prior to, or during task performance. (3) The hybrid answer is to avoid the initial planning phase by utilizing the agent's existing structure, but to fall back on more general planning methods when this structure proves to be inadequate (and possibly to acquire more structure as a result).

The purpose of this article is to report on an on-going effort at taking a next step towards answering the above question. The approach underlying this effort is to construct, and implement, a generic framework for planning by combining a particular instantiation of the hybrid approach — a recursive memory-driven agent roughly based on the approach to planning embodied in the Soar system [Laird, Newell, & Rosenbloom, 1987; Laird & Rosenbloom, 1990] — with a set of generic responses to behavioral impasses. What this yields is a space of planning behaviors that overlaps with the set of behaviors listed above. The intent in so doing is to provide an organization over this space of behaviors, characterize when the individual behaviors can arise, provide a decomposition of the behaviors that allows comparisons in terms of common primitives, and to provide the foundation for the construction of a powerful and flexible planner. Though this still leaves several important things partially or

[1]This research was sponsored jointly by the Defense Advanced Research Projects Agency (DOD) and the Office of Naval Research under contract number N00014-89-K-0155.

completely undone — the framework does not include all known planning behaviors, nor does it characterize which behaviors are necessary in an intelligent agent or precisely when the individual behaviors are desirable — the hope is that this is a step in a useful direction.

Section 2 describes the hybrid approach, with short digressions to ground the abstract characterization in Soar. Section 3 catalogues the behavioral impasses and a set of generic responses to them. This does not yet provide a complete set of response types, but is a start towards such a set. A sampling of the planning behaviors resulting from these impasses and responses are also provided from research on this framework in Soar. Section 4 relates this overall approach to other work in the field. Section 5 concludes and discusses future directions.

2. A Hybrid Planning Framework

Start with an agent capable of performing *operations* to achieve some end, where "operation" is used in a generic sense, referring to any specification of something to be accomplished. The operation can be external (a motor act) or internal (in a simulated world, for example), and either primitive (closing a gripper) or high-level (picking up a block). The specification can be procedural (directly executable) or declarative (interpretable), and specified in terms of what is to be accomplished (a goal) or how it is to be accomplished (an operator). The agent proceeds by cycling through four *steps*:

1. *Generation:* Generate a set of candidate operations.
2. *Selection:* Select an operation from the set of candidates.
3. *Applicability:* Determine if the selected operation is applicable in the current situation (or make it applicable).
4. *Execution:* Execute the selected operation.

In Soar, this corresponds to the process of selecting and applying operators in problem spaces.

So far, this describes a generic, serial, agent[2] — though one with a particularly local drive, as it is always attempting to take the next step towards some end. What is missing, and what distinguishes the various planning behaviors, is how the four steps come to be performed. In our hybrid agent, precedence is given to a context-dependent, *memory-driven* performance strategy. In a memory-driven strategy, the information required for the performance of each step is directly accessed from the agent's memory — no involved chains of reasoning are involved. In a context-dependent, memory-driven strategy, the current context — provided by perception, goals, etc. — determines what is accessed. For example, in a robot manipulation domain, the problem leads to accessing the available manipulation operations (for example, `pickup`, `putdown`, `translate`, `open`, and `close`). Based on the goal and situation, memory is then accessed about which operation to select (`close`, for example). Memory is then accessed about whether the preconditions of `close` are met in this type of situation. If the preconditions are met, then the final memory access either determines how the situation is changed by the operation (if this is an internal, or simulated, operation), or which motor actions are to be performed (if this is an external operation).

The exact form of the memory is not critical for this behavior — it could be declarative or procedural; contain information that is generalized (plans) or instantiated (cases); and be structured as boolean circuits, rules, or frames. However, what does matter is that access to the memory be computationally limited. Without this restriction, arbitrary amounts and kinds of processing — such as full first-order theorem proving — can be surreptitiously imported under the heading of "memory access". The resulting agent — the *basic* agent — is a generic, computationally-bounded, memory-driven agent. While this is a fairly simple sort of agent, it forms the core out of which more complex planning behaviors can emerge. It also serves as a useful abstraction over a set of common agent types — depending on the exact details of the memory, the basic agent can be a *reactive* agent (if the memory is sufficiently limited computationally), a *rule-based* agent (if the memory contains computationally-limited rules about step performance), a tweak-free *case-based* agent (if the memory contains previous problem instances), or a non-hierarchical *schema-based* agent (if the memory contains generalized plans).

In Soar, the memory is structured as a parallel production system. Knowledge about step peformance is stored predominantly in procedural form, as productions that generate preferences about changes to working memory; however, it can also be stored in more episodic or declarative forms [Rosenbloom, Newell, & Laird, 1990]. Likewise, what predominantly corresponds to a plan is a set of preference-generating productions that jointly determine the agent's behavior, rather than more declarative specifications that yield behavior through interpretation. The latter is possible — see [Reich, 1988], for example — but is not the predominant approach.

More complex — planning — behaviors arise when the basic agent hits *impasses*. Impasses occur whenever the memory is inadequate for the

[2]For simplicity we will stick to serial agents in this article. However, it should be possible to extend the framework to agents capable of performing operations in parallel.

performance of one of the four basic steps — for example, when the memory proves insufficient for generating a set of candidate operations. When such an impasse occurs, the agent is applied recursively to the problem of resolving the impasse. The hybrid agent consists of the basic agent — a generic, computationally-limited, memory-driven agent — plus this ability to recur on impasses. In its most flexible form, this recursion allows full meta-level — or, equivalently, reflective — processing. In Soar, this recursion occurs via the automatic generation of subgoals within which flexible meta-level processing is possible via the selection and use of further problem spaces [Laird, 1983; Rosenbloom, Laird, & Newell, 1988].

The range of responses available during the recursive processing determines the range of planning behaviors exhibited by the hybrid agent. Consider two illustrative examples. In the first example, the agent is performing operations in the external world — it is in "execution mode" — and an impasse occurs because of an inability to select the next operation from the set of candidates. This drops the agent into "planning mode" where it can, for example, perform a lookahead search; that is, execute the candidates in a simulated world to determine which ones are likely to achieve its end. The resulting behavior corresponds to a classical case of execution monitoring — detecting that the agent has reached a situation for which it does not have a preprogrammed response — and dynamic replanning. In the second example, the agent reaches an impasse because the selected operation is not applicable to the current situation. If it responds to this impasse by selecting a second operation in the recursive space that can modify the situation so that the first operation is applicable, the resulting behavior is appropriately characterized as backward chaining or operator subgoaling.

Because impasses are tackled via a recursive process, further impasses can occur within this processing, leading to yet further levels of recursion. One implication of this recursion is that many of the same phenomena will occur for both execution and planning — both involve applying sequences of operations, and both can run into impasses on the same performance steps. This uniformity not only simplifies the structure of the planner — for example, making particularly simple the transfer of information from planning to execution — it also simplifies the subsequent analyses of planning behaviors. A second implication of this recursion is that many planning behaviors may arise from a cascade of impasses and responses, rather than from just single ones. Consider the first illustrative example above, which is described in a simple two-level fashion. This example actually occurs in Soar over three levels of behavior (two levels of impasses). As above, the first level consists of task execution. However, at the second level the operations are ones that evaluate the first-level candidates. If an impasse occurs during execution of one of these second-level operations — because of a lack of memory structures about the value of the first-level operation — a lookahead search is performed, beginning with the simulated execution of the first-level candidate.

One important consequence of this overall hybrid framework is that planning occurs on an as-needed basis. Performance is predominantly memory-driven, but when memory is insufficient, planning is possible. Such a strategy can be quite effective in many situations, as it avoids expensive deliberation until it is needed, and can potentially be improved by simple learning strategies. However, in hazardous and/or time-limited domains it can get into trouble if the memory contains incorrect information, or if the memory turns out to be incomplete when attempting to perform a time-critical step (for which there is insufficient time to perform on-the-fly planning). Under such circumstances a prudent agent would be sensitive to its context, and deliberately do contingency planning prior to performing in the real domain. The results of this contingency planning could be used to alter the agent's memory structures so that when the real problems are faced, issues of incorrectness and incompleteness do not arise. If these memory alterations are persistent, this amounts to a form of learning. If the architecture is sufficiently uniform, the same approach can also be used to learn new memory structures from all planning episodes — whether it be deliberate contingency planning or as-needed planning (or replanning). If enough learning occurs in a domain, it may be possible to eliminate all impasses, and thus all planning, converting the agent into a completely memory-driven agent (for that domain). In Soar, this form of learning occurs via a chunking process that adds new productions to memory from the results of subgoal-based processing.

3. Generic Responses to Impasses

The hybrid framework presented in Section 2 implicitly defines four distinct impasse types, one for each type of step:

1. *Generation:* Failure to generate a set of candidate operations.
2. *Selection:* Failure to select an operation from the set of candidates.
3. *Applicability:* Failure to determine if the selected operation is applicable (or to make it applicable).
4. *Execution:* Failure to execute the selected operation.

The principal thesis underlying this article is that if

these impasse types are crossed with a set of generic response types, then a framework is generated which covers a significant fraction of the important planning behaviors. A complete rendering of such a framework is currently beyond our means. However, what has been accomplished so far is the isolation of four generic response types, an understanding of some of the planning behaviors that fall within their scope, and implementations of some of these behaviors within Soar.

In the remainder of this section we characterize these four generic response types:

1. *Pursuit:* Pursue resolution of the impasse.
2. *Termination:* Terminate the line of development.
3. *Suspension:* Suspend the line of development.
4. *Obviation:* Alter context to make impasse irrelevant.

A sampling of the planning behaviors generated in the context of Soar will be presented as illustrations.

3.1. Pursuit

Pursuit is the classical response to an impasse. It covers strategies that manipulate either the agent's knowledge, or the world itself, to resolve the impasse. For example, simulated lookahead is a typical pursuit strategy for selection impasses — it is a potentially unbounded method for squeezing out more information from what the agent already knows. Pursuit methods are widespread in planning. They include linear and nonlinear planning, the simulation methods (simulation, envisionment, projection, lookahead search, state-space search, operator subgoaling, and backward chaining), the decomposition methods (AND hierarchies, skeletal planning, schema-based planning, macro expansion), the deductive methods (theorem proving)[3], the inductive methods (case tweaking, analogical transfer), the attentional methods (shift of attention, extended memory search), the experimentation methods, and the advice-taking methods. Such a large and diverse list reveals that pursuit is rather a large grab bag itself, and implies that further levels of structure will eventually be needed in a complete framework. However, part of this diversity is only apparent, as it arises from the use of different terms for essentially the same behavior — such as simulation, projection, envisionment, lookahead search, and state-space search — while part of the remaining diversity arises from the development of different classes of responses for different impasse types: the just-mentioned simulation behaviors for selection impasses; the backward chaining (operator subgoaling) method for applicability impasses; the decomposition methods for execution impasses; and the attentional methods for generation impasses. So the picture isn't quite as bleak as it might at first appear.

Previous articles on Soar have already detailed the implementation of a number of these behaviors. Decomposition occurs when a problem space containing more primitive operators is used in response to an impasse. Examples of decomposition for both selection impasses — where the `selection` problem space is used to decompose the problem into one of computing evaluations for each of the candidate operations — and execution impasses — where high-level operations, such as configuring a computer backplane, are decomposed into a conjunction of more primitive operations, such as configuring a module into a slot of the backplane — can be found in [Laird, Newell, & Rosenbloom, 1987]. The use of simulation for execution impasses — for the operations in the selection problem space — can also be found in [Laird, Newell, & Rosenbloom, 1987]. The use of advice taking for selection is described in [Golding, Rosenbloom, & Laird, 1987], and for more general impasses, in [Laird *et al*, 1990].

Rather than reprise this existing material in any detail, we will make do with a brief return to the issue of lookahead search — which nicely illustrates how planning behaviors can arise out of combinations of impasses and responses — and then look at recent implementations of *linear* and *nonlinear* planning. When lookahead was first introduced in Section 2, it was described as a response to a selection impasse. Later, this was refined to two levels of impasses: a selection impasse and an execution impasse. Now it can be further refined to a decomposition response to a selection impasse, plus a simulation response to an execution impasse. In fact, under many circumstances, this pair of impasses only yields one step of lookahead — once the simulated candidate has been executed, a new selection impasse may occur because of a lack of memory about what operation to select next. Thus, each such additional level of lookahead can correspond to two new levels of impasses.

Both linear and nonlinear planning are quite familiar to the community by now. What makes them interesting here is not their novelty, but that they both turn out to be complex combinations of multiple impasses with varying pursuit responses. They thus turn out to be good examples for illustrating the framework. In addition, by decomposing each method into a set of impasses and responses, the framework makes explicit their normally implicit, fine-grained structure, and reveals their similarities and differences.

[3] Many of the other methods are also technically deductive, but it seems useful to separate them out from the pure theorem proving methods.

Both linear and nonlinear planning start with an operation that represents a conjunctive goal, and reach an execution impasse on this operation. They both then pursue this impasse, but in different fashions. In linear planning — as shown for the blocks world in Figure 1 — the agent responds to this execution impasse by decomposing the original operation into a set of operations, one for each conjunct. A selection impasse then occurs, unless there is memory about how to pick among them, or there is only one conjunct (in which case the decomposition also need not occur). In this selection impasse, a simulation begins by selecting one of the candidates.[4] Execution of the operation then consists of achieving the corresponding conjunct. If the conjunct is not already met, it cannot be achieved by simple memory access, so an execution impasse occurs ("goals" as operations have the property that they are always applicable, so no applicability impasse will occur). This execution impasse is responded to by a simulation in which means-ends analysis is used to generate a set of candidate operations from the operators in the domain. If there are multiple candidates, another selection impasse occurs. However, in this example there is only one candidate, so it is just selected. Its applicability is then tested, and if it is known to be applicable it is executed; otherwise, as is the case here, an applicability impasse occurs. The whole process then recurs, with the achievement of the operation's preconditions as the new goal, which in this case is non-conjunctive.

In nonlinear planning[5] — Figure 2 — the original operation is not decomposed. The agent instead responds immediately with a simulation in which the candidate operations are generated via means-ends analysis from the entire set of conjuncts. An operation is then selected, and if it is applicable, it is executed. If an applicability impasse occurs, the whole process then recurs, with a set of goal conjuncts corresponding to the original set, except that the conjunct responsible for generating the selected operation is replaced by the operation's preconditions.

From these descriptions it can be determined that the key differences between these methods are: (1) at any point in time, linear planning generates a candidate set of operations by looking at only one goal conjunct — the *current* one — while nonlinear planning looks at all active goal conjuncts (irrespective of their level of generation); (2) linear planning must achieve a goal conjunct before moving on to a sibling (or higher level) conjunct, while nonlinear planning can intermingle operations generated from any of the active conjuncts; and (3) when linear planning finishes with a goal conjunct, it proceeds to one of the conjunct's siblings, while nonlinear planning has no locus of control (all active conjuncts are considered at all times). The bottom line is that linear planning obeys a strict depth-first progression. It looks at only one goal conjunct at a time, continues looking at that conjunct until it is resolved, and moves on to one of the conjunct's siblings when it is resolved. Nonlinear planning, on the other hand, is free to move around at will.

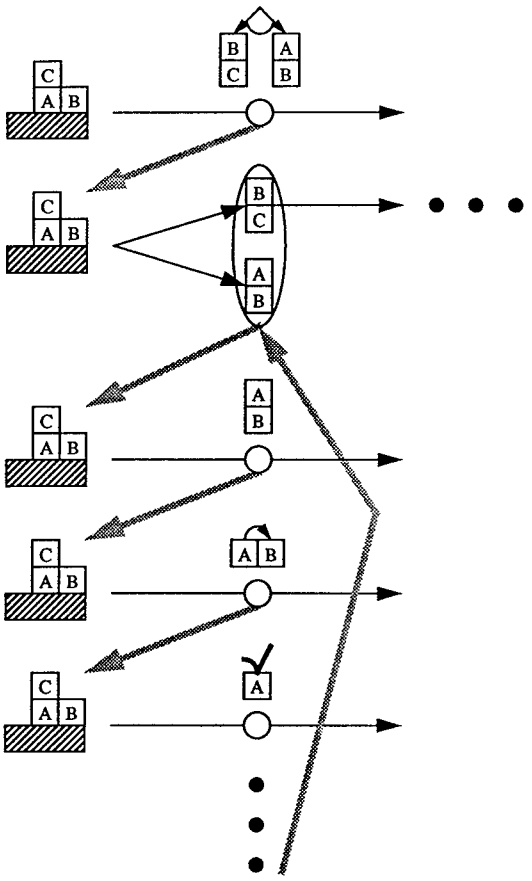


Figure 1: Trace of linear planning in the blocks world.

Given this breakdown of the differences between the two methods, it can be used to generate methods intermediate between the two extremes. One such intermediate method is like linear planning, except that no decomposition occurs before operation

[4]For simplicity of presentation, and since we have already gone over the structure of lookahead, this and later figures show simulation directly arising from a selection impasse.

[5]There is some confusion in the use of the term "nonlinear planning" in the field. Here we take it to mean the construction of plans whose ordering of operations does not respect subgoal boundaries. In other words, operations for different goals can be interleaved. Least commitment is one technique for generating nonlinear plans, but it is not what is meant by "nonlinear planning". Likewise, nonlinear plans can be built from partially ordered plans, but they need not be (see [Veloso, 1989], for example).

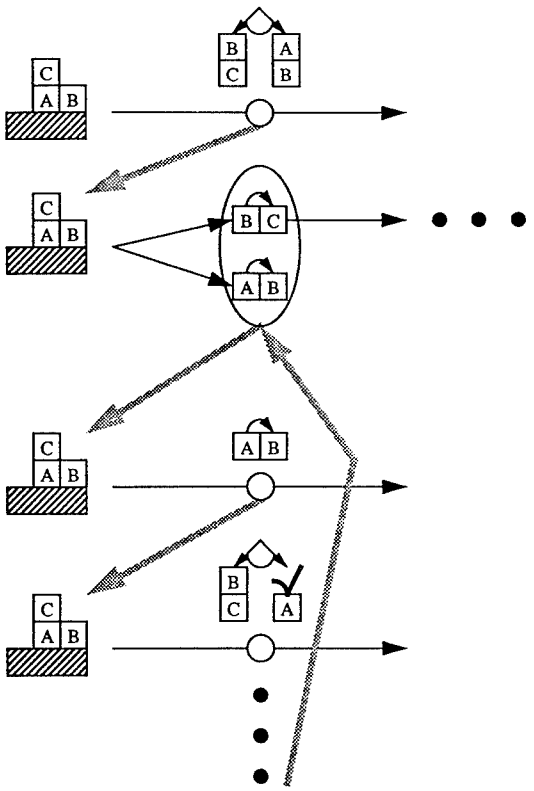


Figure 2: Trace of nonlinear planning in the blocks world.

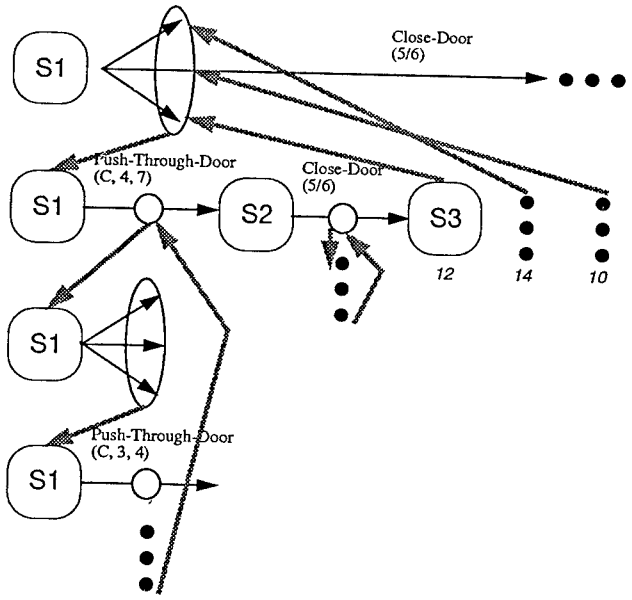


Figure 3: Trace of intermediate planning in the robot domain, starting from the pursuit of the conjunctive goal. The task is to find the shortest path for achieving the problem in Figure 4, so evaluations correspond to minimal path lengths.

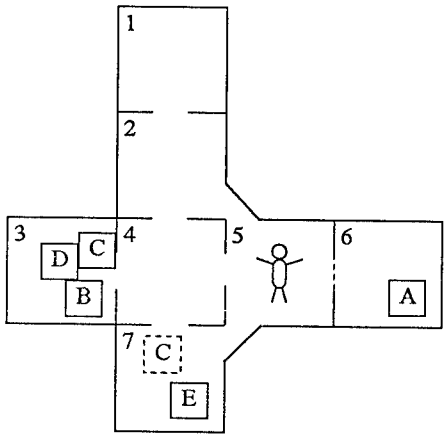


Figure 4: Initial state and goal for the robot-domain problem. The goal conjuncts are dashed (move block C into room 7 and close the door between rooms 5 and 6.)

generation; or equivalently, it is like nonlinear planning, except that the conjunctive goal in the recursive step consists of only the selected operation's preconditions. Figure 3 shows a trace of such a method in a simple robot domain. This trace also contains two simplifications with respect to the traces we have seen so far. The first simplification is that the original conjunctive goal is not represented as an operation. Instead it is represented directly as the (only) end the agent is to achieve. The second simplification is that when an applicability impasse occurs, it is pursued directly by a simulation using operations generated via means-ends analysis from the impassed operation's preconditions. This contrasts with the more involved approach used in the previous traces, where there is an intermediate step of creating a new operation for the new conjunctive goal, and simulation isn't used until an execution impasse occurs on this new operation. Other such simplifications — eliminations of impasse levels — are also possible under the appropriate circumstances.

The trace begins with the pursuit of the conjunctive goal via a simulation. When applicability impasses occur during the simulation, further simulation is performed to pursue them. This method can violate the strict depth-first progression used by linear planning, but only during operation selection for sibling conjuncts.

In addition to using simplifications such as those in this last trace, another way to eliminate impasse levels, and thus to simplify performance, is to augment the agent's memory so that the impasses don't occur. This can occur by deliberate prestructuring of the agent's memory, or it can

happen dynamically via learning. As an example, Figure 5 shows a control production learned for the initial selection impasse in the intermediate trace from Figure 3. This production, in conjunction with others that are also learned, forms a control plan that allows selection to proceed for this task in a totally memory-driven fashion. Learned productions can also potentially transfer to other situations, thus allowing other selections to be performed in a memory-driven fashion.

```
In the robot problem space,
want door <d1> closed
     and box <b> in room <r2>,

the robot is in room <r1>,
door <d1> is to room <r1>,
door <d2> is to rooms <r2> and <r3>,
door <d3> is to rooms <r1> and <r2>,
door <d4> is to rooms <r1> and <r3>,
door <d5> is to rooms <r3> and <r4>,
doors <d1> through <d5> are open,
box <b> is in room <r4>,
box <b> is next to door <d5>,
box <b> is pushable,

operators <o1> and <o2> are candidates,
operator <o1> is Close-Door(<d1>),
operator <o2> is
     Push-Through-Door(<b>, <r3>, <r2>)

  -->

prefer <o1> to <o2>.
```

Figure 5: Control production learned for the initial selection impasse in the intermediate robot trace of Figure 3. Angle brackets denote variables, and different variables bind to different objects.

3.2. Termination

Termination covers strategies that abort lines of development (that is, sequences of operation executions) that result in impasses. It leads to a class of what can be called *completeness* methods, because they assume that memory can be complete enough for successful step performance. In *applicability completeness*, lines of development are terminated if they lead to applicability impasses. When this response is used for execution-time applicability impasses, it aborts any problem in which a selected operation is not applicable. When this response is used during planning-time applicability impasses, it focuses effort on lines containing operations known to be applicable. As an example, consider the blocks-world problem presented in the previous subsection. If this is initially pursued as with linear planning, but applicability impasses are responded to with termination, the behavior shown in Figure 6 arises. This behavior can be described as a search through the permutations of (subsets of) the operations generated for the original goal conjuncts. The planning methods described in the previous section also searched through sets of permutations, but they were different sets. Those methods had the additional ability to introduce new goal conjuncts — from the preconditions of selected operations — and thus to introduce new permutations based on the operations generated from these conjuncts. To counterbalance this increased flexibility, the linear and intermediate methods then use goal boundaries to restrict the set of permutations that are considered.

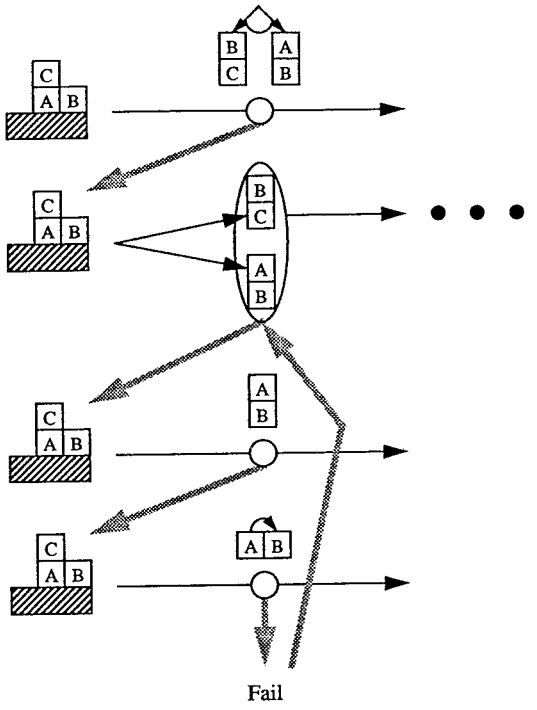


Figure 6: Trace of applicability completeness in the blocks world.

Other (not yet implemented) completeness methods yield variations on this behavior. For example, with *selection completeness*, lines of development that yield selection impasses are terminated; essentially giving up whenever an ambiguous choice is reached. If termination is used for all impasses, pure memory-driven behavior — reactive behavior(?) — is forced.

3.3. Suspension

Suspension covers strategies that delay the development of lines that lead to impasses. An example is execution suspension, where lines that lead to operations with nonobvious execution methods are delayed. It is like termination, except that it is

possible to return to the aborted lines at a later point. It also corresponds to a form of least-commitment behavior, in which there is a preference to investigate "understood" lines first. No suspension methods have been implemented so far in Soar, and it is still to be determined whether Soar provides the primitive functionality to implement such responses in a straightforward fashion.

3.4. Obviation

Obviation consists of changing the performance context so that the impasse no longer matters. It covers strategies that reformulate either the ends (the goal to be achieved) or the means (the operations) in such a way that the impasse becomes irrelevant. The difference between this and pursuit is that, in pursuit the impasse is resolved, while in obviation changes are made so that the impasse can be avoided without being resolved. One classical example of an obviation strategy is *precondition abstraction*, where applicability impasses are obviated by deleting those operation preconditions that lead to the impasses. When this is done during planning, it can yield an abstract plan which may be refined later. When this is done during execution, it corresponds to an attempt to bull through the constraints imposed by the world.

Figure 7 shows what happens when obviation (precondition abstraction) is used in response to the applicability impasses that occur during planning for the robot-domain problem of Figure 4.[6] As before, the original conjunctive goal is responded to via pursuit (simulation), but now applicability impasses are glossed over by assuming that unmet preconditions are actually met — this is a local way of deleting a precondition, without its affecting other uses of the same operation — and the operation is executed to the extent possible. Because the abstraction of a precondition can lead to partial execution of the corresponding operation, and thus to the generation of an abstract state, the abstraction propagates dynamically through the simulation (not shown).

When the abstract simulations have successfully completed, they yield an abstract plan that determines what operations to perform during execution. Figure 8 shows a control production learned for the initial selection impasse in this trace. This production is quite similar to the one learned in the corresponding non-abstract trace (Figure 5), except that many of the situational conditions are missing. This occurs because successful completion of the abstract simulation depends on fewer aspects of

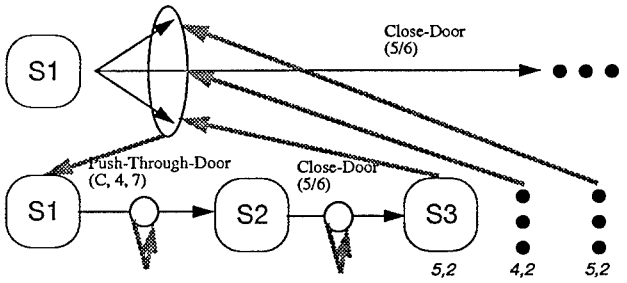


Figure 7: Trace of precondition abstraction in the robot domain, starting from the pursuit of the conjunctive goal. The task is to find the shortest path for achieving the problem in Figure 4. A lexicographic evaluation function is used whose first component is the number of preconditions that have been deleted, and whose second component is the minimal path length.

the situation. The result is a more general production, which can be used in more situations, though not necessarily always correctly.

```
In the robot problem space,
want door <d1> closed
     and box <b> in room <r2>,

the robot is in room <r1>,
door <d1> is to room <r1>,
door <d2> is to rooms <r2> and <r3>,
doors <d1> and <d2> are open,
box <b> is pushable,

operators <o1> and <o2> are candidates,
operator <o1> is Close-Door(<d1>),
operator <o2> is
     Push-Through-Door(<b>, <r3>, <r2>)

  -->

prefer <o1> to <o2>.
```

Figure 8: Control production learned for the initial selection impasse in the abstract robot trace of Figure 7. Angle brackets denote variables, and different variables bind to different objects.

Refinement of the abstract plan occurs when the operations that were abstracted during simulation are reselected during execution. When applicability impasses occur during execution, they are not obviated, since this is quite difficult in the real world; instead, they are responded to via pursuit methods. If further applicability impasses occur during the planning of this pursuit, then further abstraction

[6] See [Unruh, Rosenbloom, & Laird, 1987; Unruh & Rosenbloom, 1989; Unruh & Rosenbloom, 1990] for more details about, and results from, *impasse-driven abstraction* in Soar; including the presentation of more sophisticated methods, such as *iterative abstraction* [Unruh & Rosenbloom, 1990].

occurs (and so on). Figure 9 shows a fragment of the refinement process for the trace in Figure 7. This processing is just a continuation of the earlier trace from the point where `Close-Door(5/6)` is selected for execution. Once this is done, the operation is executed without further impasses (the robot is already in the room containing the door to be closed). A random selection is then made between the other two options, because their abstract evaluations are equal,[7] and an applicability impasse occurs. This impasse is obviated, and processing continues. The resulting behavior corresponds to multi-level refinement, where the original plan is as abstract as possible, and each level of refinement adds one more level of detail. It arises here not because of the deliberate construction of a multi-level agent, but because: (1) obviation is used for applicability impasses that occur during simulation, while pursuit is used for applicability impasses that occur during execution; and (2) the same processes occur during planning and replanning.

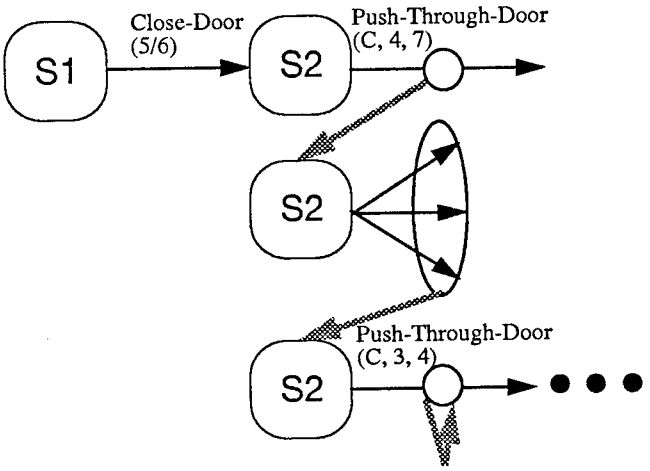


Figure 9: Refinement of the abstract plan in the robot domain.

As with as-needed planning in general, impasse-driven abstraction engages in some trade-offs with respect to the more traditional, preprocessing approaches to abstraction, such as ABSTRIPS [Sacerdoti, 1974] and ALPINE [Knoblock, 1990]. The principal advantage is that the determination of what to abstract when can be driven by the dynamic needs of the agent in the context of the problem to be solved. The principal disadvantage is that, because abstraction is focused on obviating the local impasse, rather than on optimizing global performance, potentially powerful abstractions may be missed.

In addition to obviating (abstracting) applicability impasses, research in Soar on obviating (abstracting) execution impasses in a computer-configuration domain is reported in [Unruh, Rosenbloom, & Laird, 1987; Unruh & Rosenbloom, 1989]. This leads to successive-refinement behavior for the execution of operations. More general impasse-driven goal and/or problem space reformulations are also conceivable, but have not yet been investigated in any great depth.

4. Relationship to Other Work

This work does not stand alone. Because the flexibility of the framework comes from reflective processing in response to impasses, there is a close relationship to other work on meta-level architectures and reflection [Maes & Nardi, 1988]. In focusing on a set of standardized responses to behavioral impasses, this work is very much in the spirit of repair theory [Brown & VanLehn, 1980]. Veloso has developed a nonlinear planner for the PRODIGY architecture [Veloso, 1989], has proposed to use a memory-driven (case-based) strategy to drive the planning, and has identified a set of three general responses to be made when a justification from a case fails to apply [Veloso & Carbonell, 1990]. The BUILD system employed the notion of "gripes" and of "gripe handlers" that had flexible meta-level access to the situation [Fahlman, 1974]. In TWEAK, Chapman developed an integrative framework for partial-order planning [Chapman, 1987].

5. Conclusions

Planning is an important, but still rather disorganized, subdomain of artificial intelligence. One route towards ameliorating this is to develop a framework for planning behaviors. Not only might it help bring organization to the field, but it might also provide the basis for constructing flexible and powerful planning agents.

In this article, we have described one on-going research effort aimed at doing just this. The framework is based on the combination of a basic memory-driven agent, behavioral impasses, and generic responses to these impasses. Learning is also discussed as a way to compile impasse responses into bounded memory structures, thus allowing future behavior to be more memory-driven. Though the framework is still quite a ways from completion, several methods have already been covered, and decomposed into combinations of more primitive elements.

A number of things remain to be done. First, the existing framework needs to be refined, especially in areas like pursuit, where it casts too coarse a net. Second, the scope of the framework must be expanded to cover a wider range of planning

[7] This assumption is relaxed in the iterative abstraction method.

behaviors, such as temporal planning. Covering these may only require extending our understanding of the existing framework, but it is likely that actual extensions of the framework will also be required. Third, a more systematic investigation is required in which all of the generic responses are crossed with all of the impasse types. This should reveal how additional known planning methods are covered, and, more importantly, may yield new interesting methods. Fourth, investigations are needed into how to intelligently mix the generic responses across different types of impasses, and across different impasses of the same type. We have already seen some of this sort of mixing — of different forms of pursuit, of pursuit with termination, and of pursuit with obviation — but much more is possible. This has the potential to generate both novel methods, and more sophisticated versions of existing methods. For example, more sophisticated abstraction methods could be enabled by making decisions on an impasse-by-impasse basis about whether to obviate or pursue applicability impasses. This fits well within the framework, but it is not yet clear what knowledge is needed to make such decisions in an intelligent fashion. Fifth, and finally, this framework needs to be tied in with work on weak problem solving methods and on skill acquisition. On the former, the weak problem solving methods — such as depth-first search and hill climbing — cover a large segment of the generic pursuit methods, so efforts at developing integrative frameworks for the weak methods [Laird & Newell, 1983; Bennett & Dietterich, 1986; Nau, Kumar and Kanal, 1982] are particularly relevant. On the latter, augmenting memory with new plan information is a form of skill acquisition, so work on such topics as control-rule and macro-operator acquisition [Fikes, Hart, & Nilsson, 1972; Korf, 1985; Laird, Rosenbloom, & Newell, 1986; Langley, 1985; Mitchell, Utgoff, & Banerji, 1983] is particularly relevant.

Acknowledgements

We would like to thank John Laird and Bill Swartout for helpful discussions on this topic.

References

Bennett, J. S., & Dietterich, T. G. (1986). *The test incorporation hypothesis and the weak methods* (Tech. Rep. 86-30-4). Department of Computer Science, Oregon State University.

Brown, J. S. & VanLehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science, 4,* 379-426.

Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence, 32,* 333-377.

Fahlman, S. (1974). A planning system for robot construction tasks. *Artificial Intelligence, 5,* 1-49.

Fikes, R. E., Hart, P. E., & Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence, 3,* 251-288.

Golding, A. R., Rosenbloom, P. S., & Laird, J. E. (1987). Learning general search control from outside guidance. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence.* Milan: IJCAII.

Knoblock, C. A. (1990). Learning abstraction hierarchies for problem solving. *Proceedings of the Eighth National Conference on Artificial Intelligence.* Boston: AAAI, In press.

Korf, R. E. (1985). Macro-operators: A weak method for learning. *Artificial Intelligence, 26,* 35-77.

Laird, J. E. (1983). *Universal Subgoaling.* Doctoral dissertation, Carnegie-Mellon University, (Available in Laird, J. E., Rosenbloom, P. S., & Newell, A. *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies,* Hingham, MA: Kluwer, 1986).

Laird, J. E. & Newell, A. (1983). A universal weak method: Summary of results. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence.* Karlsruhe: IJCAII.

Laird, J. E., & Rosenbloom, P. S. (1990). Integrating execution, planning, and learning in Soar for external environments. *Proceedings of the Eighth National Conference on Artificial Intelligence.* Boston: AAAI, In press.

Laird, J.E., Hucka, M., Yager, E.S., and Tuck, C.M. (1990). Correcting and extending domain knowledge using outside guidance. *Proceedings of the Seventh International Conference on Machine Learning.* Austin.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence, 33,* 1-64.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning, 1,* 11-46.

Langley, P. (1985). Learning to Search: From weak

methods to domain-specific heuristics. *Cognitive Science, 9,* 217-260.

Maes, P., & Nardi, D. (Eds.). (1988). *Meta-Level Architectures and Reflection.* Amsterdam: North Holland.

Mitchell, T. M., Utgoff, P. E., & Banerji, R. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. S. Michalski, J. G. Carbonell, T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach.* Palo Alto, CA: Tioga Publishing Co.

Nau, D. S., Kumar, V. and Kanal, L. (1982). A general paradigm for A.I. search procedures. *Proceedings of the National Conference on Artificial Intelligence.* Pittsburgh: AAAI.

Reich, Y. (1988). Learning Plans as a Weak Method for Design. Department of Civil Engineering, Carnegie Mellon University, March, 1988, Unpublished.

Rosenbloom, P. S., Laird, J. E., & Newell, A. (1988). Meta-levels in Soar. In P. Maes & D. Nardi (Eds.), *Meta-Level Architectures and Reflection.* Amsterdam: North Holland.

Rosenbloom, P. S., Newell, A., & Laird, J. E. (1990). Towards the knowledge level in Soar: The role of the architecture in the use of knowledge. In K. VanLehn (Ed.), *Architectures for Intelligence.* Hillsdale, NJ: Erlbaum. In press.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence, 5,* 115-135.

Sacerdoti, E. D. (1977). *A Structure for Plans and Behavior.* New York: Elsevier.

Stefik, M.J. (1981). Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence, 16,* 111-139.

Stefik, M.J. (1981). Planning and meta-planning (MOLGEN: Part 2). *Artificial Intelligence, 16,* 141-169.

Unruh, A. & Rosenbloom, P. S. (1989). Abstraction in Problem Solving and Learning. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* Detroit: IJCAII.

Unruh, A., & Rosenbloom, P. S. (1990). Two new weak method increments for abstraction. T. Ellman (Ed.), *Working Notes of the AAAI-90 Workshop on Automatic Generation of Approximations and Abstractions.* Boston: AAAI, In press.

Unruh, A., Rosenbloom, P. S., & Laird, J. E. (1987). Dynamic abstraction problem solving in Soar. *Proceedings of the Third Annual Aerospace Applications of Artificial Intelligence Conference.* Dayton, OH.

Veloso, M. M. (December 1989). *Nonlinear problem solving using intelligent casual-committment* (Tech. Rep. 89-210). School of Computer Science, Carnegie Mellon University.

Veloso, M. M. & Carbonell, J. G. (1990). Integrating analogy into a general problem-solving architecture. M. Zemankova and Z. Ras (Eds.), *Intelligent Systems.* .

Wilkins, D.E. (1984). Domain-independent planning: representation and plan generation. *Artificial Intelligence, 22,* 269-301.

O-Plan2: Choice Ordering Mechanisms in an AI Planning Architecture *

Austin Tate
Artificial Intelligence Applications Institute
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
United Kingdom

Abstract

O-Plan2 is an AI planning architecture which supports research into a number of aspects of planning, scheduling and control. It is based on earlier work on the O-Plan System which was directed towards plan generation. The paper explores the different types of choice ordering decisions which need to be made in an architecture for command and control. The mechanisms for choice ordering and selection in the original O-Plan system were found to be too general for efficient use for all purposes. The paper describes a number of choice ordering mechanisms provided in the O-Plan2 Architecture which provide specialised mechanisms more suited to the range of different ordering problems that arise in planning, scheduling and control applications.

1 Introduction

O-Plan is a continuation of earlier work on Nonlin [Tate, 1977] and was influenced by a number of other systems developed in the late '70s and early '80s. In particular it inherits features from:

- NOAH: [Sacerdoti, 1977] by using a *least commitment* search strategy based on a *hierarchical* representation of plans in which actions may be *partially ordered.*
- Nonlin: which introduced the notion of *goal structure* as a means of recording the rationale behind actions in the plan, and also the use of *typed preconditions* as an aid to search space control. A declarative Task Formalism (TF) was also used to provide a description of applications to the planner.
- Deviser: [Vere, 1981] itself derived from Nonlin but was extended to handle *time* and *events.*
- Molgen: [Stefik, 1981] notable for its ability to perform object selection using *least commitment principles.* This is supported by constraint formulation and propagation techniques.
- McDermott: [McDermott, 1978] provided the notion of defining a plan to encompass the decisions on plan structure already taken and outstanding problems still to be handled by the planner.
- OPM: [Hayes-Roth & Hayes Roth, 1979] provided an *opportunistic* planning framework in a *blackboard* architecture. It introduced the concept of *cognitive specialists* which can make certain kinds of decisions to alter the plan as it is being built and showed how measures of the worth of invoking these specialists could be utilised.

O-Plan borrows from these systems, but importantly it presents a framework, or architecture, which enables these techniques to be incorporated into a single system in a uniform way. The system is fully described in [Currie & Tate, 1989].

O-Plan2 is a more portable redesign and reimplementation of the O-Plan architecture in a Common Lisp, X-Windows and Unix environment. It improves on O-Plan in a number of ways. This paper will give an overview of the O-Plan2 architecture and describe the different mechanisms provided within the architecture to enable the planning and control system builder to select suitable implementation methods for describing choices, posting constraints which will restrict choice, postponing choice making decisions until the most opportune time to make them, and triggering choices that are ready to be acted upon.

2 O-Plan2 Architecture

O-Plan2 is a domain independent architecture to support the construction of planning, scheduling and control systems. By providing suitable versions of the Domain Description, Plan State, Knowledge Sources and Support Tools, the architecture can be tailored and made more efficient for specialised use. Three different systems are currently being explored using the same basic architecture.

- An activity based task planner: O-Plan is being retained as the name for the activity based task plan-

*This work is supported by the US Air Force/European Office of Aerospace Research and Development by grant number EOARD/88-0044 monitored by Dr Nort Fowler at the Rome Air Development Center. The views expressed are those of the author only.

ning application of the architecture.

- A scheduler: TOSCA is a variant of the system specialised to manufacturing scheduling applications. Here the plan state includes information on the work capacity of the machines available and resource based representations of the schedules being constructed. Knowledge sources are specialised to resource analysis and resource based planning.
- A planner with a logical temporal representation: a project is underway to employ a temporal logic used for temporal data bases as the basis for the plan state. Specialised Domain Description, Knowledge Sources and Support Tools will allow the planner to generate plans in this representation.

The basic O-Plan2 architecture with definitions of its parts suited to activity based task planning is shown in Figure 1.

The Task Formalism or TF domain description is compiled into static data structures, to be used during the plan generation process - in particular, activities are represented as *schemas*. The left hand side of Figure 1 denotes the *plan state*, which comprises the emerging plan (based on the partial order of activities), the list of plan *flaws*, and internal detail such as the Goal Structure [Tate, 1977], the effects of activities (as in NOAH's Table of Multiple Effects or TOME [Sacerdoti, 1977]) and plan variables. The flaws are posted onto *agenda* lists, which are simply lists of outstanding tasks to be performed during the planning process, and are picked off by an overall *controller* to be processed by the *knowledge sources* in the middle of the diagram. The knowledge sources provided represent the planning knowledge of the system and are referred to as *plan modification operators*. Knowledge sources run on one or more *knowledge source platforms* which are able to run some or all of the available knowledge sources. Knowledge sources in turn may add detail to the plan state, for example by expanding actions to greater levels of detail, establishing how conditions are satisfied, adding ordering links or choosing bindings for plan variables. The knowledge sources may also post new flaws as a result of discovering constraint violations, detecting goal interactions and other problems. The knowledge sources also provide the means whereby a user can assist the planner.

There are a range of flaw types and each is matched with an appropriate knowledge source which can process the particular flaw. Recognised flaw types in the activity based task planner include `expand an action`, `satisfy a condition`, `add a link`, `bind a plan variable` and even `call the user`. This approach allows for the extension of the capabilities of the system. O-Plan knowledge sources relate to plan generation only. The early versions of O-Plan2 will replicate the plan generation features of O-Plan. However, new flaw types and knowledge sources will be provided in O-Plan2 to provide an experimental platform for planning and control in a simplified distributed environment comprising a ground based planner and a space-borne execution agent. The O-Plan2 architecture will support the reasoning of both agents and the extraction and patching in of plans fragments between the on-going control environments of both agents [Tate, 1989].

O-Plan2 is built up in a succession of layers of functionality in order to support the control requirements in a concise manner. At the lowest level is a simple fact storage and retrieval database. This is used to provide support for effect and condition maintenance in a context layered fashion. In turn the effect and condition manager maintains "clouds" of (aggregated) effects and holding periods (ranges) for effects contributing to the satisfaction of necessary conditions in the plan state being developed [Tate, 1986]. Moving up the layers, this is turn provides support for QA (Question Answering) which is the basic reasoning component within the system. QA results drive plan state alterations made by the planner's knowledge sources which in turn are maintained by the net management and time point network manager module. To facilitate re-use of support tools across a range of different specialisations of the O-Plan2 architecture, there is a clear distinction between the plan state specific description (called by us the *associated data structure*) and the underlying management of time points and temporal relationships.

O-Plan2 is given tasks by adding entries to its plan state flaw list (agendas). O-Plan2 maintains a *partial plan* at all stages, and makes alterations to the partial plan and the flaw list as it proceeds. The partial plan represents a complete description of a set of possible plans which are only partially specified. The controller is responsible for selecting an outstanding flaw to process whenever a knowledge source can be activated on a waiting knowledge source platform. The domain information is consulted by knowledge sources as they run. This lets knowledge sources access task descriptions, definitions of resources and other domain constraints. The domain information also gives access to the operator schemas which define higher level activities in terms of more detailed activities. There will often be more than one plan modification possible; that is, there will often be a choice of how to remove a flaw. These choices lead to *search*. Normally, the consequences of a decision are maintained by the support tools and information about the selection made is stored as dependency information within the plan state. However, there are occasions on which alternative plan states may need to be generated to explore the options. O-Plan and O-Plan2 allow for such alternative based explorations.

O-Plan searches through a space of partial plans, modifying one plan to obtain another. It seeks a complete plan that is free of flaws - though this may not be achievable in continuous command and control applications. The plans produced by the activity based task planner variant of O-Plan2 are described in networks. The nodes in the network denote actions, and the arcs signify an ordering on action execution. Each node has information associated with it which describes the action's conditions and effects. Time and resource information can also be associated with each action in a plan network node.

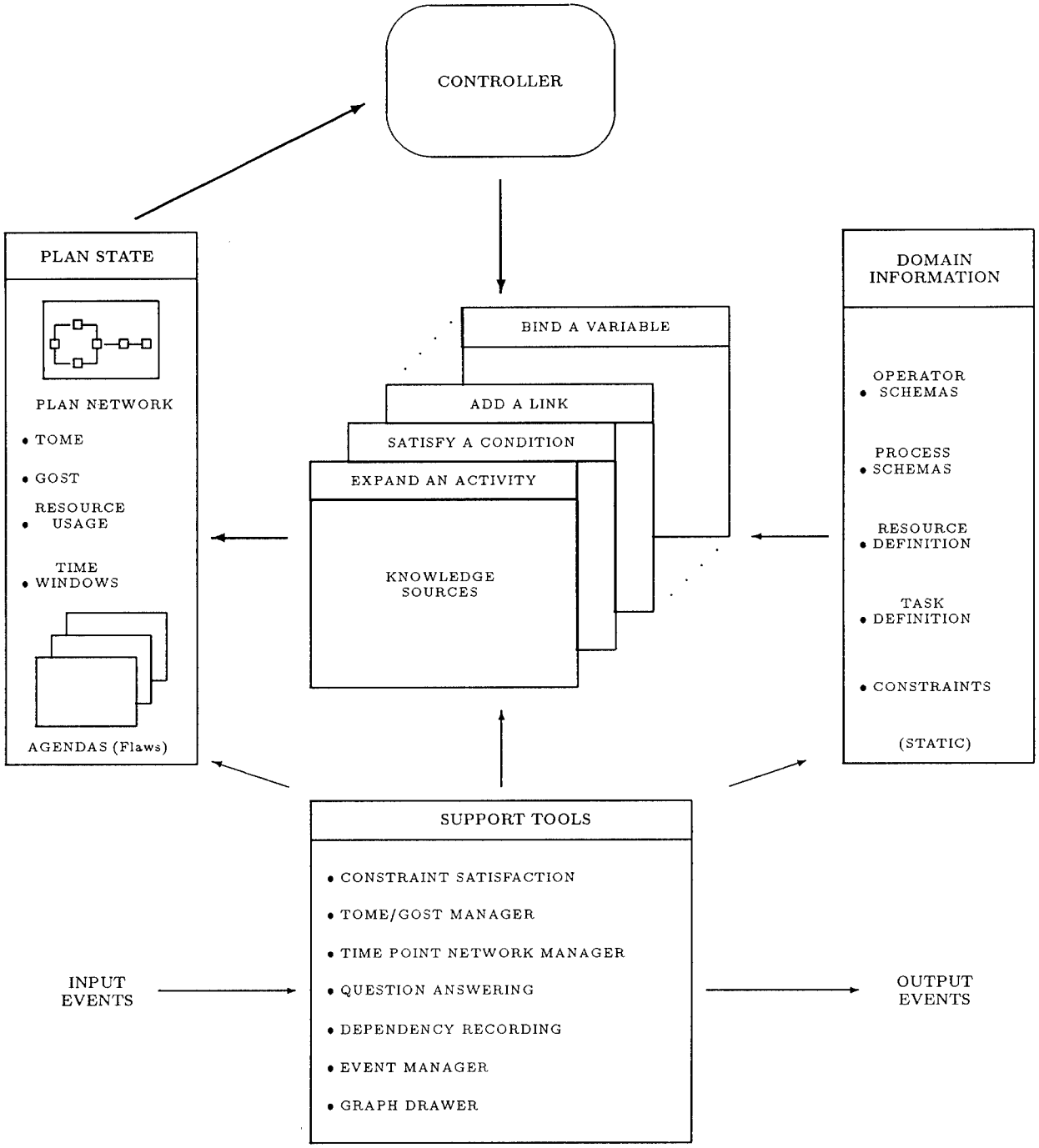


Figure 1: O-Plan2 Architecture

3 Ordering Mechanisms in O-Plan

The O-Plan system seeks to include mechanisms to allow for the implementation of an efficient planning system able to take an opportunistic approach to selecting where computational effort should be concentrated during planning. This aim was only partially achieved in the original O-Plan. The basic mechanisms are listed in the following sections.

3.1 Building up information in an Agenda Record

O-Plan included the ability to allow a knowledge source to examine a possible decision point (represented by the agenda entry it was asked to process) and to add information relating to the choice to the fields of the agenda record. If the choice did not become suitably tightly restricted as a result of the addition of this information, it is possible to put the agenda entry back onto the outstanding flaws list with improved information for deciding on the time to reselect it for processing. The ability to build up information around an agenda entry in an incremental way prior to a final knowledge source activation is an important feature that ensures that work done in accessing data bases and checking conditions can be saved as far as possible when processing is halted. There are some similarities to mechanisms within real-time responsive architectures such as RT-1 [Sridharan, 1988].

3.2 Granularity of Knowledge Sources

Each knowledge source within the O-Plan2 architecture encodes a piece of planning knowledge. For example, how to expand an action, bind a variable, check a resource, etc. From a modularity viewpoint, there is some advantage in having a very fine grain of knowledge source to implement planning knowledge. However, this can lead to tens of agenda entries and knowledge source activations with the overheads associated with such activations for even the simplest types of action expansion. In simpler planners, such as Nonlin, an expansion is efficiently handled as an atomic operation. There is a conflicting desire to have efficient large grain knowledge sources implementing planning knowledge and very fine grain knowledge sources detailing each individual step of some higher level plan modification operator.

With a finer grain of knowledge source, it was also found that ordering relationships between agenda entries left in the agenda list had to be stated to ensure efficient processing. The controller was then required to unravel the web of activation orderings that resulted. A special form of agenda entry called a *sequence* was implemented to assist the controller in this task, it would only consider the head of the sequence for activation at any time, subsequently releasing the following agenda items clustered in the sequence in the order indicated. This process is similar to the control blocks used in the Tecknowledge s.1 system [Tecknowledge, 1988].

3.3 Priority of Processing Agenda Entries

O-Plan assigns priorities to every flaw placed on the agendas at the time they are placed. The priorities are calculated from the flaw type, the degree of determinancy of the flaw and information built up in the Agenda Record as described earlier. These provide measures of choice within the flaw. Two heuristic measures are maintained in each agenda entry. One called BRANCH-1 indicates the immediate branching ratio for the choice point. An upper bound on this can be maintained quite straightforwardly. The second measure is called BRANCH-N and gives a heuristic estimate of the number of distinct alternatives that could be generated by a naive and unconstrained generation of all the choices represented by the choice point.

In O-Plan, three agendas are maintained to efficiently select between agenda entries which are ready for knowledge source activation and ones awaiting further information to bind open variables in the agenda information. This is described in [Currie & Tate, 1985]. Eventually though, the ready to run agenda entries are simply rated according to a numerical priority maintained for each agenda entry on the basis of flaw type and the BRANCH-1 and BRANCH-N estimators. This forms too simplistic a measure for allowing the controller to decide between waiting agenda entries. Consideration was given to a rule based controller with knowledge of other *measures of opportunism* but no implementation of this was done within the original O-Plan.

4 Ordering Mechanisms in O-Plan2

O-Plan2 seeks to provide a more coherent set of mechanisms to enable the planning and control system builder to select suitable implementation methods for describing choices, posting constraints which will restrict choice, postponing choice making decisions until the most opportune time to make them, and triggering choices that are ready to be acted upon.

4.1 Knowledge Source Stages

The O-Plan mechanism for building up information in an agenda entry prior to making some selection between alternatives was a very useful feature but proved difficult to use in practice. A knowledge source had to be activated to initiate processing which might simply add a little information to the agenda entries and then suspend to allow the controller to decide whether to progress. This is very inefficient.

In O-Plan2, knowledge sources are defined in a series of *stages*. There can be one or more stages, only the last may make alterations to the plan state (thus locking out other knowledge source final stages which can write to the same portion of the plan state). Any earlier stages may build up information useful to later stages. At the end of any stage, the knowledge source must be prepared to halt processing if asked to by the controller. If it is asked to halt at a stage boundary, the knowledge source may summarise the results of its computation in a field of the agenda record provided for this purpose. A controller directed support routine is called by the knowledge source at the end of each stage to identify whether it must halt or may continue. This allows the controller to dynamically re-direct computation as it considers all the information available to it, while providing a simple

and efficient way for the knowledge source to continue computation without intermediate state saving while it continues to receive a go-ahead from the end of stage continuation authorisation routine.

A *Knowledge Source Formalism* for O-Plan2 is being designed to allow for stage definition and to assist with declaring the restrictions on the plan state portions affected by the final plan state modifying stage of the knowledge source - to assist in lock management.

4.2 Knowledge Source Triggers

In O-Plan2, a mechanism of setting *triggers* on agenda entries for activating knowledge sources (and an individual stage of a knowledge source if desired) is provided. The triggers may use various "items" of data available within the plan state and other global information available to the planner. These may include things such as the availability of a specific binding for a plan variable, the satisfaction of a condition at a specific action node in the plan network, the use of a specific resource, the occurrence of an external event, information from the "clock" within the planner, etc. The Knowledge Source Formalism referred to earlier will also be used to define triggers on knowledge source stages. The triggering constructs in the language will initially be quite restrictive to ensure that efficient agenda entry triggering mechanisms can be implemented. However, as we gain experience, we expect the triggering language to be quite comprehensive. A knowledge source may also dynamically create a trigger on a continuation agenda entry when halting processing at a stage boundary.

Only agenda entries which are currently triggered will be available to the controller for decisions on which entries to activate through to a knowledge source running on a knowledge source platform.

4.3 Compound Agenda Entries

Individual *simple* agenda entries can be grouped together into *compound* agenda entries. Only the head entries in the compound agenda entry are considered at any time by the controller (and possibly by the triggering mechanism considered above), thus cutting down on the amount of processing required by the controller to select the next agenda entry to execute when such pre-defined orderings can be specified. Compound agenda entries can be made by knowledge source to implement some definite planning strategy or to implement planning algorithms with finer grain knowledge sources to provide modularity and real time response improvement.

A Support Routine is to be provided in O-Plan2 to allow any knowledge source to easily and reliably build and return a compound agenda entry.

4.4 Controller Priorities

The controller is given the task of deciding which of the current set of triggered agenda entries should be run on an available knowledge source platform. It does this by considering the priority and measures of opportunism of the agenda entry. Four priority levels are available within O-Plan2 - Low, Medium, High and Emergency. The Emergency priority level is only available to handle incoming external events. The RT-1 system has similar priority based processing arrangements [Sridharan, 1988]. In certain cases, an O-Plan2 implementation will possess knowledge source platforms dedicated to processing specific real-time responsive events appearing as agenda entries - thus allowing for reliably real-time response to events categorised as Emergency priority.

A waiting knowledge source platform will be able to run one, several, many or all knowledge sources. Any restriction on a specific platform will be known to the controller. Only triggered agenda entries at the highest priority level which can be processed on a waiting knowledge source are considered by the controller on each cycle. Where there is still choice, a range of *measures of opportunism and priority* are employed to make a selection. The underlying principle is to make a selection according to a strategy given to the controller. Initially this strategy will use user selected preferences or by default will seek to reduce search to the extent it can judge this (reflecting the opportunistic generative planning nature of the early versions of O-Plan2 - like its predecessor O-Plan). Measures such as BRANCH-1 and BRANCH-N described earlier for O-Plan are relevant to this. However, the use of a utility function guided by task specifiers given to the controller will be explored later for O-Plan2 when it is used in continuous command and control applications.

5 Summary

O-Plan2 seeks to provide a more coherent set of mechanisms to enable the planning and control system builder to select suitable implementation methods for controlling the flow and ordering of making choices in an AI planner. These mechanisms are:

- the use of *stages* in knowledge sources to allow for a linear thread of computation to be defined which can be assumed to run through to completion, but provides a means for interruption at defined staging points.
- the definition of *triggers* on knowledge sources and knowledge source stages to provide a clear means to delegate a higher level of knowledge source activation checks to the controller.
- the use of *compound agenda entries* to put direct dependencies of some tasks on others that must complete earlier. This allows complex computational dependencies and strategies to be created.
- the use of *agenda manager priorities* to allow the controller to select appropriate ready-to-run agenda entries and match these to waiting knowledge source platforms.

All the mechanisms described above are part of the O-Plan2 planner now being constructed.

Acknowledgements

My thanks to my colleagues on the O-Plan and O-Plan2 projects: Ken Currie, Brian Drabble and Richard Kirby.

References

[Currie & Tate, 1985] Currie, K.W. & Tate. A. O-Plan: control in the open planning architecture. *In proc. of the* BCS *Expert Systems '85, Warwick,* UK, *Cambridge University Press, 1985.*

[Currie & Tate, 1989] Currie, K.W. & Tate, A. O-Plan: the Open Planning Architecture. *Submitted to the AI Journal. Also* AIAI-TR-*67. 1989.*

[Hayes-Roth & Hayes Roth, 1979] Hayes-Roth, B. & Hayes-Roth, F. A Cognitive Model of Planning. *Cognitive Science,pp 275 to 310, 1979.*

[McDermott, 1978] McDermott, D.V. A Temporal Logic for Reasoning about Processes and Plans In *Cognitive Science, 6, pp 101-155, 1978.*

[Sacerdoti, 1977] Sacerdoti, E. A structure for plans and behaviours. *Artificial Intelligence series, publ. North Holland, 1977.*

[Sridharan, 1988] Sridharan, N. Practical Planning Systems, *Rochester Planning Workshop,* AFOSR, *1988.*

[Stefik, 1981] Stefik, M. Planning with constraints. *In Artificial Intelligence, Vol. 16, pp. 111-140. 1981.*

[Tate, 1977] Tate, A. Generating project networks. *In procs.* IJCAI-77, *Cambridge,* USA, *1977.*

[Tate, 1986] Tate, A. Goal Structure, Holding Periods and "Clouds". *In Reasoning about actions and plans, Morgan-Kaufmann, 1986.*

[Tate, 1989] Tate, A. Coordinating the activities of a planner and an execution agent. *In Procs. of the Second NASA Conference on Space Telerobotics, (eds. G.Rodriguez & H.Seraji), JPL Publication 89-7 Vol. 1 pp. 385-393, Jet Propulsion Laboratory, Pasadena, CA, 1989.*

[Tate, 1990] Tate, A. Interfacing a CAD system to an AI planner. *paper to the* SERC *seminar on integrating knowledge-based and conventional systems, Edinburgh, May 1990. Also Artificial Intelligence Applications Institute* AIAI-TR-76, *1990.*

[Tecknowledge, 1988] Tecknowledge, s.1 product Description, Tecknowledge Inc., 525 University Avenue, palo Alto, CA 94301. 1988.

[Vere, 1981] Vere, S. Planning in time: windows and durations for activities and goals. IEEE *Transactions on Pattern Analysis and Machine Intelligence Vol. 5, 1981.*

[Wilkins, 1988] Wilkins, D. Practical Planning. *Morgan Kaufman, 1988.*

Hypergames and AI in Automated Adversarial Planning

Russell R. Vane, III
The Young Guard Company
400 Montpelier Road
Great Falls, VA 22066

Paul E. Lehner
Center for Excellence in C3I
George Mason University
4400 University Drive
Fairfax, VA 22030
plehner@gmuvax.gmu.edu

ABSTRACT

This paper[1] examines how game theory techniques can be combined with knowledge-based planning procedures to reason about an adversary's beliefs and the extent to which a competitive agent is capable of defeating a plan. The main results are that (a) the hypergame provides a convenient mechanism for representing and reasoning about knowledge/data not available to a competitive agent and (b) automated implementation of this form of game theory-based reasoning is conceptually straightforward.

1.0 INTRODUCTION

In multi-agent environments effective planning requires that the planner have an ability to reason about beliefs, intentions, and likely actions of other agents. This fact has lead various researchers to explore alternative formalisms for reasoning about other agents' beliefs (e.g. Halpern, 1986; Vardi 1988). Unfortunately, most of these approaches only deal with categorical assertions about belief; and agent either does or does not believe some sentence. Statements such as Probably 'A believes X,' or Probably 'A is not aware of option X' cannot be reasoned about. This is a severe limitation, since it is rare that one can predict an agent's actions, particularly those of an adversary, with an precision. Practical planning requires the generation of plans that are flexible and robust against *probable* actions of other agents.

In this paper we show how game theory techniques can be combined with AI planning techniques to reason probabilistically about an agent's beliefs. At the same time, we demonstrate how AI techniques provide a new approach to the "outguessing problem" in game theory.

This discussion focuses on adversarial planning/competitive games. However, the basic approach and all the algorithms are applicable to noncompetitive planning problems as well.

The material below is decomposed into two parts. Section 2.0 is a self-contained discussion of game theory and on use of *hypergames* in adversarial planning. Section 3.0 then discusses how to implement this form of reasoning as part of an automated adversarial planning system.

2.0 GAMES AND HYPERGAMES

In this section we are going to introduce some primitive game theory concepts, develop a consistent set of terms, and give you a peek at hypergames. This system was specifically designed to address problems which are factorial or exponential in terms of possible strategies.

GAME THEORY

In game theory, an opponent is referred to as a *player*. Each player has a countable number of choices, called *strategies*. The possible *outcomes* of a game are a function of all of the strategies available to both players. The optimal solution of a game may be a pure strategy or a mixture of pure strategies. A *pure strategy* is a single strategy. A *mixed strategy* combines two or more of a player's pure strategies. If the outcomes for one player are the negative of the outcomes for the other player, then the game is called a two-player, zero-sum game. Only one value for each strategy pair is needed to describe this kind of game. We usually show the outcome from player A's perspective.

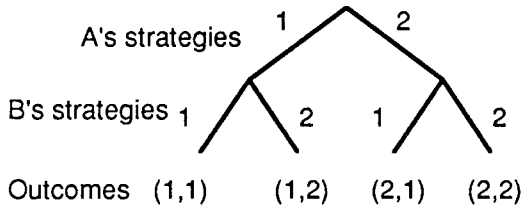
Extensive Form

The extensive game is a representation of all the possible play by players and chance in a game. For even games as simple as 2-player Monopoly™, this form must consider all die rolls for both players and their buying decisions (estimated conservatively as 11x2x2, or 44) for each turn. This form has its computational problems. However, it is the most information containing form, so a more rigorous definition follows.

[1] This research is part of an ongoing research program in automated adversarial planning. Support for this research program is provided by the Center of Excellence in Command, Control, Communications, and Intelligence at George Mason University. The Center's general research program is sponsored by the Virginia Center for Innovative Technology, MITRE Corporation, the Defense Communications Agency, CECOM, PRC/ATI, ASD(C3I), TRW, AFCEA, and AFCEA NOVA.

The extensive form of a game is a finite tree which represents all of the possible moves in a game. Its origin, o, represents the initial starting position. Each node of the tree is a possible future position. Each edge of the tree is a player's alternative. The endnodes are those that have no future alternatives, that is the game is over when an endnode position is reached. Any path from the origin to an endnode is called a play.

The tree itself can be depicted as a n-ply tree, where the number of opponents, n, are represented at each ply. Each ply is used to show the alternatives available to the player being examined. In standard games such as Monopoly, Poker, and Chess each n-ply are called a turn. Each ply can represent a simultaneous decision or a sequential one. Therefore a Planner which plans m turns ahead accounts for m x n plies in the tree.



Normal Form

The normal (or matrix) form of a game is a synopsis of the extensive form. The normal form is based on choices of strategy, which may be thought of as a complete set of contingency plans for every situation which might arise. Although it is not easy to identify all of the strategies available to a player, it is theoretically possible to do this for every finite game. In this form every player has but one move, the choice of a strategy. Most examples in Game Theory are shown in this form.

	Player B	
Player A	B's Strategy 1	B's Strategy 2
A's strategy 1	outcome(1,1)	outcome(1,2)
A's strategy 2	outcome(2,1)	outcome(2,2)

Normal form of 2 Player Game

Games represent the ultimate case of a lack of information about the reasoning process of the opponent. The result is that a very conservative criterion, the MAXIMIN and MINIMAX criterion, is usually proposed for two player zero-sum games. This criterion selects a strategy which yields the best of the worst possible outcomes.

MAXIMIN is the maximum of the minimums. It is the lower bound of the value of the game.

MINIMAX is the minimum of the maximums. It is the upper bound of the value of the game.

An optimal solution is defined as a strategy in which neither player gains a benefit from altering his strategy. In this case the game is said to be stable or in a state of equilibrium.

	Player B		
Player A	Strategy 1	Strategy 2	MAXIMIN
A's strategy 1	2	3	***2***
A's strategy 2	-1	4	-1
MINIMAX	***2***	4	2

An Example of a Game with a Pure Strategy Equilibrium

The example provides the following information:

A's MAXIMIN (rightmost column) strategy is strategy 1 with a value of 2, italicized and bolded in the MAXIMIN column.

B's MINIMAX (bottommost row) strategy is strategy 1 with a value of 2, italicized and bolded in the MINIMAX row.

The game is stable and the value of the game is 2 (in the double box). We will adopt two terms from the US Army:

Course of Action (*COA*) - a strategy for our forces.
Plan - a strategy for enemy forces.

	Enemy		
US Forces	Plan 1	Plan 2	MAXIMIN
COA 1	2	3	2
COA 2	-1	4	-1
MINIMAX	2	4	2

An Example of a US Forces Game with US Outcomes.

Mixed strategy - a probabilistically weighted use of more than one plan or COA.

		Enemy			
	%	0%	42%	58%	
%	US Forces	Plan 1	Plan 2	Plan 3	MAXIMIN
75%	COA 1	2	3	0	0
0%	COA 2	-1	4	-2	-2
25%	COA 3	1	-4	5	-4
	MINIMAX	2	4	5	0 to 2

An Example of a US Mixed Strategy

Using a linear program to solve, the US Forces should use COAs 1-3 in the mix: [75%, 0%, 25%]. The enemy is calculated to be optimal when using Plans 1-3 in the following mix: [0%, 42%, 58%]. The actual value of the mixed strategies is 1.25. The value of a game will always lie between the MAXIMIN and MINIMAX bounds (0 and 2, in this case). Several interesting properties are seen in the above case.

(1) Some strategies are dominated by other combinations of strategies, such as Plan 1 for the enemy and COA 2 for us.

(2) The mixed strategy does not even include the MINIMAX strategy of player B, that is the pure strategy which guarantees the worst result for player B.

(3) The value of the game is something other than a result in the chart.

But what if you know that the opponent is likely to use plan 1? How do we reason about an opponent with whom we are familiar? We are discussing a strategy selection in a situation which is not at the equilibrium point. What is our optimal plan against a mixed strategy which incorporates such reasoning?

HYPERGAME THEORY

Hypergame theory has been used to discuss games where both sides are playing different games (Bennett 1982). This subjective approach has many intuitive features that would be valuable if an optimal hyperstrategy could be derived. It can be used to support the reason that master game players rarely use their most sophisticated strategies on novices. Hypergame theory is necessary anytime one side knows information unavailable to the other side. This information can be a novel course of action, as well as the value of surprise, deception, or unexpected reinforcement. It could answer questions about incorporating experience and planning time differences when choosing a course of action. These questions are particularly appropriate when lots of combinations of tactical moves are available to players such as in war games. It is also valuable when enemy decisionmakers are concentrating on schemes of maneuver using predictable doctrinal templates.

Our extension to hypergame theory is the process of optimizing one's strategy using subjective probabilities (expert judgments) about the game being considered by an adversary. The basis of these judgments can be military intelligence reports, previous tendencies of an enemy commander, and our knowledge of the enemy's doctrine. There is a conceit that the game that we are using is a superset of the opponent's game, although some probability can be assigned that the opponent is using the complete hypergame. If we are certain about which game an opponent is playing then the probability used in our model is represented as 1.0 for that game. If the enemy might be considering different games then we estimate the probabilities of each. Note: an enemy only considers one game, we apply probability to describe our uncertainty about which game is being played. Game theory is still used to determine the strategies for both sides for each possible game. When the games are combined into a hypergame, a hyperstrategy can be determined.

Hypergames are analogous to a possible world framework with explicit belief assigned to each possible world. Reasoning is done on each of the frames and aggregated for an overall assessment of the strategy adopted by an opponent. Hypergames, therefore represent a modal extension to the logic of normal game theory. In fact to reason about an opponent's belief of our beliefs is an interesting modal logic oriented future step in this work.

HYPERSTRATEGIES

The general form for determining a zero-sum hyperstrategy follows. Let p_i be the probability of an opponent playing a subgame i of the hypergame, which we will call a game. Note that:

$$\sum_{i=1}^{n} p_i = 1$$

In other words, n accounts for all of the possible games that the opponent might play. Let s_i be the enemy minimax strategy vector for the each game, whether mixed or pure. This strategy is determined using standard game theory assumptions and results in a rational strategy. Unfortunately for the opponent, this is a local optimization of only one game. Use the following:

$$S_j = \sum_{i=1}^{n} p_i s_{ij}$$

where p_i is the probability weight of game i, and s_{ij} is the probability weight of playing strategy j for each game.

to determine S_j, the probabilistically weighted strategy vector for the opponent. This vector is the weighted average of the enemy strategies.

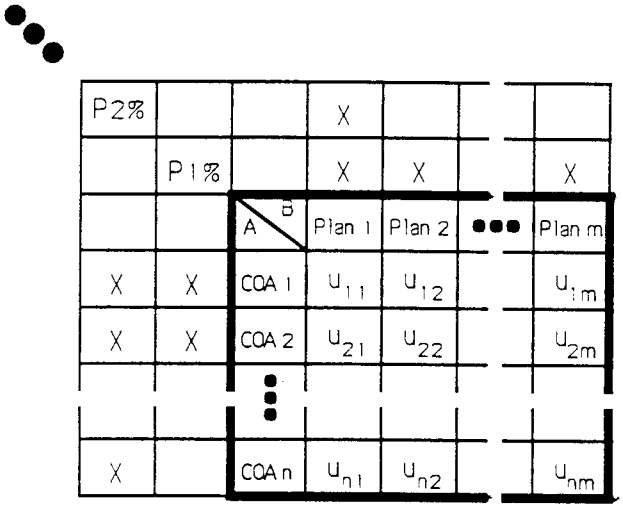
$$H = \max_k \left(\sum_{j=1}^{m} S_j u_{jk} \right)$$

where u_{jk} is the utility value for the outcome Plan j and COA k.

The hyperstrategy, H, can be chosen to maximize the expected value of each row, k, by summing the utilities, u_{jk}, weighted by the probability of the enemy strategy, S_j. The result is the new expected value of the hyperstrategy. Since the decisions of each commander are now decoupled, only a pure strategy is recommended. That a single strategy is recommended is an extremely important result. This means that we can make a definite recommendation. While hyperstrategies are an important, tractable, theoretical extension of game theory, investigations are underway to provide evidence about the practical usefulness of hyperstrategies (Vane 1990) in a wargaming environment.

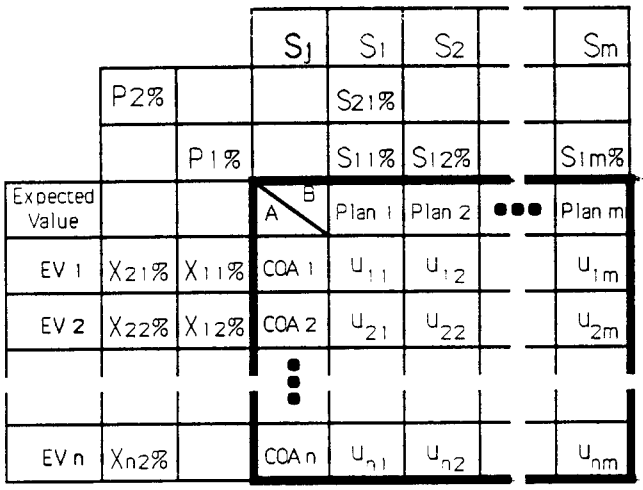
A SIMPLE NOTATION

The following diagrammatic notation, designed by Vane, is introduced to ease discussion of hypergames and hyperstrategies.



NORMAL FORM OF HYPERGAME

This notation shows two games of an m x n hypergame, game 1 weighted by P1% and game 2 by P2%. The complete game is shown in the heavy rectangle. In the above diagram, Game 1 does not include all of the friendly options in the reasoning (at least COA n is missing). Game 2 uses only one enemy strategy, Plan 1. These situations are shown by the X's above Plans and to the left of COAs to show which COAs and Plans are associated with each belief percentage.



NORMAL HYPERGAME WITH SOLUTION

The normal form with solution is a hypergame that is solved. It includes all of the information of the normal form of a hypergame, as well as the actual percentages and expected value for the game. There is a value in every place that an X occurred in the normal form. If the strategy evaluates to 0%, a '0%' placeholder is still placed in the entry, so that one can still reconstruct games. The aggregate vector is determined, S_j, and the expected values calculated. Obviously the maximum value is the pure strategy to choose. This form will not be used in the explanations which follow, so that a step by step construction of the solutions can be presented.

APPLICATIONS OF HYPERSTRATEGIES

We will explain the use of hyperstrategies in one notional case, in one historic example, and provide guidance when uncertainty about enemy reasoning is high.

THE PREDICTABLE FOE

But what if you know that the opponent is likely to use plan 1? We are discussing a strategy selection in a situation which is not at the equilibrium point. What is our optimal plan against a mixed strategy which incorporates such reasoning? First we judge that likely in this case means 70%. Since we have no other

information we assign all of the remaining probability to the hypergame (30%).

We set up a hypergame matrix incorporating the games that are being considered as noted below by X'ing the plans and COAs.

		Enemy		
Subgame 1	70%	X		
Subgame 2	30%	X	X	X

1&2	US Forces	Plan 1	Plan 2	Plan 3	MAXIMIN
X	COA 1	2	3	0	0
X	COA 2	-1	4	-2	-2
X	COA 3	1	-4	5	-4
	MINIMAX	2	4	5	0 to 2

An Example of a US Hypergame

Reasoning about both subgames, using standard game theory: we determine that in subgame 1 that the opponent plays the strategy [100%, 0%, 0%] and the rest of the time [0%, 42%, 58%]. The expected enemy strategy, S_j, is [70%,13%,17%] based on 70% for subgame 1 and 30% for subgame 2. This strategy is a probabilistic composite picture of two games. It is not subject to the equilibrium based MINIMAX argument because the percentages represent our subjective judgment of how likely the opponent is to play each subgame.

		Enemy		
Subgame 1	70%	100%		
Subgame 2	30%	0%	42%	58%

1&2	US Forces	Plan 1	Plan 2	Plan 3	MAXIMIN
X	COA 1	2	3	0	0
X	COA 2	-1	4	-2	-2
X	COA 3	1	-4	5	-4
	MINIMAX	2	4	5	0 to 2

An Example of a US Hyperstrategy

By evaluating the expected value of S_j, the enemy strategy, we see that COA 1 is our only rational choice, which is H. We have undertaken some risk for potential gain. The gain is equal to the risk, which is the expected value of 1.79 versus 1.25, or .54.

Note that the hypergame solution does not have to lie in the MAXIMIN range.

ILLUSTRATIVE HISTORIC EXAMPLE

Let's consider an illustrative game - The Flanders Campaign 1940, from a game theoretic framework. It is known to military practitioners, has been researched by others (Bennett 1979) and shows the power of hypergame theory. Germany and the Allies (France, Britain, Belgium, and the Netherlands) were in a declared war, where both sides had conducted few significant military operations. Neither side was confident of its capacity to wage a successful (decisive and economical) offensive campaign. We will examine the different games that both Allied and German planners were playing. We will trace the military planning paradigm and show where hypergame theory can be applied by decisionmakers in war games, C^2 systems, or decision support systems. Basically, analyzing avenues of approach there are 7 possible defensive plans and 3 offensive plans. The Allied counterattack option of Dupuy (Dupuy 1987) has been ignored , although it most definitely has merit.

The offensive plans are:

- attack in the north across the Belgian Plains (as in 1914),
- attack in the south through the Maginot Line (as in Franco-Prussian War 1870), or
- hey-diddle-diddle through the middle (the Ardennes Forest).

The defensive plans are:

- weighted defense in the south (adding to the already formidable Maginot complex,
- weighted defense in the north,
- weighted defense in the middle,
- defense with a strong reserve in the center,
- defense with strong reserve in the north,
- defense with strong reserve in the south, or
- flank weighted defense (use the natural terrain advantages of the Ardennes for economy of force).

Please see the following normal form matrix to view the alternatives and from an Allied perspective. For ease of exposition we will assume a zero-sum game.

	Attack North	Attack South	Attack Middle*
Defend North	1	-.8	-.5
Defend South	-1	1	-.5
Defend Center	-1	-.8	1
Reserve Center	.4	.4	.8
Reserve North	.8	-.4	.5
Reserve South	-.8	.8	.5
Weighted Flanks*	.6	.6	-.5

Flanders Campaign in normal form

* The actual choices of each side.

The utility values represent the range between complete victory(1) to absolute defeat (-1). All of the battle tuples were judged, not simulated, whether it was good from the Allied perspective. Some of the semantics which underlies the values is:

- the Ardennes (the middle) is inherently a slower battlefield,
- fortifications of the Maginot line are valuable in the protection of Allied troops,
- the inherent attritional nature of combat.

First we will look at the MINs of the Rows and the MAXs of the Columns and cull out any dominated strategies in the matrix below to determine the effective range.

	Attack North	Attack South	Attack Middle *	MIN of Rows
Defend North	1	-.8	-.5	-.8
Defend South	-1	1	-.5	-1
Defend Center	-1	-.8	1	-1
Reserve Center	.4	.4	.8	.4
Reserve North	.8	-.4	.5	-.4
Reserve South	-.8	.8	.5	-.8
Weighted Flanks*	.6	.6	-.5	-.5
MAX of columns	1	1	1	.4 to 1†

Flanders with MINIMAX

† the MAX of the MINs is a lower bound, the MIN of the MAXs is an upper.

The result is that the Allies should win with somewhere between a substantial to a decisive victory. The following linear equations can be used to describe the problem, solve for the expected value, and prescribe the proper mixed strategy.

$$\max(w) = Y_1 + Y_2 + Y_3$$
$$Y_1 - .8Y_2 - .5Y_3 \le 1$$
$$-Y_1 + Y_2 - .5Y_3 \le 1$$
$$-Y_1 - .8Y_2 + Y_3 \le 1$$
$$.4Y_1 + .4Y_2 + .8Y_3 \le 1$$
$$.8Y_1 - .4Y_2 + .5Y_3 \le 1$$
$$-.8Y_1 + .8Y_2 + .5Y_3 \le 1$$
$$.6Y_1 + .6Y_2 - .5Y_3 \le 1$$
$$Y_1, Y_2, Y_3 \ge 0$$

This formulation yields the following linear programming tableau.

Basic	w	Y_1	Y_2	Y_3	S_1	S_2	S_3	S_4	S_5	S_6	S_7	Soln
w	1	-1	-1	-1	0	0	0	0	0	0	0	0
S_1	0	1	-.8	-.5	1	0	0	0	0	0	0	1
S_2	0	-1	1	-.5	0	1	0	0	0	0	0	1
S_3	0	-1	-.8	1	0	0	1	0	0	0	0	1
S_4	0	.4	.4	.8	0	0	0	1	0	0	0	1
S_5	0	.8	-.4	.5	0	0	0	0	1	0	0	1
S_6	0	-.8	.8	.5	0	0	0	0	0	1	0	1
S_7	0	.6	.6	-.5	0	0	0	0	0	0	1	1

Initial Tableau

Calculating for the original problem:

$$v^* = \frac{1}{w}$$
$$y_i^* = \frac{Y_i}{w}$$

The dual tableau can be used to find the Allied strategy. As a result the optimal strategy for both players is a mixed strategy:

The Germans find that they should:

Attack North 62%
Attack South 25%
Attack Middle 13%.

The Allies should play:

Reserve Center 73%
Weighted Flanks 27%.

The value of the game is .45, a virtually guaranteed substantial victory.

The Allies and the Germans did not reason this way, though. Either the Allies did not consider the Ardennes (Attack Middle) or they decided that the Germans would discard it. They played the game:

	Attack North	Attack South	MIN of Rows
Defend North	1	-.8	-.8
Defend South	-1	1	-1
Defend Center	-1	-.8	-1
Reserve Center	.4	.4	.4
Reserve North	.8	-.4	-.4
Reserve South	-.8	.8	-.8
Weighted Flanks*	.6	.6	.6
MAX of columns	1	1	.6 to 1

The Allies' subgame.

The mixed strategy for the Germans would be:

Attack North 78%
Attack South 22%.

The Allied strategy is Weighted Flanks 100%.

The Germans thought that there was a significant chance that the Allies were ignoring their Attack Middle option. How confident did they have to be? From the following matrix, we see that the Germans were able to rationally attack in the middle when they were X% sure that this subgame was being played.

			Attack North	Attack South	Attack Middle *
X%	1-X%	EXPECTED VALUE			
		Defend North	1	-.8	-.5
		Defend South	-1	1	-.5
		Defend Center	-1	-.8	1
	73%	Reserve Center	.4	.4	.8
		Reserve North	.8	-.4	.5
		Reserve South	-.8	.8	.5
100%	27%	Weighted Flanks*	.6	.6	-.5

Flanders with MINIMAX

They determined that if they were only ε% sure that the Allies were ignoring the middle that they should attack there. They actually did attack in the middle and achieved even better results than the expected -.5. They broke through and achieved a decisive victory.

3.0 HYPERGAMES IN AUTOMATED PLANNING

AI research in automated game playing, and to a lesser extent automated planning, has also examined the problem of intelligent play by adversaries. Unlike game theory, AI systems generally deal with the extensive form of the game directly. Move/action generation procedures fall into two categories: game-based and knowledge-based. Game-based enumeration uses the rules of the game to generate options (e.g., a chess playing program that generates all legal moves). Knowledge-based enumeration, on the other hand, generates moves according to a model of the objectives, goals, subgoals, tactics, etc., each player might have. The knowledge-based approach is common for "games" (e.g., go, wargames, military planning, etc.) where there are far too many legal moves to examine exhaustively (e.g., Lehner, 1982, 1990; Reitman and Wilcox, 1979; Young and Lehner, 1986; Wilkins, 1980). It is of course also the approach used ion nonadversarial planning problems (Georgeff, 1987).

When a game-based move generation procedure is used, the game theoretic and AI-based formulations are equivalent. In principle, the extensive form generated by the AI programs could be summarized in normal form, and the same minimax option would be selected. On the other hand, when a knowledge-based procedure is used,

there is a great deal of useful information in the extensive for each branch of the extensive form, there is information about *how that branch was generated.*

To illustrate the usefulness of this information, imagine a context where one agent, after analyzing the situation asserts, "If I were her, I would do X," but then asserts "but determining that X will work involves knowing some advanced tactics which I don't think she is familiar with, so she will probably do Y instead." This type of reasoning involves several steps. First generate the extensive form of the game by solving the problem yourself. Second, assess whether the agent is capable of generating the same extensive form (i.e., identify the subgames). Third, select the best option given your assessment of the options that may not be in the other agent's repertoire (i.e. select a hyperstrategy). Not that the assessments in the second stage although probabilistic, are not subject to the outguessing problem. This is because the assessments determine whether or not another agent is capable of generating and option (for either side), and not how that agent will process the option once it is generated. An adversary cannot outguess a missing option.

Although this is simplified, the above discussion suggests the following steps for automated adversarial planning.

1. Generate/examine a proposed sequence of actions.

2. Identify a possible adversary action (or event) that can defeat the plan.

3. Determine what knowledge/data the adversary must have in order to determine the action identified in step 2.

4. Assess the probability that the adversary has the knowledge/data identified in 3. Each potential element of missing knowledge defines a subgame and the assessed probability is the probability of that subgame.

5. If the probability value determined in step 4 is too great, modify the plan. This can be done by (a) generating a contingency branch, (b) back tracking and selecting alternative actions, (c) inserting actions that will decrease the probability of failure (e.g., add in a deception tactic, etc.).

6. If the probability of success is less than a desired threshold, go to 1.

Steps 1,2, and 5 are well within the scope of existing AI planning techniques. In particular, steps 2 and 5 can be achieved by recursively calling the planner with the goal of defeating/repairing individual aspects of a proposed plan. In part, Step 3 is automatic. Whatever knowledge/data was used in Step 2 can be used as a basis for Step 3 processing.

Step 4, probability estimation, is by far the most problematic. However, here the hypergame representation allows a conservative strategy. The worst case situation is one where the adversary is aware of the complete game. To the extent that there is higher order uncertainty regarding the probability that the adversary is unaware of certain knowledge/data elements, the probability of the corresponding subgame can be decreased and the probability of the complete game increased.

4.0 SUMMARY

In this paper we have sketched some of our ongoing research in automated adversarial planning. In particular, we have shown how game theory techniques can be combined with knowledge-based planning procedures to reason about an adversary's beliefs and the extent to which a competitive agent is capable of defeating a plan. The main results to keep in mind are: (a) the hypergame representation provides a convenient mechanism for representing and reasoning about knowledge/data not available to a competitive agent and (b) automated implementation of this form of reasoning is conceptually straightforward.

REFERENCES

Bacchus, F. (1989) "LP: A Logic of Statistical Inference," Proceedings of the Fifth Annual Uncertainty in AI Workshop, University of Windsor.

Bennett, P.G.,and Dando, M.R., Complex Strategic Analysis: A Hypergame Study of the Fall of France, *Journal of the Operational Research Society Vol. 30* pp. 23 to 32, 1979.

Bennett, P.G., and Huxham,C.S., Hypergames and what they do: a 'soft O.R.' approach, *J. Opl. Res. Soc. Vol. 33*, pp. 41 to 50, 1982.

deKleer, J. (1986a) "An assumption-based Truth Maintenance System," Artificial Intelligence, 29, 241-288

deKleer, J. (1986b) "Extending the ATMS," Artificial Intelligence, 29, 289-318.

Dupuy, T.N., Understanding War, Paragon House, New York, 1987, pp.91-100.

Fagin, R. and Halpern, J. (1988) "Belief Awareness and Limited Reasoning," Artificial Intelligence, 34, 480-490.

Georgeff, M. "Planning," Annual Review of Computer Science, Volume 2 1987, Palo Alto, Annual Reviews, Inc.

Halpern, J. (ed.) (1986) Theoretical Aspects of Reasoning about Knowledge, Los Altos, Morgan and Kaufmann.

Halpern, J. (1989) "The Relationship Between Knowledge, Belief and Certainty," Proceedings of the Fifth Annual Uncertainty Workshop, University of Windsor.

Jackson, P., Reichgelt, H. and van Harmelan, F. (1989) Logic-Based Knowledge Representation, MIT Press, Cambridge, MA.

Jones, A. (1980) Game Theory: Mathematical Models of Conflict, Chichester, Ellis Horwood.

Laskey, K.B. and Lehner, P.E. (1989) "Assumptions, beliefs, and probabilities," Artificial Intelligence, 41, 65-77.

Laskey, K.B. and Lehner, P.E. (1990) "Belief Maintenance: An Integrated Approach to Uncertainty Management," in Readings in Uncertainty, Shafer, G. and Pearl, J. (eds.), Morgan Kaufmann Publishers, Inc., San Mateo, CA.

Lehner, P. (1983) "Strategic Planning in Go," in Computer Game Playing: Theory and Practice, ed. M. Bramer, Chichester, Ellis Horwood.

Lehner, P.E. (1990a) "Robust Inference Policies: Preliminary Report," submitted Sixth Conference of Uncertainty in AI.

Lehner, P.E. (1990b) "Inference Policies," in Uncertainty in Artificial Intelligence: Volume 5, Henrion, M. Schachter, R. Kanal, L.N., and Lemmer, J. (eds.), Elsevier, North-Holland, to appear.

Lehner, P.E. (1990c) "Automated Planning in Systems and Organizational Information Processing, " in Concise Encyclopedia of Information Processing in Organizations and Systems, A.P. Sage (ed.), Pergammon Press.

Lehner, P.E. (1990d) "Probabilities and Reasoning about Possibilities," International Journal of Approximate Reasoning, to appear.

Lehner, P.E. (1990e) "Adversarial Planning Search Procedures with Provable Properties," in New Directions in Command and Control Systems Engineering, S. Andriole (ed.), AFCEA International Press, Fairfax, VA.

Lehner, P.E. and Tosten, R. (1990) "An Autoepistemic Logic for Multiagent Reasoning," in preparation.

Lehner, P.E. and Ulvila, J.W., (1990) "A Note on the Application of Classical Statistics to Evaluating the Knowledge Base of an Expert System," submitted IEEE Transactions on Systems, Man, Cybernetics.

Pearl, J. (1990) "Which is more believable, the probably provable or the provably probable," Technical Note, January 1990.

Reitman, W. and Wilcox, B., "Modeling Tactical Analysis and Problem Solving in Go," in Proceedings of the Tenth Annual Conference on Modeling and Simulation, 1979, 2133-2144.

Shoham, Y. (1988) "Chronological Ignorance: Experiments in Nonmonotonic Temporal Reasoning," Artificial Intelligence, 36, 379-331.

Tosten, R. (1990) A Logic of Belief and Time for Multi-agent Time Dependent Belief and Reasoning, Doctoral Thesis, George Mason University, in preparation.

Vane, R. (1990) Doctoral Thesis Proposal, George Mason University, in preparation.

Vardi, M. (ed.) (1988) Proceedings of the Second Conference on Theoretical Aspects of Reasoning About Knowledge, Los Altos, Morgan Kaufmann.

Wilkins, D. (1980) "Using Patterns and Plans in Chess," Artificial Intelligence, 14, 165-203.

Young, P.R. and Lehner, P.E., (1986) "Applications of a Theory of Automated Adversarial Planning in Command and Control," IEEE Transactions on Systems Man, and Cybernetics, SMC-16, 806-812.

Nonlinear Planning with Parallel Resource Allocation

Manuela M. Veloso **M. Alicia Pérez** **Jaime G. Carbonell**

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Most nonlinear problem solvers use a least-commitment search strategy, reasoning about partially ordered plans. Although partial orders are useful for exploiting parallelism in execution, least-commitment is NP-hard for complex domain descriptions with conditional effects. Instead, a casual-commitment strategy is developed, as a natural framework to reason and learn about control decisions in planning. This paper describes (i) how NOLIMIT reasons about totally ordered plans using a casual-commitment strategy, (ii) how it generates a partially ordered solution from a totally ordered one by analyzing the dependencies among the plan steps, and (iii) finally how resources are allocated by exploiting the parallelism embedded in the partial order. We illustrate our claims with the implemented algorithms and several examples. This work has been done in the context of the PRODIGY architecture that incorporates NOLIMIT, a nonlinear problem solver.

1 Introduction

Nonlinear problem solving is desired when there are strong interactions among simultaneous goals and subgoals in the problem space. NOLIMIT, the nonlinear problem solver of the PRODIGY architecture [Carbonell *et al.*, 1990, Veloso, 1989], develops a method to solve problems nonlinearly that explores different alternatives at the operator and at the goal ordering levels. Commitments are made during the search process, in contrast to a least-commitment strategy [Sacerdoti, 1975, Tate, 1977, Wilkins, 1989], where decisions are deferred until all possible interactions are recognized. With the casual-commitment approach [Minton *et al.*, 1989], background knowledge, whether hand-coded expertise, learned control rules, or heuristic evaluation functions, guides the efficient exploration of the most promising parts of the search space. Provably incorrect alternatives are eliminated and heuristically preferred ones are explored first. Casual commitment is crucial because it provides a framework in which it is natural to reason and *learn* about the control decisions of the problem solver.

The immediate output of a problem solver that searches using a casual-commitment strategy is a totally ordered plan. It is advantageous to know the solution in terms of the least constrained partial ordering of its steps, which NOLIMIT generates by analyzing the dependencies among the different operators. The algorithm implemented constructs a directed acyclic graph that relates preconditions and effects of operators and then translates this graph into a partial order.

The independent actions shown in the partially ordered graph may not directly correspond to parallel executable actions due to resource contention. We show how a resource allocation module further analyzes the partial order and generates the final parallel plan.

This paper is organized in five sections. Section 2 briefly presents the casual-commitment search algorithm discussing its motivation and claims. In section 3, we introduce the algorithm that generates the partially ordered plan from the totally ordered one. In section 4 we describe the method to allocate resources, by analyzing the parallelism of the partially ordered solution. Finally, in section 5, we draw conclusions on this work. We illustrate our concepts, claims, and algorithms with several examples throughout the paper.

2 Nonlinear Problem Solving using Casual Commitment

NOLIMIT reasons about *totally* ordered plans that are *nonlinear*, i.e., the plans cannot be decomposed into a sequence of complete subplans for the conjunctive goal set. All decision points (operator selections, goal orderings, backtracking points, etc.) are open to introspection and reconsideration. In the presence of background knowledge – heuristic or definitive – only the most promising parts of the search space are explored to produce a solution plan efficiently [Veloso, 1989, Veloso *et al.*, 1990 forthcoming]. The skeleton of NOLIMIT's search algorithm, shown in Table 1, describes the basic cycle of the nonlinear planner.

In step 1 of the algorithm, the planner checks whether the goal is true in the current state. If so, the planner has found a solution to the problem. In step 2, it computes both the *set of pending goals* and the *set of applicable operators*. A goal is *pending*, if it is a precondition of a *chosen* operator that is not true in the state. An operator is *applicable*, if all its preconditions are true in the state. In step 3, the planner selects a goal to work on or an operator to apply. If a goal is chosen, the problem solver *expands* the goal in step 4, by generating and selecting a relevant *instantiated operator*. If an applicable operator is selected, then, in step 5, it is applied, i.e. executed in the *internal* current state to produce a new state.

1. Check if the goal statement is true in the current state, or there is a reason to suspend the current search path.
   If yes, then either return the final plan or backtrack.
2. Compute the *set* of *pending goals* $\mathcal{G}$, and the set of possible *applicable operators* $\mathcal{A}$.
3. Choose a goal G from $\mathcal{G}$ or select an operator A from $\mathcal{A}$ that is directly applicable.
4. If G has been chosen, then
   - *expand goal* G, i.e., get the set $\mathcal{O}$ of *relevant instantiated operators* for the goal G,
   - choose an operator O from $\mathcal{O}$,
   - go to step 1.
5. If an operator A has been selected as directly applicable, then
   - *apply* A,
   - go to step 1.

Table 1: A Skeleton of NOLIMIT's Search Algorithm.

PRODIGY provides a rich action representation language coupled with an expressive control language. Preconditions in the operators can contain conjunctions, disjunctions, negations, and both existential and universal quantifiers with typed variables. Effects in the operators can contain conditional effects, which depend on the state in which the operator is applied. The control language allows the problem solver to represent and learn control information about the various problem solving decisions, such as selecting which goal/subgoal to address next, which operator to apply, what bindings to select for the operator or where to backtrack in case of failure. Different disciplines for controlling decisions can be incorporated [Drummond and Currie, 1989, Anderson and Farley, 1990]. In PRODIGY, there is a clear division between the declarative domain knowledge (operators and inference rules) and the more procedural control knowledge. This simplifies both the initial specification of a domain and the incremental learning of the control knowledge [Minton, 1988, Veloso and Carbonell, 1990].

Previous work in the linear planner of PRODIGY used explanation-based learning techniques [Minton, 1988] to extract from a problem solving trace the explanation chain responsible for a success or failure and compile search control rules. We are now extending this work to NOLIMIT, as well as developing a derivational-analogy approach to acquire control knowledge [Carbonell, 1986, Veloso and Carbonell, 1990]. The machine learning and knowledge acquisition work supports NOLIMIT's casual-commitment method, as it assumes there is *intelligent* control knowledge, exterior to its search cycle, that it can rely upon to make decisions.

2.1 Example

Consider a generic transportation domain with three simple operators that load, unload, or move a carrier, as shown in Figure 1 (variables in the operators are shown in bold face).

Suppose that the operator MOVE a carrier has *constant* locations $locA$ and $locB$. This transforms the current general domain into a *one-way* carrier domain. The problem we want to solve consists in moving two given objects $obj1$ and $obj2$ from the location $locA$ to the location $locB$ using a ROCKET as the carrier, for example. Without any control knowledge the problem solver searches for the goal ordering that enables the problem to be solved. Accomplishing

```
(LOAD                  (UNLOAD                (MOVE
 (preconds              (preconds               (preconds
  (and                   (and                    (at carrier locA))
   (at obj loc)           (inside obj carrier)  (effects
   (at carrier loc)))     (at carrier loc)))     (add (at carrier locB))
 (effects               (effects                (del (at carrier locA))))
  (add (inside obj carr)) (add (at obj loc))
  (del (at obj loc))))    (del (inside obj carrier))))
```

Figure 1: A Transportation Domain.

either goal individually, as a linear planner would do, inhibits the accomplishment of the other goal, as a precondition of the operator LOAD cannot be achieved: the ROCKET cannot be moved back to the object's initial position. So interleaving of goals and subgoals at different levels of the search is needed to find a solution. NOLIMIT solves this problem, where linear planners fail (but where, of course, other least-commitment planners also succeed), because it switches attention to the conjunctive goal *(at obj2 locB)* before completing the first conjunct *(at obj1 locB)*. This is shown in Figure 2 by noting that, after the plan step 1 where the operator (LOAD ROCKET obj1 locA) is applied, NOLIMIT changes its focus of attention to the other top-level goal and applies the operator (LOAD ROCKET obj2 locA). NOLIMIT returns the totally ordered solution (LOAD ROCKET obj1 locA), (LOAD ROCKET obj2 locA), (MOVE ROCKET), (UNLOAD ROCKET obj1 locB), (UNLOAD ROCKET obj2 locB).

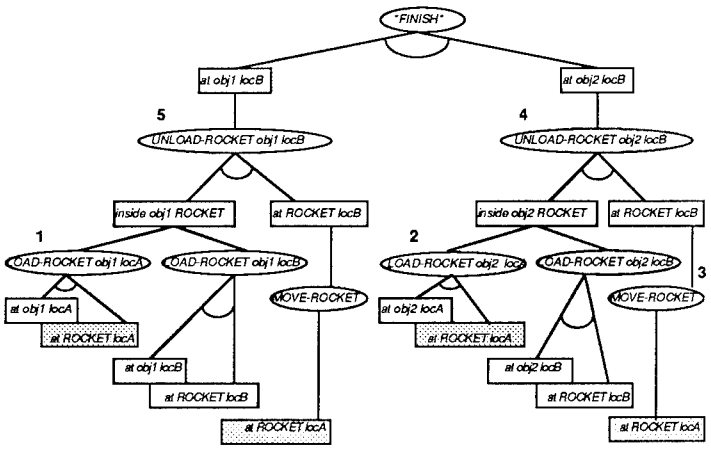


Figure 2: The Complete Conceptual Tree for a Successful Solution Path. The numbers at the nodes show the execution order of the plan steps.

Clearly, NOLIMIT solves much more complex and general versions of this problem. The present minimal form was used to illustrate the casual-commitment strategy in nonlinear planning, allowing full interleaving of goals and subgoals. We present below examples with a complex logistics domain.

3 Total and Partial Orders

A partially ordered graph is a convenient way to represent the ordering constraints that exist among the steps of the plan. Consider the partial order as a directed graph (V, E), where

V, the set of vertices, is the set of steps (instantiated operators) of the plan, and E is the set of edges (ordering constraints) in the partial order. Let $V = \{op_0, op_2, \ldots, op_{n+1}\}$. We represent the graph as a square matrix P, where $P[i,j] = 1$, if there is an edge from op_i to op_j. There is an edge from op_i to op_j, if op_i must *precede* op_j, i.e. $op_i \prec op_j$. The inverse of this statement does not necessarily hold, i.e. there may be the case where $op_i \prec op_j$ and there is not an edge from op_i to op_j. The relation $\prec$ is the *transitive closure* of the relation represented in the graph for the partial order. Without loss of generality consider operators op_0 and op_{n+1} of any plan to be the *additional* operators named *start* and *finish*, represented in the Figures below as s and f.

3.1 Transforming a Total Order into a Partial Order

A plan step op_i necessarily precedes another plan step op_j if and only if op_i adds a precondition of op_j, or op_j deletes a precondition of op_i. For each problem, the start operator s adds all the literals in the initial state. The preconditions of the finish operator f are set to the user-given goal statement. Let the totally ordered plan $\mathcal{T}$ be the sequence $op_1, \ldots, op_n$ returned by NOLIMIT as the solution to a problem. In Table 2, we show the algorithm to generate the partially ordered plan from this totally ordered one, $\mathcal{T}$.

Input: A totally ordered plan $\mathcal{T} = op_1, op_2, \ldots, op_n$, and the start operator s with preconditions set to the initial state.
Output: A partially ordered plan shown as a directed graph $\mathcal{P}$.

procedure Build_Partial_Order($\mathcal{T}$, s):
1. **for** $i \leftarrow$ **n** down-to 1 **do**
2. **for** each precond in Preconditions_of(op_i) **do**
3. supporting_operator $\leftarrow$ $\leftarrow$ Last_Op_Adding_Precond(precond,i)
4. Add_Directed_Edge(supporting_operator,op_i,$\mathcal{P}$)
5. **for** each del in Delete_Effects(op_i) **do**
6. supported_operators $\leftarrow$ $\leftarrow$ All_Ops_Needing_Effect(del,i)
7. **for** each supported_operator **do**
8. Add_Directed_Edge(supported_operator,op_i,$\mathcal{P}$)
9. $\mathcal{P} \leftarrow$ Remove_Transitive_Edges($\mathcal{P}$)

Table 2: Building a Partial Order from a Total Order

Step 1 loops through the plan steps in the *reverse* of the execution order. Lines 2-4 loop through each of the preconditions of the operator, i.e. plan step. The procedure Last_Op_Adding_Precond (not shown) searches from the operator op_i back to, at most the operator s, for the first operator (supporting_operator) that has the effect of adding the precondition in consideration. Note that one such operator must be found as the given $\mathcal{T}$ is a solution to the problem (in particular the initial state is added by the operator s). All the supporting_operators of an operator op_i must precede it. The algorithm sets therefore a directed edge from each of the former into the latter. Lines 5-8 similarly loop through each of the delete effects of the operator. The procedure All_Ops_Needing_Effect (not shown) searches for all the earlier operators that need, i.e. have as a precondition, each delete effect of the operator. We call such operators, supported_operators. Lines 7-8 capture the precedence relationships by adding directed edges from each supported_operator to the operator that deletes some of their preconditions. Finally, line 9 removes all the transitive edges of the resulting graph to produce the partial order. Every directed edge e connecting operator op_i to op_j is removed, if there is another path that connects the two vertices. The procedure Remove_Transitive_Edges tentatively removes e from the graph and then checks to see whether vertex op_j is reachable from op_i. If this is the case, then e is removed definitively, otherwise e is set back in the graph.

If n is the number of operators in the plan, p is the average number of preconditions of an operator, and d is the average number of delete effects of an operator, then steps 1-8 of the algorithm Build_Partial_Order run in $O((p+d)n^2)$. Note that the algorithm takes advantage of the given total ordering of the plan, by visiting, at each step, only earlier plan steps. The final procedure Remove_Transitive_Edges runs in $O(e)$, for a resulting graph with e edges [Aho *et al.*, 1974]. Empirical experience with test problems shows that the algorithm Build_Partial_Order runs in meaningless time compared to the search time to generate the input totally ordered plan.

We now illustrate the algorithm in the simple *one-way* rocket problem introduced in the previous section. NOLIMIT returned the totally ordered plan $\mathcal{T}$ = (LOAD ROCKET obj1 locA), (LOAD ROCKET obj2 locA), (MOVE ROCKET), (UNLOAD ROCKET obj1 locB), (UNLOAD ROCKET obj2 locB). Let op_i be the ith operator in $\mathcal{T}$. In Figure 3 we show the partial order generated by the algorithm, before removing the transitive edges. As previously seen, the goal of the problem we solved is the conjunction *(and (at obj1 locB) (at obj2 locB))*. These two predicates are added by the UNLOAD steps, namely by op_4 and op_5 respectively. The edges labelled "g" show the precedence requirement between op_4 and op_5, and the finish operator f. The numbers at the other edges in Figure 3 represent the order by which the algorithm introduces them into the graph. As an example, while processing op_5 (UNLOAD ROCKET obj2 locB), it sets the edges 1 and 2, as the preconditions of op_5, namely *(inside obj1 ROCKET)* and *(at ROCKET locB)* (see Figure 1), are added by op_2 and op_3 respectively. When processing op_3 (MOVE ROCKET), edge 5 is set because op_3's precondition *(at ROCKET locA)* is in the initial state. The edges 6 and 7 are further set, because op_3 deletes *(at ROCKET locA)* that is needed (as a precondition) by the earlier steps op_1 and op_2. Removing the transitive edges, namely edges 1, 3, and 5, in this graph results in the final partial order.

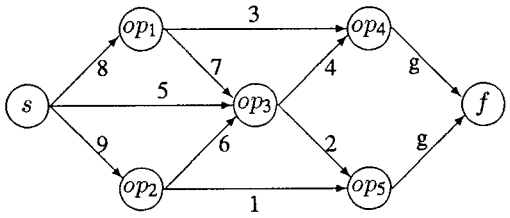


Figure 3: Partial Order with Transitive Edges.

4 Exploiting Parallelism in the Plan Steps

When there are multiple execution-time agents in a domain, they must be able to organize their activities so that they can cooperate with one another (e.g. to push a very heavy block)

and avoid conflicts (e.g. not to tyr to use the same tool at the same time).

Our approach for doing multiagent planning is a centralized one [Georgeff, 1983, Lansky and Fogelsong, 1987]. An initial planning phase produces a plan as parallel as possible by reasoning about a presumably infinite number of resources. Real available resources are then assigned to obtain the final parallel plan [Wilkins, 1989]. A problem is first solved creating generic instances of the resources. In this context, "resources" refer to agents, such as robots, or trucks or airplanes in a logistics transportation domain, or machines in a process planning domain. Control rules assign different resources to different unrelated goals to obtain a plan as parallel as possible. In some cases the same resource can be used to solve different unrelated goals. For example, it is better to load different objects in the same truck if they have the same destination), if minimization of resources usage is preferred by the control knowledge.

Let $\mathcal{T}$ be the resulting plan and s the start operator. Table 3 outlines the algorithm for resource allocation.

1. Generate the partial order graph $\mathcal{P}$ using the algorithm in Table 2 with inputs $\mathcal{T}$ and s.
2. Insert parallel split and join nodes in the partial order graph $\mathcal{P}$ obtaining a graph $\mathcal{P}'$.
3. Recursively analyze in $\mathcal{P}'$ the parallel branches inside a split-join pair. If some of the parallel branches are in conflict insert sequential split and join nodes. If all the parallel branches are in conflict, transform the parallel split-join pair into a sequential one. Let $\mathcal{P}''$ be the resulting graph.
4. From $\mathcal{P}''$, assign real resources to the generic instances.
5. Assign plans to the individual resources and monitor their execution to avoid conflicts.

Table 3: Algorithm for Resource Allocation.

In step 1 the algorithm section 3.1 generates the partial order graph from $\mathcal{T}$. Step 2 extends this graph with nodes that are not associated with steps in the plan. They only serve as guidelines to determine which actions can be executed in parallel. If a node op_i has several successors op_{i_1}, ..., op_{i_n}, a *parallel split node* is inserted having op_i as a predecessor and op_{i_1}, ..., op_{i_n} as successors. The edges between op_i and op_{i_1}, ..., op_{i_n} are removed. Similarly, if a node op_j has several predecessors op_{j_1}, ...,op_{j_n}, a *parallel join node* is inserted having op_j as only successor and op_{j_1}, ..., op_{j_n} as predecessors. The edges between op_{j_1}, ..., op_{j_n} and op_j are removed.

Step 3 analyzes the parallel branches. It may be necessary to add *sequential split* and *join nodes* to the graph, or replace some of the parallel ones. The branches inside a sequential split-join pair must be executed sequentially although any order is allowed.

A class of objects C can be declared as a possible reason for conflict. Two *actions* are in conflict if they use the same instance of C, and hence they are not allowed to occur simultaneously. A conflict between two *branches* is detected when there is not a pair of actions, one of each branch, that can be executed at the same time. If *all* the actions of the two branches are in conflict, they are enclosed in a sequential split-join pair. If only *some* of them are, the parallel split-join remains. Committing to executing the branches in sequence would constrain the parallelism in the plan, as the actions not in conflict could still be done simultaneously. As we describe below, an execution monitor is responsible for avoiding that the conflicting actions are performed simultaneously. This analysis is done recursively to deal with nested split-join pairs.

Step 4 assigns real resources to the generic instances, by recursively analyzing the branches inside a split-join pair. If enough resources are available, the algorithm assigns different ones to each branch. Otherwise the available resources are shared by several branches. These branches are put inside a sequential split-join pair so the monitor can execute them without conflicts. The planner may have to be called again to obtain the actions that situate the real resource in the same initial state as the generic one it replaces.

From the global parallel plan obtained so far, step 5 generate plans for each of the agents or resources. A monitor module is responsible for synchronizing the execution of the different plans (for example, in the case when two or more agents are necessary to perform an action). It uses the sequential split and join nodes to deal with conflicts or resource sharing among different branches. Those conflicts can be considered as critical regions. Standard operating systems methods can be used to enforce synchronization in the plans so the conflicting critical regions are not entered at the same time [Georgeff, 1983].

4.1 Example in the Extended-STRIPS Domain

To illustrate this we will consider a simple example where two robots, R1 and R2, have to move two blocks, a heavy one H, and a light one L. The two robots have to cooperate to push H. The domain is an extension of the STRIPS domain; the operators include going to locations, going through doors and pushing objects to locations. There are also "team" operators that require the cooperation of two robots to perform an action (e.g. t-push-to-location). Only one robot can go through a door at a time, therefore doors are considered reasons for potential conflicts. Figures 4 (a) and (b) show the initial state and goal statement, and (c) shows the initial state using generic robots GR1, GR2 and GR3.

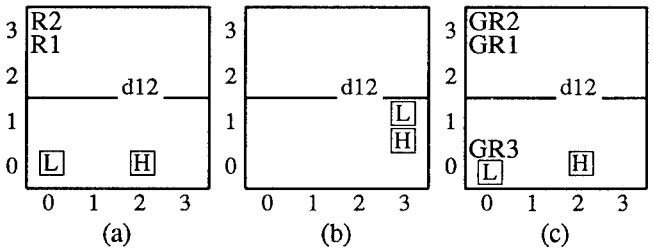


Figure 4: Initial State, Goal Statement, and Initial State with Generic Resources for the Example Problem. Coordinates represent the locations within the rooms.

The problem is first solved with generic robots. Their initial situation was decided based on domain dependent heuristics such as the initial situation of the available robots and of the objects that have to be pushed. The solution is:

1 (goto-loc GR1 3 0 2 2)
2 (go-thru-door GR1 door12 2 2 2 1)
3 (goto-loc GR1 2 1 2 0)
4 (goto-loc GR2 3 0 2 2)
5 (go-thru-door GR2 door12 2 2 2 1)
6 (goto-loc GR2 2 1 2 0)
7 (t-push-to-loc GR1 GR2 heavy-block 2 0 3 1)
8 (push-to-loc GR3 light-block 0 0 3 1)

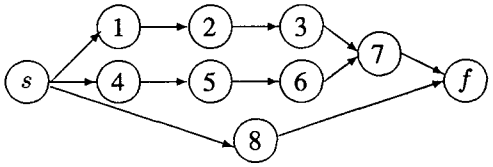


Figure 5: Partial Order Graph for the Example Problem.

Figure 5 shows the partial order generated by the algorithm in section 3. The only conflict is between 2 and 5 when GR1 and GR2 try to go through the door at the same time. As the other actions (1, 3, 4, 6) in the parallel branches do not conflict, these branches are not put inside a sequential split join pair. The resource assignment step assigns R1 to GR1, and R2 to both GR2 and GR3. After this step the graph looks like in Figure 6.

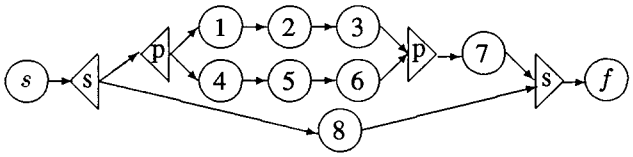


Figure 6: Graph after Assigning Resources.

Now the task of the monitor is to control the plan execution avoiding the conflict at the door and deciding which of the two branches will be executed first. The planner is called to plan the actions of R2 to join the end of branch 1-2-3 with the beginning of branch 4-5-6. A resulting parallel plan is the one shown below, where branch 1-2-3, and branch 4-5-6 are monitored to be executed in parallel avoiding the conflict between steps 2 and 5. Step 7' is added to the plan.

monitored-parallel-split
1 (goto-loc R1 3 0 2 2)
2 (go-thru-door R1 door12 2 2 2 1)
3 (goto-loc R1 2 1 2 0)

4 (goto-loc R2 3 0 2 2)
5 (go-thru-door R2 door12 2 2 2 1)
6 (goto-loc R2 2 1 2 0)
monitored-parallel-join

7 (t-push-to-loc R1 R2 heavy-block 2 0 3 1)
7' (goto-loc R2 3 1 0 0)
9 (push-to-loc R2 light-block 0 0 3 1)

4.2 Example in the Logistics Domain

We are currently implementing a complex logistics planning domain. In this domain, packages are to be moved among different cities. Packages are carried within the same city in trucks and across cities in airplanes. Trucks and airplanes may have limited capacity. At each city there are several locations, e.g. post offices (po) and airports (ap). This domain (without introducing the capacity of carriers) is an extension of the generic transportation domain (see Figure 1). Consider carriers of type TRUCK and AIRPLANE. The logistics domain consists of the operators LOAD TRUCK (LT), LOAD AIRPLANE (LA), UNLOAD TRUCK (UT), UNLOAD AIRPLANE (UA), DRIVE TRUCK (DT), FLY AIRPLANE (FA). Consider the problem shown in Figure 7 where bo, pg and sf stand for Boston, Pittsburgh and San Francisco respectively. There are three packages ($p1, p2, p3$), two airplanes ($a1, a2$), and four trucks ($tbo1, tbo2, tsf, tpg$). NOLIMIT returns the plan in Figure 8, and Figure 9 shows the partial order generated by the algorithm in Table 2.

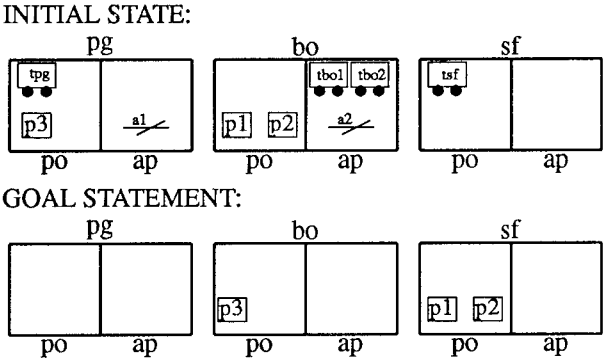


Figure 7: A Problem in the Logistics Domain.

1.(LT p3 tpg pg-po)
2.(DT tsf sf-po sf-ap)
3.(DT tpg pg-po pg-ap)
4.(UT p3 tpg pg-ap)
5.(LA p3 a1 pg-ap)
6.(FA a1 pg-ap bo-ap)
7.(UA p3 a1 bo-ap)
8.(LT p3 tbo1 bo-ap)
9.(DT tbo1 bo-ap bo-po)
10.(UT p3 tbo1 bo-po)
11.(DT tbo2 bo-ap bo-po)
12.(LT p2 tbo2 bo-po)
13.(LT p1 tbo2 bo-po)
14.(DT tbo2 bo-po bo-ap)
15.(UT p2 tbo2 bo-ap)
16.(UT p1 tbo2 bo-ap)
17.(LA p2 a2 bo-ap)
18.(LA p1 a2 bo-ap)
19.(FA a2 bo-ap sf-ap)
20.(UA p2 a2 sf-ap)
21.(UA p1 a2 sf-ap)
22.(LT p2 tsf sf-ap)
23.(LT p1 tsf sf-ap)
24.(DT tsf sf-ap sf-po)
25.(UT p2 tsf sf-po)
26.(UT p1 tsf sf-po)

Figure 8: Totally Ordered Plan - Logistics Domain.

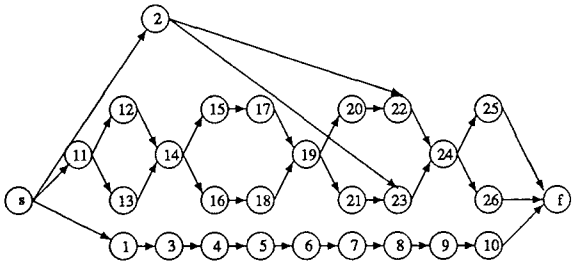


Figure 9: Partially Ordered Plan - Logistics Domain.

Suppose now that when executing this plan, there is available only one airplane (a) and only one truck in Boston (tbo). The resource allocation algorithm assigns a to both $a1$ and $a2$, and tbo to both $tbo1$ and $tbo2$ after generating the parallel serial graph. Figure 10 shows a possible solution for the plans of a and tbo. Using the information on the graph built by the algorithm, the monitor synchronizes the execution of the plans for the different agents, without violating the constraints discovered by the algorithm.

We are refining the monitor synchronization mechanism to

Plan for airplane a:	*Plan for truck tbo:*
(LA p3 a pg-ap)	(DT tbo bo-ap bo-po)
(FA a pg-ap bo-ap)	(LT p2 tbo bo-po)
(UA p3 a bo-ap)	(LT p1 tbo bo-po)
(LA p2 a bo-ap)	(DT tbo bo-po bo-ap)
(LA p1 a bo-ap)	(UT p2 tbo bo-ap)
(FA a bo-ap sf-ap)	(UT p1 tbo bo-ap)
(UA p2 a sf-ap)	(LT p3 tbo bo-ap)
(UA p1 a sf-ap)	(DT tbo bo-ap bo-po)
	(UT p3 tbo bo-po)

Figure 10: Plans for Each Resource.

deal with more complex conflict constraints, by using domain dependent heuristics.

5 Conclusion

In this paper, we first discuss the use of a casual-commitment strategy to generate plans for nonlinear problems. This strategy provides a natural framework to learn and reason about control decisions during the planning process. The method becomes increasingly efficient as the planner learns control knowledge from experience. Committing while searching generates a totally ordered solution. As it is advantageous to know the least constrained partial ordering of the plan steps, we then discuss how we efficiently generate a partial order from the total order returned by the casual-committing problem solver. Finally, we show a resource allocation strategy that reasons about the partially ordered plan to convert it into a parallel executable graph.

This work has been done in the context of the PRODIGY architecture that is designed as a testbed for machine learning research. Casual commitment relies upon learned control knowledge to efficiently make decisions. The resource allocation module is an ongoing research effort to address multiagent (or multi-resource) planning and execution.

Acknowledgements

Our special thanks to Daniel Borrajo for a major part of NOLIMIT's implementation. Without him, it would have been very difficult to include many powerful features in NOLIMIT. We acknowledge Craig Knoblock and Yolanda Gil for providing useful comments on a draft. The authors thank the whole PRODIGY research group for helpful discussions.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD) and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, OH 45433-6543 under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the U.S. Government. The second author was supported by a Fellowship from the Ministerio de Educación y Ciencia of Spain.

References

[Aho *et al.*, 1974] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[Anderson and Farley, 1990] J. S. Anderson and A. M. Farley. Partial commitment in plan composition. Technical Report TR-90-11, Computer Science Department, University of Oregon, 1990.

[Carbonell *et al.*, 1990] J. G. Carbonell, C. A. Knoblock, and S. Minton. Prodigy: An integrated architecture for planning and learning. In Kurt VanLehn, editor, *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, 1990.

[Carbonell, 1986] J. G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning, An Artificial Intelligence Approach, Volume II*. Morgan Kaufman, 1986.

[Drummond and Currie, 1989] M. Drummond and K. Currie. Goal ordering in partially ordered plans. In *Proceedings of IJCAI-89*, pages 960–965, 1989.

[Georgeff, 1983] M. Georgeff. Communication and interaction in multi-agent planning. In *Proceedings of the National Conference of the American Association for Artificial Intelligence*, pages 125–129, Washington, DC, August 1983.

[Lansky and Fogelsong, 1987] A. L. Lansky and D. S. Fogelsong. Localized representation and planning methods for parallel domains. In *Proceedings of the National Conference of the American Association for Artificial Intelligence*, pages 240–245, Seattle, Washington, August 1987.

[Minton *et al.*, 1989] S. Minton, C. A. Knoblock, D. R. Kuokka, Y. Gil, R. L. Joseph, and J. G. Carbonell. PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, 1989.

[Minton, 1988] S. Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. PhD thesis, Computer Science Department, Carnegie Mellon University, 1988.

[Sacerdoti, 1975] E. D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of IJCAI-75*, pages 206–213, 1975.

[Tate, 1977] A. Tate. Generating project networks. In *Proceedings of IJCAI-77*, pages 888–893, 1977.

[Veloso and Carbonell, 1990] M. M. Veloso and J. G. Carbonell. Integrating analogy into a general problem-solving architecture. In Maria Zemankova and Zbigniew Ras, editors, *Intelligent Systems*, 1990.

[Veloso *et al.*, 1990 forthcoming] M. M. Veloso, D. Borrajo, and M. A. Perez. NOLIMIT - The nonlinear problem solver for PRODIGY: User's and programmer's manual. Technical report, School of Computer Science, Carnegie Mellon University, 1990, forthcoming.

[Veloso, 1989] M. M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, 1989.

[Wilkins, 1989] D. E. Wilkins. Can AI planners solve practical problems? Technical Note 468R, SRI International, 1989.

SCHEDULING

Applying a Heuristic Repair Method to the HST Scheduling Problem

Steven Minton and Andrew B. Philips
Sterling Federal Systems
AI Research Branch, Mail Stop: 244-17
NASA Ames Research Center
Moffett Field, CA 94035 U.S.A.

Abstract

This paper describes a heuristic search method that has been employed to solve the Hubble Space Telescope Scheduling problem. Given an initial schedule created by a greedy algorithm, the method operates by searching though the space of possible rearrangements of the initial schedule. The search is guided by an ordering heuristic, the *min-conflicts heuristic*, that attempts to minimize the number of constraint violations after each step.

1 Introduction

In a previous paper, Minton et al.[10] described a *local search* method for solving large-scale constraint satisfaction and scheduling problems. The method operates by generating an initial, suboptimal solution and then applying a local repair heuristic, which we refer to as the min-conflicts heuristic. Local search techniques have met with empirical success on many problems, including the traveling salesman and graph partitioning problems[5]. Such techniques also have a long tradition in AI, most notably in problem-solving systems that operate by debugging initial solutions [14; 15]. However, this approach is a relatively new approach for solving constraint-satisfaction problems, and offers some important advantages over traditional methods.

The local search method described here was distilled from an analysis of a surprisingly effective neural network developed by Johnston and Adorf[1; 7]. for scheduling the use of the Hubble Space Telescope. The method is very effective at solving the Hubble Space Telescope scheduling problem, and empirical studies have demonstrated that it also performs extremely well on some standard problems. For example, we have shown that instances of the n-queens problem with one million queens can be solved very rapidly. The method also has the virtue of being extremely simple. In this paper, we describe the basic method and its application to the Space Telescope Scheduling problem.

2 The Min-Conflicts Heuristic

A constraint-satisfaction problem consists of n variables, $X_1 \ldots X_n$, with domains $D_1 \ldots D_n$, and a set of constraints. We will assume for the moment that each constraint is a binary constraint, that is, each constraint $C_\alpha(X_j, X_k)$ is a subset of $D_j \times D_k$ specifying incompatible values for a pair of variables. In this paper we consider the task of finding a single solution to a problem, i.e., an assignment for each of the variables such that the constraints are satisfied.

Our method takes an initial, inconsistent assignment for the variables in a constraint satisfaction problem (CSP) and incrementally repairs constraint violations until a consistent assignment is achieved. The method is guided by a simple ordering heuristic for repairing constraint violations: select a variable that is currently participating in a constraint violation, and choose a new value that minimizes the number of outstanding constraint violations. This heuristic can be specified as follows:

> **Min-Conflicts heuristic:**
> *Given:* A set of variables, a set of binary constraints, and an assignment specifying a value for each variable. Two variables *conflict* if their values violate a constraint.
> *Procedure:* Select a variable that is in conflict, and assign it a value that minimizes the number of conflicts. (Break ties randomly.)

As an illustration of our approach, consider the n-queens problem, a standard benchmark for testing CSP algorithms. The n-queens problem requires placing n queens on an $n \times n$ chessboard so that no two queens share a row, column or diagonal. To solve the problem, we begin by randomly placing a queen on each row of the board. This gives us an initial assignment. Then we take a queen that is currently in conflict, and move it to the column (in the same row) that has the fewest conflicts (with ties broken randomly). This "repair process" is repeated until a solution is found, or a preset iteration bound is reached.

The method outlined above is a *hill-climbing* algorithm. Thus, it is entirely possible that a local maximum may be encountered, in which case the system will typically oscillate between a small number of states. However, as described in [10], the min-conflicts

heuristic can be used with a variety of search strategies, including best-first search, simulated-annealing and backtracking. For the scheduling applications described in this paper, the hill-climbing approach was employed, due to its effectiveness and simplicity.

In [10], we analyzed the min-conflicts approach in order to determine the types of problems for which the algorithm will work well. Not surprisingly, it appears that the algorithm is likely to be more effective if the preprocessing stage can generate an assignment that is close to a solution, i.e., an assignment in which relatively few repairs need to be made. In the n-queens problem, we can generate a good initial assignment in the following manner. In the preprocessing phase, when each queen is assigned to an inital row and column, we prefer columns where there are no conflicts, rather than choosing a random column. Using this technique, we reported that the "million queens" problem could be solved in less than two minutes on a SparcStation1 (with very high probability). This is orders of magnitude better than has be achieved with traditional backtracking CSP algorithms.

3 The HST Scheduling Problem

By almost any measure, the Hubble Space Telescope (HST) scheduling problem is a complex task [16; 13; 6]. Between ten thousand and thirty thousand astronomical observations per year must be scheduled, subject to a vast variety of constraints involving time-dependent orbital characteristics, power restrictions, priorities, movement of astronomical bodies, stray light sources, etc. Because the telescope is an extremely valuable resource with a limited lifetime, efficient scheduling is a critical concern. An initial scheduling system, SPSS, developed in FORTRAN using traditional programming methods, highlighted the difficulty of the problem; among other problems, it was estimated that the system would take several weeks to schedule one week of observations. A more successful constraint-based system, the SPIKE system, was then developed to augment the original system.

The input to SPIKE is a set of detailed specifications for exposures that are to be scheduled on the telescope. These exposures are submitted by astronomers whose proposals have been approved by a peer review process. An exposure specification includes a potentially large number of configuration parameters describing how the data is to be taken. Johnston [6] outlines the problem:

> There are a variety of properties and relationships among these exposures that may be specified by the proposer [astronomer]. Their relative order and time separation may be important. Some exposures are designed as calibrations or target acquisitions for others. Some must be executed at specific times, or at specific phases in the case of periodic phenomena. Some are especially sensitive to stray or scattered light. Exposure durations may vary depending on background light intensity. Some exposures must be executed without interruption while others can be broken up as needed. In some cases a specific orientation of an instrument aperture is required. Some exposures are conditional on the results of other exposures.
>
> In addition to proposer-specified constraints, there are a large number of other constraints that must be considered when scheduling HST operations. The range from "strict" constraints that cannot be violated under any circumstances, to "good operating practices" that represent scheduling goals. HST is not allowed to point closer than 50° to the sun and 15° to the bright moon. Slewing the telescope is relatively slow (90° in $\sim$ 15 minutes) so it is important to minimize the time spent in maneuvers. Many constraints are a direct result of HST's low orbital altitude (500 km) and consequent 95 minute orbital period. A typical target is occulted by the earth for $\sim$ 40 minutes of each orbit. Up to half the orbits in a day are contaminated for up to $\sim$ 20 minutes by HST's passage through the South Atlantic Anomaly, a high particle density region during which data cannot be collected. Scattered earthlight changes dramatically over the course of an orbit...

The scheduling team at the Space Telescope Science Institute made the problem considerably more tractable by breaking it into two parts: the long-term scheduling problem and the short-term scheduling problem. The long-term problem consists of taking approximately one year's worth of exposures, and dividing them up into "bins" or time segments of a few days length. The short-term problem consists of coming up with a very detailed schedule for a time segment, which can be translated into commands that the telescope can then directly execute. As it turns out, SPIKE handles only the long-term problem. The short-term problem has a quite different nature, because it involves both planning and scheduling. (We use the term planning to refer to the *generation* of a partially-ordered set of activities to achieve a set of goals, and the term scheduling to refer to the process of placing a set of activities on a time line.) The short-term problem requires planning because an exposure may require activities such as warming up or cooling down different instruments on the telescope, pointing maneuvers, communication of data, etc. Currently, the short-term problem is handled by the original SPSS system, however, Muscettola et al. [13] are developing AI planning techniques that will hopefully do a better job. Another possibility is the extension of the SPIKE system so that it can generate a schedule for significantly smaller time buckets. The research reported here may contribute to this goal, by improving the speed of the SPIKE system.

SPIKE operates by taking the exposure specifications prepared by astronomers and validating that

they are internally consistent. It then compiles the specifications into a set of constraints, represented as relative temporal relations and "suitability functions". The relative temporal relations specify the relative before/after ordering of tasks, and the maximal/minimal amount of time between tasks. Each suitability function is a function of time whose value represents the desirability of starting an activity at a specified time, as given by the constraint in question. For example, one suitability function may represent the constraint that the telescope should not point near the moon. Thus, the suitability of scheduling an exposure when the target is close to the moon will be low (perhaps zero). Suitability functions are represented internally as piecewise constant functions, enabling the product of multiple suitabilities to be calculated efficiently.

Because of the uncertainty in calculating certain constraints, and also because the grain-size of the time segments may be relatively large, suitability functions are often used to represent the statistical or aggregate desirability of scheduling an exposure during a certain time segment. For example, a particular orbital constraint might state that an exposure must be taken when the telescope is pointing more than 5° from the earth's limb and is in the earth's shadow. The resulting suitability function might indicate, for each time segment, the average amount of time these conditions are satisfied over that segment (which could encompass many orbits). In other words, it would be preferable to schedule the exposure in a time segment in which a relatively high number of such viewing opportunities occur.

Once SPIKE has compiled the astronomers proposals into a set of constraints, it must search for a good schedule. SPIKE employs a neural network to carry out this search, the Guarded Discrete Stochastic (GDS) network[1; 7]. The GDS network is a modified Hopfield network[3]. The most significant modification is that the main network is coupled asymmetrically to an auxiliary network of *guard neurons* which restricts the configurations that the network can assume. This modification enables the network to rapidly find a solution for many problems, even when the network is simulated on a serial machine. The disadvantage is that convergence to a stable configuration is no longer guaranteed, in which case the network can fall into a local minimum involving a group of unstable states among which it will oscillate. In practice, however, if the network fails to converge after some number of neuron state transitions, it is simply stopped and started over.

To illustrate the network architecture and updating scheme, let us consider how the network is used to solve the HST scheduling problem. Each task to be scheduled (an exposure or block of exposures) is represented by a separate set of neurons, one neuron for each possible time segment in the schedule. Each neuron is either "on" or "off"; if a neuron is "on" it means the task is currently scheduled for that time segment. Inhibitory (i.e., negatively weighted) connections between the neurons are used to indicate hard constraints between tasks, where the suitability of placing two tasks in a certain configuration is zero. To insure that each task is eventually assigned a time segment there is a guard neuron for each set of neurons representing a task; if no neuron in the set is on, the guard neuron will provide an excitatory input that is large enough to turn one on. (Due to the way the connection weights are set up, it is unlikely that the guard neuron will turn on more than one neuron.) The network is updated on each cycle by randomly selecting a set of neurons that represents a task, and flipping the state of the neuron in that set whose input is *most inconsistent* with its current output (if any). When all neurons' states are consistent with their input, a solution is achieved.

The network updating scheme roughly accomplishes the following: If the task is currently in conflict then it is removed from the schedule, and if the task is currently unscheduled then the network schedules it for the time segment that has the fewest constraint violations. Note that the network only represents hard constraints (i.e. it treats suitabilities as zero or one). Soft constraints (where the suitability is between zero and one) are only consulted when there are two or more "least conflicted" places to move a task.

As discussed in [10], the min-conflicts algorithm effectively mimics the behavior of the GDS network. In fact, the algorithm was developed from an analysis of the network's performance. (The two approaches can be parallelized in a similar manner, but currently both are run on serial machines.) In the HST application, the min-conflicts algorithm operates by constructing an initial schedule in a preprocessing phase, and iteratively repairs the schedule until a conflict-free schedule is found (or the process is terminated by a preset iteration bound). Because our analysis of the min-conflicts algorithm showed that a good initial assignment could greatly improve the solution time, we use a greedy algorithm to create an initial schedule, rather than randomly assigning tasks.[1] The greedy algorithm places each task on the schedule, at each point trying to minimize the number of conflicts.

One advantage in using the min-conflicts algorithm, as compared to the GDS network, is that much of the overhead of using the network can be eliminated (particularly the space overhead). The min-conflicts algorithm has been shown to be at least as effective as the GDS network on representative data sets provided by the Space Telescope Sciences Institute. Moreover, because the min-conflicts heuristic is so simple, the scheduling program could be quickly coded in C and is extremely efficient. (The scheduling program runs at least an order of magnitude faster than the network, although some of the improvement is due to factors such as programming language differences, which makes a precise comparison difficult.) While this may be regarded as just an implementation issue, we believe that the clear and simple formulation of the

[1] We discovered the importance of a good initial assignment by analyzing the min-conflicts algorithm, but it has also been shown to hold for the network as well.

method was a significant enabling factor. We are currently experimenting with a variety of different search strategies that can be combined with the min-conflicts heuristic. Although this study is not yet complete, we expect that the improvements in speed we have observed will eventually translate into better schedules, since the search process can explore a larger number of acceptable schedules.

Several minor issues arose when implementing the HST application. First, the algorithm, as specified in section 2, deals with binary constraints. The HST scheduling problem includes non-binary constraints, i.e., constraints that may involve several variables. For example, one constraint bounds the number of tasks that may be scheduled during a given time segment. For general CSPs, the exact method of counting the number of conflicts for an assignment may depend on the particular constraint in question. As it turned out, for the HST application it sufficed to count each violated constraint as a single conflict, even though multiple tasks might be involved in the violation.

A second issue concerns a difference between the GDS network and the min-conflicts algorithm. As described earlier, the network will remove a conflicted task from the schedule and then reschedule the task in two separate steps, which may not occur consecutively. In contrast, the min-conflicts algorithm rearranges tasks on the schedule, rather than removing them and reinserting them later. It appears that this difference is not significant, except perhaps when the schedule is over-constrained (as discussed below).

4 The Over-Subscription Problem

The HST scheduling problem can be considered a constraint optimization problem where we must maximize both the number and the importance of the constraints that are satisfied [2; 12]. We note that the telescope is expected to remain highly over-subscribed, in that many more proposals will be submitted than can be accommodated by any schedule. Unfortunately, one of the problems we have had is that no clear objective exists for determining the best schedule in such cases. In particular, we would like to maximize both the overall suitability of the schedule and the number of proposals that can be accommodated – no clear policy for evaluating the tradeoff between these two goals has yet been established by the Space Telescope Science Institute.

SPIKE handles the problem in a somewhat ad-hoc manner. There is, in effect, a pool of tasks that are either unscheduled or in conflict, and SPIKE's network updating scheme is equally likely to select any of these tasks. (Unscheduled tasks will be moved onto the schedule, and tasks that are in conflict will be moved off the schedule.) Thus, the number of unscheduled tasks to likely to remain approximately equal to the number of tasks in conflict. When the algorithm is eventually interrupted (assuming a conflict-free schedule has not been found) tasks that are in conflict can be removed. One of the advantages of the min-conflicts algorithm is that it is relatively easy to try a variety of schemes for dealing with overconstrained problems. We are currently experimenting with two basic approaches. The first is to follow the approach taken by the network (where tasks are removed and later re-inserted), but vary the procedure for removing and inserting tasks. For example, we can alter the probability of choosing an unscheduled task versus an already scheduled task, or bound the number of unscheduled tasks. (If we set to the bound to zero, then tasks will never be removed from the schedule, but simply be moved from place to place on the schedule as in the normal case.) Another approach is to use a more principled method for removing conflicting tasks after coming up with an initial schedule, so that only the minimum number of conflicting tasks need to be removed.

5 Evaluating the Algorithm

There are two contributions of this research. First, we have analyzed a neural network that has been successfully applied to a complex scheduling task and derived an easily understood symbolic algorithm that captures the network's behavior. Second, the algorithm's simplicity has lead to an implementation that is apparently much faster than the network's implementation. The algorithm has not yet been field-tested, but it has been tested on sample problems.

Unfortunately, one problem in evaluating the performance of the algorithm is that it is difficult to compare against competing approaches. This, of course, is a common problem. In particular, many operations research algorithms make different assumptions about the problem. For example, the over-subscription issue introduces certain difficulties in evaluation and comparison. Nevertheless, we do have plans to conduct such experiments.

To show the generality of our approach, we have tested the min-conflicts approach on standard CSP problems such as n-queens problem, where it performs quite well [10]. The min-conflicts method has also been tested on data on the Space Shuttle Payload Scheduling problem, another complex, real-world scheduling problem. Preliminary results show that the method performs far better than a backtracking CSP program that was previously built for this task[18]. Additional corroboration comes from a parallel study by Zweben[17], who has investigated a related method for repairing schedules using simulated annealing. In general, it appears that repair-based methods fare quite well on this problem. An additional bonus, as Zweben has pointed out, is that repair-based methods can also be used for dynamic rescheduling. In many domains this capability is important because unexpected events may require frequent schedule revision.

6 Related Work

The heuristic method described in this paper can be characterized as a *local search* method[5], in that each repair minimizes the number of conflicts for an individual variable. Local search methods have been applied to a variety of important problems, often with

impressive results. For example, the Kernighan-Lin method, perhaps the most successful algorithm for solving graph-partitioning problems, repeatedly improves a partitioning by swapping the two vertices that yield the greatest cost differential. The much-publicized simulated annealing method can also be characterized as a form of local search[4]. However, it is well-known that the effectiveness of local search methods depends greatly on the particular task.

There is also a long history of AI programs that have used repair or debugging strategies to solve problems (e.g., [15; 14]). These programs have generally been successful, although the repair strategies they employ may be complex, or domain specific. In the area of scheduling, Kurtzman[9; 8] has developed a class of iterative improvement algorithms that use a hill-climbing approach, similar to our algorithm. His approach is being used commercially for several space station scheduling applications. Kurtzman's method for repairing schedules appears more "intelligent" than ours, and more complex as well. In the area of constraint-satisfaction problems, Morris[11] has also recently developed an iterative improvement algorithm. His system uses an interesting technique called "breakout" to avoid being caught in local minima. We have not yet compared our algorithm to either Kurtzman's or Morris'. We suspect that their algorithms will perform better in certain domains due to their additional "intelligence", however, the advantage of the algorithm described here is that it is simple and relatively easy to analyze (see [10]). With this in mind, we are currently investigating the the possibility of adding a learning method to our algorithm so that more informed behavior is produced.

7 Conclusions

This paper has discussed a local search technique that has been successfully applied to the Hubble Space Telescope scheduling problem. The algorithm was derived from a neural network developed at the Space Telescope Science Institute. Our technique offers two main advantages. First, it is relatively easy to understand and analyze. Second, it requires less overhead than the network. The technique has been applied to other problems, and we are continuing to investigate and evaluate its computational properties.

References

[1] H.M. Adorf and M.D. Johnston. A discrete stochastic neural network algorithm for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Neural Networks*, San Diego, CA, 1990.

[2] E.C. Freuder. Partial constraint satisfaction. In *Proceedings IJCAI-89*, Detroit, MI, 1989.

[3] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences*, 1982.

[4] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation, Part II. *To appear in Journal of Operations Research*, 1990.

[5] D.S. Johnson, C.H. Papadimitrou, and M. Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37:79–100, 1988.

[6] M.D. Johnston. Automated telescope scheduling. In *Proceedings of the Symposium on Coordination of Observational Projects*, Cambridge University Press, 1987.

[7] M.D. Johnston and H.M. Adorf. Learning in stochastic neural networks for constraint satisfaction problems. In *Proceedings of NASA Conference on Space Telerobotics*, Pasadena, CA, January 1989.

[8] C.R. Kurtzman. *Time and Resource Constrained Scheduling, with Applications to Space Station Planning.* PhD thesis, Dept. of Aeronautics and Astronautics, MIT, Cambridge, MA, 1988.

[9] C.R. Kurtzman and D.L. Aiken. The Mfive space station crew activity scheduler and stowage logistics clerk. In *Proceedings the AIAA Computers in Aerospace VII Conference*, Monterey, CA, 1989.

[10] S. Minton, M.D. Johnston, Philips A.B, and Laird P. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings AAAI-90*, Boston, MA., 1990.

[11] P. Morris. Solutions without exhaustive search: An iterative descent method for binary constraint satisfaction problems. In *Proceedings the AAAI-90 Workshop on Constraint-Directed Reasoning*, Boston, MA, 1990.

[12] Fox M.S. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Morgan Kaufmann Publishers, Inc., 1987.

[13] N. Muscettola, S.F. Smith, G. Amiri, and D. Pathak. *Generating Space Telescope Observation Schedules.* Technical Report CMU-RI-TR-89-28, Carnegie Mellon University, Robotics Institute, 1989.

[14] R.G. Simmons. A theory of debugging plans and interpretations. In *Proceedings AAAI-88*, Minneapolis, MN, 1988.

[15] G. J. Sussman. *A Computer Model of Skill Acquisition.* American Elsevier, New York, 1975.

[16] M. Waldrop. Will the Hubble space telescope compute? *Science*, 243:1437–1439, 1989.

[17] M. Zweben. *A Framework for Iterative Improvement Search Algorithms Suited for Constraint Satisfaction Problems.* Technical Report RIA-90-05-03-1, NASA Ames Research Center, AI Research Branch, 1990.

[18] M. Zweben and M. Eskey. Constraint satisfaction with delayed evaluation. In *Proceedings IJCAI-89*, Detroit, MI, 1989.

Integrating Planning and Scheduling To Solve Space Mission Scheduling Problems

Nicola Muscettola and Stephen F. Smith

Center for Integrated Manufacturing Decision Systems
The Robotics Institute, Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

In this paper, we describe HSTS, a system that constructs executable observation schedules for the Hubble Space Telescope (HST). HST observation scheduling is a complex task, requiring attendance to a myriad of constraints relating to orbit characteristics, power and thermal balance requirements, instrument capabilities, viewing conditions, guidance requirements, overall allocation objectives, and astronomer specific restrictions and preferences. HSTS provides a general framework for representing and solving such complex scheduling problems. Generally speaking, scheduling in HSTS is viewed as the process of constructing a prediction of the behavior of a physical system (e.g. the HST operating environment) that reflects specified goals and constraints. The HSTS architecture provides a *domain description language* for specifying the structure and dynamics of the physical system, a *temporal data base* for modeling possible system behaviors over time, and a *scheduling/planning framework* that flexibly integrates decision-making at different levels of abstraction to construct a system behavior (or set of behaviors) consistent with stated scheduling goals and constraints.[1]

1. Introduction

A fundamental goal of space mission scheduling is efficient use of complex systems with large operating costs. In many cases (e.g. communication satellites, orbiting observatories, manned space vehicles, manned space stations), the system of interest has been designed to provide a wide range of capabilities over an extended lifetime. Such space technology offers unique opportunities for scientific experimentation and information gathering, and, on a given mission, potential demands for system resources to accomplish specific objectives are virtually unlimited. To maximize mission cost-effectiveness, it is important to accommodate as many demands as possible. This, in turn, depends directly on an ability to construct schedules that efficiently allocate system resources to competing mission activities.

Global optimization of resource usage in such domains is a very complex problem. Candidate mission objectives specify sets of activities to perform, each having a specific priority and designating specific resource requirements, temporal ordering constraints, allowable time windows, and scheduling preferences. Moreover, complex physical constraints relating to the behavior of various components of the system dictate the actual circumstances under which target activities can be performed. For example, successful execution of an onboard experiment may require minimal spacecraft vibration, which implies restrictions on parallel activity and the prior execution of any activities (e.g. stabilizing maneuvers) required to establish a vibration-free state. Satisfaction of physical constraints depends on the particular predicted state of the overall system, thus raising the additional issue of generating, synchronizing, and allocating resources to the activities required to support (enable) accomplishment of selected mission objectives. Over any given time frame, there are typically insufficient resources to satisfy all user demands, making it necessary to selectively relax problem requirements (e.g. drop mission objectives). Such decisions must balance the preferences of individual mission objectives with overall resource allocation goals.

Space mission scheduling can be seen generally as the process of constructing a behavior (or set of behaviors) of a complex dynamical system that is consistent with specified goals and constraints [Muscettola 90]. It requires an integration of what have historically been distinguished as "scheduling" and "planning" techniques, as each offers specific strengths with respect to the required overall process.[2] Recent research in scheduling [Fox and Smith 84, Smith et al 90, Sadeh 90] has emphasized the problem of efficiently allocating resources to competing activities over time in the presence of conflicting objectives and preferences, and has produced heuristic techniques that exploit the structure of the problem constraints (in particular, implied resource contention) to opportunistically focus solution development toward an acceptable global compromise. The power of these techniques vis a vis classical dispatch-based approaches has been demonstrated in large-scale manufacturing scheduling contexts [Ow and Smith 88]. At the same time, the ability to exploit such problem structure relies on specific

[1]This work was sponsored in part by the National Aeronautics and Space Administration under contract # NCC 2-531 and the Robotics Institute

[2]We in fact do not consider scheduling and planning to be fundamentally different processes and see this distinction more as a consequence of the capabilities of current techniques.

representational assumptions held in common with classical manufacturing scheduling research [Baker 74]. In particular, it is assumed that physical constraints are "pre-compilable" so that the complete set of activities requiring resources, as well as their ordering relationships and durations, are known in advance. This leaves resource availability as the only aspect of state that must be attended to over time, and permits a model of resource availability wherein a resource is considered unavailable during any interval that it is allocated to an activity and otherwise available. These representational assumptions are insufficient in domains like mission scheduling where the ability to execute a given target activity (e.g. a step in an onboard experiment) is a complex function of the state of the underlying physical system and variably implies different networks of supporting activities and resource requirements. In such domains, techniques based on these assumptions can at best provide guidance in focusing the development of an executable schedule.

Research in planning, alternatively, has focused on the problem of "compiling" activity networks that bring about desired goal states from more basic representations of the effects of actions in the world. However, as with scheduling research, the techniques that have emerged do not fully address the requirements of the class of problems described above. The appropriateness of classical STRIPS-style representational assumptions [Fikes et al. 72, Wilkins 88] is limited given the obvious need to deal explicitly with time (in both absolute and relative senses). More recently developed representational frameworks [Allen and Koomen 83, Dean et al. 88, Lansky 88, Vere 83] do provide these capabilities. However, with few exceptions (e.g., [Lansky 88]), these frameworks have not attempted to exploit the inherent structure of the underlying physical system. Given the complexity of the systems of interest in space mission scheduling, the ability to work with decomposable models of system behavior is fundamental to managing the combinatorics of search. More generally, current planning representations and frameworks do not provide a convenient basis for reasoning globally about efficient resource allocation. Interactions in resource requirements emerge only as the represented physical constraints are applied to achieve planning goals. Allocation conflicts can be avoided, but there is no leverage to anticipate resource contention, compromise among conflicting objectives and dynamically organize planning on this basis.

In this paper we describe HSTS, a scheduling/planning architecture for solving problems that require efficient allocation of resources over time in the presence of complex physical constraints. HSTS is based on a unifying perspective of scheduling and planning as a process of predicting the behavior of a dynamical system. Accordingly, HSTS provides

- a domain description language for modeling the structure and dynamics of complex systems at different levels of abstraction - Overall system dynamics is expressed in terms of interactions among structural components, providing a modularity that facilitates incremental development of both models and scheduling/planning heuristics.
- a temporal behavior data base for representing possible evolutions of the state of the system over time - System behaviors are represented as constraint networks, providing a basis for both analysis of current solution structure and incremental construction of consistent system behaviors (via successive posting of constraints and propagation of consequences).
- a scheduling/planning framework that flexibly integrates decision-making at different levels of abstraction - Abstract models are used to globally focus the development of the final, executable schedule in accordance with overall resource allocation objectives and preferences.

An initial version of the HSTS scheduling architecture has been implemented and applied to the complex problem of generating short-term, executable observation schedules for the Hubble Space Telescope. Preliminary experimental results have been obtained relative to a simplified but representative model of the telescope and its operating environment which indicate the potential of the architecture in solving large-scale mission scheduling problems.

To provide a context for describing the HSTS scheduling architecture and its current implementation status, we first consider the nature of the space telescope observation scheduling problem and its constraints.

2. The HST Observation Scheduling Problem

The initial motivation and domain of focus for the HSTS project has been the development of executable observations schedules for the Hubble Space Telescope (HST). HST is a sophisticated orbiting astronomical observatory that was deployed in April 1990 and is expected to have an operating lifetime of 15 years. When fully operational, HST will allow the world astronomical community to observe celestial objects at distances 7 to 10 times further and with a resolution 10 times higher than is possible from Earth-based observatories. HST is expected to provide insights into some fundamental questions about our Universe, such as its age, its density, how it began, and how it might end.

The development of observation schedules for HST is a large and complex task. On an annual basis, an allocation committee at the Space Telescope Science Institute (STScI) selects, among the submitted observation program proposals, those to be considered for the coming year. In order to insure a very high utilization of the telescope, the number of proposal accepted exceeds those that can be actually executed by the telescope. The objective of the observation scheduler for HST is to accommodate as many observation programs as possible in a given scheduling horizon, taking into account assigned program and observation priorities, and satisfy all constraints relating to the physical operation of HST. The principal measure of scheduling effectiveness is the fraction of time spent actually recording data on any scientific instrument on HST.

Astronomers specify observation programs according to a specification language [STScI 86] that allows the representation of complex constraints on the execution of the component observations. The basic structure of each program is a partial ordering of observations, each specifying the collection of light from a celestial object with one of the telescope's six scientific instruments. A diverse set of temporal constraints can be imposed on the observations in a program, including precedences, windows of opportunity for groups of observations, minimum and maximum temporal separations, and coordinated parallel observations with different viewing instruments. It is possible to prioritize the observations specified in a given program, and to specify preferences with respect to observation completion levels (e.g., 25% completion is minimally acceptable, 50% completion would be desirable, 75% completion is the maximum required). Each program also has an associated priority, decided by the allocation committee, which specifies if the program has to be considered "required" (i.e., its execution has to be insured within the scheduling horizon) or "supplemental" (i.e., its execution is conditioned to the availability of time on HST).

An observation program accurately describes the performance that the user requires from the telescope but leaves unspecified the operational constraints associated with actually executing the exposure. In fact, these constraints relate directly to the "physics" of picture-taking with the telescope, and are usually independent of the particular observations to be executed.

In general, the execution of an observation requires the satisfaction of three main requirements:

1. The telescope must be pointed at the target while the picture is being taken;
2. The required scientific instrument must be operational (i.e., exposing) for the specified duration.
3. The data collected by the scientific instrument must be communicated to Earth.

Each of these conditions, in turn, places additional constraints on the required state of the telescope and/or of the surrounding environment with which the telescope interacts, which must be similarly established for the exposure to take place.

If Condition 1 is not already satisfied, it can be achieved by rotating (or slewing) the telescope from the orientation required by the previous target to that of the required target. The duration of a slewing operation depends on the position of the previous target on the celestial sphere and can therefore be calculated only when an observation sequence has been determined. In addition to orienting the telescope in the direction of the target, it is necessary to lock the target in the center of the field of view of the required instrument, a process that requires the execution of additional operations. Both locking and picture taking require the target to be unocculted by the earth, the moon or the sun; these occultation periods can be deterministically known within a 1-2 month scheduling horizon, and can therefore be considered as data of the problem. If an observation is designated as "interruptible" it can continue after an occultation period; however, it is necessary to reestablish the target lock when the target becomes visible again. These aspect of telescope behavior are graphically illustrated in Figure 2-1.

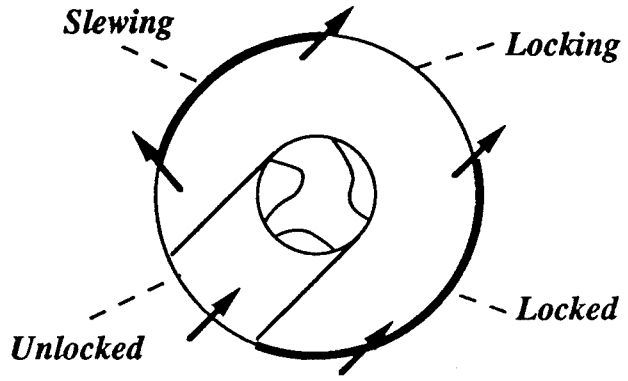


Figure 2-1: A sequence of transitions to point HST

The achievement of Condition 2 typically requires the execution of complex sequences of instrument setup operations. Some instruments are in fact composed of several independent detectors, possibly sharing some service devices (e.g., temperature control systems). Each detector and service device has an associated operating status, which can undergo warming up and cooling down transitions with several intermediate states; each state and transition typically has an associated range of possible durations. Limitations on the availability of electric power and structural characteristics the instruments require the satisfaction of parallelism and mutual exclusion constraints among the various warmup/cooldown processes. For example, a given detector might be required to be switched off while another is undergoing a warm-up or cool-down process. Similarly, while a detector is in an intermediate warmup state, the corresponding service base might be constrained to undergo only a well specified subsection of its warmup process. Figure 2-2 graphically illustrates one such constraint on the operating states of the Wide Field/Planetary Camera service base (WFPC) and one of its two detectors, the Wide Field Camera (WF).

Finally, Condition 3 implies that satisfaction of communication constraints is a function of the specified instrument and viewing mode (e.g., the rate amount of data produced by an instrument requires the use of a 1Mbyte/sec channel through a TDRSS communication satellite), as well as various user-specified special requirements (e.g., criticality of an observation requires both immediate down-linking of data and local storage on the tape recorder). Transmission of data to earth requires both visibility of one of the two currently available TDRSS satellites and the availability of an appropriate communication link. Storage of collected data for later transmission requires sufficient on-board tape recorder capacity.

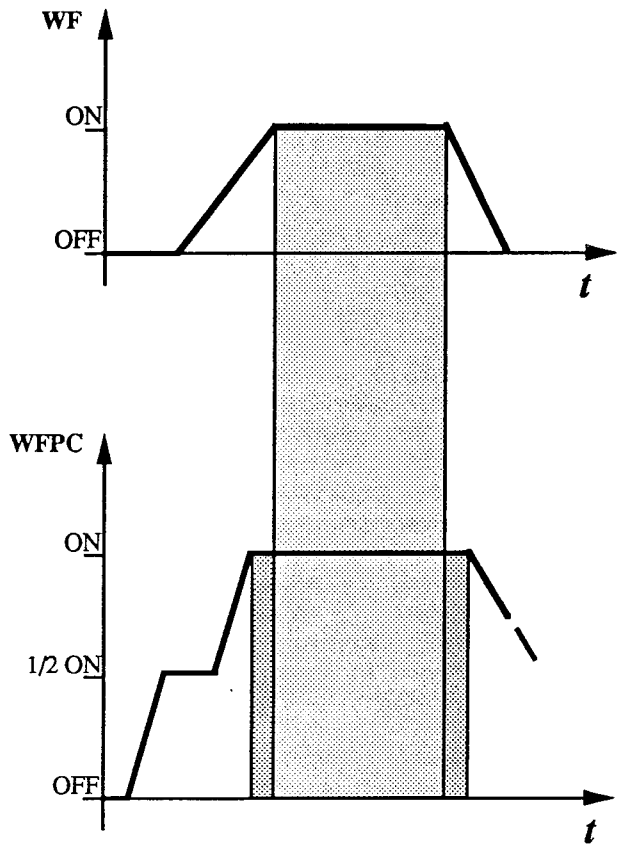


Figure 2-2: Constraint: WFPC must be on while WF is on

3. Modeling the Dynamics of Telescope Operations

As indicated at the outset of this paper, scheduling in HSTS is viewed as the process of constructing a behavior of a system that satisfies given constraints. Assuming this view, the first problem to address is that of describing the structure and dynamics of the system to be managed. Our approach reflects the following broad modeling requirements:

- **Representational adequacy:**
  In-depth analysis of the HST scheduling problem has led to the identification of several representational requirements:
  1. the ability to model actions and states that have definite, and often context-dependent, durations (e.g., slewing time).
  2. the ability to deal with actions and events that depend on the occurrence of particular combinations of states as opposed to the execution of explicit actions (e.g., a lock on a target is lost if the visibility window closes)
  3. the ability to model not only sequence constraints among actions and states but also constraints on their parallel occurrence (e.g., constraints on WFPC reconfiguration).

  The HSTS modeling framework addresses all of these issues.
- **Independence from the particular application:**
  It is evident that the representational issues identified above are not unique to the HST domain, but are instead common to a wide range of scheduling problems. For example, a representation of context-dependent durations is fundamental to management of automated factories where mobile robots are used to move parts around, as optimization of paths and travel times would necessarily play an important role.
  The HSTS modeling framework provides a general approach to the representation of physical systems, which we have used to construct a specific model of the HST operating environment.
- **Independence from the problem solving strategy:**
  Use of an opportunistic scheduling methodology implies the need to variably adopt different problem solving strategies during the construction of system behaviors (e.g. backward chaining, forward simulation). Thus, the system description must truly encompass all possible behaviors of the system, and be decoupled from any assumptions about the nature of problem solving strategy to be applied.
  The HSTS modeling framework achieves such independence by clearly separating the description of the structure and the dynamics of the system, which is of general use, from any heuristics and preferences that might be added to the model to specialize it with respect to a specific problem solving strategy.

In the following subsections, we describe the salient features of the HSTS modeling framework. We first consider the basic primitives for specifying system structure and dynamics, and then the extensions necessary to accommodate multiple levels of representation.

3.1. The HSTS Domain Description Language

Within the HSTS domain description language, a system is defined, at the basic structural level, as a collection of interacting parts or **system components**. Each component is characterized by a set of **properties** that are relevant to the scheduling problem. For example, one of the components of the current model of the HST operating environment is the HST optical system. For purposes of scheduling, the state of the optical system is fully specified once one knows what it is pointing at; thus, its sole property is POINTING STATUS. Another important class of components is that of **fixed targets** (stars, globular clusters, galaxies, etc.) (see Figure 3-1). One of their properties is the position on the the celestial sphere, identified by a <Right Ascension, Declination> coordinate pair; another is their visibility with respect to the space telescope.

At any instant of time, each property of each component of the system has one and only one associated **value**. In general, a value of a property is a description of some relation existing among several components of the system. Some properties

{{**fixed-target**
LOCATION:
VISIBILITY: }}

Figure 3-1: The class of fixed targets

are **static**, i.e., their value does not change over time. Others are **dynamic**, i.e., their value might change over time; in the following we will also refer to these as **state variables**. Referring again to the example in Figure 3-1, a fixed target's LOCATION is a static property while its VISIBILITY is dynamic (i.e., some times it will be visible from HST while other times it will be occulted). Other examples of dynamic properties in the HST domain include the POINTING STATUS of the telescope's optical system and the OPERATING STATUS of an instrument.

It is important to note here that in order to determine a behavior of a system that achieves specified goals it is necessary to model not only the dynamical behavior of the system to be managed but also the dynamical behavior of those elements of the environment (in the HST case, the targets) whose behavior affects our capability to achieve those goals. Within the HSTS domain description language, both the environment and the system to be managed are represented uniformly with the same primitives, leaving to the problem solver the responsibility to decide what is to be considered accessible and modifiable and what has to be considered as given.

The HSTS Domain Description Language requires explicit declaration of the set of possible values that can be assumed by each dynamic property in the model. Since a value represents the existence of a particular relationship among system components at a certain instant of time, a set of values is represented as a set of tuples belonging to one or more relations. Each set of tuples is represented as a set of predicate calculus assertions, with predicate names designating specific relations and arguments denoting variables or constants; by convention variable arguments are preceded by a question mark. Returning to our examples from the HST domain, the VISIBILITY state variable of a given fixed target $?T$ is defined to take on one of the two possible values at any point in time: *VISIBLE* ($?T$) or *NOT–VISIBLE* ($?T$). The set of possible values for the POINTING STATUS of the optical system of the telescope is given in Figure 3-2, where the variables $?T$, $?T1$ and $?T2$ designate arbitrary targets.

LOCKED (*HST*, $?T$)
UNLOCKED (*HST*, $?T$)
LOCKING (*HST*, $?T$)
SLEWING (*HST*, $?T1$, $?T2$)

Figure 3-2: Possible values of POINTING STATUS

A **behavior** of the system is an evolution over time of the values of its state variables. A behavior of the system is completely specified once a value has been associated with each state variable for each instant of time. Scheduling is concerned with the construction of such behaviors, and we will consider their representation in HSTS in Section 4. For now, we are concerned with specification of the *possible* behaviors that can be realized by the system.

The HSTS domain description language allows the specification of the "laws" that govern the possible behaviors of the system, as constraints on the values that the state variables can assume over time. Each possible value of each state variable has an associated **value descriptor**, which collectively specify the legal patterns of values relative to the variables of the system that may occur over time. More precisely, in order for a value v to be present in a behavior B of the system, it must be possible to recognize in B one and only one of the patterns specified in the value descriptor of v. Value descriptors allow the specification of simultaneity and sequentiality constraints on the occurrence of specific values.

A value descriptor specifies two distinct pieces of information; the **duration** and the **compatibility specification**.

The duration of a value is a constraint on the amount of time during which a value can appear continuously in a behavior of the system; it is represented as a pair of temporal distances $[d, D]$, $D \geq d \geq 0$, where d and D are respectively the **lower bound** and the **upper bound** on the duration. For example, the pair $[0, +\infty]$ denotes an indefinite duration; in this case the duration of the value is totally determined by the occurrence of other values that constrain its start and end time. On the other hand, $[c, c]$, where c is a constant, denotes a definite duration for a value; in this case the duration of the value is totally independent of the rest of the behavior of the system.

In general, both d and D may be functions of the parameters determining the associated value. The duration of *SLEWING* (*HST*, $?T1$, $?T2$), for example, is:

$$[d_{slew}(HST, ?T1, ?T2), d_{slew}(HST, ?T1, ?T2)]$$

This constraint returns a definite duration only when the both targets $?T1$ and $?T2$ are completely specified.

The **compatibility specification** of a given value determines how the continuous occurrence of that value is constrained over time by the occurrence of other state values. A compatibility specification may consist of one or more sets of compatibilities (not necessarily disjoint). The meaning of a compatibility specification for a value v is the following: for each possible behavior b of the system, if the value v appears in b over an interval of time, then there is a compatibility set in the compatibility specification such that all the compatibilities in the set are satisfied in b.

Each **compatibility** is expressed as a temporal relation between two values, indicating the existence of one or more temporal separation constraints between the start and/or end of the continuous occurrences of the two values. The temporal relations used in the HSTS domain description language are equivalent to those in [Allen and Koomen 83] but also allow the specification of temporal distances among the extremes of the intervals [Dean and McDermott 87]. For example, the fact that a target $?T$ must be visible in order to take a picture of it with viewing instrument $?I$ in operational status $?S$, is expressed as:

VISIBLE (?*T*)

{ *contains*, [0, +∞], [0, +∞] }

EXPOSE (?*I*, ?*S*, ?*T*)

This indicates that, for any ?*I*, ?*S* and ?*T*, if *EXPOSE* (?*I*, ?*S*, ?*T*) appears in the behavior of the system, then the value *VISIBLE* (?*T*) has to appear continuously during an interval of time such that its start precedes the start of *EXPOSE* (?*I*, ?*S*, ?*T*) by an indefinite amount of time and its end follows the end of *EXPOSE* (?*I*, ?*S*, ?*T*) by an indefinite amount of time. Another temporal relation available in the domain description language is { *before*, [*d*, *D*] }, which specifies that the end of the constraining value must precede the start of the constrained value, and the time interval separating the two events is constrained by [*d*, *D*]; The relation { *before*, [0, 0] }, for example, requires the simultaneity of the two events.

Figure 3-3 illustrates the sole compatibility set in the compatibility specification of *EXPOSE* (*WF*, 4*n*, ?*T*), which corresponds to taking a picture of a given target with the WF in operational state 4*n*.

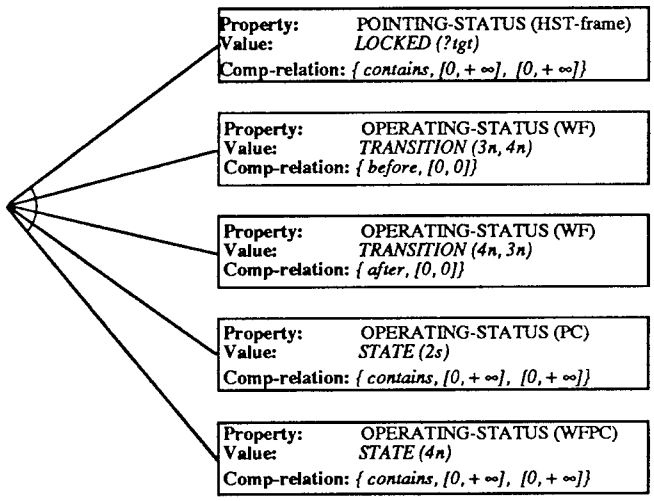


Figure 3-3: Compatibility spec for *EXPOSE* (*WF*, 4*n*, ?*T*)

3.2. Levels of Representation

The complexity of large scale scheduling requires the analysis of the problem at an aggregate level, in order to focus the problem solving effort on the more pressing issues. While detailed models are fundamental to generate schedules that satisfy all the constraints imposed by the physics of system operation, abstract models can facilitate the focusing effort. The HSTS domain description language provides primitives for specifying models of a system at different levels of aggregation and establishing their correspondence.

An abstract model consists of system components and state variables that aggregate several components and state variables of the detailed model. In the HST domain, for example, at the abstract level the telescope is modeled as a single capacity resource, with a single system component, the HST itself, and a single property, HST OBSERVING STATE, which provide a summary of the state of all the components of the detailed model.

Abstract values provide high level descriptions of entire segments of detailed behavior. In general, several detailed activities (e.g., the "setup" sequences) can be determined only with respect on the fully detailed system model. Therefore, the mapping between abstract and detailed representation is given through **refinement descriptors** that for each abstract value give the necessary conditions to be satisfied in the corresponding detailed behavior, i.e., a set of detailed state values, and a set of temporal relations constraining their occurrence over time. In the HST domain, one of the values of the abstract HST OBSERVING STATUS variable refers to specific "observations" (e.g., *OBSERVE*(?*P*, ?*I*, ?*S*, ?*T*, ...), where ?*P* designates an observation program, ?*I* designates a viewing instrument, ?*S* designates the required operating state of ?*I*, and ?*T* designates a target); the corresponding refinement descriptor listed in Figure 3-4 requires the occurrence of two values, *EXPOSE*(?*I*, ?*S*, ?*T*) and *READOUT*(1*Mb–link*, ?*I*) on the OPERATING STATUS state variables of the instrument and communication device respectively, such that the end time of the *EXPOSE* and the start time of the *READOUT* are within the duration of a single orbit. Furthermore, the start and end time of the *EXPOSE* coincide respectively with the start and end time of the abstract *OBSERVE*. These latter constraints provide a basis for downward imposition of time constraints, as well as upward propagation of detailed scheduling decisions.

At the abstract level, the description language primitives allow the generation of rough estimates of the characteristics of the corresponding detailed behaviors before their complete expansion on the detailed model. For example, in order to account for the duration of the context-dependent "setup" activities associated to an *OBSERVE*(?*P*, ?*I*, ?*S*, ?*T*, ...), the lower and upper bounds of the duration of the abstract value are modeled as functions of the proximity of the previous and current targets, and of the previous and current configurations of the scientific instruments and communication devices.

{{**refine-desc-1**
VALUE:
OBSERVE(?*P*, ?*I*, ?*S*, ?*T*, ...)
SUBVALUES:
V1 (OPERATING-STATUS *EXPOSE*(?*I*, ?*S*, ?*T*))
V2 (OPERATING-STATUS *READOUT*(1*Mb–link*,?*I*))
ORDERING-CONSTRAINTS:
V1 { *before*, [0, **orbit–duration**] } V2
SAME-START: V1
SAME-END: V1 }}

Figure 3-4: Refinement descriptor for *OBSERVE*

4. Representing System Behaviors

Given a description of the system to be managed, a second broad architectural issue concerns the manner in which specific system behaviors (i.e. schedules) constructed by the

scheduler are represented. Within HSTS, this is accomplished through the use of an underlying temporal data base. The HSTS temporal data base has the following general characteristics:

1. **It stores behaviors of a system**: In fact, the data base satisfies a stronger requirement, since its *only* legal states are those that satisfy the constraints on the dynamics of a pre-specified system model;
2. **It is a constraint network**: Behaviors are represented implicitly by a series of constraints that have been either externally imposed (e.g. by requirements of the problem) or directly extracted from the system model. This provides a representation of a partial schedule as a state of the database where several aspects of the system behavior currently under construction are left underspecified.
3. **It supports opportunistic scheduling**: At any point during scheduling, several parts of the data base might require refinement (through additional constraint posting) to produce a complete specification of the final schedule. The database leaves complete freedom as to the order in which these refinements are made.

The HSTS temporal data base extends in several ways the philosophy of the time map formalism developed in [Dean and McDermott 87]. Perhaps the most fundamental departure in our approach is the tight connection that is established between the state of the data base and the model of a system This association provides a strong basis to support planning and enforce database consistency.

The process of building a system behavior that satisfies a given set of scheduling goals involves the determination of sequences of values for system state variables that include these designated values and coordinate temporally in a manner consistent with the compatibility constraints specified in the system model. At any stage of this process, the HSTS Temporal Behavior Data Base represents the set of values, sequences and compatibility constraints that have been posted so far and the specifications of the compatibility constraints that are known to be needed but have not yet been posted.

For each state variable, the scheduling horizon is subdivided into a sequence of intervals, or **tokens**, each being a triple $< st, et, type >$, where *st* and *et* represent respectively the token's start and end time and *type* is a set of values.

We distinguish two distinct kinds of tokens:

- **value token**: a value token indicates that the interval represents the occurrence of a single constant value; therefore, the type of a value token has cardinality one. A value token originates either from the external posting of a scheduling goal or from the direct implementation of a compatibility constraint. For example, in the HST domain, value tokens of type $EXPOSE(?I, ?S, ?T)$ generally correspond to a proposer's request to take a picture, while tokens of type $LOCKED\,(HST, ?T)$ are typically justified as enabling conditions for a corresponding *EXPOSE*. Notice, however, that the distinction between tokens that are scheduling goals and tokens justified by compatibilities depends on the specific scheduling problem that is being solved and are not at all intrinsic to the system model. In other words, it could be perfectly reasonable to formulate problems that require as external goal the occurrence of a *LOCKED* token, e.g., during calibration and/or instrument maintenance routines.
- **constraint token**: a constraint token denotes a segment of the evolution of a state variable that has not yet been constrained to any value token. Therefore, this token implicitly represents a set of sequences of values. No restriction is imposed on the length of the sequence, and it can possibly be empty. Each value of the sequence is constrained to belong to the set *type*, while *st* and *et* represent respectively the start of the first value and the end of the last value in the sequence. A constraint token can be considered as a "hole" in the evolution of a state variable within which it is possible to find room for a new value.

As mentioned earlier, any value posted in the temporal database must be consistent with the compatibility constraints in the corresponding system model. When a value token is introduced into the sequence of a state variable, it is connected with instances of the duration and compatibility specifications that are associated with its type in the system model. In order for the value token to be justified, there must be an implementation of these specifications such that the overall schedule is consistent. Given a duration constraint $[d,D]$ for the token *TOK*, its implementation implies the introduction of the following temporal separation constraint;

$$d \leq et(TOK) - st(TOK) \leq D$$

Given a token TOK_1, implementing a compatibility constraint of the kind:

TEMPORAL RELATION: $\{contains, [0,+\infty], [0,+\infty]\}$
TYPE: *P*

corresponds to selecting or generating a token TOK_2 of type *P* and introducing the temporal constraints;

$$st(TOK_1) - st(TOK_2) \geq 0$$

$$et(TOK_2) - et(TOK_1) \geq 0$$

A fundamental aspect of the operation of the HSTS Temporal Behavior Data Base is that constraints can be posted irrespective of the existence of an overall consistent assignment of values to each variable. Consistency verification can be an expensive operation and it can be redundant if it is known that the addition of a constraint will not make the network inconsistent (even if a complete assignment of values to state variables is not yet known). Constraint propagation is decoupled from constraint posting to allow the scheduler to take advantage of such knowledge.

One important consequence of representing a schedule as a network of temporal constraints is that it allows commitment to a specific assignment of start and end times for each

scheduled value token to be avoided whenever possible. An HSTS schedule explicitly represents a window of opportunity for the occurrence of each event in a system behavior. Moreover, the explicit representation of the network of temporal constraints reduces the solution of some simple reactive scheduling problems to polynomial constraint satisfaction processes. For example, Figure 4-1 represents the network of reconfiguration activities needed on the WF/PC state variables and on the telescope pointing status in order to take a picture with the WF (the black value token corresponds to the EXPOSE value). The four temporal constraints connected to the target visibility correspond to the selection of the orbit during which the exposure is scheduled to be taken. If, after the development of a schedule, it becomes necessary to delay the exposure of one orbit (e.g., because the scheduler has been suddenly required to make room for a higher priority exposure), we would just need to redirect the four orbit selection links and repropagate through the resulting network of temporal constraints.

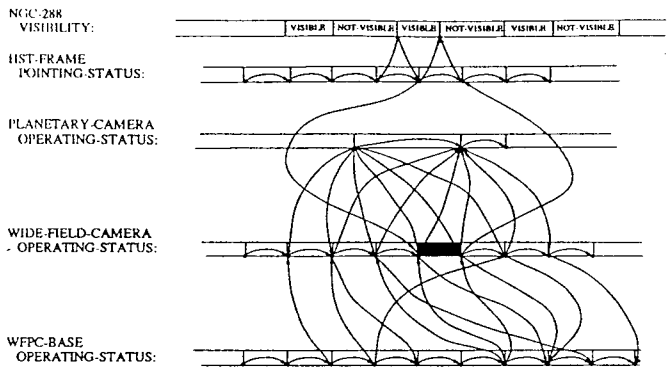


Figure 4-1: Network of temporal constraints for exposure.

5. Integrating scheduling and planning

The temporal behavior data base is partitioned into a series of layers, each corresponding to a level of abstraction of the system model, to reflect the current state of the solution during the schedule generation process. Within the current HST scheduler two layers are distinguished:

- **Abstract layer**: This is a representation of the scheduling goals and temporal scheduling constraints that constitute the current scheduling problem (e.g., programs of observations) and of the abstract model of the HST. Initially no observations are scheduled and the OBSERVING STATUS of HST is *AVAILABLE* over the entire scheduling horizon.
- **Detailed layer**: This represents a set of behaviors that are consistent with the refinements of the scheduling goals selected so far. Initially, the temporal data base reflects external events that are deterministically known (e.g., periods of target visibility) and assumptions regarding the initial state of other system state variables (e.g., the initial state of each instrument, the initial pointing status, etc.)

Generation of a schedule proceeds incrementally, by repeatedly selecting one or more as yet unachieved scheduling goals (or alternatively selecting one or more previously achieved goals to be retracted), inserting them into the abstract state variable sequence (or extracting them, if the goals have to be retracted), communicating the refined goals to the detailed layer, and constructing the detailed system behavior that extends the detailed layer to include the achievement of the newly posted goals (or desired retraction of previously achieved goals). This cycle is repeated until either all of the scheduling goals have been achieved or it has been determined that it is not possible to achieve those that remain.

The different layers are suitable for different problem solving activities according to the level of detail of their constraint representation. At the abstract level, telescope reconfigurations needed to achieve a scheduling goal (e.g., telescope slewing, instrument warm ups and shut downs) are implicitly modeled as adjustments on the duration of each observation. Here, decision-making is concerned with allocating available slots of time on the telescope operating status to unscheduled observations. This level of abstraction is well suited for global focusing activities such as distributing observations over the scheduling horizon to minimize the possibility of resource contention (i.e., avoiding resource bottlenecks) and sequencing observations to minimize estimated telescope reconfiguration (setup) times. On the other hand, the detailed representation of the system supports the actual expansion of required setup activities and their mutual synchronization. Depending on the accuracy of the abstractions of the detailed constraints in the abstract model, it may not be possible to reliably make certain decisions (e.g., the selection of the orbit during which an exposure is taken) until all detailed constraints have been elaborated. In these cases, decision-making at the abstract layer will involve the communication of preferences (e.g., "schedule the observation in the earliest possible orbit within the imposed time constraints") to be implemented on the detailed layer.

At any point during problem solving, there may be several value tokens in a given layer of the temporal data base whose occurrence is still incompletely justified. This is the case if there are value tokens in the temporal data base that do not have a compatibility set with all compatibilities implemented. To establish if the current temporal data base actually contains some consistent behavior of the system, it is necessary to select and implement additional compatibilities. This process is carried out through a heuristic search that combines two principal selection steps:

1. Selection of an open compatibility to implement. We refer to the value token to which the compatibility is connected as the the constrained token;
2. Selection of a value token to be connected to the constrained token according to the directives of

the selected compatibility. If a value token does not already exist in the current temporal data base, a constraint token whose type matches the value required by the compatibility is selected and a new value token is inserted into it.

After some number of temporal constraints have been posted, a temporal constraint propagation process spreads the consequences to the rest of the temporal data base and possibly detects the inconsistency of the current state of the data base. In the latter case, backtracking is needed before the search process can continue.

The architecture provides several mechanisms to encode heuristic knowledge to govern the search process. Such knowledge includes:

1. selection among alternative value refinements during translation of scheduling goals at the abstract layer into networks of goals at the detailed layer. For example, in the current HST scheduler, direct communication of data to earth is preferred to local storage on the tape recorder when the choice exists, and these alternative refinements are explored in this order.
2. decomposition of the overall search into subproblems. For example, in the current HST scheduler, the search to achieve an EXPOSE and READOUT goal pair that results from the refinement of an OBSERVE is partitioned into a search to achieve the EXPOSE, a search to achieve the READOUT and a final search to select the target and communication satellite visibility windows. Each of these subproblems can be further sub-divided into still smaller subproblems. For example, to achieve an EXPOSE requiring the WF detector, the planner first builds the network of values switching on the WF/PC and then that switching off the other instruments. To switch on the WF, the planner considers the reconfiguration of the state variables associated to the whole WF/PC instrument, first reconfiguring the WF, PC and WFPC state variables independently and then mutually synchronizing them.
3. selection among mutually exclusive compatibilities that discriminate among different compatibility sets within an open compatibility specification; For example, in the current HST scheduler, communication of data to earth requires the visibility of either of the two TDRSS satellites, and a heuristic that exploits the degree of overlap with the viewing target's visibility is employed.
4. selection of value or constraint tokens to implement the selected compatibility. For example, in the current HST scheduler, the goal of executing an observation as soon as possible translates into a preference for the earliest token in time (value or constraint) that locally satisfies the requirements of the current compatibility.
5. execution of the temporal constraint propagation. It is possible to specify when the propagation is needed with respect to the current decomposition of the problem into subproblems. Moreover, it is possible to specify different preferences with regard to the traversal of parallel propagation paths in the network of temporal constraints, with the intent of speeding up the propagation by traversing the most restrictive paths first.

Heuristic knowledge is also required in order to relax the detailed layer of the temporal data base in situations where a revision to the constraints posted on the abstract layer requires the retraction of a network of detailed values (e.g., the insertion of a new observation between two observations previously considered to be consecutive might require substantial changes in the telescope reconfiguration sequences). After the detailed values are retracted, constraint tokens take their places on the corresponding state variables, providing the "holes" into which the values required by the new reconfiguration networks can be placed.

6. Current Status and Preliminary Results

The development of a scheduling system for HST has been pursued by building and experimenting with increasingly realistic models of the operating environment. The model of the Hubble Space Telescope over which the system is currently operational consists of 14 state variables: 1 in the abstract model and 13 in the detailed model. The state variables contained in the detailed model relate to the following system components:

- 5 represent the operating status of the 2 instruments that are currently modeled; 3 for the WF/PC and 2 for the Faint Object Spectrograph (FOS)
- 1 represents the HST pointing status
- 4 represent the status of instrument data buffers
- 2 represent the status of the two data transmitters
- 1 represents the status of local tape recorder.

Additional state variables represent the visibility status of each target and of the two TDRSS satellites.

The current automatic scheduler operates according to a "dispatching" strategy. At each cycle, an observation is selected among the current unscheduled tasks and appended to the current sequence of scheduled observations. The selection is made according to a dispatching rule that attempts to minimize "dead time" (i.e., an estimate taking into consideration instrument reconfiguration, telescope repositioning, and target and TDRSS visibility windows). Figure 6-1 describes some preliminary results on a scheduling problem consisting of 16 single observation programs, with requirements for all the 4 detectors of the two modeled instruments and all the communication links and the tape recorder. The cumulative viewing time required by this set of observation was 5 hours and 9 minutes. The system constructs a schedule that covers a scheduling horizon of 23 hours and 35 minutes, with schedule efficiency (i.e., the ratio between the science time and the covered scheduling horizon) of about 22 %. The schedule efficiency is artificially low with respect to the theoretically estimated upper bound of 30-35% [Johnston

85] due to the fact that the model includes the requirement that the FOS (respectively WF/PC) must be switched off every time an exposure is performed on the WF/PC (FOS). Experiments conducted with a less restrictive (and more realistic) model allowing both the WF/PC and the FOS to be on in parallel yielded a schedule efficiency of 29.43 %.

Notice that the temporal data base minimizes the number of time points needed in the temporal distance graph. When an equality constraint (i.e., [0, 0] distance) is posted, the two connected time points are collapsed. Figure 4-1 gives an indication of the topology of the network of temporal constraints that need to be built for each scheduled observation.

CPU time:	5 minutes 40 seconds
Value tokens:.....	306
Time points:.....	273
Temporal distances:.....	1138

Figure 6-1: Preliminary results

The current HST scheduler can also be operated in an interactive mode. The user is allowed complete freedom in the development and revision of the overall observation sequence. The user need only specify, at each step, the unscheduled observations to be added, the position in the current sequence of scheduled observations where they should be inserted, which currently scheduled observations (if any) should be removed, and the preference with respect to the time of occurrence of the selected observations (e.g., as soon as possible, after a calendar date). The interactive scheduler provides the basic capabilities on which the development of a more flexible and opportunistic automatic scheduler will rely (see below).

The modularity of the HSTS framework has allowed an incremental development of both the model and the scheduling and planning knowledge. This development took place in several stages, each consisting in the construction of a complete scheduler on increasingly more complex models. The first stage involved a model consisting only of the WF/PC and the HST pointing status; then the model was expanded to include the FOS; finally the rest of the state variables was added to account for the communication of data to earth.

7. Concluding Remarks

In this paper, we have presented HSTS, a problem-solving architecture aimed at the solution of complex problems, like space mission scheduling, that require efficient allocation of resources in the presence of complex physical constraints. The HSTS architecture is grounded in a uniform view of planning and scheduling as a process of constructing behaviors of a complex dynamical system. Following this view, the HSTS architecture provides a *domain description language* for modularly specifying the structure and dynamics of complex systems at different levels of abstraction, a *temporal behavior data base* that represents possible system behaviors over time and flexibly supports the incremental construction of solutions, and a *scheduling/planning framework* that flexibly integrates decision-making at different levels of abstraction to produce consistent system behaviors that attend to overall allocation objectives.

An initial version of the HSTS architecture has been applied to the problem generating short-term observation schedules for the Hubble Space Telescope, and preliminary experimental results obtained relative to an incomplete but substantial model of the telescope and its operating environment indicate the utility of the architecture as a basis for effectively managing the combinatorics of large-scale problems.

One area of current research concerns the development of more sophisticated, constraint-directed strategies for globally structuring the scheduling process. The currently implemented HST scheduler relies strictly on a local greedy heuristic for optimizing overall resource usage. While the schedule efficiency results given in Section 6 are respectable, the effectiveness of the local heuristic is due in part to the absence of many of the more complex temporal constraints that may be specified in observation programs in the scheduling problem that was solved. More importantly, schedule efficiency is not the sole objective in HST observation scheduling, it must ultimately be balanced against other allocation objectives (e.g. program and observation priorities). We believe that a key to better solutions to the full problem lies in the ability to dynamically direct problem solving according to the evolving structure of the underlying solution space. In fact, it was the desire to exploit such opportunistic problem structuring techniques that, in large part, originally motivated the HSTS scheduling architecture. [Muscettola et al. 89] We are currently investigating the use of previously developed preference representations [Muscettola and Smith 87, Johnston 90, Sadeh 90] as a basis for more general characterizations of current solution constraints, and the development of problem decomposition heuristics that operate with respect to these representations. On a related note, we currently have little insight into performance tradeoffs concerning the relative amount of problem solving effort expended at different levels of abstraction, and the relative amount of problem solving responsibility that should be apportioned to each level. Experimental analysis of this tradeoff is required.

Acknowledgements

We gratefully acknowledge the efforts of the other members of the HSTS project team: Gilad Amiri, Amedeo Cesta, Daniela D'Aloisi, and Dhiraj Pathak.

References

[Allen and Koomen 83]
Allen, J. and Koomen, J.A.
Planning Using a Temporal World Model.
In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 741-747. 1983.

[Baker 74] Baker, K.R.
Introduction to Sequencing and Scheduling.
John Wiley and Sons, New York, 1974.

[Dean and McDermott 87]
Dean, T.L. and McDermott, D.V.
Temporal Data Base Management.
Artificial Intelligence 32:1-55, 1987.

[Dean et al. 88] Dean, T. and Firby, R.J. and Miller, D.
Hierarchical Planning Involving Deadlines, Travel Time, and Resources.
Computational Intelligence 4:381-398, 1988.

[Fikes et al. 72] Fikes, R.E., Hart, P.E. and Nilsson, N.J.
Learning and Executing Generalized Robot Plans.
Artificial Intelligence 3:251-288, 1972.

[Fox and Smith 84]
Fox, M.S. and Smith, S.F.
ISIS: A Knowledge-Based System for Factory Scheduling.
Expert Systems 1(1):25-49, 1984.

[Johnston 85] Johnston, M.D.
A Note of ST Observing Efficiency.
unpublished internal note, STScI, Baltimore, MD.

[Johnston 90] Johnston, M.D.
SPIKE: AI Scheduling for NASA's Hubble Space Telescope.
In *Proceedings of the 6th Conference on Artificial Intelligence Applications*, pages 184-190. IEEE Computer Society Press, 1990.

[Lansky 88] Lansky, A.
Localized Event-based Reasoning for Multiagent Domains.
Computational Intelligence 4:319-340, 1988.

[Muscettola 90] Muscettola, N.
Planning the Behavior of Dynamical Systems.
Technical Report CMU-RI-TR-90-10, The Robotics Institute, Carnegie Mellon University, 1990.

[Muscettola and Smith 87]
Muscettola, N. and S.F. Smith.
A Probabilistic Framework for Resource-Constrained Multi-Agent Planning.
In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 1063-1066. Morgan Kaufmann, 1987.

[Muscettola et al. 89]
Muscettola, N. and Smith, S.F. and Amiri, G. and Pathak, D.
Generating Space Telescope Observation Schedules.
Technical Report CMU-RI-TR-89-28, The Robotics Institute, Carnegie Mellon University, 1989.

[Ow and Smith 88]
Ow, P.S. and Smith, S.F.
Viewing Scheduling as an Opportunistic Problem Solving Process.
In R.G. Jeroslow (editor), *Annals of Operations Research 12*. Baltzer Scientific Publishing Co., 1988.

[Sadeh 90] N. Sadeh, and M.S. Fox.
Variable and Value Ordering Heuristics for Activity-based Job-shop Scheduling.
In *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management, Hilton Head Island, S.C.*. 1990.

[Smith et al 90] Smith, S.F. and Ow, P.S. and Muscettola, N. and Potvin, J.Y. and Matthys, D.
An Integrated Framework for Generating and Revising Factory Schedules.
Journal of the Operational Research Society 41(6):539-552, 1990.

[STScI 86] STScI.
Proposal Instructions for the Hubble Space Telescope.
Technical Report, Space Telescope Science Institute, 1986.

[Vere 83] Vere, S.
Planning in Time: Windows and Durations for Activities and Goals.
IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-5, 1983.

[Wilkins 88] Wilkins, D.E.
Practical Planning.
Morgan Kaufmann, 1988.

SOLUTION OF TIME CONSTRAINED SCHEDULING PROBLEMS WITH PARALLEL TABU SEARCH

by

E. L. Perry, Ph. D.
Ford Aerospace
9970 Federal Drive
Colorado Springs, Colorado 80921
Phone: 719 594-1911

Abstract

Many critical decisions involve the solution of scheduling problems where time does not permit the consideration of all alternatives. Military examples include weapon-target pairing and evaluation of courses of action. Civilian examples are found in the assignment of computing jobs to processors and humans to tasks. Often the size of the solution space grows exponentially with the number of threats and a schedule must be produced within seconds. The new technique of Tabu Search offers a method for systematically searching the solution space to find an optimal or near optimal solution in a short period of time. This paper combines Tabu Search with parallel processing to increase the number of feasible schedules that can be considered in a short time period. The technique also overcomes the problem of local optimality. A Naval Anti-Air Warfare scheduling problem is used to illustrate the method.

1.0 INTRODUCTION

Most scheduling problems have extremely large solution spaces which cannot be searched by traditional methods in any reasonable amount of time. We give an example from the military which is representative of the numerous time constrained scheduling problems.

1.1 Illuminator Scheduling in Naval Anti-Air Warfare

In Naval Anti-Air Warfare, suppose that we have n incoming threats (missiles or planes). Surface to air missiles (SAMs) are to be launched against the threats. There are m terminal illuminator's (TIs) which serve as terminal homing radars for the SAMs. During the last three to fifteen seconds of the SAM's flyout, one of the TIs must be locked onto the threat. The reflected beam of the TI guides the SAM into the threat for a certain kill. The problem is to schedule the illuminators so that one is always available for each SAM at the proper time. The objective is to maximize the depth of fire; that is, to get the most possible shots at the incoming threats. The illuminators may have times when they are unavailable due to previous scheduling. It has been shown that there are $m^n * n!$ possible schedules [Boyer, et al, 1990]. This problem is an example of a very complex assignment problem called the time constrained scheduling problem. In non-military terms it can be described as a many-to-one assignment of computing jobs to usable processors where the cost of assigning job i to processor j is time dependent. It is assumed that the processors may

already have a preset schedule that must be considered as new jobs are added. Furthermore, the assignments must be made by some deadline (the time constraint on processing) that prevents consideration of all the feasible schedules. It is therefore necessary to use an iterative method where the best schedule found to date is always available in case processing is cut short. In this process, the initial feasible solution must be found quickly to insure availability of a solution. We must also overcome the problem of local optimality. The search through the solution space can become fixed on a locally optimal solution and completely miss the globally optimal one.

The problem itself is a member of the set of problems known to be NP-Complete [Garry and Johnson, 1979]. This means that no efficient algorithm has been developed which will always find the solution. However, several techniques have been found to yield an approximate solution in a reasonable amount of time [Glover, 1988, 1989 and Davis, 1987]. We will extend Glover's Tabu Search technique to a particular form that is appropriate for implementation on a parallel processor. We show the application of the general method to problems of the type described above.

We now describe the Parallel Tabu Search method and then show its application to the example described above. A summary of the method and its wide applicability to other problems is described in the last section.

2.0 PARALLEL TABU SEARCH

Tabu Search has been successfully implemented in a wide range of settings as a metastrategy to guide other heuristics to overcome limitations of local optimality [Glover, 1988, 1989 & 1990]. It utilizes a form of short term memory called a tabu list to assure the search will not revisit a previous solution except by a path not traveled before. Attributes of the solution space are identified which, if prevented from recurring in a future move, will assure the present move cannot be reversed. These attributes are recorded on the tabu list, where they reside for a specified number of iterations before they are removed.

In order to implement Tabu Search, we must have:

ts1) a graph theoretic description of the solution space;
ts2) a characterization of a solution in the space;
ts3) a heuristic method of generating a new solution from an old one in the solution space; and
ts4) a way to compare one solution against another to decide which is better.

In the Parallel Tabu Search method, we divide the solution space into approximately equal size segments using a permanent tabu list. The number of segments is determined by the number of processors that are available. One processor will search each segment thus assuring that all processors are doing useful and non-redundant work. Each processor does the following:

p1) get a solution from my segment of the solution space;
p2) use ts3) to get a new solution; put the old solution on the tabu list; the tabu list is used to make sure that we do not repeat solutions and the perma-

nent tabu list is used to make sure that we remain within our allocated segment; the tabu list also overcomes the problem of local optima [Glover, 1990];
p3) use ts4) to compare the new solution to the best one found to date; record the best solution found;
go to ps2);

This process continues until time expires. After the allotted time, each processor reports the best solution it found. These are compared and the best overall solution is returned.

3.0 PARALLEL TABU SEARCH AND THE ILLUMINATOR SCHEDULING PROBLEM

For an example, we turn now to the illuminator scheduling problem.

Assume we are given a set of n threats

$$T = \{ t(1),...,t(n) \}$$

and a set of m terminal illuminators

$$TI = \{ illum(1),...,illum(m) \}$$

For each of the threats t(i) we have a collection of p engageability intervals

$$eng(i) = \{ eng(i,1), eng(i,2), ... ,eng(i,m) \}$$

where

$$eng(i,k) = \{ t: fe(i,k) <= t <= se(i,k) \}$$

indicates that t(i), can be engaged by a SAM using a terminal illuminator illum(k) during the period of time between fe(i,k) and se(i,k). We use E for the union of the engageability intervals.

The depth of fire, d(i) for a given threat t(i) based on a set of engageability intervals eng(i) is the maximum number of SAM's that could be launched against this threat employing a specified firing doctrine.

Also, there is a plan function, P, such that for each threat i and each terminal illuminator j, we can determine whether or not the illuminator can be used for terminal guidance against the ith threat. The plan function is defined by

$$P : TxTI -> BxRxRxRxIxI$$

where B represents the set {TRUE, FALSE}, R represents the real numbers, and I is the integers. The first component of P(i,j) is TRUE if and only if threat t(i) can be intercepted using terminal illuminator illum(j) to guide the SAM to the threat during the final seconds of flyout. When the first component is TRUE, the second component is set to the earliest available starting time (computed from j's que of illumination intervals) that can be used to address threat t(i) (i.e., the earliest possible time for beginning illumination, bill(i), that is consistent with a feasible launch time) while the third component of P(i,j) is set to the predicted intercept time, pint(i). Of course, pint(i) must be in eng(i, j). The fourth component of P(i,j) is set to the launch time, ltime(i), for the SAM while the fifth and sixth components of P(i,j) give the indices of the participating units that provide the launch and terminal illuminator respectively for the SAM. The launch time must be chosen so that a launch device is available on unit x at ltime(i). The interval

(bill(i), pint(i))

is called the illumination interval and it includes the time for slewing the terminal illumination device and locking onto the ith threat. We often write delta(i) = pint(i) - bill(i). Thus,

P(i,j) = (TRUE, bill(i), pint(i), ltime(i), x, k)

represents a plan for an engagement of the ith threat using the jth terminal illuminator on unit k. In this plan, the SAM is launched from participating unit x at time ltime(i), the jth illuminator provides terminal illumination in the time interval (bill(i), pint(i)). When there is only one ship involved, we can suppress the last two terms of P(i,j) because the source of the launch is readily apparent and there is only one participating unit.

Also, there is a cost function

c:TxTIxE -> R

in which c(t(i), illum(j), pint(i)) gives the cost of engaging t(i) at time pint(i) using terminal illuminator illum(j). The cost function gives the method for evaluating one possible schedule for the terminal illuminators against another. This cost function together with the plan function provide much of the intelligence of the heuristic underlying our Tabu Search method. We note that the cost function can be any function of the threat, the terminal illuminator and the intercept time that accomplishes the desired objective.

A linear cost function is given by

c(t(i), illum(j), pint(i)) = d(i) * (pint(i) - fe(i))

where fe(i) is the earliest possible intercept time for threat i, pint(i) is the predicted intercept time for threat i and d(i) is the depth of fire for threat i. This cost function will schedule the launches and terminal illuminators to maximize the depth of fire.

A schedule for terminal illuminator illum(j) is a set

$$S(j) = \{ (t(j_i), P(j_i, j)): i = 1,2,\ldots,n(j) \}$$

of threats paired with outputs of the plan function such that:

a0) the first component of $P(j_i, j)$ is TRUE for all i;
a1) $t(j_i)$ is in T for $i = 1,..,n(j)$;
a2) $t(j_i)$ is different from $t(j_k)$ when i is not equal k;
a3) the illumination intervals are all mutually disjoint.

Conditions a1 and a2 together guarantee that each threat in T appears, at most, once in a schedule for the jth illuminator.

An optimal schedule is a collection of schedules

S = {S(j): j = 1,...,m}

such that

b1) S(j) is a schedule for illum(j), j = 1,...,m;
b2) n(1) + n(2) + ... +n(m) = n;
b3) The sum of $c(t(j_i), illum(j), pint(j_i))$ is minimal where the sum is taken over all i and j.

A feasible schedule is a collection of schedules satisfying

b1) and b2).

Then in an optimal schedule, all threats in T are addressed and the total cost is minimal. In a feasible schedule, all threats are addressed but the total cost is not necessarily minimal.

The problem is to produce an optimal schedule. If an optimal schedule is not possible in the allotted amount of processing time then we shall relax condition b3 and produce a feasible schedule satisfying b1, b2, and with the double sum in b3 as small as possible. Such a schedule will be referred to as a best schedule.

3.1 The Solution Space

We describe the solution space via a directed acyclic graph [Bertsekas and Tsitsiklis, 1989].

The problem can be formulated as a shortest route problem from a start node s to an end node t through a network. Each node in the network corresponds to a threat paired with a terminal illuminator, an output of the planning function and a cost for the engagement. Thus a node in the network can be thought of as a tuple of the form

$$(t(i), illum(j), P(i,j), c(t(i), illum(j), pint(i))),$$

where

$$P(i,j) = (TRUE, bill(i), pint(i), ltime(i), x, z).$$

The network itself is a tree and also a directed acyclic graph if we disregard the end node. Each path from s to level n through the network will correspond to a feasible schedule. The totalcost of a path is simply the sum of the costs of the individual nodes on that path. The shortest path (according to cost) through the network from s to level n will satisfy b3 (above) and give us the optimal schedule that we want. The tree is made up of n levels (recall that n is the number of threats). Each of these levels correspond to the appending of an interval to the schedule for the terminal illuminators.

From s (the start node), we build the nodes in level 1 of the network. The following actions are performed.

For each i = 1,2,...,n and for each j = 1,2,...,m we compute P(i,j). If the first component is TRUE then a node

$$(t(i), illum(j), P(i,j), c(t(i), illum(j), pint(i))$$

is placed into level 1. Otherwise, no node is constructed.

Note that the cost for the node is computed only if the plan function is successful in planning this engagement. Each path from s to t will pass though exactly one node in level 1. Hence each feasible schedule will include exactly one of these nodes.

Now, by induction, we continue the node construction process. Suppose that levels 1,...,k have been constructed. We construct level k+1 as follows.

1. For each node N at level k, we expand that node by connecting N to all of its children. These children constitute level n. They are found by the following action.
2. For each i = 1,2,...,n, if t(i) occurs as a threat in a node on the unique path from N

to s we do nothing. If t(i) does not occur on this path then for each j = 1,2,...,m we compute P(i,j). If the first component is TRUE then the node

(t(i), illum(j), P(i,j), c(t(i) illum(j), pint(i))

is placed as a child of N.

The network is complete when n levels have been constructed. Each of the nodes in level n is connected to the end node t. The problem mathematically is to find the shortest path from s to t through the network. The fact the solution space (disregarding t) forms a tree allows us to generate feasible schedules easily. We start with any node in level 1 and find a child of that node. We continue finding children until we reach level n. Figure 1 below shows the generation of solutions for a 2 illuminator, 3 threat case. For simplicity, in this example, we assume that Plan always returns TRUE. Nodes are denoted by listing the illuminator, the threat and the level of the node.

All threats are placed on the illuminator schedules in the order shown by the levels. That is, the pairing in level 1 is formed first, followed by the pairing from level 2, and so forth. In Figure 1, suppose that the required illumination times



Figure 1. Generation of schedules in a 2 illuminator, 3 threat example. In the left hand schedule, threat 1 is put on illuminator 1's schedule at the earliest possible time, then threat 2 is put on illuminator 2's schedule and finally threat 3 is placed on the schedule for illuminator 1 at the earliest possible time assuming that threat 1 is already scheduled there. To generate the right hand schedule, we choose two of the nodes in the left hand schedule at random (the asterisks indicate the choices). Then the threats for these two are interchanged. The illuminators are changed with probability 0.5. In this example the illuminators remained the same.

for the threats are given by delta(1) = 5, delta(2) = 10, and delta(3) = 4. Suppose further that illuminator 1 is already busy between the times of 5 and 10 and illuminator 2 was previously scheduled between times 0 and 5. Then the left hand schedule from Figure 1 could be represented as shown in Figure 2.

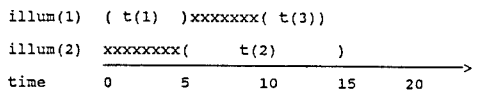


Figure 2. Another representation of the left hand illuminator schedule from Figure 1. Here the required illuminator times are taken as delta(1) = 5, delta(2) = 10 and delta (3) = 4. It is also assumed that illuminator 1 was previously scheduled between times 5 and 10 as shown by the x's and illuminator 2 was busy between times 0 and 5.

Threat 1, (shown as t(1)) is first inserted on the schedule for illuminator 1. The period from time 0 to time 5 just exactly gives us the required 5 units of illumination time. Threat 2 (shown as t(2)) is then inserted from time 5 to time 15 on illuminator 2. Threat 3 (shown as t(3)) then must be placed on illuminator 1 from 10 to 14.

Figure 3 shows the right hand schedule in Figure 1 displayed on a time line using the same required illumination times. First threat 3 is placed on the schedule for illuminator 1. Since threat 3 requires 4 units of illumination, it can be placed in illuminator 1's schedule prior to the busy period. Threat 2 is thus scheduled from 5 to 15 on illuminator 2 and finally threat 1 is placed on illuminator 1 from 10 to 15.

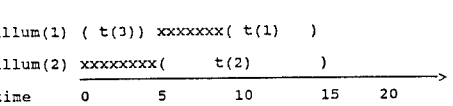


Figure 3. The right hand schedule from Figure 1. The same deltas are used as in Figure 2.

Figure 1 also illustrates the method of generating new schedules from old ones. Two nodes on the old schedule are chosen at random. These choices are shown with asterisks in Figure 1. The threats for the nodes are interchanged. Although the illuminators were not changed in Figure 1, they can also change in the creation process. A random number is generated and compared to a preset threshold which in this case is 0.5. If the random number is less than the threshold, the illuminator is changed to another legal value. This technique gives us a heuristic method of generating new schedules from old ones.

3.2 Comparing Schedules

We compare schedules using the minimum of a cost function. For example let us consider the linear cost function described in Section 3.0:

$$c(t(i), illum(j), pint(i)) = d(i) * (pint(i) - fe(i))$$

where d(i) is the depth of fire and fe is the first possible time for engagement. Suppose, in Figure 2 & 3, that the engageability intervals are:

$$eng(1) = (0, 20),\ eng(2) = (3, 18) \text{ and } eng(3) = (0, 15)$$

while the depth of fire is given by:

$$d(1) = 2,\ d(2) = 1, \text{ and } d(3) = 3.$$

Then the cost values for each of the schedules shown in Figures 2&3 are computed by:

$$totalcost = 2*(5 - 0) + 1*(15 - 3) + 3*(14 - 0) = 64$$

for Figure 2 and

$$totalcost = 2*(15 - 0) + 1*(15 - 3) + 3*(4 - 0) = 54$$

for Figure 3. Thus the schedule in Figure 3 is considered better than the one in Figure 2. The construction of the cost function is an important part of this algorithm but is not pertinent to the current discussion. It is described in other literature [Boyer, et al, 1990].

3.3 Parallel Tabu Search for the Illuminator Scheduling Problem

We can now use the Tabu Search technique to solve the Illuminator Scheduling Problem. First we partition the search space among the processors using the permanent tabu. This is done by restricting the level 1 nodes that each processor can use. For example, with 2 processors, the permanent tabu list for processor 1 could be { (2, 1, 1), (2, 2, 1), (2, 3, 1) } using the notation of Figure 1 in the 2 illuminator and 3 threat example. This means that processor 1 only considers schedules which use illuminator 1 in level 1. Processor 2 would then have the permanent tabu list { (1, 1, 1), (1, 2, 1), (1, 3, 1) }. This assures that the processors are searching different areas of the solution space. The (temporary) tabu list can be of any length but a length of 2 is good for the 2 illuminator, 3 threat example. As nodes are changed to get new schedules, the old nodes are put on the tabu list. Before any change is made, the prospective new nodes are compared to those on the tabu list and the permanent tabu list. The change is not completed if any of the new nodes would produce nodes on either tabu list unless that change produces a smaller cost than the best one to date. Glover [Glover, 1989]

has suggested the use of an aspiration list. This aspiration list contains conditions that can override the tabu list. A new schedule may be produced even if some of the nodes are on the tabu list (not the permanent tabu list) if the conditions in the aspiration list are satisfied. In this case, our aspiration list contains only one condition. If the new schedule has a smaller totalcost than the best one to date, it is constructed.

In general, the algorithm proceeds as follows:

x0) generate any solution that does not violate the permanent tabu list; record it as the best one found to date;
x1) generate a new solution from the old one;
x2) if the new solution has any nodes on the permanent tabu list then goto x1);
x3) if the new solution has any nodes on the tabu list and the condition of the aspiration list is not satisfied then goto x1);
x4) compare the new solution to the best old one; If the new solution is better record it as the best found to date; Put nodes that were changed on the tabu list;
x5) make this new solution the old one and goto x1).

The algorithm terminates when time expires. Each processor reports the best solution that it found. The best of these is reported as the nearest to the optimal. In many cases it will be the global optimal solution.

4.0 SUMMARY

The example of Tabu Search which we presented illustrates the potential usefulness of the approach, especially when combined with parallel processing. Glover [Glover, 1990] has described many more applications such as mixed integer programming and multi-variable decision problems. As the number of applications for this new search technique continues to grow, we learn more and better ways to apply it. This paper serves the purpose of introducing it to those that solve time constrained scheduling problems.

5.0 REFERENCES

Bertsekas D. and Tsitsiklis, J (1989), Parallel and Distributed Computation: Numerical Methods, Prentice-Hall, Englewood Cliffs, N. J.

Boyer, D., Price, E., and Perry, E. L (1990), "Force Level Control in Naval Anti-Air Warfare", Proceeding of the 1990 IEEE Symposium on Command and Control, Ford Aerospace Corporation, 9970 Federal Drive, Colorado Springs, Colorado.

Davis, L. (1987), Genetic Algorithms and Simulated Annealing, Pitman Publishing, London.

Garey, M. R. and Johnson, D. S. (1979), Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman and Company, New York.

Glover, F. (1988), "Tabu Search", CAAI Report 88-3, Center for Applied Artificial Intelligence, University of Colorado at Boulder, Boulder, CO.

Glover, F. (1989), "Tabu Search, Part I", ORSA Journal on Computing, Vol. 1, No. 3, pp. 190-206.

Glover, F. (1990), "Tabu Search, Part II", ORSA Journal on Com-

puting, Vol. 2, No. 1, pp. 4-32.

Hillier, Frederick S. and Lieberman, Frederick J. (1980), Introduction to Operations Research, Third Edition, Holden-Day, San Francisco.

Managing Resource Allocation in Multi-Agent Time-Constrained Domains

Katia Sycara, Steve Roth, Norman Sadeh, Mark Fox

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present an approach to perform asynchronous, opportunistic, constraint-directed search in multi-agent time-bound, and resource limited domains. Such domains are extremely complex because of the presence of temporal and resource constraints that give rise to tightly interacting subproblems. In a distributed environment lacking a global system view and global control, the complexity increases further. Our approach relies on a set of *textures* of the problem space being searched. Textures provide a probabilistic, graph theoretic definition of the complexity and importance of decisions in the local problem space of each agent. In other words, they provide sophisticated local control. In addition, textures provide good predictive measures of the impact of local decisions on system goals. As a result, textures can be used to make control decisions that significantly reduce the amount of search required to solve complex distributed problems. We explore the utility of the approach in the context of cooperative multi-agent job-shop scheduling.

1. Introduction

In this paper we present mechanisms to enable efficient distributed search for multi-agent, time-bound and resource-limited problems. Such problems are characterized by the presence of temporal precedence constraints and resource constraints. These constraints result in conflicts over the use of shared resources and make the local decisions of distributed agents highly interdependent and interacting. Our investigation is conducted in the domain of job-shop scheduling. Our work addresses concerns in three research areas: (1) managing resource allocation in multi-agent planning, (2) constraint satisfaction, and (3) job-shop scheduling. Research in multi-agent planning has primarily focused on problems where agents contend only for computational resources, such as computer time and communication bandwidth (e.g., [Cammarata 83, Durfee 87a]). In most real world situations, however, allocation of (non-computational) resources that are needed by a planner to carry out actions in a plan is of central concern. Conry [Conry 86] has investigated (non-computational) static resource allocation not involving temporal constraints. The constraint satisfaction research community has investigated the efficiency of heuristics for incrementally building a solution to a constraint satisfaction problem by instantiating one variable after another within a single agent setting [Haralick 80, Mackworth 85, Purdom 83, Dechter 88]. Job-shop scheduling has been the subject of intense investigation by both Operations Research and AI communities (e.g., [Smith 85, Ow et. al. 88, Baker 74, French 82, Rinnooy Kan 76]). With few exceptions [Parunak 86, Smith&Hynynen 87], there has been almost no research in distributed scheduling. Prosser [Prosser 89] has investigated job-shop scheduling within a hierarchical distributed architecture where the high level agent has a global view and can act as conflict arbiter. In our system, the agents form a *heterarchy*, where no agent has a global view of the problem and actions of others. We provide mechanisms both for conflict avoidance and conflict resolution.

Our model enables a set of agents to structure their individual agent problem space and focus their attention during search so as to optimize decisions in the global search space. The beneficial effects of sophisticated local control on the coordination of distributed problem solving have been recognized by prior research [Durfee 87a, Durfee 87b]. Our approach, based on problem space *textures* [Fox 89], allows agents to make rapid, intelligent local decisions without the need of excessive information exchange or the availability of detailed models of each other's problem solving activities. Our hypothesis is that these textures provide good predictive measures of the impact of local decisions on system goals and constitute abstract information summaries of expectations concerning the decision making activities of other agents. Basing local decisions on such predictive measures is very important in distributed problem solving by opportunistic scheduling agents. Since the agents operate in an asynchronous and opportunistic manner, and since each local decision

[1]This research has been supported, in part, by the Defense Advance Research Projects Agency under contract #F30602-88-C-0001, and in part by grants from McDonnell Aircraft Company and Digital Equipment Corporation.

interacts with subsequent decisions of other agents, each agent must predict and take into consideration in its local decision making the future resource needs and problem solving behavior of other agents.

2. The Distributed Scheduling Problem

The scheduling task can be described as assigning resources to the activities present in a plan over time in a consistent manner, i.e., so as to avoid the violation of resource and precedence constraints. In our model, a group of autonomous opportunistic schedulers build a schedule in order to synchronize their activities to avoid and resolve conflicts. The schedule is built in a cooperative fashion through local computation and communication. There is no single agent with a global system view, nor any agents whose role is coordination. In distributed job-shop scheduling, each agent has a set of orders to schedule on a given set of resources. Each order consists of a set of activities (operations) to be scheduled according to a process plan which specifies a partial ordering among these activities. Additionally, an order has a release date and a due date. Each activity also requires one or several resources, for each of which there may be one or several substitutable resources. There is a finite number of resources available in the system. Some resources are only required by one agent, and are said to be *local* to that agent. Other resources are *shared*, in the sense that they may be allocated to different agents at different times[2].

We distinguish between two types of constraints: activity precedence constraints and capacity constraints. The activity precedence constraints together with the order release dates and due dates restrict the set of acceptable start times of each activity. Capacity constraints restrict the number of activities that can be allocated to a resource at one time. Typically the limited capacity of the resources induces interactions between orders competing for the possession of the same resource at the same time. In such an environment, schedules are constructed in an *incremental fashion*. Agents make local decisions about assignments of resources to particular activities at particular time intervals and a complete schedule for an order is formed by incrementally merging partial schedules for the order. If the merging of partial schedules results in constraint violations, the resulting schedule is infeasible.

Distributed scheduling has the following characteristics:

- To achieve global solutions, agents must make consistent allocations of resources needed to perform system activities. Conflicts in the system arise due to contention over optimal allocation of limited capacity shared resources.
- Because of conflicts over shared resources it is impossible for each agent to optimize the scheduling of its assigned orders using only local information.
- Due to limited communication bandwidth, it is not possible to exchange detailed constraint information during problem solving.
- All the given orders have to be scheduled. In other words, agents cannot drop any local goals. In addition, constraints cannot be relaxed (e.g., precedence constraints among the operations of an order, resource capacity constraints, and due dates).
- Because of the tightly interacting nature of scheduling decisions, an agent's problem solving context is rapidly changing. Moreover, an agent's decisions can produce constraint violations for other agents which may lead to backtracking. Backtracking can have major ripple effects on the multi-agent system since it may invalidate resource reservations that other agents have made.

A consequence of the above characteristics is that agents need methods to deal efficiently with incomplete information and a rapidly changing problem solving context. In addition, agents must maintain coherent behavior [Durfee 87b] in a *heterarchical* setting. To address these requirements, our approach gives the agents mechanisms to enable them to accomplish the following: (1) predict and evaluate the impact of local decisions on global system goals, (2) develop and communicate in a concise form robust expectations and predictions about the resource needs and decision-making behavior of other agents, (3) avoid and resolve conflicts over resources and time intervals, and (4) help focus the attention of the agents opportunistically on parts of their search space where it is expected that good solutions, in terms both of schedule quality and minimal interactions, will be found. These mechanisms, based on problem textures, result in search and communication efficiency.

3. Constrained Heuristic Search

Our approach to scheduling relies on the combination of local constraint propagation techniques with texture-based heuristic search. We have developed a formal model of this search mechanism which we call Constrained Heuristic Search (CHS) [Fox 89]. CHS provides a methodology for solving Constraint Satisfaction Problems (CSPs) and

[2]This model mirrors actual factory floor situations where the factory is divided into work areas that might share resources, such as machines, fixtures and operators in order to process orders.

Constrained Optimization Problem (COPs). A CSP is defined by a set of variables, each with a predefined domain of possible values, and a set of constraints restricting the values that can simultaneously be assigned to these variables [Montanari 71, Mackworth 77, Dechter 88]. A solution to a CSP is a complete set of assignments that satisfies all the problem constraints. COPs are CSPs with an objective function to be optimized. The general CSP is a well-known NP-complete problem [Garey 79]. There are however classes of CSPs and COPs that do not belong to NP, and for which efficient algorithms exist. The CHS methodology is meant for those CSPs/COPs for which there is no efficient algorithm. A general paradigm for solving these problems consists in using Backtrack Search (BT) [Golomb 65, Bitner 75]. BT is an enumerative technique that incrementally builds a solution by instantiating one variable after another. Each time a new variable is instantiated, a new search state is created that corresponds to a more complete partial solution. If, in the process of building a solution, BT generates a partial solution that it cannot complete (because of constraint incompatibility), it has to undo one or several earlier decisions. Partial solutions that cannot be completed are often referred to as deadend states (in the search space).

Because the general CSP is NP-complete, BT may require exponential time in the *worst-case*. CHS provides a methodology to reduce the *average* complexity of BT by interleaving search with *local constraint propagation* and the computation of *texture-based heuristics*. Local constraint propagation techniques are used to prune the search space from alternatives that have become impossible due to earlier decisions made to reach the current search state. By propagating the effects of earlier commitments as soon as possible, CHS reduces the chances of making decisions that are incompatible with these earlier commitments [Mackworth 85]. Typically, pruning the search space can only be done efficiently on a local basis [Nadel 88]. Hence local constraint propagation techniques are not sufficient to guarantee backtrack-free search. In order to avoid backtracking as much as possible as well as reduce the impact of backtracking when it cannot be avoided, CHS analyzes the pruned problem space in order to determine critical variables, promising values for these variables, promising search states to backtrack to, etc. The results of this analysis are summarized in a set of textures that characterize different types of constraint interactions in the search space. These textures are operationalized by a set of heuristics to decide which variable to instantiate next (so-called variable ordering heuristics), which value to assign to a variable (so-called value ordering heuristics), which assignment to undo in order to recover from a deadend, etc.

In the factory scheduling domain, variables are activities whose values are reservations consisting of a start time and a set of resources (e.g. a human operator, a milling machine, and a set of fixtures). Local constraint propagation techniques are used to identify reservations that have become unavailable for an unscheduled activity due to the scheduling of another activity (e.g. a resource that has been allocated to an activity over some time interval, or a start time that has become infeasible due to the scheduling of an earlier activity in a process plan). Within this context, texture-based heuristics are concerned with such decisions as which activity to schedule next, which reservation to assign to an activity, which reservation assignments to undo if the current partial schedule cannot be completed.

4. Distributed CHS Scheduling

The model concerns a set of scheduling agents, $\Gamma=\{\alpha, \beta, ...\}$. Each agent α is responsible for the scheduling of a set of orders $O^{\alpha}=\{o_1^{\alpha},...,o_{N_\alpha}^{\alpha}\}$. Each order o_l^{α} consists of a set of activities $A^{l\alpha}=\{A_1^{l\alpha},...,A_{n_{l\alpha}}^{l\alpha}\}$ to be scheduled according to a process plan (i.e. process routing) which specifies a partial ordering among these activities (e.g. $A_p^{l\alpha}$ BEFORE $A_q^{l\alpha}$). Additionally an order has a release date and a latest acceptable completion date, which may actually be later than the ideal due date. Each activity $A_k^{l\alpha}$ also requires one or several resources $R_{ki}^{l\alpha}$ $(1 \le i \le p_k^{l\alpha})$, for each of which there may be one or several alternatives (i.e. substitutable resources) $R_{kij}^{l\alpha}$ $(1 \le j \le q_{ki}^{l\alpha})$. There is a finite number of resources available in the system. Some resources are only required by one agent, and are said to be *local* to that agent. Other resources are *shared*, in the sense that they may be allocated to different agents at different times.

We distinguish between two types of constraints: activity precedence constraints and capacity constraints. The activity precedence constraints together with the order release dates and latest acceptable completion dates restrict the set of acceptable start times of each activity. The capacity constraints restrict the number of activities that a resource can be allocated to at any moment in time to the capacity of that resource. For the sake of simplicity, we only consider resources with unary capacity in this paper. Typically the limited capacity of the resources induces interactions between orders competing for the possession of the same resource at the same time. These interactions can take place either between the order of a same agent or between the orders of different agents.

With each activity, we associate preference functions that map each possible start time and each possible

resource alternative onto a preference. These preferences [Fox 83, Sadeh 88] arise from global organizational goals such as reducing order tardiness (i.e. meeting due dates), reducing order earliness (i.e. finished goods inventory), reducing order flowtime (i.e. in-process inventory), using accurate machines, performing some activities during some shifts rather than others, etc. In the cooperative setting assumed in this paper, the sum of these preferences over all the agents in the system and over all the activities to be scheduled by each of these agents defines a common objective function to be optimized. The sum of these preferences over all the activities under the responsibility of a single agent can be seen as the agent's local view of the global objective function. In other words, the global objective function is not known by any single agent. Furthermore, because they compete for a set of shared resources, it is not sufficient for an agent to try to optimize his own local preferences. Instead, agents need to consider the preferences of other agents when they schedule their activities. This is accomplished via a communication protocol described in section 6.

4.1. Activity-based Scheduling

In our model we view each activity $A_k^{l\alpha}$ as an aggregate *variable* (or vector of variables). A *value* is a reservation for an activity. It consists of a start time and a set of resources for that activity (i.e. one resource $R_{kij}^{l\alpha}$ for each resource requirement $R_{ki}^{l\alpha}$ of $A_k^{l\alpha}$, $1 \le i \le p_k^{l\alpha}$).

Each agent asynchronously builds a schedule for the orders he has been assigned. This is done incrementally by iteratively selecting an activity to be scheduled and a reservation for that activity. Each time a new activity is scheduled, new constraints are added to the agent's initial scheduling constraints that reflect the new activity reservation. These new constraints are then propagated (local constraint propagation step). If an inconsistency (i.e. constraint violation) is detected during propagation, the system backtracks. Otherwise the scheduler moves on and looks for a new activity to schedule and a reservation for that activity. The process goes on until all activities have been successfully scheduled.

If an agent could always make sure that the reservation that he is going to assign to an activity will not result in some constraint violation forcing him or other agents to undo earlier decisions, scheduling could be performed without backtracking. Because scheduling is NP-hard, it is commonly believed that such look-ahead cannot be performed efficiently. The most efficient constraint propagation techniques developed so far [LePape&Smith 87] for scheduling do not guarantee total consistency. In other words the reservation assigned by an agent to an activity may force other agents or the agent himself to backtrack later on[3]. Consequently it is important to focus search in a way that reduces the chances of having to backtrack and minimizes the work to be undone when backtracking occurs. This is accomplished via two techniques, known as *variable* (i.e. activity) and *value* (i.e. reservation) ordering heuristics.

The variable ordering heuristic assigns a *criticality measure* to each unscheduled activity; *the activity with the highest criticality is scheduled first.* The criticality measure approximates the likelihood that the activity will be involved in a conflict. The only conflicts that are accounted for in this measure are the ones that cannot be prevented by the constraint propagation mechanism. By scheduling his most critical activity first, an agent reduces his chances of wasting time building partial schedules that cannot be completed (i.e. it will reduce both the frequency and the damage of backtracking). The value ordering heuristic attempts to leave enough options open to the activities that have not yet been scheduled in order to reduce the chances of backtracking. This is done by assigning a *goodness* measure to each possible reservation of the activity to be scheduled. Both activity criticality and value goodness are examples of *texture measures.* The next two paragraphs briefly describe both of these measures[4].

4.1.1. Variable Ordering

Each agent's constraint propagation mechanism is based on the technique described in [LePape&Smith 87]. It always ensures that unscheduled activities within an order can be scheduled without violating activity precedence constraints. This is not the case however for capacity constraints: there are situations with insufficient capacity that may go undetected by this constraint propagation technique. Accordingly a critical activity is one whose resource requirements are likely to conflict with the resource requirements of other activities. [Sadeh 88, Sadeh 89] describes a technique to identify such activities. The technique starts by building for each unscheduled activity a probabilistic *activity demand.* An activity $A_k^{l\alpha}$'s demand for a resource $R_{kij}^{l\alpha}$ at time t is determined by the ratio of reservations that remain possible for $A_k^{l\alpha}$ and require using $R_{kij}^{l\alpha}$ at time t over the total number of reservations that

[3]This is already the case in the centralized version of the scheduling problem. Because of the additional cost of communication it is even more so in the distributed case.

[4]For a more complete description of these measures, the reader is referred to [Sadeh 90].

remain possible for $A_k^{l\alpha}$. Clearly activities with many possible start times and resource reservations tend to have smaller demands at any moment in time, while activities with fewer possible reservations tend to have higher ones. In a second step, each agent aggregates his activity demands as a function of time, thereby obtaining his *agent demand*. This demand reflects the need of the agent for a resource as a function of time, given the activities that he still needs to schedule[5]. Finally, for each shared resource, agent demands are aggregated for the whole system thereby producing *aggregate* demands that indicate the degree of contention among agents for each of the (shared) resources in the system as a function of time. Time intervals over which a resource's aggregate demand is very high correspond to violations of capacity constraints that are likely to go undetected by the constraint propagation mechanism. The contribution of an activity's demand to the aggregate demand for a resource over a highly contended-for time interval reflects the reliance of the activity on the possession of that resource/ time interval. It is taken to be the *criticality of the activity*.

To choose the next activity to schedule, each agent looks among the resource/ time intervals that he may need and selects the one with highest aggregate demand. He then picks his activity with the highest contribution (i.e. highest criticality) to the aggregate demand for that resource/time interval. In other words, each agent looks for the resource/time interval over which he has some demand that is the most likely to be involved in a capacity constraint violation. He then picks his activity with the highest probability of being involved in the conflict.

4.1.2. Value Ordering

Once an agent has selected an activity to schedule next, it must decide which reservation to assign to that activity. Here several strategies can be considered. In particular, we distinguish between two extreme strategies: (1) a least constraining value ordering strategy (LCV) and (2) a greedy value ordering strategy (GV). Under LCV an agent will select the reservation that will be the least constraining both to itself and to other agents. LCV is a mechanism for *avoiding conflicts* over resources and over time intervals. This heuristic can be viewed as resulting in *altruistic* behavior on the part of an agent. Under the GV strategy, an agent can select reservations based solely on its local preferences, irrespectively of its own future needs as well as those of other agents. This heuristic results in *egotistic/myopic* behavior on the part of the agent. In this paper, we report experimental results obtained using the LCV value ordering strategy.

[5]Notice that, an agent's demand at some time t for a resource is obtained by simply summing the demand of all his unscheduled activities at time t. Because these probabilities do not account for limited capacity, their sum may actually be larger than 1

5. Using Textures for Decentralized Scheduling

This section describes additional theoretical concerns and new mechanisms that arose in our application of the texture approach to decentralized, multiagent, resource-constrained scheduling. The issues that we addressed include:

- scheduling with incomplete information about the intentions and future behavior of other agents.
- scheduling with uncertain/changing information (i.e. even when detailed information regarding other agents' intentions is communicated, this information is not stable over time, since agents are scheduling asynchronously), and
- scheduling *without* the help of coordinating agents for avoiding conflicts and achieving global goals.

The following subsections describe our approach to addressing these issues.

5.1. Incomplete information

In a multi-agent system, complete information is unavailable to each agent about the constraints, partial plans/schedules and heuristic analyses of other agents. Incomplete information results because of limitations in the amount of inter-agent communication that can reasonably occur. Hence, some level of summarization and abstraction is needed. In our approach, summarization information is expressed in terms of the texture measures that have been effective for centralized problems. Specifically, we represent an agent's intentions with respect to resource use in terms of that *agent's demand density* for the resource for different time intervals. All agent densities are further abstracted to produce an *aggregate density*, which represents the system-wide expectation for resource utilization over time.

An important outcome of this approach is the ability to efficiently communicate those aspects of an agent's partial schedules which are most relevant to each of the other agents in a system, without the need to explicitly determine relevance. An element of a partial schedule is relevant to another agent if it influences the agent's expectations regarding the demand for resources the agent requires. Since the effects of most scheduling decisions indirectly

influence the computation of an agent's expected demand for shared resources, these implicitly include an abstraction of all relevant decision making elements.

5.2. Rapidly changing information

The continuous, asynchronous behavior of agents can reduce the validity of information they exchange, regardless of how complete that information may be. Therefore, an agent cannot depend on the certainty of information when it elects to use it, because other agents' decisions interact with its already constructed partial schedules as well as with its future scheduling decisions thus producing new expectations. In addition, because of the associated communication costs, agents cannot afford to communicate, update and evaluate information with every change that occurs. Hence the information communicated must remain predictive robustly in the face of communication lags.

There are several aspects of the texture approach that address the problem of rapid information obsolescence in asynchronous, multi-agent systems: First, texture measures produce relatively accurate early predictions of agent behavior, as long as expectations are communicated by all agents at the initiation of scheduling and constraints remain constant. Second, the uniform representation of expectations as densities and the incremental nature of activity scheduling allows changes in expectations to be incorporated as soon as they are received. Third, agents can monitor their current expectations to determine when these have changed significantly from those that were last communicated.

In the multi-agent system, other agents can make reservations throughout an agent's search, making it difficult to determine which set of previous reservations were responsible for a constraint violation when it is eventually detected. The task facing an agent at this point is to find the last set of reservations it made which, together with those made by other agents, does not violate constraints. A simple backtracking procedure will eventually find this state, but is extremely inefficient.

In order to deal with this problem, we have developed a variation of backjumping [Dechter 89] for uncertain, multi-agent environments. In our approach, backjumping involves iteratively undoing each activity's scheduled reservation and determining whether constraint violations remain, until the set of acceptable activity reservations has been partitioned. No alternative values are tried for any one activity until this set has been determined. This procedure avoids the inefficient testing of alternate values for variables when, in fact, violations already exist for values assigned to previously addressed variables. Our version of backjumping locates the appropriate search point with computation that is just a linear function of the number of variables traversed, a tremendous saving over chronological backtracking.

5.3. Absence of explicit coordination

Coordination within the texture approach to multi-agent scheduling is achieved through mutual acceptance and adherence to shared policies of decision-making. In our system, the goal of supporting other agents' attempts to achieve a solution to their portions of the global scheduling problem is realized through three policies. First, agents use information about other agents' expectations to avoid over-constraining them through the application of LCV value ordering heuristics. Second, reservations for resources are granted without contest when requested by an agent (i.e. reservations granted on a first-come, first served basis). Reservations are also surrendered promptly by agents if they decide not to use them as a result of local constraint violations. Third, once an agent has made a reservation, it is not required to surrender it i.e., no provision is made for one agent to request another to backtrack. An important principle is that all agents assume that the global good is best realized through the application of these policies and therefore, do not depart from them to maximize local objective functions.

6. A Communication Protocol for Distributed Scheduling

The agents make decisions using local available knowledge as well as information communicated by the other agents. In our model, resources are passive objects that are *monitored* by active agents. Each resource has a monitoring agent and each agent monitors one or more resources. Thus, monitoring responsibility is distributed among many agents. Monitoring resources does not give an agent either a global view or preferential treatment concerning the allocation of the monitored resources but is simply a mechanism that enables agents to perform load balancing in bookeeping efforts and efficient detection of capacity constraint violations. Since there is no single agent that has a global system view, the allocation of the shared resources must be done by collaboration of the agents that require these resources (one of which is the monitoring agent).

The multi-agent communication protocol is as follows:

1. Each agent determines required resources by checking the process plans for the orders it has to schedule. It sends a message to each monitoring agent informing it that it will be using shared resources.

2. Each agent calculates its demand profile for the resources (local and shared) that it needs.

3. Each agent determines whether its new demand profiles differ significantly from the ones it sent previously for shared resources. If its demand has changed, an agent will send it to the monitoring agent.

4. The monitoring agent for each resource combines all *agent demands* when they are received and communicates the *aggregate demand* to all agents which share the resource[6].

5. Each agent uses the most recent aggregate demand it has received to find its most critical resource/time-interval pair and its most critical activity (the one with the greatest demand on this resource for this time interval). Since agents in general need to use a resource for different time intervals, the most critical activity and time interval for a resource will in general be different for different agents. The agent communicates this reservation request to the resource's monitoring agent and awaits a response.

6. The monitoring agent, upon receiving these reservation requests, checks the calendar of the resource it is monitoring to find out whether the requested intervals are available. There are two cases:

- If the resource is available for a requested time interval, the monitoring agent of the resource (a) communicates "Reservation OK" to the requesting agent, (b) marks the reservation on the resource calendar, and (c) communicates the reservation to all concerned agents (i.e. the agents that had sent positive demands on the resource).
- If the resource had already been reserved for the requested interval, the request is denied. The agent whose request was denied will then attempt to substitute another reservation, if any others are feasible, or otherwise perform backjumping.

7. Upon receipt of a message indicating its request was granted, an agent will perform consistency checking to determine whether any constraint violations have occurred. If none are detected, the agent proceeds to step 2. Otherwise, backjumping occurs with undoing of reservations until a search state is reached which does not cause constraint violations. Any reservations which were undone during this phase are communicated to the monitor for distribution to other agents. After a consistent state is reached, the agent proceeds to step 2.

The system terminates when all activities of all agents have been scheduled i.e. when all demands on resources become zero. In this version of the protocol we assume that reservations are not changed because of backtracking.

[6]With the exception of the first time demands are exchanged, agents do not wait for aggregate demands to be computed and returned prior to continuing their scheduling operations (although they can postpone further scheduling if desired).

7. Experimental Results

The main goal of our experiments was to determine the feasibility of the texture approach to multi-agent scheduling across a number of different scheduling experiments and across a variety of system configurations. We have developed a testbed and performed experiments with 1-, 2- and 3-agent configurations. The experiments were run asynchronously on a number of machines corresponding to each of the agent configurations. In addition, we wanted to test particular mechanisms and parameters that influence system performance. In particular, our experiments considered:

- the effects of agents' incomplete knowledge of each other's plans (i.e. the robustness of texture measures when aggregated across multiple agents and with the resulting loss of detailed information),
- the effects of rapidly changing expectations on performance (i.e. the robustness of these measures with respect to delays in the communication of densities),
- the consequences of asynchronous scheduling (e.g., asynchronous use of variable-ordering strategies) without external coordination.

The experiments summarized here were created from problems found to be difficult in previous research on centralized scheduling [Sadeh 89] and they reflect system performance with respect to search efficiency rather than schedule optimality. We selected problems on which more traditional constraint satisfaction approaches performed poorly (e.g. Purdom's dynamic search rearrangement technique [Sadeh 89, Purdom 83]). The problems were also selected and distributed across the agents in a way that maximized *resource coupling* within orders and across agents. The problems were constructed so that a change in reservation for any activity or resource would influence expectations for every other.

All experimental problems were selected so that orders could be distributed evenly between two agents, all resources were shared by the two agents (high inter-agent resource coupling), every order used all resources (high intra-order resource coupling), and problems ranged from 40 - 100 activities.

Over 100 experiments were run in order to vary several properties of each problem. The asynchrony in the system prevents exact replication of experiments. So, we repeated each experimental run a minimum of three times. If different runs of the same experiment produced wide

variations in the results, we repeated the experiment five times. The reported results (see figure) are the average of these runs. In each case, the dependent variable was the efficiency with which the scheduling system found a solution. Efficiency is expressed in terms of the total number of states needed to reach a solution. For example, for a problem with 40 activities, the minimum number of states needed to assign a reservation to each activity is 40. Every reservation that needed to be redone added an additional state to the total. This allowed comparing a 40-activity 1-agent problem to a pair of 20-activity problems solved simultaneously by 2 agents, or a 10, 15, 15 split of the 40 activities among three agents. There were 5 resources (all shared among all the agents). These resources were used by 8 orders, each having 5 activities.

Problem versions differed in several ways. First, to establish a baseline, we created a 1-agent system, which was similar to the 2-agent and 3-agent systems in every way, except that the aggregate densities were constructed from a single agent. This was still different from the original centralized system in that decisions were based on an *abstract aggregate demand profile* that did not include detailed information about the number of activities which contributed to the densities. Furthermore, we varied the frequency with which the aggregate was computed, thereby isolating the effect of uncertain expectations caused by infrequent and delayed communication of densities in the 2-agent and 3-agent systems.

Specifically, we implemented several simplified versions of the heuristic used by agents to determine when to communicate their changed densities. In the *minimum* delay condition, a single reservation on *any resource by any agent* initiated the exchange of densities for all resources. In the *increased* delay conditions, densities were exchanged *for each resource independently*, whenever N reservations were made on it, where N = 1, 3, and 5. This provided a way to observe the effects of wide ranges in communication bandwidth in the 2-agent and 3-agent systems and comparable conditions in a 1-agent system.

Another version of the 1-agent system was created which used a *semi-random* version of the variable-ordering heuristic. The goal was to isolate and assess the effects of less accurate variable ordering that might occur in a multi-agent system. Recall that variable ordering is performed in parallel in a multi-agent system (each agent selects the best activity to schedule from its subset of all activities which require a critical resource). Agents do not coordinate the selection of activities to schedule to ensure that the globally most critical ones are scheduled first. As a result, variable ordering is probably less effective than in a 1-agent system. The semi-random heuristic still selects activities to schedule from those which require the most critical resource/time-interval (which narrows the selection to a maximum of 20% of the activities in these problems). However, it then randomly selects from this subset, instead of selecting the activity with the greatest demand for the critical resource. Relative to completely random variable ordering, the semi-random condition is still highly selective in that only activities which use the most critical resource/time interval are considered. In fact, we found that random variable ordering resulted in terrible performance, even in the 1-agent case. Solutions were not found in over 500 states.

Two system versions were created to compare the use of backtracking and backjumping search techniques. As expected, the use of a backjumping strategy substantially reduced the search in the 2- and 3-agent systems. The results presented in the Figure are results of the backjumping version. The reported results are for a representative group of 40-activity experiments (8 orders, 5 activites per order, and 5 shared resources). The four curves represent the effects of increasing the delay (from 0 to 5) prior to initiating creation of aggregate demand densities for 1-, 2- and 3-agent configurations and for a 1-agent case with a semi-random variable ordering strategy (labelled 1-agent SR variable ordering in the figure).

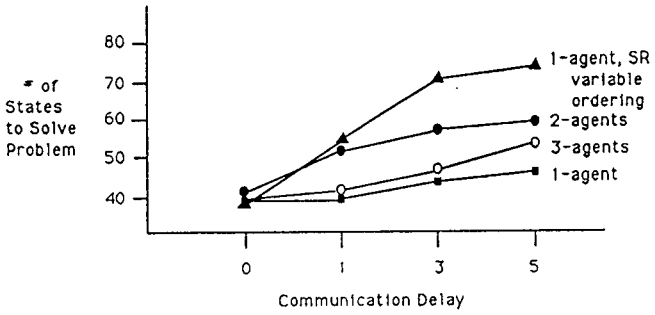


Figure 7-1: Experimental Comparisons of Distributed Scheduling Systems

The first important observation is that the use of abstracted texture measures was sufficient to allow near perfect performance (solving the problem in 40 states) when the texture information was updated frequently (minimum delay conditions, expressed as 0 on the x-axis) in all experimental configurations. This matches performance obtained in the original centralized scheduling system [Sadeh 89]. Thus, despite the incompleteness of information available in the 2- and 3-agent systems, texture measures provide satisfactory summarizations. Second, as expected, performance of the 2- and 3-agent systems does

deteriorate as the communication of changing texture information is delayed. Since current texture information is used to perform both variable and value ordering, it is likely that both these processes deteriorate. An interesting observation is that in this set of experiments, the 3-agent system did better in terms of search efficiency than the 2-agent system[7].

The effect of delaying communication/computation of demand densities is greater for the 2- and 3-agent systems than the 1-agent system. This interaction may reflect the compensatory relation between variable and value ordering observed in [Sadeh 89]. Note that 2- and 3-agent performance is still better than the semi-random condition, suggesting that variable ordering strategy is robust with respect to the conditions of the multi-agent environment (incomplete, changeable information and asynchronous behavior without external coordination).

8. Concluding Remarks

In this paper we have presented mechanisms to guide distributed search. The domain of investigation is distributed job-shop scheduling. In particular, we have presented measures of characteristics of a search space, called textures, that are used to focus the attention of agents during search and allow them to efficiently find scheduling solutions that satisfy all constraints. In addition, the textures express the impact of local decisions on system goals and allow agents to form expectations about the needs of others. This ability is critical in multi-agent complex environments, such as the factory floor, where agents have to plan under considerable uncertainty. We have presented two types of textures (activity criticality and value goodness), their operationalization into variable and value ordering heuristics and their use in distributed problem solving. In addition, a communication protocol that enables the agents to coordinate their decisions has been presented.

A testbed has been implemented that allows for experimentation with a variety of distributed protocols that use variable and value ordering heuristics. The testbed also provides unique opportunities to compare closely matched single- and multi-agent scheduling systems. This comparison helps establish baseline performance measures and isolate conditions that influence performance in multi-agent systems.

Our results demonstrated that a texture approach to multi-agent scheduling can produce search efficiency that approximates that of a centralized system, even for problems that are difficult for traditional approaches to constraint satisfaction. Furthermore, the texture approach proved to be robust in the face of decreasing communication frequency, thus substantially decreasing communication overhead.

[7]This was true in the majority of comparisons between the 2- and 3-agent systems. No easy generalization can be made, however, since in some of the experimental groupings, the 3-agent system performance was very bad in the increased delay conditions, whereas the 2-agent system performed with graceful deterioration for the corresponding increased delay conditions.

References

[Baker 74] K.R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, 1974.

[Bitner 75] J.R. Bitner and E.M. Reingold. Bakctrack Programming Techniques. *Communications of the ACM*18(11):651-655, 1975.

[Cammarata 83] Cammarata, S. McArthur, D. and Steeb, R. Strategies of cooperation in distributed problem solving. In *IJCAI-83*, Pages 767-770. IJCAI, Karlsruhe, W. Germany, 1983.

[Conry 86] Conry, S.E., Meyer, R.A., and Lesser, V.R. *Multistage Negotiation in Distributed Planning.* Technical Report COINS-86-67, COINS, University of Massachusetts, december, 1986.

[Dechter 88] Rina Dechter and Judea Pearl. Network-Based Heuristics for Constraint Satisfaction Problems. *Artificial Intelligence*34(1):1-38, 1988.

[Dechter 89] Dechter, R., and Meiri, I. Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Pages 271-277. American Association of Artificial Intelligence, Detroit, MI., August, 1989.

[Durfee 87a] Durfee, E.H.
A Unified Approach to Dynamic Coordination: Plannign Actions and Interactions in a Distributed Problem Solving Network.
PhD thesis, COINS, University of Massachusetts, 1987.

[Durfee 87b] Durfee, E.H, Lesser, V.R., and Corkill, D.D.
Coherent Cooperation Among Communicating Problem Solvers.
*IEEE Transactions on Computers*C(36):1275-1291, 1987.

[Fox 83] Fox, M.S.
Constraint-Directed Search: A Case Study of Job Shop Scheduling.
PhD thesis, Computer Science Department, Carnegie-Mellon University, 1983.

[Fox 89] Mark S. Fox, Norman Sadeh, and Can Baykan.
Constrained Heuristic Search.
In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Pages 309-315. 1989.

[French 82] S. French.
Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop.
Wiley, 1982.

[Garey 79] M.R. Garey and D.S. Johnson.
Computers and Intractability: A Guide to the Theory of NP-Completeness.
Freeman and Co., 1979.

[Golomb 65] Solomon W. Golomb and Leonard D. Baumert.
Backtrack Programming.
*Journal of the Association for Computing Machinery*12(4):516-524, 1965.

[Haralick 80] Robert M. Haralick and Gordon L. Elliott.
Increasing Tree Search Efficiency for Constraint Satisfaction Problems.
*Artificial Intelligence*14(3):263-313, 1980.

[LePape&Smith 87] LePape, C. and S.F. Smith.
Management of Temporal Constraints for Factory Scheduling.
In *Proceedings IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems (TAIS 87)*, Elsevier Science Publishers, held in Sophia Antipolis, France, May, 1987.

[Mackworth 77] Consistency in Networks of Relations.
*Artificial Intelligence*8(1):99-118, 1977.

[Mackworth 85] Mackworth, A.K., and Freuder, E.C.
The Complexity of some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems.
*Artificial Intelligence*25(1):65-74, 1985.

[Montanari 71] Ugo Montanari.
Networks of Constraints: Fundamental Properties and Applications to Picture Processing.
Technical Report , Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1971.

[Nadel 88] Bernard Nadel.
Tree Search and Arc Consistency in Constraint Satisfaction Algorithms,
In L. Kanal and V. Kumar, *Search in Articial Intelligence*. Springer-Verlag, 1988.

[Ow et. al. 88] Ow, P.S., S.F. Smith, and A. Thiriez.
Reactive Plan Revision.
In *Proceedings AAAI-88*, St. Paul, Minnesota, August, 1988.

[Parunak 86] Parunak, H.V., P.W. Lozo, R. Judd, B.W. Irish.
A Distributed Heuristic Strategy for Material Transportation.
In *Proceedings 1986 Conference on Intelligent Systems and Machines*, Rochester, Michigan, 1986.

[Prosser 89] Prosser, P.
A Reactive Scheduling Agent.
In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Pages 1004-1009. American Association of Artificial Intelligence, Detroit, MI., August, 1989.

[Purdom 83] Paul W. Purdom, Jr.
Search Rearrangement Backtracking and Polynomial Average Time.
*Artificial Intelligence*21:117-133, 1983.

[Rinnooy Kan 76] A.H.G. Rinnooy Kan.
Machine Scheduling Problems: Classification, complexity, and computations.
PhD thesis, University of Amsterdam, 1976.

[Sadeh 88] N. Sadeh and M.S. Fox.
Preference Propagation in Temporal/Capacity Constraint Graphs.
Technical Report CMU-CS-88-193, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, 1988.
Also appears as Robotics Institute technical report CMU-RI-TR-89-2.

[Sadeh 89] N. Sadeh and M.S. Fox.
Focus of Attention in an Activity-based Scheduler.
In *Proceedings of the NASA Conference on Space Telerobotics*, 1989.

[Sadeh 90] N. Sadeh, and M.S. Fox.
Variable and Value Ordering Heuristics for Activity-based Job-shop Scheduling.
In *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management, Hilton Head Island, S.C.*, 1990.

[Smith 85] Stephen F. Smith and Peng Si Ow.
The Use of Multiple Problem Decompositions in Time Constrained Planning Tasks.
In *Proceedings of the Ninth International Conference on Artificial Intelligence*, Pages 1013-1015. 1985.

[Smith&Hynynen 87]
Smith, S.F. and J.E. Hynynen.
Integrated Decentralization of Production Management: An Approach for Factory Scheduling.
In *Proceedings ASME Annual Winter Conference: Symposium on Integrated and Intelligent Manufacturing*, Boston, MA, December, 1987.

Anytime Rescheduling

Monte Zweben
NASA Ames Research Center
M.S. 244-17
Moffett Field, California 94035
zweben@pluto.arc.nasa.gov

Michael Deale and Robert Gargan
Lockheed AI Center
Palo Alto, California 94305
deale@laic.lockheed.com

Abstract

This paper discusses an *anytime* rescheduling algorithm based upon constraint-based simulated annealing. Rescheduling is the process of resolving conflicts in a modified schedule. The algorithm has been implemented and tested upon a NASA scheduling problem and has performed well. We describe the algorithm in terms of its speed, its optimization criteria, and its disruption to the original schedule. Finally, we step through a simple example of the algorithm.

1 Introduction

This paper describes an *anytime* [Dea88] rescheduling algorithm based upon constraint-based simulated annealing [Zwe90a]. In an anytime algorithm, the search process can be halted at virtually any point and return a useful solution. Our rescheduling algorithm iteratively improves a complete but flawed schedule until a satisfactory schedule is found, an arbitrary bound on the number of search iterations is reached, or the user terminates the search. The algorithm is heuristic and not complete, but it converges quickly to a solution when tested on the NASA Kennedy Space Center (KSC) space shuttle processing problem and is expected to be used operationally at KSC. For more information on the application of the system please refer to [Zwe90b].

In this paper, we first formulate the general scheduling problem and present our iterative improvement algorithm used for rescheduling. Then, we step through a small example of the algorithm. Finally, we describe various perturbations to our approach in order to handle real-time and over-constrained problems.

2 Problem Formulation

Scheduling systems typically model temporal precedence between relations and the resource requirements between tasks. In contrast, planning systems typically reason about more general forms of information, but usually with respect to partial orderings of tasks, ignoring metric times. Few systems have been developed that can reason about arbitrary information with respect to metric time. Some notable exceptions include [Dru90, Dea85, Mil88].

In addition to modeling resource availability, our scheduling system has been extended to model state variables. State variables represent attributes of domain objects that change state over time. Examples from the domain include the position of switches, doors, landing gear, elevons, or flaps, the configuration of a system such as whether it is hazardous or clear and finally the location of an object. Tasks can be constrained by these state variables (i.e., they may require certain state variables to have certain values, ranges, or properties during some time interval) and tasks can also change state variables (i.e, once scheduled, they can force some state variables to take on certain values with a given persistence). These constraints and effects are analogous to the action schemata of traditional planning systems [Fik72]. Constraints on resources combined with constraints on state-variables, allow us to represent goals of maintenance and goals of achievement.

We formulate scheduling as a constraint satisfaction problem. Variables represent the assignments required to complete a schedule as well as any *a priori* fixed information affecting the schedule. Constraints are used to represent the desired relationships between these variables and scheduling is the process of instantiating the unassigned variable subject to the constraints. Variables represent the attributes of tasks, the attributes of resources, and the attributes of domain objects such as the Space Shuttle. The attributes of tasks include start times, end times and durations, calendars of legal times that account for work schedules, and resource requirements. The calendar indicates whether the task can be active during the first, second, and third shift of the day, as well as, whether the task is active on weekends and on holidays. Resource requirements are specified by a resource type, a quantity, and a particular pool [1]. For example, an inspection task could require six technicians that can be drawn from one of ten different technician pools. The attributes of resources include their type, their users, and their availability over time. The attributes of domain objects are largely dependent upon the type of objects. For example, a space shuttle has elevons (left and right), landing gear (nose and main), flaps, power, hydraulics and numerous other parts.

[1] A pool is an indistinguishable collection of resources. A resource requirement also indicates whether the resource is replenishable.

Variables are either static or functions of time. The domain of possible values for each static variable is either a time[2], a real number, an integer, or a discrete set of values. Some variables, such as the availability of resources and attributes of domain objects, are properly viewed as functions over time. These variables are represented as histories [Wil86] or time lines, which are lists of tuples. The first element of the tuple represents a time interval and the second represents a value. For example,

```
( ([0 100] open) ([200 :pos-infinity] closed))
```

represents the value of a door status. From time 0 to 100, the door is open and in the absence of any other information it persists until time 200 when it is permanently shut. Other variables such as a task's time variables and resource requirement variables are static and do not require a timeline representation.

The following constraints relate variables to each other:

1. less-than(?x ?y): The value of variable ?x must be less than the value of variable ?y.
2. less-than-or-equals(?x ?y): The value of variable ?x must be less than or equal to the value of variable ?y.
3. equals(?x ?y): The value of variable ?x must equal the value of variable ?y.
4. not-equals(?x ?y): The value of variable ?x must not equal the value of variable ?y.
5. plus(?x ?y ?z): The sum of the value of variable ?x and the value of variable ?y must equal the value of variable ?z.
6. capacity(?start ?end ?resource): The values of variables ?start and ?end are times. The value of variable ?resource is a resource pool. In the time interval spanning from ?start to ?end, ?resource must not exceed its maximum capacity..
7. temporal-equals(?t1 ?t2 ?a ?v): ?a must equal ?v during the interval ranging from ?t1 to ?t2.
8. temporal-less-than(?t1 ?t2 ?a ?v): ?a must be less than ?v during the interval ranging from ?t1 to ?t2.
9. calendar-point(?st ?et ?calendar): The ?st and ?et must be legal with respect to the ?calendar.
10. calendar-extend(?st ?et ?workduration ?calendar) There must be at least ?workduration amount of active time with respect to ?calendar during the interval ?st through ?et

Other constraints can be added easily to the default set above.

As stated previously, tasks consist of variables (denoting resource and temporal information), and constraints that relate this information to other tasks and other domain objects. Additionally, tasks have *effects* which represent the changes made to domain objects. In particular, tasks can set or increment a state-variable, and can initiate these changes either at the start or end of the task.[3] These task effects can persist indefinitely, only for the duration of the task, or until they are clipped by some other effect [Dea85].

The input to a CSP scheduling problem is a set of tasks and objects that are related by constraints. A solution to the problem is an assignment to all unassigned variables such that after all tasks have their resource usages and changes to state variables asserted, all constraints are satisfied. The input to a *rescheduling problem* is a complete legal schedule and a schedule modification. We define a schedule modification as any combination of the following changes:

1. The shift of a task (the start and end times of a task are displaced by some positive or negative constant).
2. A change in duration (either an extension or a restriction)
3. A change in some resource capacity.
4. A change in the value of some attribute of a domain object.
5. A temporary delay and projected resumption of a task.
6. The addition/removal of a task.
7. The addition/removal of a constraint.

A solution to a rescheduling problem is a reassignment of variable values, such that after all modifications are made, and all tasks have their resource usages and changes to state variables asserted, all constraints are satisfied.

3 Rescheduling

Our approach to rescheduling is based upon the general iterative improvement algorithm described in [Zwe90a]. We now describe the algorithm used to respond to task changes, which is similar to that used to respond to resource and object state changes. The basic algorithm consists of a two phase process. The first phase is a systematic repair of all violated temporal constraints. This results in a temporally consistent schedule, but with outstanding resource and state-variable constraint violations. This schedule is then input to the second phase – constraint-based simulated annealing. Here the schedule is incrementally improved by repairing violated resource and state-variable constraints. It is important to note that whenever the annealing process must shift a task, it employs the temporal shift algorithm used for the first phase of the rescheduling process. We now describe each phase in detail.

3.1 Phase One: Temporal Shift

The temporal shift is a heuristic procedure that takes as input a desired change in the start and end of a task and creates a temporally consistent schedule as output. It achieves this by systematically shifting tasks

[2]Time is represented as the number of minutes since some anchor point.

[3]Our implementation will be extended to allow arbitrary functional changes. We will also allow these effects to be anchored with some delay.

```
Solve(T){
   Old = Cost(T);
   Repeat until Old <= *THRESHOLD* {
          Next = Find_New_Solution(T);
          New = Cost(Next);
          If New < Old
             Then { Old = New;
                    T = Next;
                  }
             Else { With probability P
                    do
                      { Old = New;
                        T = Next;
                      }
                  }
          SaveBestSolutionIfNecessary;
   }
}
```

Figure 1: Constraint-based Simulated Annealing.

that are involved in temporal constraint violations, in a fashion similar to the techniques used in OPIS [Ow 88]. The algorithm begins by rescheduling the changed task; that is, it shifts it by some displacement that is sufficient to resolve all temporal constraints. This process is equivalent to achieving arc-consistency using the Waltz algorithm on temporal constraints [Fre82, Wal75, Dav87].[4] Following the achievement of arc-consistency, the system assigns the earliest legal start time for each activity (or the latest if the initial task was moved earlier). The advantage of this approach is that it rapidly synthesizes viable schedules with minimal changes to the original schedule; only those tasks with constraint violations are shifted. More radical changes to the original schedule are unrealistic and unacceptable in many real-world scheduling domains, including shuttle processing.

Before a task is shifted, its resources are deallocated, and state effects are removed. After the task is shifted, the resources are reallocated and effects reasserted with new start and end times. Remaining resource violations are removed in phase two.

It is important to note that the requested move is not guaranteed to be carried out. If the temporal shift attempts to move a task that is marked as permanent (such as a natural event like the sunrise) then the the move is deemed implausible. The algorithm then reverts all changed variables back to their original state

3.2 Phase Two: Constraint-based Simulated Annealing

The second phase is based on simulated annealing [Kir83]. It begins with the scheduling assignment that results from phase one of rescheduling and then evaluates a "cost" of the assignment. The cost function for our experiments is a weighted sum of violated constraints for the given assignment. The system then suggests a new solution by repairing constraints. If the new cost is an improvement, it adopts the new assignment and continues. If the new solution is more costly, the algorithm will adopt it according to a probabilistic measure described later. This last step allows the algorithm to escape local minima. We have customized this general approach to constraint satisfaction problems which is described in more detail elsewhere [Zwe90a]. Figure 1 illustrates the basic algorithm (where T is a set of tasks with assignments made in phase one).

In over-constrained problems, when it is impossible to converge to a solution below the desired threshold, the system terminates after some bound on the number of iterations. The threshold used for our rescheduling experiments was zero – all constraints must be satisfied. If the algorithm is interrupted, the best solution found prior to the interruption is returned.

3.2.1 Systematic Repairs: Finding a New Schedule

The system synthesizes a new schedule by repairing a subset of the violated constraints. In our KSC domain, we require fast rescheduling with a heuristic bias against schedules with excessive work-in-process (WIP) time, and against schedules that require radical perturbations to the original schedule. This bias is enforced by the repair strategies. For example, tasks are not moved drastically and are only moved if they are involved in constraint violations.

Our repair strategy exploits the knowledge that any task move is likely to violate temporal constraints. As a result, after any constraint repair causes a task to move, the temporal constraint violations are resolved first by executing the temporal shift algorithm given above.

Figure 2 illustrates the repair strategy for the *capacity* constraint. The constant c is a small, fixed time unit (an 8 hour shift in the KSC processing domain) and d is a direction (1 or -1) that is set randomly. The strategy attempts to substitute a new resource pool, but if this is impossible, it moves a task back or forward in time. After the task is moved, the temporal shift algorithm of phase one is executed – this systematically propagates the change caused by the repair to all temporal dependents.

The computational overhead of the repair is proportional to the complexity of the choice of what task to move. One can simply move the task associated with the constraint or move another task that is also using the resource during the interval specified by the constraint. Any heuristic used for this choice can draw upon the following criteria:

Fitness: Move the task that is using an amount closest to the amount that is overallocated. A task using a smaller amount is not likely to have a large enough impact and a task using a far greater amount is likely to be in violation wherever it is moved.

Temporal Slack: Any task that is highly constrained temporally is likely to cause temporal constraint violations and therefore could result in large perturbations to the schedule.

[4] Our algorithm also considers calendar constraints but only in a limited manner. It applies these constraints only to the extrema of the intervals found by propagating temporal constraints.

Temporal Dependents: Similar to temporal slack, a task with many dependents is likely to cause temporal constraint violations, if moved.

Severity of Bottleneck: Prefer tasks that do not need to be moved drastically to avoid conflict.

Priority: The system should avoid delaying important tasks, conversely it should prefer moving them earlier.

In-Process: A task that has already begun should be completed as soon as possible, rather than temporarily stopping it, and then continuing later.

Chronological Proximity: It is better to move activities that start later in the schedule than those that are about to begin.

Cycles: It is better to avoid moving tasks that have been moved frequently in previous iterations because the iterative improvement algorithm can potentially cycle.

In our initial implementation of this repair strategy, the system moves the task associated with the constraint, but our expectation is that an informed choice is likely to be more effective without introducing substantial overhead. We are currently performing experiments that address this hypothesis.

Figure 3 illustrates the repair strategy for the *temporal-equals* constraint that maintains arbitrary temporal conditions on state variables. This repair is analogous to the modal truth criterion of non-linear planning [Cha87] but without the flexibility of adding actions. The preferred repair is to move a task that sets the state-variable appropriately to a time interval preceding the task with the requirement. If this is impossible, the task with the requirement is moved to a point in time when the state variable is set appropriately.

In either case, to perform a move, the temporal shift of phase one is employed, which results in a temporally consistent schedule.

It should be noted that schedule modifications may require planning, that is, the addition of new tasks into the schedule may be required. For example, if a task opens a door in support of a later task, and then the door is mysteriously closed, it could be impossible to find a new *door open* task to shift back in time. In this case, a new *door open* task must be added to the schedule. We intend to augment our scheduler with repair strategies similar to those in the GEMPLAN planner [Lan88] that enables new tasks to be added to the schedule. Whenever a repair is impossible because of the need to add a task, the repair is rejected.

During each iteration, a subset of the outstanding violations is retrieved and then repaired. Currently, we repair ten availability constraints and all the violated state-variable constraints. A more informed strategy is also possible and is discussed below.

3.2.2 Noise: Escaping Local Minima

In the algorithm presented above, we accept "worse" solutions with some probability P. This allows the algorithm to jump out of local minimum in its search. Additionally, allowing the system to temporarily follow paths that extend into the space of poor solutions appears to help the search converge to a solution quicker. The probability P is defined as follows: $P = e^{-\Delta/T}$; $\Delta = NewCost - OldCost$; T is a temperature parameter [5] that controls the likelihood that poor solutions will be accepted; higher temperatures are more aggressive.

We have adopted a schedule of temperature reductions that begins with a relatively high temperature which is later reduced. As a result, it commences by jumping around the search space frequently but then makes more careful repairs. Currently, we begin with a temperature of 100 and reduce it after several iterations to 75. When the cost is low, we then reduce the temperature to 25.

As stated previously, we introduce noise in the search process to escape local minima. One explanation for this is that the cost function does not accurately reflect how "close" a candidate solution is to the actual solution; it is only a measure of the number of flaws in a candidate solution. For example, a logical assumption is that if only one availability constraint is violated, the algorithm is quite close to a solution, however this is incorrect. It may take over 20 repairs to achieve the overall goal of zero constraint violations, because 20 tasks must be moved back in time. Thus we allow the algorithm to jump out of local minimum in a conservative manner. It should be noted that this is the distinguishing factor between simulated annealing and classical hill-climbing search.

4 Anytime Characteristics

When searching for a solution, the annealing algorithm saves its best solution to date and returns it when the algorithm is interrupted. This approach meets the criteria put forth in [Dea88] to be classified as an anytime algorithm. Their criteria classifies anytime algorithms as those that:

1. lend themselves to preemptive scheduling (i.e. can be stopped and restarted)
2. can be terminated at any time and will output an answer
3. return answers that improve in a well-behaved manner over time.

An additional consideration is that the solution output must be useful to the user. It makes no sense to be *anytime* if the solution can not be utilized effectively.

Our algorithm is interruptible, restartable, and outputs a solution when terminated. The solution quality increases as a step-function of time. These solutions are useful in our domains because human schedulers can manually resolve conflicts in the schedule, especially when there are few conflicts that tend to be over-allocations of resources. Typically, the humans will multiplex or share resources between two tasks that are physically proximate. The system does not handle this sharing capability and we believe this would be very difficult to model. A system that can quickly converge to an

[5] The name of this parameter is reminiscent of the algorithm's chemistry origin.

```
capacity(?start ?end ?resource):

1. Deallocate this current resource.
2. Try to find a pool that is available from ?start to ?end.
3. If one exists, change ?resource to be that pool and reallocate.
4. Otherwise task = ChooseTaskToMove(constraint);
             new-start = ?start + random(1 .. 10) * c * d;
             new-end = new-start + duration(task);
             TemporalShift(task, new-start, new-end);
```

Figure 2: The repair strategy for the *capacity* constraint.

```
temporal-equals(?t1 ?t2 ?a ?v):

First strategy:

1. supporter = the first task
               after ?t1 that  sets ?a = ?v;
2. task = the task associated with this constraint;
3. new-end = start(task) - c;
4. new-start = new-end - duration(supporter);
5. TemporalShift(supporter, new-start, new-end);

If unsuccessful:

1. task = the task associated with this constraint;
2. new-start = the first time of a state transition, t,
               (away from ?t1 in the direction of d) where ?a is set to ?v;
3. new-end = new-start + duration(task);
4. TemporalShift(task, new-start, new-end);
```

Figure 3: The repair strategy for the *temporal-equals* constraint.

acceptable schedule with few resource problems is a very useful tool for the human schedulers at Kennedy Space Center (KSC). In fact, they rarely work with schedules that do not have violated constraints.

5 Example

We now step through an example of rescheduling. The initial schedule shown in Figure 4 contains 7 activities. Task T_1 has one effect which is to change the state of S_1 to the "on" position. Tasks T_2, T_3, T_4, T_5, and T_6 all require S_1 to be in the on position. Task T_7 changes the state of S_1 back to the off position. T_7 is also tied to a milestone meaning that it should end before this point in time. The only temporal constraints in the example are *after* constraints that exist between the end time of T_2 and the start time of T_3 and between the end time of T_4 and the start time of T_5. Tasks T_2, T_4, and T_6 each request resources R_1 and R_2. Tasks T_3 and T_5 each request resource R_2. Finally, we will assume that both resources (R_1 and R_2) have capacities of 2.

The top portion of Figure 4 displays the initial schedule. Just below the timeline for the initial schedule are two histograms depicting the allocation of R_1 and R_2. Gray bars indicate the positions of activities before the system rescheduled them.

The user reschedules task T_2 one shift (i.e., 8 hours) later, so that it is in parallel with task T_4. The temporal shift algorithm then shifts T_3 eight hours also, resulting in the second schedule in the diagram. As a result of these moves, there are now three constraint violations in the corresponding R_2 histogram (based on requests by T_3, T_5, and T_6). The cost of this new solution is three (assuming all constraint violations cost the same).

The last two portions of Figure 4 illustrate two iterations of annealing. In the first iteration, the system decides to move T_6 earlier in time by one shift. After checking for temporal constraint violations (there were none) the first iteration completes having resolved the three constraint violations. As shown in the third graph in Figure 4, the first iteration actually results in more constraints violations (6) then there were previously but nevertheless, the annealing algorithm has chosen to take this "worse" solution probabilistically. Now tasks T_2, T_4, and T_6 all have resource constraint violations for resources R_1 and R_2.

During the second annealing iteration, the system again chooses to advance T_6 by a shift. This change causes all constraints to be satisfied completing the rescheduling process. The final schedule is shown at the bottom of Figure 4.

In our example, if the system initially moved T_6 later in time, then it would eventually discover violations involving the state of S_1; specifically T_6 requires S_1 in the on position while T_7 changes it to the off position. In repairing this constraint, the system could have tried to delay the start time of T_7, possibly missing the milestone. Since milestone constraints have a large weight (we always want to achieve milestones on schedule), the system would repair them, effectively "backtracking" through its previous actions, returning the schedule back to the state shown in the second graph in Figure 4.

5.1 Relevance to Real World Problems

While we simplified certain aspects of the example described above, it corresponds to scenarios within the Space Shuttle ground operations domain. In the main processing facility, the rear of the orbiter is often supported by large hydraulic jacks and at other times it rests upon its own main landing gear. Certain activities require the jacks with the landing gear raised. These requirements are typically represented as constraints on state variables such as S_1 in the example. Given the state of the landing gear and hydraulic jacks, many independent sets of activities can be now be performed (our example depicted 3 such independent processes). Changes in the schedule occur frequently because: 1) many unanticipated technical problems arise, 2) resources become unavailable when they are broken, sick, or performing unexpected duties, or finally 3) because of unpredicted bad weather. When these changes occur, our system will reschedule activities with respect to state variables, resource availability, and temporal constraints.

6 Heuristic Bias

In [Ow 88], three criteria were stated for evaluating the utility of various reactive revisions to a schedule:

- Attendance to scheduling objective: what is the quality of the revision with respect to the desired optimization criteria?
- Amount of disruption: how many changes to the original schedule are made?
- Efficiency of reaction: how quick is the reaction process?

These criteria must be balanced and are usually inversely related. In our domain, we sacrifice optimality of the schedule to reduce response time and disruption. We balance these criteria by biasing the initial solution, the cost function, the temperature reduction strategy, and the repair strategies. We bias the initial annealing solution to be temporally consistent with a one pass propagation of constraints and temporal shifts. To minimize the disruption of the existing schedule, the algorithm only modifies tasks that affect constraint violations. We minimize WIP by finding a proximate time for any constraint repair. Unfortunately, the optimality of the schedule is sacrificed because repairs could be more global and thus more informed; they could suggest global changes in the schedule that would reduce total WIP time. However, such an approach is likely to be too expensive and overly disruptive, but this remains to be proven.

In other domains, other criteria might be dominant, requiring different biases. Below, we present two examples of such domains and discuss the strategies that our system would adopt to address these criteria.

6.1 Over-constrained Problems

In many scientific domains, the problem is over-constrained; there are more tasks than can be successfully scheduled given the domain constraints. Examples of these domains are telescope science observation

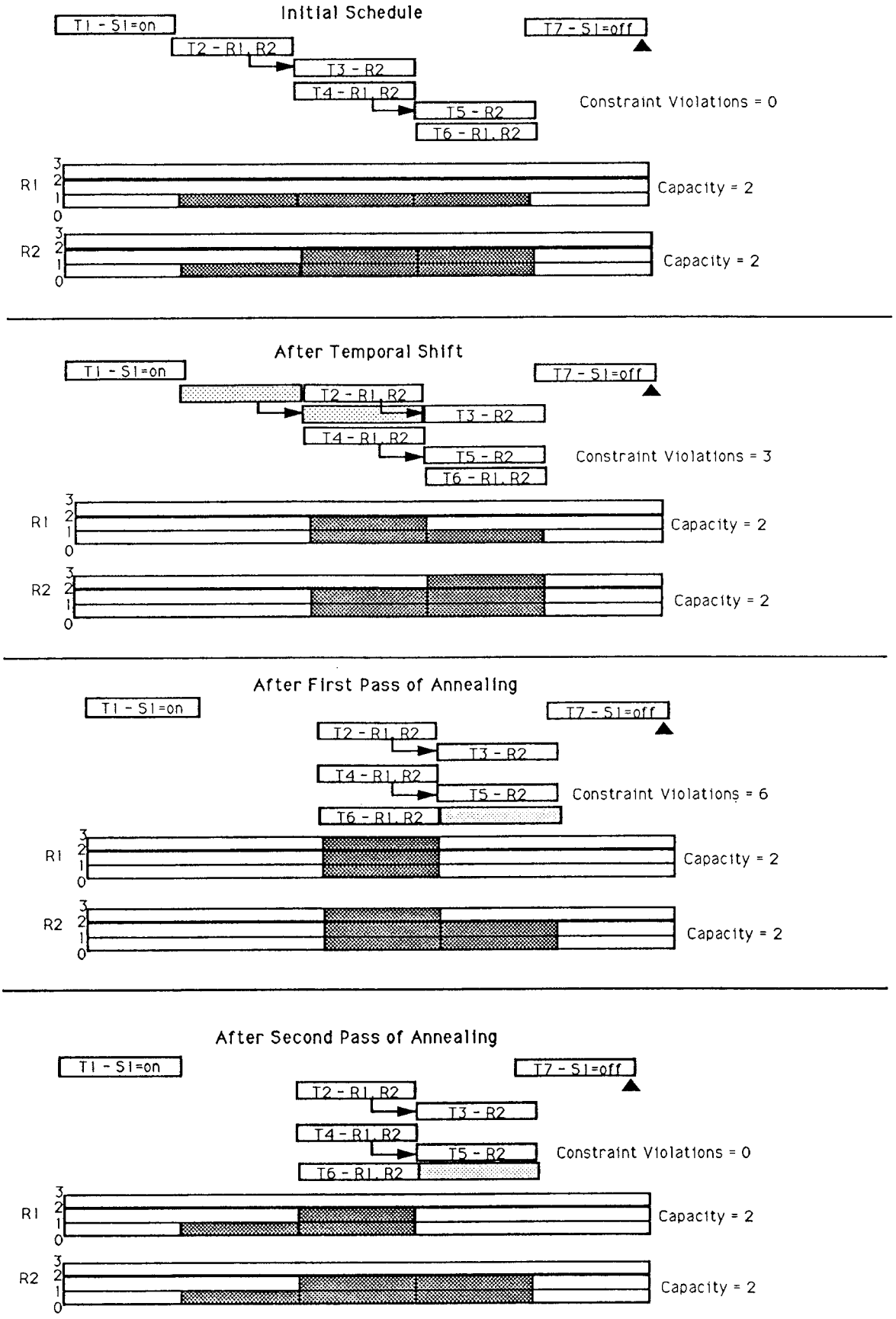


Figure 4 An example of rescheduling

scheduling and Space Shuttle and Space Station Freedom crew activity scheduling. In these domains, low priority tasks must often be dropped from the schedule. We would address this is two main ways. First, we would order repairs so that the higher priority tasks are addressed first. Second, we would penalize schedules (via the cost function) that have a large proportion of high priority tasks with constraint violations. Using this scheme, the execution time will rely heavily on the cost threshold for an acceptable solution and the iteration bound.

6.2 Real-Time Problems

In many real-time problems, priority must be given to those tasks that are about to take place. We could use the same strategy as given above, where the constraints associated with imminent tasks are repaired first and the cost function penalizes schedules with flawed tasks close to the event horizon. We could also augment the repair phase with a bound on its execution time. As soon as this time bound expires, no new constraints would be repaired until the next iteration.

7 Relation to Other Work

Our work was heavily influenced by the criteria put forth in the OPIS scheduler [Ow 88]. We have introduced a new simulated annealing search framework to these criteria that compares favorably with systematic search [Zwe90a] and corroborates with a parallel study [Min90]. The work is also related to that of Miller et. al. [Mil88] in that deadline and resource requirements are addressed, but it differs in that our representations and search techniques are quite different. They represent time-changing information as propositions maintained by the TMM - Time Map Manager [Dea85] and use traditional graph search algorithms to maintain consistency among these propositions. We also have similar goals as Drummond and Bresina [Dru90]. They are developing an *anytime* agent architecture based upon beam search that explicitly represents uncertainty. They are also developing more complex criteria to judge the *anytime* characteristics of an algorithm.

8 Summary

This paper reports a rescheduling algorithm based upon Constraint-based Simulated Annealing. The system can respond to schedule modifications and can quickly resolve problems with temporally dependent conditions. It meets the criteria put forth by Dean et. al. to be classified as *anytime*; in addition, we have addressed the quality of the rescheduling according to the criteria presented by Ow et. al. in [Ow 88]. We plan to experiment further with the technique concentrating on overconstrained and real-time problems. The system we have developed will be evaluated by the Kennedy Space Center as an operational tool for streamlining Space Shuttle Processing.

Acknowledgements

Thanks to Nils Nilsson for advising much of this work. Thanks to Amy Lansky, Peter Friedland, Bill Mark, and Jeanne Cunicelli for their careful review of this paper. We also thank Steve Smith, Steve Minton, Andy Phillips, Mark Drummond, John Bresina, and Mark Ringer for all the useful suggestions they have contributed.

References

[Cha87] Chapman, D. Planning for Conjunctive Goals. *Artificial Intelligence*, 32(4), 1987.

[Dav87] Davis, E. Constraint Propagation with Interval Labels. *Artificial Intelligence*, 32(3), 1987.

[Dea85] Dean, T. *Temporal Imagery: An Approach to Reasoning about Time for Planning and Problem Solving.* PhD thesis, Yale University, January 1985.

[Dea88] Dean, T., and Boddy, M. An Analysis of Time-Dependent Planning. In *Proceedings of AAAI-88*, 1988.

[Dru90] Drummond, M. and Bresina J. An Anytime Temporal Projection Algorithm for Maximizing Expected Situation Coverage . In *Proceedings of AAAI-90*, 1990.

[Fik72] Fikes, R.E., Hart, P.E., and Nilsson, N.J. Learning and Executing Generalized Robot Plans. *Artificial Intelligence*, 3, 1972.

[Fre82] Freuder, E. C. A Sufficient Condition for Backtrack-Free Search. *J. ACM*, 29(1), 1982.

[Kir83] Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P. Optimization by Simulated Annealing. *Science*, 220(4598), 1983.

[Lan88] Lansky, A. Localized Event-based Reasoning for Multiagent Domains. *Computational Intelligence*, 4(4), 1988.

[Mil88] Miller, D., Firby, R. J., Dean, T. Deadlines, Travel Time, and Robot Problem Solving. In *Proceedings of AAAI-88*, St. Paul, Minnesota, 1988.

[Min90] Minton, S., Phillips, A., Johnston, M., Laird., P. Solving Large Scale CSP and Scheduling Problems with a Heuristic Repair Method. In *Proceedings of AAAI-90*, 1990.

[Ow 88] Ow, P., Smith S., Thiriez, A. Reactive Plan Revision. In *Proceedings AAAI-88*, 1988.

[Wal75] Waltz, D. Understanding Line Drawings of Scenes with Shadows. In P. Winston, editor, *The Psychology of Computer Vision*, McGraw-Hill, 1975.

[Wil86] Williams, B.C. Doing Time: Putting Qualitative Reasoning on Firmer Ground. In *Proceedings of AAAI-86*, 1986.

[Zwe90a] Zweben, M. A Framework for Iterative Improvement Search Algorithms Suited for Constraint Satisfaction Problems. In *Proccedings of AAAI-90 Workshop on Constraint-Directed Reasoning*, 1990.

[Zwe90b] Zweben, M., and Gargan, R. The Ames-Lockheed Orbiter Processing Scheduling System. In *Proceedings of the Space Operations, Applications and Research Symposium*, 1990.

CONTROL

Real-Time Control of Reasoning: Experiments with Two Control Models

Anne Collinot and Barbara Hayes-Roth*
Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, California 94304

Abstract

An intelligent agent must identify and perform logically correct actions in response to external events, and it must perform them at appropriate times. The top-level objective of such an agent would be to maximize (or ensure a lower bound on) the value of some global utility function that integrates the values of its responses to events, weighted by the importance of those events, over time. In this paper, we focus on four properties that might facilitate an agent's achievement of its global utility objective: selectivity, responsiveness, robustness, and scalability. We assume a very general agent architecture, and we focus on its reasoning component. As opposed to a *best-next control model* we propose a *satisficing control model* for the reasoning process. We have conducted preliminary experiments to test the following hypotheses: the satisficing model provides good selectivity, responsiveness, robustness, and scalability (both when measured against the best-next model and when measured in absolute terms); therefore, the satisficing model provides a high global utility for the agent's performance. The results of our experiments confirm our hypotheses.

1. The problem

In many situations, an intelligent agent must coordinate its actions with the behavior of other agents or processes over which it has no direct control. The agent must identify and perform logically correct actions in response to external events, and it must perform them at appropriate times.

Given limited computational resources and a complex environment, an agent generally cannot identify and perform optimal actions in response to all environmental events in a timely fashion. Instead, to guarantee satisfactory responses to *important* events, it must compromise its responses to other events. It might reduce the *value* of its response (precision, completeness, correctness, certainty, or timeliness) to some events. It might not respond to some unimportant events at all. The top-level objective of such an agent would be to maximize (or ensure a lower bound on) the value of some global utility function that integrates the values of its responses to events, weighted by the importance of those events, over time. The exact form of this function might vary among agents or circumstances and need not concern us here.

There are many behavioral properties that might be hypothesized to facilitate an agent's achievement of its global utility objective [Dodhiawala, 1989, Hayes-Roth 1990]. In this paper, we focus on four properties: selectivity, responsiveness, robustness, and scalability. *Selectivity* refers to differential responses to events based on importance. Other things being equal, the agent should respond more reliably and more quickly to more important events than to less important events. *Responsiveness* refers to modulation of response latency based on urgency. Other things being equal, the agent should respond more quickly to more urgent events than to less urgent events. The agent should maintain selectivity and responsiveness under a variety of conditions. *Robustness* refers to the maintenance of these properties despite increases in environmental complexity. Other things being equal, the agent's behavior should degrade gradually, if at all, with increases in the rate or variability of environmental events. *Scalability* refers to the maintenance of these properties despite increases in knowledge. Other things being equal, the agent's behavior should degrade gradually, if at all, with increases in its own knowledge. In fact, in many situations, the agent's behavior should improve with increases in knowledge.

We assume a very general agent architecture with component processes for *perception*, to obtain information about environmental events, *reasoning*, to interpret perceived events, solve problems, and plan actions, and *action*, to influence the environment [Hayes-Roth, 1987, Hayes-Roth, 1990]. Because reasoning mediates perception and action, the reasoning process must have the properties of selectivity, responsiveness, robustness, and scalability--if the agent's overall behavior is to have those properties.

Thus, we have two questions. First, what kind of control model will give an agent's reasoning the properties of selectivity, responsiveness, robustness, and scalability? Second, will a control model that provides these properties

* This research was supported by DARPA grant #NAG 2-581 through NASA and Boeing contract #W-289988. Anne Collinot was supported by a fellowship from INRIA (France). We gratefully acknowledge contributions by the Guardian project members. Rich Washington provided helpful comments on various drafts of this paper. We thank Edward Feigenbaum for sponsoring the work at the Knowledge Systems Laboratory.

facilitate the agent's achievement of its global utility objective?

2. Alternative Control models

We assume an agent architecture in which independent perception, reasoning, and action processes communicate via globally accessible I/O buffers. Perception processes deliver continuous streams of perceived events, with each event labeled by its *type*, *value*, *importance*, and *deadline* [Washington and Hayes-Roth, 1989, Washington *et al.*, 1990], to input buffers in the reasoning process. Action processes retrieve and execute intended actions from output buffers in the reasoning process. The scope of our investigation into alternative control models for the reasoning process will be bounded by the entry of perceptual events into input buffers and their exit from output buffers.

We assume a cyclic reasoning process in which the agent repeatedly: **(1)** identifies and rates potential reasoning operations given the current state and recent "triggering" events (both perceptual and cognitive events), **(2)** chooses one identified operation based on the ratings, and **(3)** executes the chosen operation. This view of reasoning is quite general and is instantiated in a range of specific AI system models. Many of these more specific models emphasize the need for intelligence in the cycle, so that the agent chooses to execute a "good" or "correct" operation on each cycle. Such models are variously called "conflict resolution" [Newell, 1973, McDermott and Forgy, 1978, Forgy, 1982], "intelligent control" [Hayes-Roth and Hayes-Roth, 1979, Hayes-Roth, 1985], "meta-reasoning" [Genesereth and Smith, 1982, Russell and Wefald, 1989] or "deliberation scheduling" [Dean and Boddy, 1988]. In our model, scheduling criteria are, themselves, dynamically established and modified through base-level reasoning operations [Hayes-Roth *et al.*, 1986, Johnson and Hayes-Roth, 1987].

By definition, intelligent control of reasoning allows an agent to reason about and control its own reasoning behavior. For example, the agent can achieve a degree of selectivity by choosing operations based on the importance of their triggering events. It can achieve a degree of responsiveness by choosing operations whose computational demands are compatible with current deadlines. It can achieve a degree of robustness in the face of increasing event rates by choosing operations that respond to a smaller percentage of more important events and make smaller computational demands. It can achieve a degree of scalability over a growing knowledge base by choosing only operations that apply the most appropriate knowledge, given task demands and real-time constraints.

However, these claims are limited by the fact that the basic reasoning cycle, including the use of intelligent control, entails a computational overhead. Most formal analyses assume that the time spent on the cycle itself is insignificant, but this assumption probably is not valid in general [Dean, 1989]. Experimental evaluations [Durfee and Lesser, 1986, Garvey *et al.*, 1987] typically conclude that time spent on the cycle is more than balanced by the time saved in reasoning. Again, these observations probably are not valid in general.

In fact, the computations underlying the basic reasoning cycle are inherently complex. Let us illustrate this point with an informal complexity analysis of step (1) of the basic control cycle: the identification and rating of potential reasoning operations. An instantiation of an operation results from triggering a particular type of operation for a particular event. A particular event can trigger different types of operations and a particular operation can be triggered by many different events. We consider three complexity parameters: n is the number of events (both perceptual and cognitive events) the agent processes during step (1); k is the number of possible types of operations the agent knows; r is the number of scheduling criteria used in the rating process. In the worst case, the time spent identifying potential reasoning operations is $O(nk)$. If m is the number of identified operations, then the time spent rating these operations is $O(mr)$. In addition, the pattern-matching process involved in triggering an operation for a given event, as well as in rating an instantiated operation against a scheduling criterion, is itself a complex process. Although the details of this complexity factor need not concern us here, it is important to emphasize that this factor increases the complexity of both terms $O(nk)$ and $O(mr)$. Therefore, if an agent is to achieve selectivity, responsiveness, robustness, and scalability in its reasoning behavior, it must control the reasoning cycle, as well as its reasoning operations. In this paper, we compare two models for controlling the reasoning cycle, a best-next model and a satisficing model.

First consider the *best-next control model*. In step (1) of the reasoning cycle, the agent identifies all possible reasoning operations and rates each one against the current scheduling criteria. Scheduling criteria are determined dynamically by reasoning. Step (1) terminates when all operations triggered by all recent events have been identified. Therefore, given real-time constraints on performance, the algorithm for this process must be highly efficient. Our best-next algorithm appears in Figure 1. It assumes unlimited-capacity buffers for perceptual and cognitive events and for possible reasoning operations. In step (2), the agent chooses the highest-rated identified operation. In step (3), the agent executes the chosen operation. Thus, on each cycle, the agent invariably executes the best possible operation, but it may do so following an unbounded delay since the occurrence of the associated triggering events.

For each event present in the Event Buffer at the time the agent enters step (1),
  For each possible type of reasoning operation O,
    If all the triggering conditions of O are true,
      Then record a potential reasoning operation in the Agenda (operation buffer).

For each potential operation P recorded in the Agenda,
  Compute and record the ratings of P against the current scheduling criteria.

Figure 1. Algorithm for the Best-Next Control Model

Now consider the *satisficing control model*. In step (1) of the reasoning cycle, the agent identifies and rates a promising subset of possible operations, based on the current scheduling criteria. Again, scheduling criteria are determined dynamically by reasoning. Step (1) terminates when either: (a) an operation with a *criterial rating* has been identified; (b) a *deadline* has occurred; or (c) all possible operations have been identified. Both criterial ratings and deadlines are determined dynamically through reasoning. Given real-time constraints on performance and the possibility of an interrupt, the step (1) algorithm should identify possible operations in descending order of ratings, to the extent that is possible. Our satisficing algorithm appears in Figure 2. It assumes limited-capacity buffers for perceptual and cognitive events and for possible reasoning operations. In our experiments, event buffers had capacities of 7 items each and the operation buffer (agenda) had a capacity of 10 items. In step (2), the agent chooses the highest-rated identified operation. In step (3), it executes the chosen operation. Thus, on a given cycle, the agent may execute the first satisfactory operation, the best available operation within the given deadline, or the best possible operation--depending upon the interrupt condition.

Until
- *either an operation with a criterial rating has been identified,*
- *or a deadline has occurred,*
- *or there are no more events to process*

Select the best available event in the limited-capacity Event Buffers,
Until
- *either an operation with a criterial rating has been identified,*
- *or a deadline has occurred,*
- *or there are no more possible types of operations to consider*

Select the best possible type of reasoning operation O,
If all the triggering conditions of O are true,
Then
Compute and record the ratings of P against the current scheduling criteria,
Record and order P in descending order of ratings in the limited-capacity Agenda.

Figure 2. Algorithm for the Satisficing Control Model

Let us summarize the strengths and weaknesses of the best-next and satisficing control models. The best-next model guarantees execution of the best possible operation on each cycle and provides a very efficient algorithm for identifying that operation. On the other hand, it entails unbounded cycle times. The satisficing model conserves cycle time by executing satisfactory operations as soon as possible and guaranteeing execution of some operation by deadline. On the other hand, it explicitly allows the possibility of executing sub-optimal--costly or ineffective or even undesirable--operations. Moreover, in the worst case, where there is no deadline and no criterial operation is identified, the satisficing model must identify all possible operations, but with an algorithm that is not optimized for that purpose.

The remainder of this paper presents the results of initial experiments we conducted to compare the performance of the best-next and satisficing models with respect to the behavioral properties discussed above. In fact, the two models are not simply equal contenders in this evaluation. We have been working with the best-next cycle for several years in the BB1 architecture and used it for static problem-solving applications, where its performance has been largely satisfactory [Hayes-Roth *et al.*, 1986, Tommelein *et al.*, 1987, Darwiche *et al.*, 1988]. Its main weakness has been speed. However, for static applications, speed is only a pragmatic issue and we assume that the best-next algorithm could be optimized to provide satisfactory speed for particular applications. However, we believe that more radical changes are required to meet the requirements of the real-time problem-solving applications that we have been studying more recently. The satisficing cycle was designed in an effort to meet those requirements [Hayes-Roth, 1987]. Thus, in the conventional terms of experimental science, it is appropriate to view the best-next cycle as the "control condition" against which we will measure the effects of the "experimental treatment", which is the satisficing cycle. Accordingly, we conducted the experiments described below to test the following hypotheses:

(a) The satisficing model provides good selectivity, responsiveness, robustness, and scalability--both when measured against the best-next model and when measured in absolute terms;

(b) Therefore, the satisficing model provides a high global utility for the agent's performance--again, both when measured against the best-next model and when measured in absolute terms.

3. Overview of the Experiments

We tested our hypotheses in the context of the Guardian system for intensive care monitoring [Hayes-Roth *et al.*, 1989]. Guardian originally was implemented on top of the BB1 architecture [Hayes-Roth, 1985], with the best-next control cycle. For these experiments, we made a new version of Guardian in which the satisficing cycle replaces the best-next cycle. Thus, the two versions of Guardian differ only in whether they use the best-next cycle or the satisficing cycle. As Figures 1 and 2 show, the two algorithms share many of the same component functions. We tried to make distinctive elements of both algorithms reasonably efficient. However, because we have been working with the best-next algorithm for several years, it is reasonable to assume that any efficiency advantage due to the implementation *per se* would favor it over the satisficing algorithm.

To test our hypotheses, we measured the performance of each version of Guardian on a set of monitoring scenarios that require the targeted behavioral properties and compared the results. The following sections describe the Guardian system and the experimental scenarios, manipulations, and measurements.

3.1 The Guardian System

Functioning in a simulated intensive-care environment, Guardian monitors ventilator-supported patients and consults with physicians and nurses. The current version of Guardian monitors about sixteen automatically sensed variables (e.g., pressures, temperature) and a few irregularly sensed variables (e.g., lab results). It performs several tasks, such as interpreting sensed data, noticing and diagnosing exceptional events, predicting future states and events, planning therapy actions, explaining its reasoning, and carrying out actions in closed-loop control of the simulated patient. To perform these tasks, Guardian uses several kinds of knowledge: heuristic knowledge related to common respiratory problems; structure/function knowledge of the respiratory, circulatory, metabolic, and mechanical ventilator systems; and structure/function knowledge of generic flow, diffusion, and metabolic systems. In some cases, Guardian is capable of performing a given task in alternative ways. For example, given an exceptional event, it can diagnose the cause of that event relatively quickly, using probabilistic associations between disease conditions and observable signs and symptoms. Alternatively, it can take more time to identify plausible diagnoses based on a model of the relevant organ systems and the underlying physical principles. In such situations, Guardian makes smart choices in order to meet real-time (or other) constraints on its reasoning.

3.2 The Monitoring Scenario

In our experiments, the simulated patient has just returned from the operating room. Ventilator settings (i.e., the number of breaths delivered to the patient per minute and the volume of air blown into the patient's lungs on each breath) plus a set of sixteen parameters (e.g. temperature, heart rate, inspiratory peak pressure) are continually and automatically sensed. In addition, Guardian perceives irregularly reported lab results and asynchronous user requests. Several display drivers manage Guardian's communication with human users. These communications include dynamic graphical displays of: the patient's history; ongoing reasoning and results related to Guardian's reasoning tasks; and structure/function explanations of the patient's conditions. Each of these displays is interactive, permitting the user to pose particular kinds of questions. Therapeutic actions include changing the ventilator settings (e.g. decrease the number of breaths delivered to the patient per minute), adjusting the ventilator tube, and other sorts of interventions.

We distinguish key events and context events in the scenario. A *key event* requires a response. A *context event* could produce a response, but does not require one.

Four key events occur in the experimental scenario. The first key event occurs at the beginning of the scenario, when the patient has a low body temperature. Although this condition is not life-threatening, it can have undesirable consequences for the patient and requires a response. Guardian should predict the undesirable consequences (low arterial CO2, a condition called hypocapnia) and that the temperature will naturally return to normal over a period of hours. The prediction of hypocapnia is the second key event requiring a response. Guardian should adjust the patient's breathing rate in accordance with the low temperature to correct the hypocapnia and maintain the arterial CO2 within normal ranges as the patient's temperature rises. The third key event requiring a response is a user request for explanation of hypocapnia and the associated breathing rate adjustments. Guardian should give an appropriate explanation. The fourth key event, which occurs during Guardian's explanation, is an observation that the patient's peak inspiratory pressure (PIP) is very high, a potentially life-threatening situation. Guardian should diagnose the problem as a pneumothorax (hole in the lung) and immediately (within eight minutes) take an appropriate action, inserting a chest tube to release accumulated air in the chest cavity and reduce the PIP, thereby permitting normal ventilation.

There are also many context events in the scenario, which permit a response but do not require one. For example, there are many minor deviations of observed patient data from expected patient data. For any of these deviations, Guardian could predict present and future consequences. However, these events are much less important than the key events mentioned above and Guardian ordinarily would not have time to attend to them.

3.3 Manipulations

Variables manipulated in the experiments are summarized in Table 1. *Criticality of events* is defined by the importance and urgency of sensed data. In our scenario, the high PIP is both very important and very urgent and, therefore, of high criticality. The other three key events are moderately important and not very urgent and, therefore, of low criticality. *Environmental complexity* is defined by the number of sensed parameters and the resulting range of global event rates. High environmental complexity involves 16 parameters and 4-83 events every ten minutes, while low environmental complexity involves 8 parameters and 4-42 events every ten minutes. *Amount of knowledge* is defined by the number of different types of reasoning operations known to Guardian. High knowledge involves reasoning operations for diagnosis, reaction, prediction, explanation, planning, and global control, while low knowledge involves all of these except planning. Table 2 shows how four versions of the experimental scenario are defined in terms of these variables.

In all scenarios, Guardian used the same control decisions to guide its scheduling of possible reasoning operations and, in the case of the satisficing cycle, to guide its identification of possible reasoning operations. Several of these decisions were active throughout the scenario: (d1) favor operations depending on their type of reasoning (by default, global control is preferred to planning, which is preferred to prediction and explanation, which are preferred to diagnosis and reaction); (d2) favor operations that respond to a user request; (d3) favor operations that respond to important or abnormal signs. The decisions d1, d2 and d3 were weighted 100, 50 and 1, respectively. The following decision was active only during the critical period of high PIP: (d4) favor operations that respond quickly to high PIP. This decision resulted from the execution of a global control operation triggered by the critical event, high PIP, and remained active

until there are no more executable operations responding to this critical event. The weight of decision d4 was 1000.

Criticality of Events	Patient Condition
Yes	High PIP
No	No High PIP

Amount of Knowledge	# of Types of Operations
High	39
Low	28

Environmental Complexity	# of Sensed Parameters	Global Event Rate
High	16	[4, 83]
Low	8	[4, 42]

Table 1. Manipulations of Scenario Variables

Experimental Conditions	Criticality of Events	Environmental Complexity	Amount of Knowledge
Base Scenario	Yes	High	High
Non Critical	No	High	High
Low Complexity	Yes	Low	High
Low Knowledge	Yes	High	Low

Table 2. Definition of Experimental Conditions

3.4 Measurements

To make the appropriate measurements, we instrumented BB1 for the following variables. Response value is the value of a response given by Guardian to a key event. Speed of response to the critical event, high PIP, is a difference measure: the deadline for responding to the critical event minus the total time Guardian took to respond to it. Number of critical cycles is the number of reasoning cycles Guardian executed in order to respond to the critical event. Average agenda time per critical cycle is the average time used for agenda management during critical cycles. For the non-critical scenario, we measured agenda management time during cycles at a corresponding time during the run. Average priority of executed operations is the average priority for executed operations across all reasoning cycles. Average cycle time is similarly computed across all reasoning cycles, and average agenda time is that part of the cycle used in agenda management. Average number of new operations per cycle is the average number of possible reasoning operations identified on each reasoning cycle. All times are given in seconds.

4. Results

The results are summarized in Table 3. In the following sections, we evaluate each of our four hypotheses by comparing numbers from particular cells of the table.

4.1 Selectivity

We measure selectivity as: (a) correct vs. incorrect (or no) response to the critical event, high PIP; and (b) speed of the correct response to the critical event. With the best-next cycle, Guardian produced the correct response (insert chest tube) in the low complexity and low knowledge scenarios, but not in the base scenario. With the satisficing cycle, it produced the correct response in all three scenarios. Moreover, in the two scenarios where both cycles produced the correct response, the satisficing cycle produced lower response latencies than the best-next cycle. Thus, in this experiment, the satisficing cycle produced better selectivity than the best-next cycle, enabling Guardian to respond more reliably and more quickly to critical events. As we shall see later, the satisficing cycle achieved high selectivity of critical events by allowing Guardian to miss some less important events.

4.2 Responsiveness

We measure responsiveness as the difference in response latency for critical and non-critical events. As shown in Chart 1, with the best-next cycle, Guardian actually spends more time on agenda management and, therefore, on the entire cycle, during critical cycles than during non-critical cycles. By contrast, with the satisficing cycle, Guardian spends much less time on agenda management during critical cycles than during non-critical cycles. Thus, in this experiment, the satisficing cycle provides better responsiveness, allowing Guardian to reason more quickly when faced with a critical event than when faced with only non-critical events.

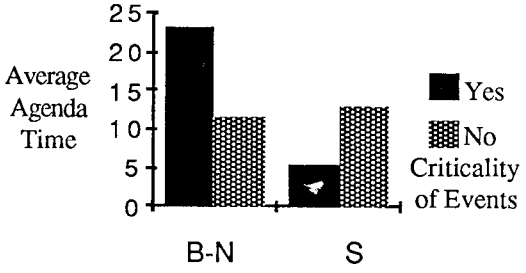


Chart 1. Responsiveness

Experimental Conditions	Response Value to the Critical Event		Speed of Response		# of Critical Cycles		Average Agenda Time per Critical Cycle	
	B-N	**S**	**B-N**	**S**	**B-N**	**S**	**B-N**	**S**
Base Scenario	Not Most Specific	Most Specific	-242	199	16	21	23	5.3
Non Critical Sc.	---	---	---	---	16	21	11.5	12.8
Low Complexity Sc.	Most Specific	Most Specific	165	221	21	21	7.4	5.2
Low Knowledge Sc.	Most Specific	Most Specific	15	165	21	21	15.3	6.4

Table 3. Results of the Experiments

4.3 Robustness

We measure robustness as the difference in response latency for critical events in low and high complexity scenarios. As shown in Chart 2, with the best-next cycle, Guardian spends much more time on agenda management and, therefore, on the entire cycle, in the high complexity scenario than in the low complexity scenario. This result is consistent with complexity analysis of Section 2, where the complexity of agenda management increases with event rate. By contrast, with the satisficing cycle, Guardian maintains stable agenda management times in both complexity conditions. Presumably, it does so by selectively responding to critical events, ignoring non-critical events regardless of their number. Thus, in this experiment, the satisficing cycle provides better robustness, allowing Guardian to respond quickly to critical events regardless of environmental complexity.

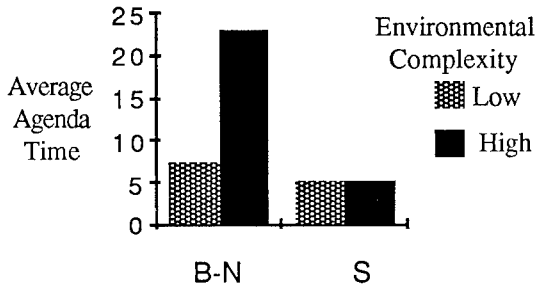


Chart 2. Robustness

4.4 Scalability

We measure scalability as the difference in response latency for critical events in low and high knowledge scenarios. As shown in Chart 3, with the best-next cycle, Guardian spends more time on agenda management and, therefore, on the entire cycle, in the high knowledge scenario than in the low knowledge scenario. This result is compatible with complexity analysis of Section 2, where the complexity of agenda management increases with the number of types of operations. By contrast, with the satisficing cycle, Guardian maintains stable agenda management times in both scenarios. Presumably, it does so by selectively applying the most important knowledge and ignoring irrelevant knowledge regardless of quantity. Thus, in this experiment, the satisficing cycle provides better scalability, allowing Guardian to maintain stable response times despite increases in knowledge.

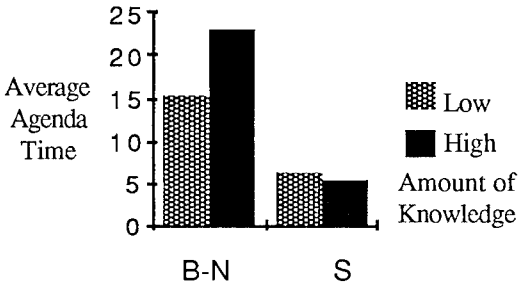


Chart 3. Scalability

4.5 Global Utility

We propose two classes of global utility functions:

U1 = Sum of (Response Value * Event Importance) for Key Events,

U2 = If Satisfactory Response to Critical Events, Then U1 Else 0.

Table 4 gives additional information necessary to characterize the global utility of Guardian's performance under each control cycle, with respect to these two classes of functions. Response correctness, 1 if correct and 0 otherwise, is given for the four key events for each scenario. For the critical event, Table 4 also gives the speed of response. We assume that at least for critical events, response value is a function of correctness and speed. In the experimental scenario, the critical event, high PIP, signals a pneumothorax, a life-threatening condition. Without specifying a combining function, we can see that the numbers in Table 4 clearly favor the satisficing cycle for the critical event, where the response is always correct and the latencies are always shorter (i.e., the speed of response is always higher) than those for the best-next cycle. Therefore, the U2 class of utility functions would generally favor the satisficing cycle over the best-next cycle, and with a wide range of combining functions, the U1 class of functions would favor it as well.

On the other hand, the best-next cycle produces responses to more non-critical key events than the satisficing cycle. Especially in the non-critical scenario, where no high PIP and pneumothorax occurred, the best-next cycle would produce higher global utility values in this experiment. However, we would not extend this conclusion to the general class of scenarios involving no critical events. In the present experiments, the control plans (i.e., the sets of control decisions) used by the satisficing cycle referred only to the critical events and provided no guidance at all in how to reason about non-critical events. With more comprehensive control plans, the satisficing cycle probably would have performed better on those events as well.

5. Interpretation of Results

We can analyze the performance of the satisficing cycle in terms of underlying variables. As shown in Table 5, the average priority of executed operation is comparable for the satisficing and best-next cycles, except in the base scenario, where the priority is higher for the satisficing cycle. Except in the non-critical scenario, the average cycle time and agenda time are shorter for the satisficing cycle than for the best-next cycle. The average number of newly identified possible reasoning operations is always smaller for the satisficing cycle. Because of this and the limited-capacity agenda buffer, the satisficing cycle always has a shorter total agenda than the best-next cycle. Thus, the satisficing cycle provides the desired behavioral properties by identifying a smaller number of high-priority reasoning operations in less time than the best-next cycle can identify all reasoning operations. Two factors allow it to do this: a good satisficing algorithm and an effective control plan.

Experimental Conditions	Best-Next Control Model				Satisficing Control Model			
	Non Critical Events			Critical Evt	Non Critical Events			Critical Evt
	Low Temp.	Hypocapnia Prediction	User Request	High PIP	Low Temp.	Hypocapnia Prediction	User Request	High PIP
Base Scenario	1	1	1	f(0, -242)	0	1	0	f(1, 199)
Non Critical Sc.	1	1	1	---	0	1	1	---
Low Complexity Sc.	1	1	1	f(1, 165)	0	1	0	f(1, 221)
Low Knowledge Sc.	1	---	1	f(1, 15)	0	---	0	f(1, 165)

Table 4. Response Values to Key Events

Experimental Conditions	Average Priority of Executed Operations		Average Cycle Time		Average Agenda Time		Average # of New Operations per Cycle	
	B-N	**S**	**B-N**	**S**	**B-N**	**S**	**B-N**	**S**
Base Scenario	25	36.3	22.1	14	12.1	7.1	4.6	2.5
Non Critical Sc.	3.8	3.9	10.5	15.4	6.5	10	3.6	3.1
Low Complexity Sc.	27	27.3	10.4	9.7	5.3	4.6	2.4	1.9
Low Knowledge Sc.	29.1	33.2	13.8	13.3	8.6	7.2	3.4	2.3

Table 5. Results for Interpretation

6. Future Work

The experiments reported in this paper are the first of a program of experiments we plan to conduct. We need to replicate the present results in a variety of monitoring scenarios and with a wider variation of experimental variables (e.g., number of critical events, environmental complexity, amount of knowledge). We are particularly interested in further substantiating our findings regarding robustness and scalability, key properties of a truly general approach. In addition, we wish to evaluate other desirable properties of real-time reasoning, such as coherence and flexibility [Hayes-Roth, 1990].

We also wish to analyze the roles of different parts of the satisficing cycle in allowing it to achieve the desired behavioral properties, in particular, the limited-capacity buffers and agenda, the knowledge-based heuristic search for possible operations, and interruptability of the cycle. Some other preliminary experiments show that limited-capacity buffers and agenda greatly reduce agenda management time. However, with only these design features in the cycle, Guardian fails to respond to important events.

Finally, we believe that the quality of the control plan plays a crucial role in the effectiveness of the satisficing cycle. Very good control plans allow the agent to find the best operations quickly. But what if the agent does not have a good control plan? One possibility is for the agent to recognize that its control plan will not help it identify good reasoning operations. In that case, rather than exhaustively identifying all reasoning operations in a vain search for a good one, the agent can choose to execute an arbitrary operation early. In an intermediate situation, with a moderately good control plan, the satisficing cycle should behave like an "anytime algorithm" [Dean and Boddy, 1988], trading the amount of time it spends identifying new operations for the expected value of the "best-so-far" operation identified. In sum, much of our future work will be directed toward understanding the nature of good and bad control plans and understanding how an agent can adapt its performance to control plan quality in order to maintain a satisfactorily high global utility.

References

[Darwiche *et al.*, 1988] A. Darwiche, R. E. Levitt, and B. Hayes-Roth. Oarplan: generating project plans by reasoning about objects, actions and resources. *AI EDAM*, 2(3):169-181, 1988.

[Dean and Boddy, 1988] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49-54, Saint Paul, Minnesota, 1988.

[Dean, 1989] T. Dean. Decision-theoretic control of inference for time-critical applications. Technical Report CS-89-44, Brown University, 1989.

[Dodhiawala *et al.*, 1989] R. Dodhiawala, N. S. Sridharan, P. Raulefs, and C. Pickering. Real-time AI systems: a definition and an architecture. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 256-261, Detroit, Michigan, 1989.

[Durfee and Lesser, 1986] E. Durfee and V. R. Lesser. Incremental planning to control a blackboard-based problem-solver. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 58-64, Philadelphia, Pennsylvania, 1986.

[Forgy, 1982] C. L. Forgy. RETE: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, 19:17-32, 1982.

[Genesereth and Smith, 1982] M. R. Genesereth and D. E. Smith. Meta-level architecture. Technical Report HPP-81-6, Stanford University, 1982.

[Garvey *et al.*, 1987] A. Garvey, C. Cornelius, and B. Hayes-Roth. Computational costs versus benefits of control reasoning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 110-115, Seattle, Washington, 1987.

[Hayes-Roth and Hayes-Roth, 1979] B. Hayes-Roth and F. Hayes-Roth. A cognitive model of planning. *Cognitive Science*, 3:275-310, 1979.

[Hayes-Roth, 1985] B. Hayes-Roth. A Blackboard architecture for control. *Artificial Intelligence*, 26(3):251-321, 1985.

[Hayes-Roth *et al.*, 1986] B. Hayes-Roth, B. G. Buchanan, O. Lichtarge, M. Hewett, R. Altman, J. Brinkley, C. Cornelius, B. Duncan, and O. Jardetzky. Protean: Deriving protein structure from constraints. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 904-909, Philadelphia, Pennsylvania, 1986.

[Hayes-Roth, 1987] B. Hayes-Roth. A multi-processor interrupt-driven architecture for adaptive intelligent systems. Technical Report KSL-87-31, Stanford University, 1987.

[Hayes-Roth *et al.*, 1989] B. Hayes-Roth, R. Washington, R. Hewett, M. Hewett, and A. Seiver. Intelligent real-time monitoring and control. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 243-249, Detroit, Michigan, 1989.

[Hayes-Roth, 1990] B. Hayes-Roth. Architectural foundations for real-time performance in intelligent agents. *Real-Time Systems*, pages 99-125, 1990.

[Johnson and Hayes-Roth, 1987] M. V. Johnson and B. Hayes-Roth. Integrating diverse reasoning methods in the BB1 blackboard control architecture. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 30-35, Seattle, Washington, 1987.

[McDermott and Forgy, 1978] J. McDermott and C. L. Forgy. Production system conflict resolution strategies. In Waterman, D.A., and Hayes-Roth, F. (eds), *Pattern-Directed Inference Systems*, Academic Press, 1978.

[Newell, 1973] A. Newell. Production systems: models of control structures. In Chase W.G. (ed), *Visual Information Processing*, Academic Press, 1973.

[Russel and Wefald, 1989] S. J. Russel and E. H. Wefald. Principles of Metareasoning. In Brachman et al. (eds), *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufman, 1989.

[Tommelein *et al.*, 1987] I. D. Tommelein, M. V. Johnson, R. E. Levitt, and B. Hayes-Roth. SightPlan: a blackboard expert system for the layout of temporary facilities on construction site. In *Proceedings of the IFIP WG5.2 Working Conference on Expert Systems in Computer-Aided Design*, 1987.

[Washington and Hayes-Roth, 1989] R. Washington and B. Hayes-Roth. Managing input data in real-time AI systems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 250-255, Detroit, Michigan, 1989.

[Washington *et al.*, 1990] R. Washington, L. Boureau, and B. Hayes-Roth. Using knowledge for real-time input data management. Technical Report KSL-90-14, Stanford University, 1990.

Planning and Active Perception

Thomas Dean* Kenneth Basye Moises Lejter
Department of Computer Science
Brown University, Box 1910, Providence, RI 02912

Abstract

We present an approach to building planning and control systems that integrates sensor fusion, prediction, and sequential decision making. The approach is based on Bayesian decision theory, and involves encoding the underlying planning and control problem in terms of a compact probabilistic model for which evaluation is well understood. The computational cost of evaluating such a probabilistic model can be accurately estimated by inspecting the structure of the graph used to represent the model. We illustrate our approach using a robotics problem that requires spatial and temporal reasoning under uncertainty and time pressure. We use the estimated computational cost of evaluation to justify representational tradeoffs required for practical application.

Introduction

In this paper, we view planning in terms of enumerating a set of possible courses of action, evaluating the consequences of those courses of action, and selecting a course of action whose consequences maximize a particular performance (or *value*) function. We adopt Bayesian decision theory [Raiffa and Schlaifer, 1961] as the theoretical framework for our discussion, since it provides a convenient basis for dealing with decision making under uncertainty.[1]

One interesting thing about most planning problems is that the results of actions can increase our knowledge, potentially improving our ability to make decisions. From a decision theoretic perspective, there is no difference between actions that involve sensing or movement to facilitate sensing and any other actions; a decision maker simply tries to choose actions that maximize expected value. In the approach described in this paper, an agent engaged in a particular perceptual task selects a set of sensor views by physically moving about [Bajcsy, 1988, Ballard, 1989].

Having committed to a decision theoretic approach, there are specific problems that we have to deal with. The most difficult concern representing the problem and obtaining the necessary statistics to quantify the underlying decision model. In the robotics problems we are working on, the latter is relatively straightforward, and so we will concern ourselves primarily with the former.

In building a decision model for control purposes, it is not enough to write down all of your preferences and expectations; this information might provide the basis for constructing some decision model, but it will likely be impractical from a computational standpoint. It is frustrating when you know what you want to compute but cannot afford the time to do so. Some researchers respond by saying that eventually computing machinery will be up to the task and ignore the computational difficulties. It is our contention, however, that the combinatorics inherent in sequential decision making will continue to outstrip computing technologies.

In the following, we describe a concrete problem to ground our discussion, present the general sequential decision making model and its application to the concrete problem, show how to estimate the computational costs associated with using the model, and, finally, describe how to reduce those costs to manageable levels by making various representational tradeoffs.

Mobile Target Localization

The application that we have chosen to illustrate our approach involves a mobile robot navigating and tracking moving targets in a cluttered environment. The robot is provided with sonar and rudimentary vision. The moving target could be a person or another mobile robot. The mobile base consists of a holonomic (turn-in-place) synchro-drive robot equipped with a CCD camera mounted on a pan-and-tilt head, and 8 fixed Polaroid sonar sensors arranged in pairs directed for-

*This work was supported in part by a National Science Foundation Presidential Young Investigator Award IRI-8957601 with matching funds from IBM, and by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Air Force Office of Scientific Research under Contract No. F49620-88-C-0132.

[1] See Dean and Wellman [1989] for a discussion concerning the use of goals in artificial intelligence and the use of value functions in decision theory.

ward, backward, right, and left.

The robot's task is to detect and track moving objects, reporting their location in the coordinate system of a global map. The environment consists of one floor of an office building. The robot is supplied with a floor plan of the office showing the position of permanent walls and major pieces of furniture such as desks and tables. Smaller pieces of furniture, potted plants and other assorted clutter constitute obstacles that the robot has to detect and avoid.

We assume that there is error in the robot's movement requiring it to continually estimate its position with respect to the floor plan so as not to become lost. Position estimation (*localization*) is performed by having the robot track *beacons* corresponding to walls and corners and then use these beacons to reduce error in its position estimate.

Localization and tracking are frequently at odds with one another. A particular localization strategy may reduce position errors while making tracking difficult, or improve tracking while losing registration with the global map. The trick is to balance the demands of localization against the demands of tracking. The mobile target localization (MTL) problem is particularly appropriate for planning research as it requires considerable complexity in terms of temporal and spatial representation, and involves time pressure and uncertainty in sensing and action.

Model for Time and Action

In this section, we provide a decision model for the MTL problem. To specify the model, we quantize the space in which the robot and its target are embedded. A natural quantization can be derived from the robot's sensory capabilities.

The robot's sonar sensors enable it to recognize particular patterns of free space corresponding to various configurations of walls and other permanent objects in its environment (e.g., corridors, L junctions and T junctions). We tessellate the area of the global map into regions such that the same pattern is detectable anywhere within a given region. This tessellation provides a set of locations $\mathcal{L}$ corresponding to the regions that are used to encode the location of both the robot and its target.

Our decision model includes two variables S_T and S_R, where S_T represents the location of the target and ranges over $\mathcal{L}$, and S_R represents the location and orientation of the robot and ranges over an extension of $\mathcal{L}$ including orientation information specific to each type of location. For any particular instance of the MTL problem, we assume that a geometric description of the environment is provided in the form of a CAD model. Given this geometric description and a model for the robot's sensors, we generate $\mathcal{L}$, S_R, and S_T.

We encode our decision models as a *Bayesian networks* [Pearl, 1988]. A Bayesian network is a directed

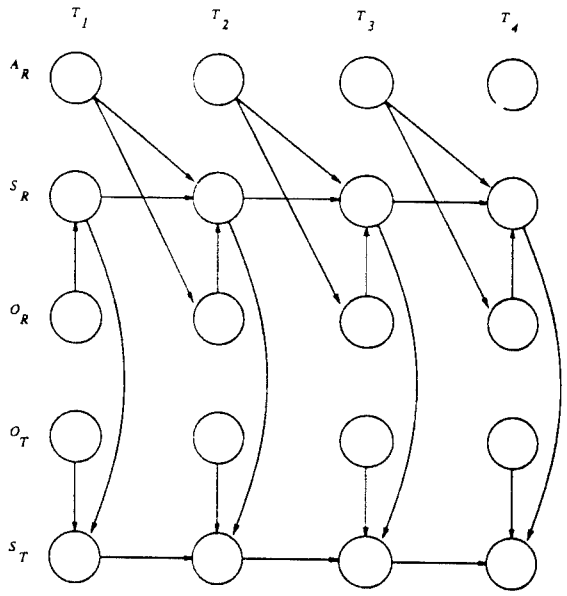


Figure 1: Probabilistic model for the MTL problem

graph $G = (V, E)$. The vertices in V correspond to random variables and are often called *chance nodes*. The edges in E define the causal and informational dependencies between the random variables. In the model described in this paper, chance nodes are discrete valued variables that encode states of knowledge about the world. Let Ω_C denote the set of possible values (*state space*) of the chance node C. There is a probability distribution $\Pr(C = \omega, \omega \in \Omega_C)$ for each node. If the chance node has no predecessors then this is its marginal probability distribution; otherwise, it is a conditional probability distribution dependent on the states of the immediate predecessors of C in G.

The model described here involves a specialization of Bayesian networks called *temporal belief networks* [Dean and Kanazawa, 1989]. Given a set of discrete variables, $\mathcal{X}$, and a finite ordered set of time points, $\mathcal{T}$, we construct a set of chance nodes, $\mathcal{C} = \mathcal{X} \times \mathcal{T}$, where each element of $\mathcal{C}$ corresponds to the value of some particular $x \in \mathcal{X}$ at some $t \in \mathcal{T}$. Let C_t correspond to the subset of $\mathcal{C}$ restricted to t. The temporal belief networks discussed in this paper are distinguished by the following Markov property:

$$\Pr(C_t | C_{t-1}, C_{t-2}, \ldots) = \Pr(C_t | C_{t-1}).$$

Let S_R and S_T be variables ranging over the possible locations of the robot and the target respectively. Let A_R be a variable ranging over the actions available to the robot. At any given point in time, the robot can make observations regarding its position with respect to nearby walls and corners and the target's position with respect to the robot. Let O_R and O_T be variables ranging these observations with respect to the robot's surroundings and the target's relative location.

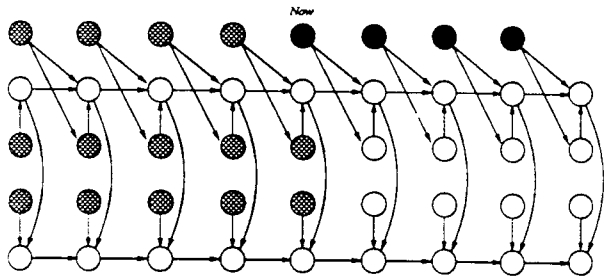


Figure 2: Evidence and action sequences

Figure 1 shows a temporal belief network for $\mathcal{X} = \{S_R, S_T, A_R, O_R, O_T\}$ and $\mathcal{T} = \{T_1, T_2, T_2, T_4\}$. To quantify the model shown in Figure 1, we have to provide distributions for each of the variables in $\mathcal{X} \times \mathcal{T}$. We assume that the model does not depend on time, and, hence, we need only provide one probability distribution for each $x \in \mathcal{X}$. For instance, the conditional probability distribution for S_T,

$$\Pr(\langle S_T, t\rangle | \langle S_T, t\text{–}1\rangle, \langle O_T, t\rangle, \langle S_R, t\rangle),$$

is the same for any $t \in \mathcal{T}$. The numbers for the probability distributions can be obtained by experimentation without regard to any particular global map.

In a practical model consisting of more than just the four time points shown in Figure 1, some points will refer to the past and some to the future. One particular point is designated the current time or *Now*. Representing the past and present will allow us to incorporate evidence into the model. By convention, the nodes corresponding to observations are meant to indicate observations *completed* at the associated time point, and nodes corresponding to actions are meant to indicate actions *initiated* at the associated time point. The actions of the robot at past time points and the observations of the robot at past and present time points serve as evidence to provide conditioning events for computing a posterior distribution. For instance, having observed σ at T, denoted $\langle O_R{=}\sigma, T\rangle$, and initiated α at T–1, denoted $\langle A_R{=}\alpha, T\text{–}1\rangle$, we will want to compute the posterior distribution for S_R at T given the evidence:

$$\Pr(\langle S_R{=}\omega, T\rangle, \omega \in \Omega_{S_R} | \langle O_R{=}\sigma, T\rangle, \langle A_R{=}\alpha, T\text{–}1\rangle).$$

To update the model as time passes, all of the evidence nodes are shifted into the past, discarding the oldest evidence in the process. Figure 2 shows a network with nine time points. The lighter shaded nodes correspond to evidence. As new actions are initiated and observations are made, the appropriate nodes are instantiated as conditioning nodes, and all of the evidence is shifted to the left by one time point.

The darker shaded nodes shown in Figure 2 indicate nodes that are instantiated in the process of evaluating possible sequences of actions. For evaluation purposes, we employ a simple *time-separable* value function. By time separable, we mean that the total value is a (perhaps weighted) sum of the value at the different time points. If V_t is the value function at time t, then the total value, V, is defined as

$$V = \sum_{t \in \mathcal{T}} \gamma(t) V_t,$$

where $\gamma : \mathcal{T} \rightarrow \{x | 0 \leq x \leq 1\}$ is a decreasing function of time used to discount the impact of future consequences. Since our model assumes a finite $\mathcal{T}$, we already discount some future consequences by ignoring them altogether; γ just gives us a little more control over the immediate future. For V_t, we use the following function

$$V_t = -\sum_{\omega_i, \omega_j \in \Omega_{S_T}} \Pr(\langle S_T{=}\omega_i, t\rangle) \Pr(\langle S_T{=}\omega_j, t\rangle) \mathrm{Dist}(\omega_i, \omega_j),$$

where $\mathrm{Dist} : \Omega_{S_T} \times \Omega_{S_T} \rightarrow \Re$ determines the relative Euclidean distance between pairs of locations. The V_t function reflects how much uncertainty there is in the expected location for the target. For instance, if the distribution for $\langle S_T, t\rangle$ is strongly weighted toward one possible location in Ω_{S_T}, then V_t will be close to zero. The more places the target could be and the further their relative distance, the more negative V_t.

The actions in Ω_{A_R} consist of tracking and localization routines (e.g., move along the wall on your left until you reach a corner). Each action has its own termination criteria (e.g., reaching a corner). We assume that the robot has a set of strategies, $\mathcal{S}$, consisting of sequences of such actions, where the length of sequences in $\mathcal{S}$ is limited by the number of present and future time points. For the network shown in Figure 2, we have

$$\mathcal{S} \subset \Omega_{A_R} \times \Omega_{A_R} \times \Omega_{A_R} \times \Omega_{A_R}.$$

The size of $\mathcal{S}$ is rather important, since we propose to evaluate the network $|\mathcal{S}|$ times at every decision point. The strategy with the highest expected value is that strategy, $\varphi = \alpha_0, \alpha_1, \alpha_2, \alpha_3$, for which V is a maximum, conditioning on $\langle A_\tau{=}\alpha_0, Now\rangle$, $\langle A_\tau{=}\alpha_1, Now{+}1\rangle$, $\langle A_\tau{=}\alpha_2, Now{+}2\rangle$, and $\langle A_\tau{=}\alpha_3, Now{+}3\rangle$. The best strategy to pursue is reevaluated every time that an action terminates.

We use Jensen's [1989] variation on Lauritzen and Spiegelhalter's [1988] algorithm to evaluate the decision network. Jensen's algorithm involves constructing a hyper graph (called a *clique tree*) whose vertices correspond to the (maximal) cliques of the chordal graph formed by triangulating the undirected graph obtained by first connecting the parents of each node in the network and then eliminating the directions on all of the edges. The cost of evaluating a Bayesian network using this algorithm is largely determined by the sizes of the state spaces formed by taking the cross product of the state spaces of the nodes in each vertex (clique) of the clique tree.

Following Kanazawa [Forthcoming], we can obtain an accurate estimate of the cost of evaluating a Bayesian network, $G = (V, E)$, using Jensen's algorithm. Let $C = \{C_i\}$ be the set of (maximal) cliques in the chordal graph described in the previous paragraph, where each clique represents a subset of V. We define the function, card : $C \rightarrow \{1, \ldots, |C| - 1\}$, so that card$(C_i)$ is the rank of the highest ranked node in C_i, where rank is determined by the maximal cardinality ordering of V (see [Pearl, 1988]). We define the function, adj : $C \rightarrow 2^C$, by:

$$\text{adj}(C_i) = \{C_j | (C_j \neq C_i) \wedge (C_i \cap C_j \neq \emptyset)\}.$$

The clique tree for G is constructed as follows. Each clique $C_i \in C$ is connected to the clique C_j in adj(C_i) that has lower rank by card(.) and has the highest number of nodes in common with C_i (ties are broken arbitrarily). Whenever we connect two cliques C_i and C_j, we create the *separation set* $S_{ij} = C_i \cap C_j$. The set of separation sets S is all the S_{ij}'s. We define the function, sep : $C \rightarrow 2^S$, by:

$$\text{sep}(C_i) = \{S_{jk} | S_{jk} \in S, (j = i) \vee (k = i)\}.$$

Finally, we define the *weight* of C_i, $w_i = \prod_{n \in C_i} |\Omega_n|$, where Ω_n is the state space of node n. The cost of computation is proportional to $\sum_{C_i \in C} w_i |\text{sep}(C_i)|$. We refer to this cost estimate as the *clique-tree cost.*

The approach described in this section allows us to integrate prediction, observation, and control in a single model. It also allows us to handle uncertainty in sensing, movement, and modeling. Behavioral properties emerge as a consequence of the probabilistic model and the value function provided, not as a consequence of explicitly programming specific behaviors. The main drawback of the approach is that, while the model is quite compact, the computational costs involved in evaluating the model can easily get out of hand. For instance, in our model for the MTL problem, the clique-tree cost is bounded from below by the product of $|\mathcal{T}|$, $|\Omega_{S_T}|^2$, and $|\Omega_{S_R}|^2$. In the next section, we provide several methods that, taken together, allow us to reduce computational costs to practical levels.

Coping with Complexity

To reduce the cost of evaluating the MTL decision model, we use the following three methods: (i) carefully tailor the spatial representation to the robot's sensory capabilities, reducing the size of the state space for the spatial variables in the decision model, (ii) enable the robot to dynamically narrow the range of the spatial variables using heuristics to further reduce the size of the state space for the spatial variables, and (iii) consider only a few candidate action sequences from a fixed library of tracking strategies by taking into account the reduced state space of the spatial variables. In the rest of this section, we consider each of these three methods.

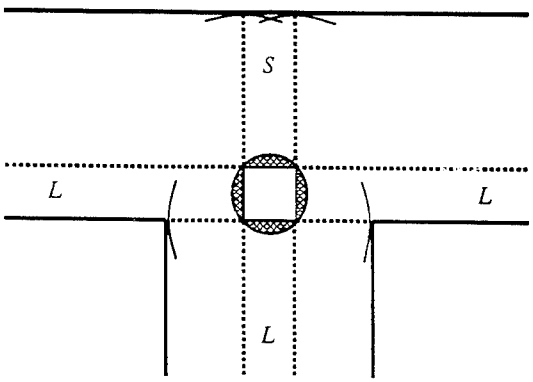


Figure 3: Sonar data entering a T junction

The use of a high-resolution representation of space has disadvantages in the model proposed here: increasing the resolution of the representation of space results in an increase in the sizes of Ω_{S_R} and Ω_{S_T}, and thus raises the cost of evaluating the network. Keeping the sizes of Ω_{S_R} and Ω_{S_T} small makes the task of evaluating the model we propose feasible.

A further consideration arises from the real-world sensory and data processing systems available to our robot. Finer-resolution representations of space place larger demands on the robot's on-board system in terms of both run-time processing time and sensor accuracy. To allow our robot to achieve (near) real-time performance, it seems appropriate to limit the representation to that level of detail that can be obtained economically from the hardware available.

In our current implementation, we have 8 sonar transducers positioned on a square platform, two to a side, spaced about 25 cm. apart. We take distance readings from each transducer, and threshold the values at about 1 meter. Anything above the threshold is "long," anything below is "short." The readings along each side are then combined by voting, with ties going to "long." In this way, the data from the sonar is reduced to 4 bits. Figure 3 shows the result of this scheme on entering a T junction. In addition, we use the shaft encoders on our platform to provide very rough metric information for the decision model. Currently, 2 additional bits are used for this purpose, but only when the robot is positioned in a hallway, which corresponds to only one sonar configuration. So the total number of possible states for O_R is 19, 15 for various kinds of hallway junctions and 4 more for corridors.

This technique results in a tessellation of space like that shown in Figure 4. Our experiments have shown that this tessellation is quite robust in the sense that the readings are consistent anywhere in a given tile. The exception to this occurs when the robot is not well-aligned with the surrounding walls. In these cases, reflections frequently make the data unreliable. One of the tasks of the controllers that underlie the actions described in

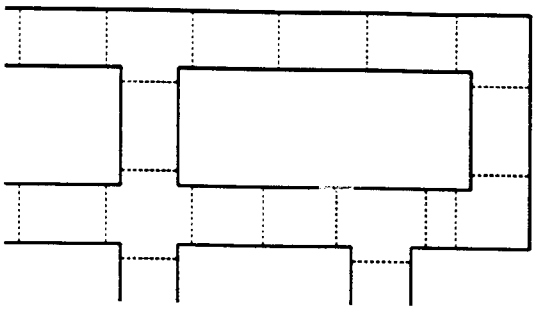


Figure 4: Tessellation of office layout

the previous sections is to maintain good alignment, or achieve it if it is lost.

In addition to reducing the size of the overall spatial representation, we can restrict the range of particular spatial variables on the basis of evidence not explicitly accounted for in the decision model (e.g., odometry and compass information). For instance, if we know that the robot is in one of two locations at time 1 and the robot can move at most a single location during a given time step, then $\langle S_R, 1\rangle$ ranges over the two locations, and, for $i > 1$, $\langle S_R, i\rangle$ need only range over the locations in or adjacent to those in $\langle S_R, i\text{-}1\rangle$. Similar restrictions can be obtained for S_T. For models with limited lookahead (i.e., small $|\mathcal{T}|$), these restrictions can result in significant computational savings.

Consider a temporal Bayesian network of the form shown in Figure 1 with n steps of lookahead. Let $\langle X, i\rangle$ represent an element of $\{S_R, S_T, A_R, O_R, O_T\} \times \{1, \ldots, n\}$. The largest cliques in one possible[2] clique tree for this network consist of sets of variables of the form:

$$\{\langle S_R, i\rangle, \langle S_R, i\text{+}1\rangle, \langle S_T, i\rangle, \langle S_T, i\text{+}1\rangle\}$$

for $i = 1$ to $n-1$, and the size of the corresponding cross product space is the product of $|\Omega_{\langle S_R,i\rangle}|$, $|\Omega_{\langle S_R,i+1\rangle}|$, $|\Omega_{\langle S_T,i\rangle}|$, and $|\Omega_{\langle S_T,i+1\rangle}|$. For fixed state spaces, this product is just $|\Omega_{S_R}|^2|\Omega_{S_T}|^2$. However, if we restrict the state spaces for the spatial variables on the basis of some initial location estimate and some bounds on how quickly the robot and the target can move about, we can do considerably better.

Table 1 shows the clique-tree costs for three MTL decision model networks of size $n = 3$, 5, and 8 time points. For each size of model, we consider cases in which $\Omega_{\langle S_R,i\rangle}$ and $\Omega_{\langle S_T,i\rangle}$ are constant for all $1 \geq i \geq n$, and cases in which $|\Omega_{\langle S_R,1\rangle}| = |\Omega_{\langle S_T,1\rangle}| = 1$ and the sizes of the state spaces for subsequent spatial variables, $\Omega_{\langle S_R,i\rangle}$ and $\Omega_{\langle S_T,i\rangle}$, for $1 > i \geq n$ grow by

[2]The triangulation algorithm attempts to minimize the size of the largest clique in the resulting chordal graph. There may be more than one way to triangulate a graph so as to minimize the clique size.

State space size	Number of time points 3	5	8
Constant (6)	40914 (0.58)	78066 (1.11)	133794 (1.90)
Constant (16)	624944 (8.87)	1232176 (17.49)	2143024 (30.42)
Constant (30)	3846330 (54.60)	7669530 (108.86)	13404330 (190.26)
Linear $(2t+1)$	5844 (0.08)	55088 (0.78)	433759 (6.16)
Quadratic (t^2+1)	3691 (0.05)	160701 (2.28)	3756559 (53.32)
Exponential (2^t)	2875 (0.05)	107515 (1.53)	4131611 (58.64)

Table 1: Clique-tree costs for sample networks

linear, quadratic, and exponential factors bounded by $|\Omega_{S_T}| = |\Omega S_R| = 30$. For these evaluations, $|\Omega_{A_R}| = 6$, $|\Omega_{O_T}| = 32$, and $|\Omega_{O_R}| = 19$ in keeping with the sensory and movement routines of our current robot. The number in brackets underneath the clique tree cost is the time in cpu seconds required for evaluation.

Our current idea for restricting the present location of the robot and the target involves using a fixed threshold and the most up-to-date estimates for these locations to eliminate unlikely possibilities. Occasionally, the actual locations will be mistakenly eliminated, and robot will fail to track the target. There will have to be a recovery strategy and a criterion for invoking it to deal with such failures.

There are certain costs involved with evaluating Bayesian networks that we have ignored so far. These costs involve triangulating the graph, constructing the clique tree, and performing the storage allocation for building the necessary data structures. For our approach of dynamically restricting the range of spatial variables, the state spaces for the random variables change, but the sizes of these state spaces and the topology of the Bayesian network remain constant. As a consequence, these ignored costs are incurred once, and the associated computational tasks can be carried out at design time. Dynamically adjusting the state spaces for the spatial variables is straightforward and computationally inexpensive.

The third method for reducing the cost of decision making involves reducing the size of $\mathcal{S}$, the set of sequences of actions corresponding to tracking and localization strategies. For an n step lookahead, the set of useful strategies of length n or less is a very small subset of ${\Omega_{A_R}}^n$. Still, given that we have to evaluate the network $|\mathcal{S}|$ times, even a relatively small $\mathcal{S}$ can cause problems. To reduce $\mathcal{S}$ to an acceptable size, we only evaluate the network for strategies that are possible given the current restrictions on the spatial vari-

ables. For instance, if the robot knows that it is moving down a corridor toward a left-pointing L junction, it can eliminate from consideration any strategy that involves it moving to the end of the corridor and turning right. With appropriate preprocessing, it is computationally simple to dynamically reduce $\mathcal{S}$ to just a few possible strategies in most cases.

Related Work

Probabilistic decision models of the sort explored in this paper are just beginning to see use in planning and control. Agogino and Ramamurthi [1988] describe the use of probabilistic models for controlling machine tools. Dean *et al* [1990] show how to use Bayesian networks for building maps and reasoning about the costs and benefits of exploration. Kanazawa and Dean [Kanazawa and Dean, 1989] extend temporal Bayesian networks to handle sequential decision making tasks. Levitt *et al* [1988] describe an approach to implementing object recognition using Bayesian networks that accounts for the cost of sensor movement and inference. Wellman [1987] shows how to integrate qualitative knowledge in probabilistic network models. For some previous approaches to using decision and probability theory in planning, see [Feldman and Sproull, 1977, Langlotz *et al.*, 1987]. For some recent work on temporal reasoning under uncertainty, see [Cooper *et al.*, 1988, Dean and Kanazawa, 1988, Hanks, 1988, Weber, 1989].

References

[Agogino and Ramamurthi, 1988] A. M. Agogino and K. Ramamurthi. Real-time influence diagrams for monitoring and controlling mechanical systems. Technical report, Department of Mechanical Engineering, University of California, Berkeley, 1988.

[Bajcsy, 1988] R. Bajcsy. Active perception. *Proceedings of the IEEE*, 76(8):996–1005, 1988.

[Ballard, 1989] Dana H. Ballard. Reference frames for animate vision. In *Proceedings IJCAI 11*, pages 1635–1641. IJCAI, 1989.

[Cooper *et al.*, 1988] Gregory F. Cooper, Eric J. Horvitz, and David E. Heckerman. A method for temporal probabilistic reasoning. Technical Report KSL-88-30, Stanford Knowledge Systems Laboratory, 1988.

[Dean and Kanazawa, 1988] Thomas Dean and Keiji Kanazawa. Probabilistic temporal reasoning. In *Proceedings AAAI-88*, pages 524–528. AAAI, 1988.

[Dean and Kanazawa, 1989] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.

[Dean and Wellman, 1989] Thomas Dean and Michael Wellman. On the value of goals. In Josh Tenenberg, Jay Weber, and James Allen, editors, *Proceedings from the Rochester Planning Workshop: From Formal Systems to Practical Systems*, pages 129–140, 1989.

[Dean *et al.*, 1990] Thomas Dean, Kenneth Basye, Robert Chekaluk, Seungseok Hyun, Moises Lejter, and Margaret Randazza. Coping with uncertainty in a control system for navigation and exploration. In *Proceedings AAAI-90*. AAAI, 1990.

[Feldman and Sproull, 1977] Jerome Feldman and Robert Sproull. Decision theory and artificial intelligence ii: the hungry machine. *Cognitive Science*, 1:158–192, 1977.

[Hanks, 1988] Steve Hanks. Representing and computing temporally scoped beliefs. In *Proceedings AAAI-88*, pages 501–505. AAAI, 1988.

[Jensen, 1989] Finn V. Jensen. Bayesian updating in recursive graphical models by local computations. Technical Report R-89-15, Institute for Electronic Systems, Department of Mathematics and Computer Science, University of Aalborg, 1989.

[Kanazawa and Dean, 1989] Keiji Kanazawa and Thomas Dean. A model for projection and action. In *Proceedings IJCAI 11*, pages 985–990. IJCAI, 1989.

[Kanazawa, Forthcoming] Keiji Kanazawa. *Probability, Time, and Action.* PhD thesis, Brown University, Providence, RI, Forthcoming.

[Langlotz *et al.*, 1987] Curtis P. Langlotz, Lawrence M. Fagan, Samson W. Tu, Branimir I. Sikic, and Edward H. Shortliffe. A therapy planning architecture that combines decision theory and artificial intelligence techniques. *Computers and Biomedical Research*, 20:279–303, 1987.

[Lauritzen and Spiegelhalter, 1988] Stephen L. Lauritzen and David J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, 50(2):157–194, 1988.

[Levitt *et al.*, 1988] Tod Levitt, Thomas Binford, Gil Ettinger, and Patrice Gelband. Utility-based control for computer vision. In *Proceedings of the 1988 Workshop on Uncertainty in Artificial Intelligence*, 1988.

[Pearl, 1988] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan-Kaufmann, Los Altos, California, 1988.

[Raiffa and Schlaifer, 1961] Howard Raiffa and R. Schlaifer. *Applied Statistical Decision Theory.* Harvard University Press, 1961.

[Weber, 1989] Jay C. Weber. A parallel algorithm for statistical belief refinement and its use in causal reasoning. In *Proceedings IJCAI 11.* IJCAI, 1989.

[Wellman, 1987] Michael P. Wellman. Dominance and subsumption in constraint-posting planning. In *Proceedings IJCAI 10.* IJCAI, 1987.

A Cooperative Approach to Planning for Real-Time Control

Edmund H. Durfee
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109
durfee@caen.engin.umich.edu

Abstract

Research into combining real-time control with AI planning has typically involved attempting to embed "intelligence" in a real-time system or "reactivity" in an AI system. We argue in favor of an alternative approach in which distinct real-time and AI systems perform the functions each is suitable for and cooperate to achieve overall real-time intelligent behavior. The real-time system guarantees the performance of some schedulable subset of important actions—a subset judiciously chosen by the asynchronously running AI system. We describe a preliminary implementation of our cooperative approach for planning routes and controlling the behavior of a hall-following robot. We conclude by outlining important open issues, including building faster AI systems, real-time communication, and ties between real-time AI and distributed problem solving.

1 Introduction

In dynamically changing worlds, intelligent decision making cannot be divorced from time. The best decision can result in failure if the world has changed substantially by the time the decision is enacted. For example, if I see no cars coming, I might decide to cross the street. Although this decision might be correct given the initial situation, if I spend too much time making the decision or putting it into action, I might still get hit. A fast and simple rule such as "If you see no cars coming then cross the street" can reduce decision-making time, but it also might lead to disaster on a foggy day or near a blind turn. Hence, an important challenge in building autonomous systems for dynamic worlds is in combining mechanisms for making rapid "reactive" decisions in time-critical situations with AI techniques for planning and anticipating outcomes given more time to reason.

We have developed a new approach to real-time AI that combines intelligent planning and real-time control. Our approach is based on a premise that is fundamentally different from previous work: We view AI and real-time computing to be incompatible by nature, and hence attempts to build an integrated, real-time AI system will never fully succeed. Instead, we propose a cooperative approach, where distinct real-time and AI systems cooperatively solve problems requiring high-level planning and reactive control.

In this paper, we briefly survey previous work on intelligent planning and real-time control to illustrate how our perspective differs (Section 2). We then discuss our approach in general terms (Section 3), and contrast it in more depth with other approaches (Section 4). Then we describe our prototype implementation for controlling a robot in an uncertain and dynamic environment (Section 5). Using our prototype as a starting point, we discuss the promises and pitfalls of our new approach (Section 6). Finally, we conclude by summarizing the current status of this work and our ongoing research (Section 7).

2 Intelligent Real-Time Systems

Intelligent real-time systems are critically important in most real-world tasks where it is not enough for a system to eventually respond to a situation; situations change over time, so a timely response to the current situation is useful while a late response is useless. By real-time, therefore, we mean that a system must carry out its actions before the environment has a chance to change substantially. Put another way, a system must act on its environment more quickly than its environment can unpredictably act on it. If we can measure the expected (or minimum) amount of time that the environment needs to change substantially, then we can place hard real-time deadlines on a system [Stankovic and Ramamritham, 1987].

2.1 Real-Time AI: A Contradiction?

Intelligent real-time systems are elusive for several reasons. One reason stems from the fact that both AI and real-time computing are relatively young disciplines without established, broadly-based principles to act as a foundation for developing intelligent real-time systems. A second reason is that the philosophies of the two communities are widely divergent. While the AI community looks to continually blur the limits of what computers can do, the real-time community attempts to set clearly-defined limits on computing capabilities and re-

[0]This research was sponsored, in part, by the University of Michigan under a Rackham Faculty Research Grant.

quirements in order to guarantee a desired level of performance.

A third reason that building intelligent real-time systems is hard is the different, essentially incompatible, objectives of researchers in the two communities. A long-standing goal of AI research is to build systems that can change based on new experiences and that thus could in time develop better but possibly unexpected solutions to problems. By gathering more knowledge, the system becomes capable of new chains of inference and must retrieve appropriate knowledge from a growing knowledge base. If its knowledge base could potentially grow without any clearly defined bounds (especially if it has access to effectively unlimited archival memory), then it is unlikely that a system could guarantee a known, bounded retrieval and response time. Meanwhile, a goal of real-time computing research is to clearly define the system's capabilities and resources in order to predictably guarantee that important deadlines and timing constraints will be met. Thus, while intelligence appears to imply inherent *unpredictability*, real-time computing demands worst-case *predictability*.

These reasons are not proof that intelligent real-time systems cannot be built; the reasons are purely intuitive and rife with underlying assumptions about what it means for a system to be intelligent and real-time. In addition, current AI systems are far from the creative and adaptive intelligent systems that we might hope to build in the future, so our current systems are probably not as incompatible with real-time systems as they might become. Our goal in this research, therefore, is not to condemn attempts to combine real-time and AI concepts into a single system, but instead to keep an open mind and begin to consider alternatives approaches to building real-time intelligent systems. Before describing the alternative approach that we have developed to date, we first present an overview of approaches to building integrated real-time AI systems.

2.2 Real-Time AI: Embedded Approaches

One approach to combining real-time and AI concepts has been to engineer AI systems to meet real-time needs [Laffey *et al.*, 1988]. Typically, this means simplifying a system's knowledge-base and inference mechanism so that the system will respond to all expected inputs within some maximum time. Unfortunately, while these systems might retain some of the languages and algorithms of AI, it could be argued that whatever intelligence they began with has been engineered out in order to guarantee predictable real-time responses.

Another approach has been to develop AI systems that use iterative improvement algorithms, so that at any given time the system can return some approximation of the desired response [Dean and Boddy, 1988; Horvitz, 1987]. Systems that use this approach attain goals within real-time, but this approach is limited to applications that admit to successive-refinement algorithms. In many applications, successfully meeting time constraints might mean that the system generates a useful but unexpected result, rather than an approximation of the expected result. For example, when navigating a vehicle through a congested area, an approximation such as "turn 90 degrees, plus or minus 45 degrees" might lead to disaster, while a completely different response such as "honk your horn and slam on the brakes" might be better.

Both of these approaches have combined AI and real-time by *embedding* an AI system within a real-time system. As part of the real-time processing, any AI reasoning must also return a response within a deadline. This view can be contrasted with the view in which real-time (reactive) capabilities are *embedded* within an AI system. For example, Cohen [Cohen *et al.*, 1989] describes an AI architecture which includes a real-time component that can be triggered by certain input and that rapidly responds to time-critical situations by "short-circuiting" the more general reasoning mechanisms. A more unified approach, such as Soar [Laird *et al.*, 1987], encodes reactive knowledge just like any other knowledge, with the stipulation that, when it is applicable, the reactive knowledge should take priority. To make real-time guarantees, these systems must ensure some upper bound on the time it will take the AI system to invoke the reactive component or to retrieve and execute a reactive "rule." In a system with changing (especially growing) knowledge but limited computing resources, establishing such a bound could be problematic.

3 A Cooperative Approach

An alternative approach is to view the real-time and AI components as being separate, concurrent, and asynchronous systems. Because neither is embedded in the other, we do not need to alter the basic behavior of either. The AI system is free to change unpredictably and need not satisfy any hard real-time guarantees, while the real-time system can ensure rigid timing constraints on its own well-defined behavior. The challenge, then, is to enable the two individual systems, with their own goals and restrictions, to cooperate so that real-time intelligent behavior emerges.

In our cooperative approach, the real-time system follows a schedule of tasks, where those tasks have known effects, resource requirements, and worst-case time needs. We view the purpose of the real-time system as reacting and adapting to domain dynamics in prespecified ways so as to ensure that some behavioral goals are maintained. As a simple example, a mobile robot generally has a goal of avoiding collisions. This goal leads to ongoing obstacle avoidance behavior. The purpose of the real-time system is to guarantee that the periodic task of detecting objects looming ahead and stopping when they appear will be carried out with some worst-case frequency. Because the detection actions (arithmetic comparisons using certain sonar readings) and reactions (setting specific motor parameters to 0) are rigidly encoded, the time needs of the periodic task can be bounded. The real-time system takes a cyclic schedule of such periodic tasks and guarantees its timely performance. This ensures that a set of reactions to particular changes in the domain will occur quickly enough to maintain at least some minimal level of performance, such as keeping a mobile robot in a "safe" state until more reasoned re-

sponses can be developed.

But where does the well-defined set of tasks comes from, and how can we ensure that it can be accomplished? The answer is the AI system. The AI system has the knowledge and reasoning power to interpret the current situation, to consider the overall system objectives, to plan and anticipate for the future, and to thus decide on the active and reactive behaviors that should be maintained at any given time. Of course, given unlimited resources, the AI system might prefer to maintain all reactive behaviors, but this might not be feasible. To determine whether a desired set of behaviors can be guaranteed by the real-time system, the AI system uses established real-time scheduling techniques to generate the real-time system's schedule. If these techniques cannot form a guaranteed schedule for the chosen behaviors, then the AI system must use this feedback to modify its expectations.

For example, let us say the AI system decides that the most relevant behaviors for a robot's next activity are (1) moving forward at speed s while (2) checking for obstacles and stopping before collisions and (3) checking for sensor readings indicating the arrival at a desired location without overshooting the location by more than distance d. Behaviors (2) and (3) must be repeated at a frequency determined by the AI system based on the desired parameters s and d. These behavioral tasks and their periods are passed to the real-time scheduling algorithms. If the algorithms can successfully schedule the tasks, they return an executable schedule which the AI system can then pass to the real-time system. If the tasks cannot be scheduled, then it is up to the AI system to relax expectations so that a satisfactory schedule can be formed. For example, if it can afford to spend more time traveling, then the AI system can reduce s, which in turn reduces the frequency at which behaviors (2) and (3) need to be repeated. On the other hand, if arrival at the destination must proceed in haste, then the AI system could choose to drop behavior (2) from the schedule and just hope that no obstacles will get in the way. Because time and other resources are limited, sometimes the AI system must intentionally choose to ignore some reactive behaviors in order to ensure more important behaviors.

In essence, our cooperative approach trades away complete flexibility in reactive behavior in favor of guaranteeing a subset of reactions. The AI system must choose what to guarantee wisely. For example, if an action to look out for cliffs was not included in the schedule, the system might very well fall off a cliff before the AI system recognizes its error and modifies the schedule. If the probability of encountering a cliff and the costs of falling off of it are high enough, however, the AI system should have planned for it. Although the AI system is free to revise the guaranteed subset of behaviors at any time, we are not restricting the AI system to meet any hard real-time requirements. This allows the AI system to apply any knowledge and inferences it chooses in deciding how to act. If it has already downloaded an appropriate schedule of behaviors that are guaranteed to keep the overall system in a safe state,[1] the AI system has the time and flexibility to carefully craft the next schedule of behaviors.

4 Comparison to Related Work

Our emphasis on using real-time scheduling algorithms to guarantee a well-defined subset of reactions differs from more typical pattern-directed invocation approaches, exemplified in rule-based and blackboard-based systems [Hayes-Roth *et al.*, 1989b; Laird *et al.*, 1987]. In pattern-directed invocation, changing state information is matched to the rules or knowledge sources to trigger appropriate responses. An advantage of pattern-directed invocation, therefore, is that unexpected events can trigger any applicable knowledge. Our approach instead forces the AI system to restrict the reactive knowledge that will be considered to only the best subset that can be considered within time constraints.

In pattern-directed invocation, the pattern-matching overhead of using state information to trigger knowledge can potentially be costly. Although faster algorithms are continually being developed and the use of parallel hardware can further speed this process, providing absolute upper bounds for pattern-matching time in a system with a changing knowledge base and fixed hardware is problematic. The advantage of our approach is that the real-time schedule explicitly targets specific reactive knowledge and the criteria for applying that knowledge. Thus, the real-time system knows exactly what patterns it will attempt to match. It also knows what state information to collect, which can lead to more focussed sensing than in the typical pattern-directed invocation approaches where all changes to state information are generally collected. Our approach is therefore more focussed and less opportunistic than pattern-directed invocation approaches, but this allows it to also be more predictable and to guarantee performance of limited behavior.

In other reactive approaches, overall system behavior emerges from the collective responses of a number of simple reactive components [Agre and Chapman, 1987; Arkin, 1987; Brooks, 1986]. As in our approach, these approaches delegate the responsibility for recognizing specific issues of concern in the current situation to separate behaviors. However, these approaches usually view the separate reactive behaviors as running concurrently on parallel processors, whereas we make no assumptions about having enough processors to give each behavior its own. Our approach is based on the expectation that, as we continually extend the range of situations that our systems will face, the amount of computing we would like to do will exceed the available computing resources. Thus, resource allocation and scheduling are critical in uniprocessor and multiprocessor implementations, and our approach using real-time scheduling algorithms guarantees performance given time and resource constraints.

Firby's RAP planner [Firby, 1987] is another example of a reactive approach where overall system behavior

[1] We also assume that the real-time system is bootstrapped with a suitable schedule.

emerges from the responses of several reactive components, but, like our approach, the RAP planner makes no assumptions about parallel hardware. Instead, the separate reactive behaviors, called RAPs, sit on an execution queue and an interpreter decides at any given time which will execute next. The RAP planner and our approach thus differ in how and where the decisions about real-time execution of reactive behaviors is accomplished. In the RAP planner, the interpreter must decide which RAP to execute next, considering its time constraints and relationships to other RAPs. While this has considerable flexibility, making such decisions might take considerable time as well. In our approach, the AI system decides which reactive behaviors are needed and uses real-time scheduling algorithms to develop a fixed schedule for execution. Moreover, our approach incorporates strategic planning capabilities in the AI system, allowing strategic and reactive planning to be combined in a more natural manner.

Our cooperative approach concentrates on guaranteeing real-time control reactions that keep an overall system in a safe state, and makes no assumptions about the timing characteristics of the AI system. As such, our approach concentrates on time constraints at the reactive control level rather than at the task level. In meeting task-level time constraints, issues in hastening decision making and balancing time spent reasoning and acting [Boddy and Dean, 1989] come to the fore. While providing real-time guarantees about AI systems is not possible in general, for some tasks and some AI algorithms it is possible to strictly bound reasoning time. For example, anytime algorithms [Dean and Boddy, 1988; Horvitz, 1987] allow the formulation of some decision within a deadline, with decisions improving as allotted time is increased. The work on approximate processing has similar goals [Lesser *et al.*, 1988]. Meanwhile, Hendler is developing an AI system on top of a real-time operating system [Hendler, 1990]. The operating system allocates time to reacting and reasoning in his system, so that in highly dynamic environments the reasoning tasks might get little or no time, while in more relaxed domains less reaction might be needed. Like ours, his approach builds on real-time computing techniques, but he uses an embedded rather than a cooperative approach.

Finally, research on using multiple systems, or "cooperating experts" has predominantly focussed on speeding up overall system performance through parallelism and filtering information [Durfee *et al.*, 1989; Hayes-Roth *et al.*, 1989b; Smith and Broadwell, 1987]. For example, the work of Hayes-Roth and her colleagues decomposes a real-time intensive-care monitoring task into a number of intelligent subsystems for interpreting sensory data, evaluating trends, enacting changes in treatment, and so on [Hayes-Roth *et al.*, 1989a]. This decomposition enables faster responses (through parallelism and information reduction), but does not separate real-time and AI capabilities as our approach does.

5 Preliminary Implementation

As a preliminary investigation of our approach, we have been experimenting with planning and control for a mobile robot that must navigate through the halls of our building using only sonar sensors and a floor plan. Our experimental environment consists of a Heathkit HERO robot connected by an RS232 cable to a TI Explorer II. Using multitasking, the TI Explorer simulates concurrency in executing both cooperating processes, the AI system and the real-time system (Figure 1).

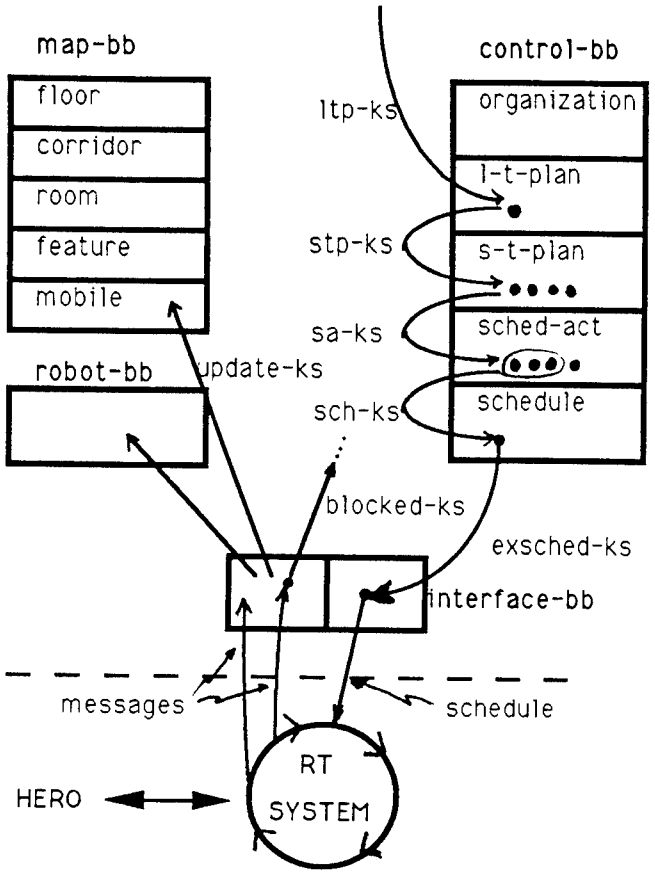


Figure 1: AI and Real-Time Systems

5.1 AI System

The AI system is responsible for taking high-level task specifications and planning routes and control behaviors for the robot. The system is implemented in a blackboard architecture using the Generic Blackboard (GBB) shell [Corkill *et al.*, 1986], and has several blackboards for control and data (Figure 1).

The map blackboard contains information about the floor plan at different levels of detail. The map provides abstract information about corridors and rooms, and more detailed information about static features (doors, partitions, etc.) and about mobile features (the robot, objects to retrieve, etc.). Knowledge sources (KSs) can access the map blackboard to plan routes, generate expectations about sonar readings, plot robot and object locations, and so on.

The control blackboard contains information about

the current and pending behaviors of the robot. A behavior is represented along the dimensions of: who (currently the single robot, but we are working toward multi-robot domains); what (the goals of the behavior); when (the time interval over which the behavior will occur); where (the spatial region in which the behavior will occur); how (the methods employed to achieve the goals); and why (the source of the behavior and its importance). The behavioral specification provides us with a common representation for organizations, plans, and schedules [Durfee and Montgomery, 1990]. The control blackboard is thus divided into levels for:

organization: general responsibilities (such as *deliveries on the first floor*);

long-term-plan: behaviors requiring substantial time (such as *pick up and deliver package x*);

short-term-plan: subcomponents of long-term-plans (such as *follow hall1 to the intersection with hall2*);

schedulable-action: behaviors to maintain during short-term-plans (such as *orient to wall* or *avoid obstacles*);

schedule: scheduled actions to meet real-time requirements.

The control KSs use information from the map and control blackboards to generate new behaviors. Currently, we have only a few KSs, and these allow the top-down elaboration of plans. Given a pickup-and-delivery request, the long-term-planning KS builds a long-term-plan behavior. The short-term-planning KS decomposes it into temporal subplans (go to x, go to y, pick up object, etc.). The schedulable-action KS generates behaviors to activate for each subplan, and the schedule KS takes the schedulable-actions associated with the current time and builds a schedule. Currently, a schedule is constructed as a lisp procedure where the functions associated with active schedulable-actions are appended within a loop construct. While frequencies and time-costs are associated with the schedulable-actions, our simplistic initial scheduling algorithm simply includes all actions.

The execute-schedule KS causes the schedule to be downloaded to the real-time system, which in our current implementation means that the schedule is transferred to the interface blackboard. This KS also adds in an additional function call to periodically test to see if a new schedule is downloaded. Thus, the AI system explicitly tells the real-time process how often to periodically poll their connecting stream to see whether a new schedule is ready to supercede the current schedule.

Communication from the real-time process to the AI system is accomplished through the interface blackboard. A control function associated with an action can generate a message, which the AI system receives through a stream and posts on the interface blackboard. For example, the avoid-collision function takes two actions when it detects an object closer than 15 inches away: it causes the robot's wheels to stop; and it generates a BLOCKED message that gets sent to the AI system.

The appearance of a message such as BLOCKED on the interface blackboard triggers additional KSs to respond to the event. For example, when blocked, the blocked KS suspends previously active behaviors and builds a short-term-plan to swivel the robot back and forth to detect the left and right boundaries of the obstruction. This leads to a new schedule that is downloaded to the real-time system. The function for sizing up the new obstruction returns a message containing its dimensions, which the AI system's update-map KS uses to add the obstruction to the map blackboard.

Finally, the robot blackboard contains information about the current state of the robot, as received from the robot via the interface blackboard. Once again, to receive such information the AI system must explicitly include commands in the schedule telling the real-time system to send messages containing the desired information.

5.2 Real-Time System

The real-time system is a separate process running on the TI Explorer. The process simply reads from the stream connecting the processes and evaluates the s-expression representing the schedule that it receives. Once the evaluation returns, it reads again from the stream, and so on. As an example schedule, the schedule for hall following has the form:

```
(loop (avoid-collisions)
      (orient-to-wall)
      (check-landmarks)
      ...
      (when (new-sched-ready-p)
        (return)))
```

The real-time process communicates via the RS232 link with the HERO robot by using predefined BASIC command templates. For example, the avoid-collisions function can generate a request for a forward sonar reading, and once it receives the response, it compares the value with some threshold to decide whether to stop the motors because the path is obstructed. As described previously, the detection of an obstruction causes the real-time process to send a message to the AI system. Fortunately for the HERO, the schedule supplied by the AI system specified the immediate reaction to take if blocked, which is simply to stop moving. If instead the process had to wait for the blackboard system to trigger and execute KSs and issue a command to stop, that command would arrive well after the HERO would have collided with the obstruction.

6 Discussion

Our experiments with the HERO robot have concentrated primarily on moving between locations in the same or adjacent hallways (limited by the length of our RS232 connection), and in adding an encountered obstacle to the map. The experiences we have gained have illustrated some of the advantages of our approach. After the AI system develops a schedule of the current behaviors, the real-time process begins maneuvering the robot down the hall. Concurrently, the AI system is incrementally planning [Durfee and Lesser, 1986] the

next set of behaviors to pursue once the current schedule has completed successfully. Thus, our blackboard system's planning and prediction activities are not directly competing with the more time-critical robot control actions of avoiding walls and obstacles. More importantly, the time-critical control actions are not scheduled through the blackboard's opportunistic but time-consuming agenda mechanisms.

The schedule of control actions clearly delineates what sensor readings are needed at any given time, and this improves reactivity. For example, the robot's head sonar needs nearly two seconds to make a complete revolution if it is to scan in all directions. The control actions of the schedule, however, avoid the time-consuming collection of unnecessary information by directing the sonar in only those directions where readings are desired. Although it might remain ignorant of some phenomenon in an area that it only would have checked had it been scanning completely around, the robot using our system more frequently performs the control activities that the AI system chose.

Navigating a HERO robot down a hallway is made difficult by the imprecision of the robot's sensors and effectors. The sonar readings are often errorful, and the robot's wheels often slip. To compensate for this, the functions for orienting the HERO and checking for landmark readings must integrate information about the current sensor data, past readings, known wheel movements, previous orientation, and features of the floor plan in order to develop a reasonable estimation of the robot's current position and orientation in the hall. While much of this can be done by the AI system, we have incorporated some of this into the real-time system's control functions themselves. We need to study these issues further to discover how much of the reasoning about uncertainty and data fusion can be included in the control code, and how much must be done by the AI system.

On a related note, one important area that we have not adequately addressed is the issue of responsiveness of the AI system. Our approach emphasizes guarantees about the control behavior, and our system can guarantee that the robot will stop before a head on collision with a stationary object and will reorient before colliding with a wall. However, once the hard-coded reaction is taken, the blackboard system might require a significant amount of time to develop a reasoned response to the situation. Thus, for example, on encountering an obstacle the hero will stop and do nothing for a short while before the AI system downloads the commands to collect data about the obstacle's boundaries. While this is acceptable behavior in our environment, we still need to examine techniques such as deliberation scheduling and approximate processing to speed up this reasoning so that we can better address time constraints at the task as well as the control level.

7 Summary and Current Directions

In summary, we are developing a cooperative approach to combining techniques from the real-time computing and AI fields in order to integrate high-level planning with scheduling low-level control actions. To evaluate this work, we have begun implementing our ideas in a real, robot system. Our preliminary experiments have shown that for our limited task domain, our cooperative approach allows us to combine task planning and reasoned responses to unexpected events with tight-loop control and rapid, hard-coded reactivity. Important current directions for this work include: enlarging the set of KSs to expand the range of behaviors; incorporating more complete timing knowledge and testing alternative real-time scheduling algorithms; more completely analyzing the dividing line between the AI and real-time components in information-rich, uncertain domains; and using techniques such as approximate processing [Lesser *et al.*, 1988] and cooperating intelligent systems [Hayes-Roth *et al.*, 1989b] for reducing AI system time needs to address task-level time constraints.

We are also exploring related issues in controlling and coordinating multiple robots. For example, multiple robots typically need to communicate in order to synchronize their actions. To make intelligent communication decisions, these robots need knowledge about the underlying communication mechanisms, such as whether messages will ever be lost and how long communication takes in the worst (or average) case. To ensure timely communication, we might need to adopt a port-based communication architecture [Shin and Epstein, 1987], allowing different priority channels. In dynamically changing environments, where agents come and go over time, reasoning about messaging capabilities, needs, and priorities will be a complex problem.

As a simple example of the types of tasks we are concerned with, consider several mobile robots that are following each other in a line. If the robot in the front discovers that it soon must stop unexpectedly, what should it do? It could stop immediately and, at the same time, send messages to the robot behind it to halt. But if the message takes too long to arrive and process, the robot behind might crash into the leader. The robot behind also must ensure that the robot following it will not crash into it. To avoid a chain reaction of rear-end collisions, therefore, a robot that is being followed must decide how quickly the following robot can stop, and a crucial aspect of this decision is using knowledge about the communication channels. If we are to guarantee real-time responsiveness in dynamic domains, then the capabilities and the use of communication channels must be appropriate.

The final direction that we are exploring is the role that reasoning about coordination plays in real-time AI. Although many deadlines an agent faces are based on aspects of the physical world that are beyond the agent's control, other deadlines are based on coordination decisions with other agents. For example, if two robots have arranged to pass a part from one to the other at a specific time and place, they have imposed deadlines on themselves for this rendezvous. If one robot is slowed by some unanticipated obstacles, it could try alternative means of meeting the deadline (such as increasing its speed) but this might have drawbacks (such as increasing the chances that it will be unable to avoid a collision). The robot could instead attempt to modify the deadline; it could ask for an extension.

We believe that reasoning about the timing of interactions between intelligent systems is a key aspect of intelligent behavior in dynamic domains. Our expectation is that real-time AI and distributed AI have many connections between them, and that studying these connections will lead to important insights and progress in both fields.

7.1 Conclusion

7.1.1 Acknowledgements.

Many of the ideas in this paper arose out of discussions with Kang Shin, Dave Musliner, and Tom Tsukada. The low-level techniques for sonar-based navigation using the HERO robot were developed jointly with Terry Weymouth and Yuval Roth-Tabak.

References

[Agre and Chapman, 1987] P. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the National Conference on Artificial Intelligence*, pages 268–272, 1987.

[Arkin, 1987] Ronald C. Arkin. *Towards Cosmopolitan Robots: Intelligent Navigation in Extended Man-Made Environments.* PhD thesis, University of Massachusetts, September 1987.

[Boddy and Dean, 1989] Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 979–984, Detroit, Michigan, August 1989.

[Brooks, 1986] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, RA-2(1):14–22, March 1986.

[Cohen *et al.*, 1989] Paul R. Cohen, Michael L. Greenberg, David M. Hart, and Adele E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3):32–48, Fall 1989.

[Corkill *et al.*, 1986] Daniel D. Corkill, Kevin Q. Gallagher, and Kelly E. Murray. GBB: A generic blackboard development system. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1008–1014, Philadelphia, Pennsylvania, August 1986. (Also published in *Blackboard Systems*, Robert S. Engelmore and Anthony Morgan, editors, pages 503–518, Addison-Wesley, 1988.).

[Dean and Boddy, 1988] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 49–54, St. Paul, Minnesota, August 1988.

[Durfee and Lesser, 1986] Edmund H. Durfee and Victor R. Lesser. Incremental planning to control a blackboard-based problem solver. In *Proceedings of the National Conference on Artificial Intelligence*, pages 58–64, Philadelphia, Pennsylvania, August 1986.

[Durfee and Montgomery, 1990] Edmund H. Durfee and Thomas A. Montgomery. A hierarchical protocol for coordinating multiagent behaviors. In *Proceedings of the National Conference on Artificial Intelligence*, July 1990.

[Durfee *et al.*, 1989] Edmund H. Durfee, Victor R. Lesser, and Daniel D. Corkill. Trends in cooperative distributed problem solving. *IEEE Transactions on Knowledge and Data Engineering*, 1(1):63–83, March 1989.

[Firby, 1987] R. James Firby. An investigation into reactive planning in complex domains. In *Proceedings of the National Conference on Artificial Intelligence*, pages 202–206, Seattle, Washington, August 1987.

[Hayes-Roth *et al.*, 1989a] Barbara Hayes-Roth, Micheal Hewett, Richard Washington, Rattikorn Hewett, and Adam Seiver. Distributing intelligence within an individual. In Les Gasser and Michael N. Huhns, editors, *Distributed Artificial Intelligence*, volume 2 of *Research Notes in Artificial Intelligence*, pages 385–412. Pitman, 1989.

[Hayes-Roth *et al.*, 1989b] Barbara Hayes-Roth, Richard Washington, Rattikorn Hewett, Micheal Hewett, and Adam Seiver. Intelligent monitoring and control. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 243–249, Detroit, Michigan, August 1989.

[Hendler, 1990] James Hendler. Abstraction and reaction. In *Working Notes of the 1990 AAAI Spring Symposium on Planning*, pages 54–56, March 1990.

[Horvitz, 1987] Eric J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *Proceedings of the 1987 Workshop on Uncertainty in Artificial Intelligence*, 1987.

[Laffey *et al.*, 1988] Thomas J. Laffey, Preston A. Cox, James L. Schmidt, Simon M. Kao, and Jackson Y. Read. Real-time knowledge-based systems. *AI Magazine*, 9(1):27–45, Spring 1988.

[Laird *et al.*, 1987] John E. Laird, Allen Newell, and Paul S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, pages 1–64, 1987.

[Lesser *et al.*, 1988] Victor R. Lesser, Jasmina Pavlin, and Edmund H. Durfee. Approximate processing in real-time problem solving. *AI Magazine*, 9(1):49–61, Spring 1988.

[Shin and Epstein, 1987] Kang G. Shin and Mark E. Epstein. Intertask communications in an integrated multi-robot system. *IEEE Journal of Robotics and Automation*, RA3(2):90–100, April 1987.

[Smith and Broadwell, 1987] David Smith and Martin Broadwell. Plan coordination in support of expert systems. In *Proceedings of the DARPA Knowledge-based Planning Workshop*, Austin, Texas, December 1987.

[Stankovic and Ramamritham, 1987] J. Stankovic and K. Ramamritham. *Tutorial on Hard Real-Time Systems.* IEEE Computer Society Press, 1987.

Managing Deliberation and Reasoning in Real-Time AI Systems

François Félix Ingrand
Artificial Intelligence Center
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
E-mail: felix@ai.sri.com

Michael P. Georgeff
Australian AI Institute
1 Grattan Street
Carlton, Victoria 3053
Australia
E-mail: georgeff@aaii.oz.au

Abstract

This paper describes some recent research[1] on architectures for situated (embedded) systems that need to deliberate and reason in real time. One of the most difficult problems in the design of such architectures is how to manage the reasoning performed by such a system while still meeting the real-time constraints of the problem domain. We present an architecture, based on the Procedural Reasoning System (PRS), that provides mechanisms for the management and control of deliberation and reasoning in real-time domains. In particular, we show how deliberation and reasoning strategies can be represented in the form of metalevel plans, and describe an interpreter that selects and executes these in a way that retains bounded reaction time. In addition, this approach allows us to represent different types of situated system by varying the metalevel deliberation strategies. Finally, we provide some statistical measures of performance for one such type of situated system applied to a complex real-time application.

1 Introduction

The design of reasoning and planning systems that are situated (embedded) in real-time, dynamic environments has recently been the focus of expanded research efforts in Artificial Intelligence. A critical issue is to identify the architectural features that would enable such systems to exhibit rational behavior in these domains. In this paper, we describe a uniform architecture that we believe addresses many of the difficult problems in this area.

Computer systems, like human beings, have resource limitations: they have only partial knowledge of their environment and bounded computational (or reasoning) capabilities. When situated in dynamic environments, these limitations become important, because the environment may change in significant ways while the system attempts to gather more information or to reason about what actions to pursue, given the information it already has. If the system (or agent) does not act in a timely manner, it may not be able to recover from a deteriorating situation or may miss positive opportunities.

One way to cope with stringent time constraints is to determine ahead of time how the system should act in every possible situation [Kaelbling, 1987; Rosenschein and Kaelbling, 1986]. However, in domains requiring complex responses to different patterns of events, it is unlikely that such precompilation of plans of action will be practically possible. In such cases, the system must be able to reason about what courses of action to pursue as it observes the changing environment and performs its various tasks.

In particular, at any given time, the system will have to decide what tasks are important enough to initiate or continue, choose among the various means for accomplishing each task, and determine how to order the chosen tasks for execution. In some cases, these decisions may be relatively simple and straightforward. But in other cases they may involve consideration of the likelihood of success of the task, the utility of performing the task, the resources required, the task's expected execution time, the availability and reliability of information upon which the performance of the task depends, the task's dependence on other tasks that are also to be performed, etc.

Moreover, these deliberative tasks themselves are subject to the same constraints on time and information as any other task the system is performing. Thus, the agent will need to decide when and how to seek more information, when and how to deliberate, and when to simply go ahead and act on the basis of whatever reasoning and deliberation it has already performed using whatever information it has at the time. And these deliberations, in turn, need to be reasoned and deliberated about.

An important question, then, is to determine how one can design a situated system that provides for the execution and management of such deliberative processes, yet meets the real-time demands and information constraints of its environment. In this paper, we describe how one such architecture, the Procedural Reasoning System (PRS), provides the mechanisms for handling this problem.

[1]This research is supported by the National Aeronautics and Space Administration, Ames Research Center, under Contract No. NAS2-12521.

2 An Architecture for Situated Deliberation

The architecture of a PRS module or agent consists of (1) a database containing the system's current beliefs about the world; (2) a set of current goals; (3) a library of plans, called Knowledge Areas (KAs), which describe particular sequences of actions and tests that may be performed to achieve given goals or to react to certain situations; and (4) an intention structure, consisting of a [partially] ordered set of all those plans chosen for execution. An interpreter or inference mechanism manipulates these components, selecting an appropriate plan (KA) based on system beliefs and goals, placing those selected KAs on the intention structure, and finally executing them.

PRS interacts with its environment both through its database, which acquires new beliefs in response to changes in the environment, and through the actions that it performs as it carries out its intentions. Different instances of PRS, running asynchronously, can be used in an application that requires the cooperation of more than one subsystem.

The PRS interpreter runs the entire system. From a conceptual standpoint, it operates in a relatively simple way. At any particular time, certain goals are established and certain events occur that alter the beliefs held in the system database. These changes in the system's goals and beliefs trigger (invoke) various KAs. One or more of these applicable KAs will then be chosen and placed on the intention structure. Finally, PRS selects a task (intention) from the root of the intention structure and executes *one step* of that task. This will result either in the performance of a primitive action, the establishment of a new subgoal, or the conclusion of some new belief.

At this point the interpreter cycle begins again: the newly established goals and beliefs trigger new KAs, one or more of these are selected and placed on the intention structure, and again an intention is selected from that structure and partially executed.

PRS has several features that make it particularly powerful as a situated reasoning system, including: (1) The semantics of its plan (procedure) representation; (2) Its ability to construct and act upon partial (rather than complete) plans; (3) Its ability to pursue goal-directed tasks while at the same time being responsive to changing patterns of events in bounded time; (4) Its facilities for managing multiple tasks in real-time; (5) Its default mechanisms for handling stringent real-time demands of its environment; and (6) Its metalevel (or reflective) reasoning capabilities. Some of these features have been discussed in earlier reports and papers [Georgeff and Ingrand, 1989; Georgeff and Ingrand, 1990a; Georgeff and Ingrand, 1990b; Georgeff and Lansky, 1986; Rao and Georgeff, 1990]. In this paper, we consider in more detail the way the system architecture supports deliberative reasoning and provide some statistics on the system's real-time performance capabilities.

3 Making Decisions in Real Time

At each interpreter cycle, the changing beliefs and goals of PRS trigger certain KAs (plans) which, upon execution, either perform certain primitive actions or modify the internal state (the beliefs, goals, and intentions) of the system. At this level of abstraction, PRS acts like a situated automaton [Rosenschein and Kaelbling, 1986].

However, one of the most critical aspects of the PRS architecture is the way in which its beliefs, goals, and intentions evolve and change over time. It is here that a number of strong commitments in the design of PRS have been made and these, we believe, are crucial to its successful performance as a situated, real-time system.

Given that the system needs to be able to deliberate in various ways and at various times, one of the most difficult problems to overcome is how to reduce the amount of potential deliberation that need be undertaken. In particular, how can we avoid deliberation on every action (internal or external) taken by the system and, recursively, how can one avoid or reduce deliberation on those deliberation processes themselves?

Most existing situated reasoning systems use one or a combination of the following approaches: (1) They do not allow any form of deliberation—the considerations important to such deliberation are compiled into the triggering parts of the plans or knowledge sources themselves [Firby, 1989]; (2) The deliberation is performed at one level only and is done at every cycle irrespective of the constraints on time and information existing at that moment in time [Hayes-Roth, 1989]; and (3) The deliberation occurs at one level only and is performed by a separate module of the system, unconstrained by the real-time demands of the application and thus not bounded in reaction or response time [Dodhiawala *et al.*, 1989; Fehling and Wilber, 1989; Hayes-Roth, 1989].

PRS takes a quite different approach. We consider that the first task of the system should be *to keep the number of options open to deliberation under control.* To achieve this, the PRS interpreter uses certain default decision-making mechanisms that are stringently bounded in execution time. For example, once a certain means has been chosen for achieving a particular goal, and as long as the system has not already failed to achieve the goal using those means, that means *will not be reconsidered* — despite possible changes in the environment that may indicate the existence of better options. These option-reducing decision mechanisms execute in bounded time and, in most real-world situations, substantially reduce the set of options available for deliberation. Some of the more important of these mechanisms are discussed elsewhere [Georgeff and Ingrand, 1989].

Of course, even after this filtering of options, some options remain open to consideration. Furthermore, the filtering may have removed some options that should really have been considered more carefully. Thus, it is necessary to provide the system with a capability for performing a possibly unbounded amount of deliberation and for reconsidering some of the options that have possibly been discarded by the default decision mechanisms.

In PRS, both these tasks are achieved by the use of so-called metalevel KAs. Metalevel KAs use exactly the same knowledge representation as application-level KAs; they differ only in that they operate on the system's internal state (i.e., its beliefs, goals, and intentions)[2] rather than the external world.

The way these metalevel KAs are brought to bear on any particular problem is via their invocation criteria. These criteria may depend both on conditions obtaining in the external world and, more typically, on conditions relating to the internal state of the system. Such conditions might include, for example, the applicability of multiple KAs in the current situation, the failure to achieve a certain goal, or the awakening of some previously suspended intention.

The body of a metalevel KA can be used to represent any kind of decision-making procedure and can be of arbitrary complexity. However, because it is executed in the same manner as any other KA, it will be interrupted whenever any external events modify the system's beliefs or goals. The system can thus continue to react in bounded time, irrespective of the complexity of the decision procedure. This is unlike other existing situated reasoning systems, whose bound on reaction time is determined by the complexity of the decision-making procedures incorporated in the system.

Moreover, further metalevel KAs can be invoked to make decisions about the decision-making procedures themselves. Again, the representation of these higher levels of metalevel procedures is as for any other procedure, and the system's reaction time remains bounded. Of course, one has to be careful in the design of such metalevel procedures if one wants the system to *respond* to events — rather than just notice them — in some given time frame.

It is also important to note that the decision-making behavior of PRS is strongly influenced by the choice of the invocation conditions of metalevel KAs. For example, if these conditions are such that the decision-making metalevel KAs are frequently invoked, PRS will act in a *cautious* manner, spending more time making decisions than otherwise [Bratman *et al.*, 1988]. If, on the other hand, these metalevel KAs are rarely invoked, PRS will act in a *bold* manner, rapidly choosing its actions in response to the changing world in which it is embedded. Thus, by varying the metalevel KAs, we can study different types of situated systems and determine which are best suited for which problem domains.

The question remains as to how to invoke the metalevel KAs and how to ensure their execution as appropriate. We look at this problem in the next section.

4 Invoking MetaLevel Procedures

Our aim in designing PRS was to hardwire as little as possible into the interpreter; i.e., to make it as simple as possible. This provides us with the potential to investigate many different types of agents simply by varying the default decision procedures and the metalevel KAs.

The main loop of the system interpreter determines which KAs are applicable and chooses which to place on the intention structure. It can be viewed as the topmost metalevel KA; it is the final arbiter of which KAs reach the intention structure and thus which can be executed.

The major problem is how to allow KAs to be deliberated upon by other [metalevel] KAs and how to place the chosen ones on the intention structure. The basis of our approach is to allow the main interpreter loop to place at most *one* KA on the intention structure and to require it be placed at the root of that structure.

At first sight, this seems unduly restrictive—one often wants to attend to more than one task, and one often wants to order these tasks for later execution rather than have them executed immediately (which placing at the root of the intention structure entails). The way around this problem lies in the metalevel KAs: *these are the means by which one can place multiple intentions on the intention structure and order them as one pleases.*

The next problem is how to actually invoke metalevel KAs. The difficulty is that, while some of the invocation conditions of metalevel KAs will be known at the beginning of each selection cycle, others (such as the number of KAs applicable at a given moment) can only be determined part way through this cycle. The way we solve this problem is to allow the system to continue to reflect on its changing beliefs about its own internal state within a single cycle of the interpreter, breaking out of this self reflection only when the process of KA activation ceases.

Figure 1 shows a simplified version of the main interpreter loop. Its purpose is to select a KA, place it on the intention structure, and invoke its execution (of which we have more to say later). The basic idea of the algorithm is that the system continuously reflects on itself until no new KAs are applicable. When this state is reached, a KA is chosen at random from those applicable at the previous reflection cycle. If there are no KAs to choose from (i.e., the set of applicable KAs is empty), the execution phase is invoked and the outer cycle repeated. Otherwise, the chosen KA is placed on the intention structure, the execution phase invoked, and the outer cycle repeated.

To enable this scheme to work, the system has to determine which KAs are applicable on each self-reflection cycle. This information becomes a new system belief. In particular, on each cycle, the system concludes a belief about the set of KAs applicable on that cycle, expressed as `(soak` x`)`, where x is the list of applicable KAs. It is then determined whether or not the acquisition of this new belief (i.e., `(soak` x`)`), and possibly other events, triggers any new [metalevel] KAs. If it does, the system acquires a new belief about the applicability of these metalevel KAs. In fact, it does so simply by updating the belief `(soak` x`)` so that the list x now contains exactly those metalevel KAs that are now applicable. (The previous belief about applicable object-level KAs is removed from the database and so, in a sense, is forgotten. However, if needed, it can be captured in the variable

[2] It is important to note that these include beliefs goals, and intentions toward various properties of the system state, such as the number of applicable KAs at the current time point, the success or otherwise of a particular KA instance, the ordering of the intention structure, or the status of some specific intention.

```
(loop ;Loop continuously.
      do (loop for soak = (set-of-applicable-ka) ;Set soak to the set of applicable KAs
               when (or previous-soak soak)
                    do (conclude-fact '(soak ,soak)) ;Post the soak metalevel fact
               if (null soak) ;No new KAs are applicable
                  then if (null previous-soak)
                           ;If previous soak is empty then either no KAs were relevant
                           ;or there is nothing to do (no new goals).
                           then ;Continue to execute the intentions in the Intention Structure
                                (activate-intention-structure)
                                return ;Exit the reflective loop.
                           else;Else, intend one of the KAs selected randomly,
                                (intend (select-randomly previous-soak))
                                ;Go and execute something in the intention structure,
                                (activate-intention-structure)
                                ;Set previous-soak to nil
                                (setq previous-soak nil)
                                return) ;Exit the reflective loop
                  else (setq previous-soak soak)) ;Swap previous-soak and soak
               do (get-new-facts)) ;Get any new facts generated by metalevel matching
      (get-new-facts-goals-messages)) ;Get any new messages, goals or facts
```

Figure 1: KA and Intention Selection in PRS

bindings of the invoked metalevel KAs.)

As PRS places no restrictions upon the invocation conditions of metalevel KAs, it is quite possible that more than one metalevel KA will be invoked at this stage. If this happens, we shall now be left with the problem of deciding which of these metalevel KAs to invoke. There are a number of possible solutions to this problem. One would be simply to select one of the metalevel KAs at random, on the assumption that all are equally good at making the decision about which object-level KAs should be invoked. Another alternative would be to preassign priorities to the metalevel KAs and to invoke the one with the highest priority. However, in keeping with our aim of providing maximum flexibility, the solution we chose to adopt is to allow further metalevel KAs to operate on these lower-level metaKAs in the same way that the lower-level metaKAs operated on the object-level KAs.

The process of invoking metalevel KAs is thus continued until no further KAs are triggered. At that point, there may still be a set of applicable KAs from which to choose. It is then, and only then (i.e., only after failing to find any more applicable metalevel KAs), that we select one of these KAs at random.

Thus it is seen that, when more than one KA is applicable, and in the absence of any information about what is best to do, the system interpreter defaults to selecting one of these KAs at random. With no metalevel KAs, the system would thus randomly select one of the applicable object-level KAs. However, one usually provides metalevel KAs to help make an informed choice about the object level KAs. The applicable metalevel KAs themselves are subject to the same default action (i.e., one will be randomly selected) unless there are yet other metalevel KAs available to make a choice among them. In the end, at some level in the meta-hierarchy, the default action will be taken.

Once selected, the chosen KAs must be inserted into the intention structure. If a selected KA arose due to an external goal or a new belief, it will be inserted into the intention structure as a new intention at the root of the structure. For example, this will be the case for any metalevel KA that is invoked to decide among some set of applicable lower-level KAs. Otherwise, the KA instance must have arisen as a result of some subgoal of some existing intention, and will be "grown" (i.e., attached) as a subKA of that intention. Finally, we are left with the execution phase. This is relatively straightforward.[3] First, an intention at one of the (possibly multiple) roots of the intention structure is selected for further execution. The next step of that intention will comprise either a primitive action or one or more unelaborated subgoals. If the former, the action is directly initiated; if the latter, these subgoals are posted as new goals of the system.

While we have focussed above on metalevel KAs that react to changes in the type or number of applicable KAs, other beliefs about the environment or the internal system state can trigger other kinds of metalevel KAs. For example, beliefs about changing intentions could trigger metalevel KAs to reorder the intention structure, or beliefs about failed goals could trigger a metalevel KA to deliberate on the utility of reattempting the goal.

5 Measures of Performance

Definitions of real-time systems revolve around the notion of *response time*. For example, Marsh and Greenwood [Marsh and Greenwood, 1986] define a real-time system as one that is "predictably fast enough for use by the process being serviced" and O'Reilly and Cromarty [O'Reilly and Cromarty, 1985] require that "there is a strict time limit by which the system must have produced

[3] In fact, the execution algorithm is somewhat more complicated than we indicate here. For example, it needs to handle in different ways the failure and success of attempting to accomplish its goals, what goals need to be reestablished, etc.

a response, regardless of the algorithm employed." This measure is most important in real-time applications; if events are not handled in a timely fashion, the operation can go out of control.

Response time is the time the system takes to recognize and respond to an external event. Thus, a bound on *reaction time* (that is, the ability of a system to recognize or notice changes in its environment) is a prerequisite for providing a bound on response time. PRS has been designed to operate under a well-defined measure of reactivity. Because the interpreter continuously attempts to match KAs with any newly acquired beliefs or goals, the system is able to notice newly applicable KAs after every primitive action it takes.

Some useful performance metrics for evaluating the performance of real-time situated systems are provided by Dodhiawala [Dodhiawala *et al.*, 1989]. Not all of these are of relevance in the applications to which PRS has so far been applied, but the following five probes provide important measures of performance:

1. *sending-time(e)* is the time at which an event e is signalled;
2. *receiving-time(e)* is the time at which e is received by the system;
3. *begin-ack-time(e)* is the time at which e is noticed by the system;
4. *end-soak-time-cycle(e)* is the time at which all the events occurring in the same cycle as e have been noticed and the corresponding set of applicable KAs determined;
5. *event-execution-time(e)* is the time at which the first action following KA selection has terminated;
6. *event-response-time(e)* is the time at which the execution of all the procedures initiated by e have terminated.

Then we defined:

$R1$ = *receiving-time(e)* − *sending-time(e)*,
$R2$ = *begin-ack-time(e)* − *receiving-time(e)*,
$R3$ = *end-soak-time-cycle(e)* − *begin-ack-time(e)*,
$R4$ = *event-execution-time(e)* − *end-soak-time-cycle(e)*,
$R5$ = *event-response-time(e)* − *sending-time(e)*,

Assuming a bounded number of events occurs in any time interval, we can prove that $R1$, $R2$, $R3$, and $R4$ are bounded. $R1$ is the time used to communicate the event to the system and is bounded by definition of the communication function (independently of PRS). The operations performed in $R3$ and $R4$ form a cycle, so $R2$ is actually bounded by $R3 + R4$. So if we prove that $R3$ and $R4$ are bounded, we can conclude that $R2$ is also bounded.

$R4$ is bounded by the maximum time required to execute the longest primitive action in PRS or the time required to post a goal. The time to post a goal is bounded by definition and is negligible. Therefore, the bound on $R4$ is determined by the choice of primitive actions and thus by the user. As the user can choose any level of granularity he or she desires, this bound can be made arbitrarily small. (In the application described below, a maximum action execution time of one second was found to be quite satisfactory, though other applications may well require finer granularity.)

$R3$ is the time used by the system to parse the invocation part and the context part of relevant KAs. As we have a bounded number of events and a bounded number of KAs, we can guarantee that $R3$ is bounded[4].

To estimate the bound on $R2$, let p be an upper bound on the execution times of the primitive actions that the system is capable of performing. Let's also assume that n is an upper bound on the number of events that can occur in unit time, and that the PRS interpreter takes at most time t to select the set of KAs applicable to each event occurrence. The maximum *reactivity delay*, Δ_R, is then given by: $\Delta_R = p + y \times t$, where y is the maximum number of events that can occur during the reaction interval. We have $y = \Delta_R \times n$ and thus obtain $\Delta_R = p/(1 - nt)$ where we assume that $t < 1/n$. This means that, provided the number of events that occur in unit time is less than $1/t$, PRS will notice *every* event that occurs [that is capable of triggering some KA] and is guaranteed to do so within a time interval Δ_R.

Because metalevel procedures are treated just like any other, they too are subject to being interrupted after every primitive metalevel action. Thus, reactivity is guaranteed even when the system is choosing between alternative courses of action or performing deliberations of arbitrary complexity.

$R5$ is the time one would like most to see bounded. However, as the time taken to respond to an event can be arbitrarily large, no such guarantee can be given in general. Let's consider this in a little more detail.

Having reacted to some event, it is necessary for the system to respond to this event by performing some appropriate action. As the system can be performing other tasks at the time the critical event is observed, a choice has to be made concerning the possible termination or suspension of those tasks while the critical event is handled. Furthermore, if there are a number of different ways in which the event can be handled, it might be necessary to choose among those alternatives.

Such choices can be made by appropriate metalevel KAs. However, in general, these decision procedures may take an unbounded amount of time to reach a decision. There are two possible ways to overcome this problem. One is to require that all decision procedures complete in a bounded time. In many domains, this provides adequate decision-making capability and yields a bound on response time. As a particular case, it is not difficult to construct metalevel KAs that yield the same functionalities as Ladder Logic[5].

Alternatively, one could construct a special metalevel KA to act as a task scheduler. This KA would have the capability to preempt all executing decision tasks (and any other tasks for that matter) within a bounded time and begin execution of an event handler. It could utilize whatever information was available (such as any incremental decisions made by anytime decision algorithms [Dean and Boddy, 1988]) to select the most appropriate event handler and the manner in which to suspend or terminate other tasks. It could also take into account

[4] As selection of KAs does not involve any general deduction beyond unification and evaluation of a boolean expression, an upper bound does indeed exist.

[5] Ladder Logic is one of the most widely used program languages for real-time systems.

the different constraints on response time that may exist in different situations. The only requirement is that this KA have a guaranteed upper bound on execution time.

In summary, PRS is guaranteed to react to critical events in a bounded time interval. With appropriate metalevel and application-level KAs, it is also possible to guarantee a bound on response time.

6 Experimentation

As mentioned earlier, one of the valuable features of this design is the ability to realize different types of situated system by varying the default decision rules and the metalevel procedures. In particular, one could then examine the behavioral properties of different types of agents in different environments. We have begun this process by creating one particular type of agent [Georgeff and Ingrand, 1989] and applying it to various real-time applications. In this section, we briefly describe one such application and provide statistics on the performance of the system.

The application domain we choose for experimentation is the task of malfunction handling for the Reaction Control System (RCS) of NASA's space shuttle. This is a relatively complex propulsion system that is used to control the attitude of the shuttle. A wide range of problems can occur in this system and, in a normal shuttle mission, no less than four mission controllers are continuously monitoring and controlling its operation.

Two PRS modules (agents) were used for the application. The resulting system was able to detect and recover from most of the possible malfunctions of the RCS, including sensor faults, leaking components, and regulator and jet failures. It is presently under testing at NASA's Johnson Space Center.

The following performance measures have been made on a SUN Sparcstation, with 20 Mega Bytes of central memory, running Sun Common Lisp, development Environment 4.0.0 Beta-0, Sun4 Version for SunOS 4.0. The code was not optimized by the compiler, and the probing itself affects system performance (the probes defined in section 5 are activated for every event and goal posted by PRS).

For the series of tests given below, we ran the following RCS scenarios: a pressure transducer failure, a regulator failure with both regulators open, and a leaking manifold. This set of scenarios exercises most of the major features of PRS and is representative of the kind of problems occurring in the RCS system. The whole test set took approximately six minutes to run.

Figure 2 shows some statistics on the run. The % measurement indicates how busy the PRS agent were. During the six minutes, RCS ran for 31 seconds, and INTERFACE for 2 minutes 31 seconds. Clearly, each PRS module has plenty of time to work on other problems. (On this machine, with this configuration, this application can be run three times faster that real time without any difficulty). The number of facts, metalevel facts, goals and messages indicate the flow of input to the two PRS modules. We have separated metalevel facts, such as `(soak ...)`, and application facts. The statistics on the goals and messages refer only to the application level. *Relevant-KAs* represents the number of relevant KAs (selected by the indexing mechanism as being potentially applicable) and *Applicable-KAs* represents the number of KAs that were actually applicable. *Intentions* indicates the number of intentions the PRS agent has formed, and *Satisfied Goals in DB* represents the number of goals that were directly satisfied in the database (and thus did not require KA activation).

Name	RCS	INTERFACE
%	8.07	36.33
Facts	20	49
Meta-Facts	357	2364
Goals	202	1257
Messages	66	355
Relevant-KAs	923	8359
Applicable-KAs	105	1108
Intentions	6	14
Satisfied Goals in DB	28	164
Total Run Time	00:00:31	00:02:29

Figure 2: Performance Statistics in the RCS application

RCS				
	Number of facts	Average	Distribution	Maximum
$R1$	486	3.04	6.34	42.
$R2$	486	2.55	4.38	36.
$R3$	7	2222.37	2630.15	6863.
	Number of goals	Average	Distribution	Maximum
$R1$	202	0.57	2.03	21.
$R2$	202	2.16	3.77	26.
$R3$	186	341.21	783.54	5983.
INTERFACE				
	Number of facts	Average	Distribution	Maximum
$R1$	2982	4.23	13.14	94.
$R2$	2982	5.84	10.67	52.
$R3$	124	441.20	479.65	2355.
	Number of goals	Average	Distribution	Maximum
$R1$	1257	0.42	1.05	21.
$R2$	1257	1.88	2.23	41.
$R3$	1165	33.11	115.06	1494.

$$\text{Average} = \frac{\sum_{i=1}^{n} x_i}{n},$$

$$\text{Distribution} = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n-1}}, \overline{x} = \frac{\sum_{i=1}^{n} x_i}{n} \text{ and}$$

$$\text{Maximum} = \max(x_1, \ldots, x_n)$$

Figure 3: $R1$, $R2$ and $R3$ for the RCS application

Figure 3 shows the values of $R1$, $R2$ and $R3$ (see Section 5). All the values are given in sixtieths of a second. The average $R1$ are usually very low (a few sixtieths of a second), and even the maximum values stay under one second[6]. $R2$ is also quite small and never exceeds one second. The values of $R3$, which represents response time, are very difficult to interpret. This is because many of the procedures executed in the RCS application are supposed to "wait" for certain external events to occur.

[6]The high maximum value can be explained by the quantum (300 ms) of the scheduler used under SUN lisp 4.0. That means that if both PRS modules are runnable, one will have to wait at least 300 ms before getting a chance to run.

For example, certain procedures require the system to wait for the pressure to drop under 300 psi, or to wait for the astronauts to flip a switch.[7] Nevertheless, the experience and the evaluation of the system by mission controllers shows that PRS executes its procedures much faster than either an astronaut or a mission controller could. Moreover, in this application, metalevel KAs have been written to ensure that the most important procedures get executed first, thus guaranteeing that the response time is shortest for the most urgent procedure [Georgeff and Ingrand, 1990b].

7 Review of related works

Some researchers have sought to deal with resource limitations in dynamic environments by considering all contingencies at design time. This approach obviates the need for explicit reasoning at execution time: all such reasoning is effectively *compiled* into the structure of the executing program [Agre and Arge, 1987; Brooks, 1986; Firby, 1989; Rosenschein and Kaelbling, 1986; Kaelbling, 1987]. It is very likely that these techniques are optimal in certain applications. However, many researchers believe that, in complex domains, the knowledge-compilation approach will lead to brittle, inflexible systems if used without any real-time deliberative processing [D'Ambrosio and Fehling, 1989; Doyle, 1988; Pollock, 1989].

Blackboard architectures have been used in certain systems that are intended to perform real-time behavior[Dodhiawala *et al.*, 1989; Hayes-Roth *et al.*, 1989]. They use a collection of knowledge sources (tasks) sharing a common data structure. There are a number of interesting features of these systems that could be important in providing fast response in real-time domains that do not require significant amounts of deliberation. However, in current blackboard systems, the actions carried out by the system are not interruptible. This poses serious problems for maintaining realistic bounds on reaction time whenever complex or lengthy tasks need to be performed [Georgeff and Lansky, 1986]. Keeping the blackboard consistent when knowledge sources are asynchronous is also a serious problem that has yet to be addressed. In addition, most blackboard architectures use an agenda of pending tasks that are run serially. The problem is that the agenda manager (i.e., the component that deliberates on what tasks to execute, how to execute them, and when to execute them) is invoked in each cycle and for each task present on the agenda. Thus it runs with considerable overhead, again seriously restricting the real-time capabilities of the system. Moreover, it is difficult to include any lengthy deliberation procedures and there are no mechanisms for reasoning about the deliberation processes themselves.

Schemer-II [Fehling and Wilber, 1989] is in some way similar to PRS, but utilizes specific managers and handlers (deliberation processes) to control the system. As with the blackboard approach, these task handlers cannot reason about themselves. Consequently, the architecture is not as general or flexible as PRS. However, it is an interesting approach and may be optimal for some real-time domains.

[7] These waits are asynchronous and do not block system execution.

8 Conclusion and Future Developments

In this paper, we have attempted to show how the uniform knowledge representation for both application-level knowledge and metalevel knowledge, the default decision rules, and the algorithm used for handling metalevel procedures provides a good framework for managing deliberation and reasoning in real-time environments. We have presented some results regarding the real-time performance of the system when used in a real application (RCS), and briefly reviewed some related works.

Although we have presented an architecture that supports real-time deliberation and reasoning, we have so far not investigated how different default decisions and different metalevel strategies affect system behavior; nor have we examined sufficient real-time domains to determine which kind of situated system best suits which kind of domain. The current RCS application used a set of default decision rules and metalevel procedures that proved to be particularly successful *in that domain*. While we believe these to be of wide applicability, that conjecture has yet to be tested.

Of particular interest would be to incorporate as metalevel procedures various algorithms that have recently been proposed for deliberating in real-time environments. These include the work of Whitehair and Lesser on approximate reasoning [Lesser *et al.*, 1989], Dean and Boddy's work in anytime algorithms [Dean and Boddy, 1988], and the work of a number of researchers [Agogino and Ramamurthi, 1989; Horvitz *et al.*, 1988; Russell and Wefald, 1989] in decision-analysis techniques.

We intend to explore some of these issues in our future research. In particular, by varying metalevel strategies, we aim to experiment with different types of system (such as the IRMA agent architecture [Bratman *et al.*, 1988]) in different kinds of environments, thus leading to a better understanding of situated systems and agent rationality.

References

[Agogino and Ramamurthi, 1989] A. M. Agogino and K. Ramamurthi. Real time reasoning about time constraints and model precision in complex distributed mechanical systems. In *Proceedings of the AAAI Symposium on Limited Rationality*, pages 96–100, Stanford, California, 1989.

[Agre and Arge, 1987] P. E. Agre and D. Arge. Pengi: An implementation of a theory of activity. In *Proceedings of the National Conference on Artificial Intelligence*, pages 268–272, Seattle, Washington, 1987.

[Bratman *et al.*, 1988] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computation Intelligence*, 4(4), 1988.

[Brooks, 1986] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, pages 14–23, 1986.

[D'Ambrosio and Fehling, 1989] B. D'Ambrosio and M. Fehling. Resource bounded-agents in an uncertain world. In *Proceedings of the AAAI Symposium on Limited Rationality*, pages 13–17, Stanford, California, 1989.

[Dean and Boddy, 1988] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 49–54, Saint Paul, Minnesota, 1988.

[Dodhiawala *et al.*, 1989] R. Dodhiawala, N. S. Sridharan, P. Raulefs, and C. Pickering. Real-time ai systems: A definition and an architecture. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 256–261, Detroit, Michigan, U.S.A, August 1989. International Joint Conference on Artificial Intelligence.

[Doyle, 1988] J. Doyle. Artificial intelligence and rational self-government. Technical Report CS-88-124, Carnegie Mellon University, Pittsburgh, Pa., 1988.

[Fehling and Wilber, 1989] M. R. Fehling and B. M. Wilber. Schemer-II: An architecture for reflective, resource-bounded problem solving. Technical Report 837-89-30, Rockwell International Science Center, Palo Alto Laboratory, Palo Alto, California, 1989.

[Firby, 1989] R. James Firby. *Adaptative Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, Department of Computer Science, Yale University, May 1989.

[Georgeff and Ingrand, 1989] M. P. Georgeff and F. F. Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, Michigan, 1989.

[Georgeff and Ingrand, 1990a] M. P. Georgeff and F. F. Ingrand. Real-time reasoning: The monitoring and control of spacecraft systems. In *Proceedings of the Sixth IEEE Conference on Artificial Intelligence Applications*, Santa Barbara, California, March 1990.

[Georgeff and Ingrand, 1990b] M. P. Georgeff and F. F. Ingrand. Research on procedural reasoning systems. Final Report, Phase 2, for NASA Ames Research Center, Moffet Field, California, Artificial Intelligence Center, SRI International, Menlo Park, California, March 1990.

[Georgeff and Lansky, 1986] M. P. Georgeff and A. L. Lansky. Procedural knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation*, 74:1383–1398, 1986.

[Hayes-Roth *et al.*, 1989] B. Hayes-Roth, R. Washington, R. Hewett, M. Hewett, and A. Seiver. Intelligent monitoring and control. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 243–249, Detroit, Michigan, U.S.A, August 1989. International Joint Conference on Artificial Intelligence.

[Hayes-Roth, 1989] B. Hayes-Roth. Architectural foudations for real-time performance in intelligent agents. Technical Report KSL 89-63, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, Stanford, California 94305, December 1989.

[Horvitz *et al.*, 1988] E. J. Horvitz, J. S. Breese, and M. Henrion. Decision theory in expert systems and artificial intelligence. *Journal of Approximate Reasoning*, 2:247–302, 1988.

[Kaelbling, 1987] L. P. Kaelbling. An architecture for intelligent reactive systems. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 395–410. Morgan Kaufmann, Los Altos, California, 1987.

[Lesser *et al.*, 1989] V. R. Lesser, D. D. Corkill, R. C. Whitehair, and J. A. Hernandez. Focus of control through goal relationships. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 497–503, Detroit, Michigan, U.S.A, August 1989. International Joint Conference on Artificial Intelligence.

[Marsh and Greenwood, 1986] J. Marsh and J. Greenwood. Real-time AI: Software architecture issues. In *Proceedings of the IEEE National Aerospace and Electronics Conference*, pages 67–77, Washington, DC, 1986.

[O'Reilly and Cromarty, 1985] C. A. O'Reilly and A. S. Cromarty. "Fast" is not "real-time" in designing effective real-time AI systems. In *Applications of Artificial Intelligence II*, pages 249–257, Bellingham, Washington, 1985. Int. Soc. of Optical Engineering.

[Pollock, 1989] J. L. Pollock. Oscar: A general theory of rationality. In *Proceedings of the AAAI Symposium on Limited Rationality*, pages 96–100, Stanford, California, 1989.

[Rao and Georgeff, 1990] A. S. Rao and M. P. Georgeff. Intention and rational commitment. Technical Report 8, Australian AI Institute, Carlton, Australia, 1990.

[Rosenschein and Kaelbling, 1986] S. J. Rosenschein and L. P. Kaelbling. The synthesis of digital machines with provable epistemic properties. In *Proceedings of the Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 83–98, 1986.

[Russell and Wefald, 1989] S. Russell and E. Wefald. Principles of metareasoning. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, 1989.

An Architecture for Coordinating Planning, Sensing, and Action

Reid Simmons
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
reids@cs.cmu.edu

Abstract

To handle multiple, complex tasks in dynamic, uncertain environments, robot systems need to combine planning and reactive behaviors. The Task Control Architecture (TCA) provides facilities for extending the classical planning framework to include capabilities for interleaving planning and execution, monitoring, error recovery, and handling multiple tasks. To date, TCA has been used to coordinate three mobile robot systems at CMU. The paper focuses on why these capabilities are necessary and how they are realized using TCA. We also describe future research goals for incrementally learning the capabilities, based on the robot's experiences.

1. Introduction

There has been much discussion recently on the utility of plans and planning for intelligent agents that interact with the real world (e.g., [Agre 87, Chapman 90, Ginsberg 90, Kaelbling 86]). The "reactive" camp contends that long-range plans are useless in dealing with dynamic, uncertain domains. The "planning" camp contends that complex tasks are difficult to perform without reasoning about interactions between subtasks.

Both frameworks have advantages. Plans provide a natural language (goal/subgoal hierarchies) for describing complex tasks. In particular, they enable planners to compensate for interactions between subtasks by coordinating their execution. An advantage of the reactive framework is its attentiveness to change, which is clearly important in dynamic environments. This capability, however, can also be achieved within a planning framework. In particular, the framework must be extended to allow plans to be executed before they are wholly specified, and must facilitate monitoring for and adapting to changes in the environment.

The Task Control Architecture (TCA) was developed to explore combining reactivity within a planning framework [Lin 89a, Simmons 90a]. TCA was designed to facilitate building and controlling mobile robot systems that have multiple, complex tasks, limited sensors relative to their tasks, and that operate in dynamic, but relatively benign, environments.

TCA consists of a task-independent central control process and utilities for communicating between the central control and task-specific processes. More importantly, TCA provides facilities for maintaining, scheduling and executing hierarchical plans, for coordinating concurrent monitors and exception handling strategies, and for managing physical and computational resources. The facilities were designed by analyzing the requirements of several mobile robot systems (e.g., [Lin 89b]). We noted several important capabilities needed to extend the planning framework to achieve the necessary reactivity. These capabilities include:

- **Interleaving Planning and Execution:** While the world is in general too complex and uncertain to plan down to primitive actions, there are often times when advance planning is desirable, or even necessary. Robot systems need flexibility in specifying when to plan and when to act. This flexibility can be achieved in a hierarchical planning framework by placing temporal constraints on the planning and execution of tasks.
- **Detecting Changes:** Reacting to change is basic to survival. In rich environments, however, it is often difficult to continually check all relevant features. To manage with limited sensors, systems must selectively choose which features to monitor, based on their current tasks and environment.
- **Error Recovery:** Purely reactive systems do not do error recovery, since they treat each situation afresh. Planning systems, however, must notice when plans are going astray and modify them accordingly. In addition, reflexive behaviors should be provided to safeguard the robots.
- **Coordinating Multiple Tasks:** Unexpected opportunities and contingencies may give rise to multiple tasks. Robot systems must decide if tasks can occur concurrently and, if not, in which contexts one task has priority over another. In addition, they should be able to interrupt lower-priority tasks and smoothly transition to new ones.

The above capabilities have all been implemented using the facilities provided by TCA. An important design feature of TCA is that the capabilities can be added incrementally. The idea is to build basic behaviors first, and then add concurrency, monitors, error recovery procedures, etc. For example, our methodology is to first develop systems having sequential sense-plan-act cycles, then use the TCA facilities to add concurrency. Similarly, we first implement behaviors that handle "normal" situations, then add monitors and error recovery procedures for handling the exceptions. Although these extensions are currently encoded manually, our research

program is geared towards automatically learning strategies that increase the performance and competence of the robots [Mitchell 90, Tan 90].

To date, we have used TCA in three mobile robot systems: the CMU Planetary Rover [Bares 89], a single leg of the rover [Krotkov 90], and an indoor mobile manipulator [Lin 89a]. The Planetary Rover project is developing the Ambler, a novel six-legged robot, to be used for navigation, exploration, and sample acquisition in rugged environments. To demonstrate competence in rough-terrain walking, we built a prototype leg of the Ambler (Figure 1) and a software system for autonomously walking the leg over rough terrain.

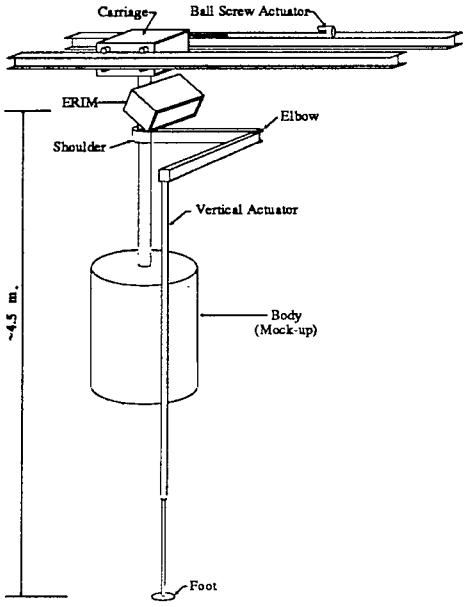


Figure 1: The Single-Leg Testbed

The mobile manipulator testbed is based on a Heathkit/Zenith Hero 2000. The robot operates in a peopled, unstructured laboratory and nearby corridors. The system is currently able to perform a variety of navigation and manipulation tasks. Its main tasks are to identify and collect cups from the floor, retrieve printer output, fetch and deliver objects from workstations, avoid collisions with static and dynamic obstacles, and recharge when necessary. Within the lab, navigation is performed by following a path planned using a 2D map obtained from an overhead camera. To traverse the corridors, the system uses local sonar navigation techniques.

2. The Task Control Architecture

A robot system built using TCA consists of task-specific processes, called *modules*, and a general-purpose *central control* module. Our testbeds all use the same central control module, but have different, robot-specific modules for controlling the robot, acquiring and processing images, and planning and error recovery (Figure 2). The modules communicate with one another by passing messages through the central control, which routes them to be handled by the appropriate modules. Routing information is determined dynamically when modules connect with the central control: modules register with TCA message names, descriptions of the data formats associated with the messages, and the names of procedures for handling the messages.

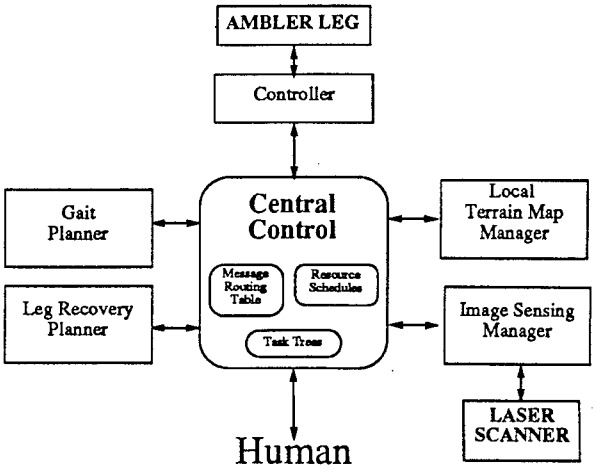


Figure 2: Modules for the Single-Leg Walking System

The facilities of TCA are built around the framework of hierarchical *task trees*. A task tree encodes parent/child relationships between messages: for each message sent, TCA records which message handler issued it. The non-leaf nodes in the task tree represent subgoals and monitors; the leaf nodes are sensor queries and executable commands (Figure 3).

Tasks are coordinated by specifying temporal constraints between nodes in the tree. For example, a module can constrain one subtask to follow another sequentially. For non-leaf nodes (goals and monitors), this *sequential-achievement* constraint implies that all the leaf nodes of the first task must be executed before the second task can begin. In Figure 3, for instance, the sequential-achievement constraint between goals *B* and *C* implies that commands *E* and *F* must be completed before command *G* can be scheduled for execution. Note, however, that the lack of constraint between *E* and *F* implies that they can be executed concurrently.

The *delay-planning* constraint indicates that the subsequent goal should not be handled until the previous task has been completely achieved. Without this constraint, TCA is free to create a plan (by expanding the goal into subgoals), although the plan will not, of course, be executed until the previous task is completed. For example, the delay-planning constraint between goals *C* and *D* indicates that *D* should not be expanded until commands *G*, *I*, and *J* are all executed.

TCA also provides facilities for examining the structure of the task trees and modifying them by killing subtrees, changing temporal constraints, and adding new nodes to the tree. These facilities are useful for error recovery,

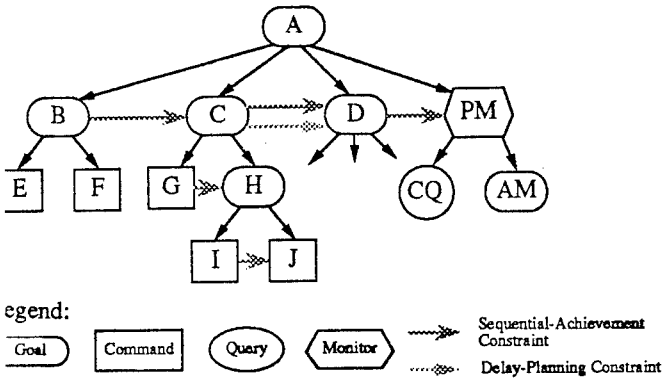


Figure 3: A TCA Task Tree

since they enable robot systems to patch plans.

Facilities are provided for specifying and scheduling monitors. The start time of a monitor can be constrained in relationship to another task (e.g., in Figure 3, the monitor *PM* begins after *D* is achieved), and it can be set to run for a specified duration (e.g., 3 minutes), or until a specified task is completed.

TCA supports polling and interrupt-driven monitors, both of which are specified by a condition query and an action message (Figure 3). For polling monitors, the central control module issues the condition query at a fixed frequency. If the condition holds, the action message is sent. The action message typically is used to replan by modifying the task tree. For example, we use a polling monitor that checks the Hero's battery every 20 seconds and inserts a "recharge" task if the battery level is low.

For interrupt-driven monitors, TCA informs modules when to set up new monitors and when to cancel them. The modules have responsibility for informing TCA whenever the monitor's condition holds, at which point TCA will issue the associated action message. For example, whenever an image is acquired, the Hero's perception module scans the image and informs TCA if new cups are detected.

Context-dependent exception handling is supported by TCA. Modules can associate exception handlers with nodes in the task tree. When an exception message is issued, TCA searches up the tree to find a handler for that exception. If the handler cannot deal with the situation, it reissues the exception and the search continues up the tree. If the root node is reached, TCA simply terminates the task.

Facilities are also provided for defining and managing resources. A TCA *resource* is a collection of message handlers, together with a *capacity*. TCA ensures that the resource capacity is never exceeded, queuing messages if necessary until the resource becomes available. The resource facility can also be used for synchronization. A module can *lock* a resource, which prevents the resource from handling messages until it is unlocked. For example, a concurrent vision module might want to ensure that images are acquired while the robot is stationary. This can be accomplished by locking the resource for the robot's actuators before acquiring an image.

3. Capabilities

To cope with dynamic and uncertain worlds, the classical planning framework must be extended to include several new capabilities. These capabilities include 1) interleaving planning and execution, to enable partially specified plans to be executed, 2) using monitors to detect environmental changes, 3) recovering from execution errors (plan failures), and 4) dealing with multiple tasks when unexpected opportunities or contingencies arise.

The following sections describe these capabilities and why they are needed to achieve reactivity within a planning framework. Each section indicates how the TCA facilities can be used to implement the capabilities. We also present some research goals that are directed towards having robot systems incrementally produce the capabilities themselves, based on their experiences.

Interleaving Planning and Execution

In dynamic, uncertain domains, it is unreasonable to have a planner specify plans down to the minutest detail [Agre 87, Chapman 90]. Instead, it is often advocated to use the environment to dictate actions [Brooks 86, Kaelbling 86]. The optimal strategy is probably some combination of both: the system should plan to the limits of its knowledge of the environment, but no further.

Rather than being antithetical to planning, this strategy is actually well-suited to a framework based on hierarchical decomposition and temporal constraints. The idea is to treat the decomposition of a goal into subgoals as an action in its own right. Thus, planning can be tightly controlled by adding temporal constraints between planning and execution actions.

In TCA, for instance, the delay-planning constraint (see Figure 3) can be used to coordinate planning and execution. A purely reactive system can be implemented by adding delay-planning constraints between every subgoal. With more judicious application, one can specify fairly arbitrary strategies for interleaving planning and execution. For example, the Hero's cup-collection task is expanded into four sequentially executed subtasks: navigate to the cup; pick it up; navigate to the trashbin; deposit the cup. A delay-planning constraint is added between the first two subtasks, since the system cannot plan how to grasp the cup until it gets near enough to make measurements with its wrist sonar. In addition, a constraint is added that the second navigation task cannot be planned until after the first. With these constraints, the Hero uses its overhead vision map to plan a path from the cup to the trashbin concurrently with picking up the cup. The plan is cached by TCA until the cup is grasped, at which point it is executed.

We also explored concurrency in the single-leg walking system. After developing and debugging the system using sequential planning and execution, several of the delay-planning constraints were removed to enable the system to execute one step while planning the next. In addition, we added a constraint to prevent the planning from getting too far ahead of execution, and used resource locking to prevent the robot from moving during image acquisition. The addition of concurrency increased performance by over 30%, with only minor modifications to the existing walking system [Simmons 90b].

Detecting Changes

The capability to detect changes in the environment is obviously central to being reactive. While a simple scheme is to continually monitor all relevant features (e.g., [Brooks 86, Kaelbling 86]), this is not feasible for robots with many tasks and limited sensors. In such cases, monitors must be carefully selected and scheduled (cf. [Firby 89, Noreils 89]).

TCA facilitates selective monitoring by providing mechanisms for creating polling and interrupt-driven monitors, and for synchronizing them with respect to other tasks in the task tree. In addition, since the TCA monitors run concurrently, a wide range of conditions can be monitored without impeding the robot's main tasks. For example, after the Hero system spots a new cup (and until the cup is grasped) it monitors whether the cup remains visible; after the cup is grasped, it periodically checks whether the cup remains in its gripper.

Given the monitor facilities, the problem remains to decide what to monitor, and how frequently. By monitoring only selected conditions, the robot could miss important changes. The challenge is to minimize that possibility. One idea we are exploring is the use of coarse-to-fine sensing strategies. For example, the Hero system uses its coarse 2D vision map to find cup-like regions. It then navigates near the object, and uses multiple sonar readings to determine if the object is actually the size and shape of a cup. In related work, this strategy is being generalized by using inductive methods to learn information-sensitive and cost-sensitive strategies for classifying objects [Tan 90].

We are also developing methods for automatically deriving the parameters of sensing strategies, such as the polling frequency or sensor resolution. The idea is to construct a causal explanation for why a sensing strategy works, and then reason about the uncertainties in that strategy. In monitoring the robot's battery, for example, we must determine how often to poll the battery level and the threshold for heading back to the charger. We can construct an explanation (an equation) that relates the polling frequency, monitor threshold, expected distance to the charger, speed of the robot, and expected rate of discharge. Since some of the terms are random variables, we end up with an equation that enables us to trade off probability of success (risk), polling frequency (sensor utilization), and monitor threshold (urgency).

Error Recovery

Monitors can detect when the assumptions underlying a plan are no longer valid (either the world changed, or the plan was based on inaccurate information). Error recovery strategies can then be employed to change the plan to reflect reality. In general, they are quite dependent on the current task and environment. The TCA exception handling facilities support context-dependent error recovery by enabling different error handlers to be associated with different nodes of the task tree. Typically, errors are handled in a TCA-based system by collecting information about the current environment, analyzing the task tree, and then manipulating the task tree by killing subtrees, adding nodes and temporal constraints, and resending messages.

For example, Figure 4 illustrates part of the task tree and associated exception handlers for the Hero's cup-collection task. An "object in path" exception message is first handled by *EH1*, which tries to plan a detour. If a detour is found, a new *Path Segment* node is added between *Path Segment1* and *Path Segment2*; this node will then be expanded and executed, as usual. Otherwise, *EH2* tries to replace the current plan with a new path to *Cup1*. If this fails, *EH3* is invoked to terminate the task by killing the subtree rooted at *Collect Cup1*.

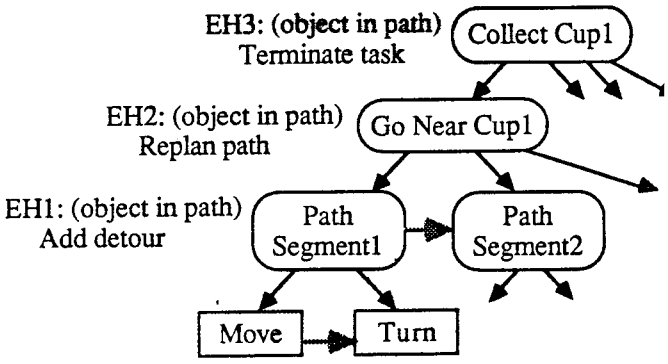


Figure 4: Context-Dependent Error Recovery Strategies

TCA's use of centralized communication places an upper limit on its reaction time. For more reflexive behaviors, we implement routines within the controller module that act to stabilize the robot if the sensors detect anomalies, and then report the error via TCA (cf. [Miller 89]). In the single-leg testbed, for instance, the controller monitors the force on the leg and stops the mechanism immediately if slippage is detected. Similarly, on-board routines check the Hero's wheel encoders and sonars while it is moving. If a collision, or imminent collision, is detected the robot reflexively stops and an exception message is issued, which invokes the error recovery strategies described above.

An alternative method is to use a debugging algorithm

to determine how to patch plans (e.g., [Hammond 89, Simmons 88]). While the debugging methodology tends to be fairly general, the TCA method is more efficient in finding applicable strategies. We are currently working to combine the methods: the algorithms of [Simmons 88] would be used to debug plans, which would then be generalized and "compiled" into error recovery strategies usable by TCA (cf. [Mitchell 90]).

Multiple Tasks

In addition to reacting to plan failures, robot systems must handle changes in the environment that signal new opportunities or contingencies, such as a new cup appearing on the floor, or the battery getting low. These new tasks should be coordinated with the current task(s) in an intelligent manner.

A straightforward approach is to temporally order all tasks based on their priorities. As with error recovery, prioritization is often context dependent. For example, we can prioritize two cup-collection tasks based on which task ordering yields the shortest overall path length (Figure 5). If we approximate path length by the straight-line distance between objects, we can infer that the robot should collect *Cup1* if:

$$|Loc_{Robot} - Loc_{Cup1}| + |Loc_{Cup2} - Loc_{Bin}| < |Loc_{Robot} - Loc_{Cup2}| + |Loc_{Cup1} - Loc_{Bin}|.$$

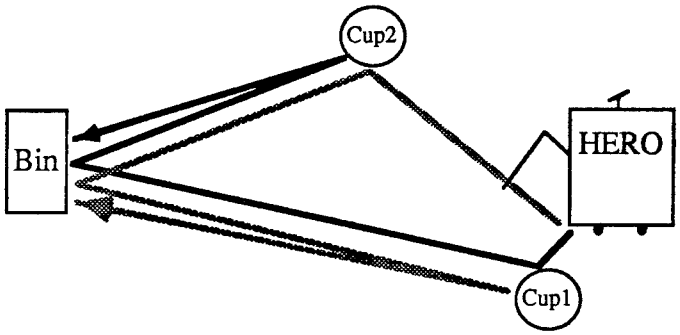


Figure 5: Ordering Two Cup-Collection Tasks

A generalization of this strategy is to prioritize tasks by estimating their relative costs and benefits (including opportunity costs). We have begun developing specific prioritization strategies for the Hero's tasks, and hope that the effort will help us discover a general, yet efficient, algorithm for context-dependent prioritization of tasks.

It is not always necessary to prioritize tasks, since tasks can occur concurrently if the resources they use are disjoint. For example, the Planetary Rover could conceivably navigate and communicate with Earth simultaneously. The TCA resource mechanism provides an efficient method for detecting conflicting tasks: contention occurs when the capacity of a resource is exceeded. While our Hero system currently orders all top-level tasks, we intend to explore alternative strategies that prioritize tasks only if TCA detects resource contention between their subtasks. Again, some combination of strategies might be best: if the system knows from prior experience that two tasks utilize common resources, it can prioritize them immediately; otherwise, it can wait until TCA detects contention.

4. Conclusions

This paper describes the Task Control Architecture, a general-purpose framework plus set of utilities for coordinating the planning, sensing, and action of mobile robot systems. Hierarchical plans are central to TCA, providing a natural and flexible foundation for handling multiple, complex tasks. TCA provides facilities for 1) creating and manipulating hierarchical plans (task trees and temporal constraints), 2) specifying polling and interrupt-driven monitors, 3) resource management, and 4) context-dependent exception handling.

We present several capabilities that must be added to the classical planning framework in order to handle uncertain and dynamic environments: interleaving planning and execution, detecting changes, error recovery, and coordinating multiple tasks. The first capability enables robot systems to act on partially specified plans, allowing them to plan in advance in spite of uncertainty. The other three capabilities enable systems to detect and intelligently handle plan failures, unexpected opportunities and contingencies. The paper focuses on how TCA supports the implementation and coordination of the four capabilities.

TCA was designed to provide a framework for combining deliberative and reactive behaviors. Its demonstratable success with the Planetary Rover and Mobile Manipulator projects is an encouraging indication of its utility in coordinating complex behaviors. Our next step is to automate the incremental addition of monitors, error recovery strategies, and task prioritization, along the directions outlined in the paper.

Acknowledgements

We thank the members of the Planetary Rover and Mobile Manipulator projects for their efforts in designing and implementing the robot systems, and their patience at accepting frequent changes in TCA. This research is supported by NASA under contract NAGW-1175.

References

[Agre 87] Agre, P.E., Chapman, D. Pengi: An Implementation of a Theory of Activity. In *Proc. of AAAI-87*, Pages 268-272. Seattle, WA, 1987.

[Bares 89] Bares, J., et al.
Ambler: An Autonomous Rover for Planetary Exploration.
In *IEEE Computer, Vol. 22, No. 6*, 1989.

[Brooks 86] Brooks, R.A.
A Robust Layered Control System for a Mobile Robot.
In *IEEE Journal of Robots and Automation, vol. RA-2, no. 1*, 1986.

[Chapman 90] Chapman, D.
Penguins Can Make Cake.
In *AI Magazine, Vol. 10, No. 4*, Pages 45-50. Winter, 1990.

[Firby 89] Firby, R.J.
Adaptive Execution in Complex Dynamic Worlds.
Technical Report YALEU/CSD/RR #672, Yale University,
1989.

[Ginsberg 90] Ginsberg, M.
Universal Planning: An (Almost) Universally Bad Idea.
In *AI Magazine, Vol. 10, No. 4*, Pages 45-50. Winter, 1990.

[Hammond 89] Hammond, K.
Case-Based Planning: Viewing Planning as a Memory Task.
Academic Press, 1989.

[Kaelbling 86] Kaelbling, L.P.
An Architecture for Intelligent Reactive Systems.
Technical Note 400, AI Center, SRI International,
1986.

[Krotkov 90] Krotkov E., Simmons, R., Thorpe, C.
Single-Leg Walking with Integrated Perception, Planning, and Control.
In *Proc. of IEEE International Workshop on Intelligent Robots and Systems*, Tsuchiura, Japan, July, 1990.

[Lin 89a] Lin, L.J., Simmons, R., and Fedor, C.
Experience with a Task Control Architecture for Mobile Robots.
Technical Report CMU-RI-89-29, Robotics Institute, Carnegie Mellon University,
1989.

[Lin 89b] Lin, L.J., Mitchell, T.M., Phillips, A., and Simmons, R.
A Case Study in Autonomous Robot Behavior.
Technical Report CMU-RI-89-1, Robotics Institute, Carnegie Mellon University,
1989.

[Miller 89] Miller, D.
Execution Monitoring for a Mobile Robot System.
In *Proc. of SPIE Conference on Intelligent Control*, Society of Photo-Optical Instrumentation Engineers, Cambridge, Massachusetts, 1989.

[Mitchell 90] Mitchell, T.M.
Becoming Increasingly Reactive.
In *Proc. AAAI 1990*, Morgan-Kaufmann, Cambridge, MA, August, 1990.

[Noreils 89] Noreils, F. and Chatila, R.
Control of Mobile Robot Actions.
In *Proc. IEEE Robotics and Automation*, Pages 701-712. 1989.

[Simmons 88] Simmons, R.
A Theory of Debugging Plans and Interpretations.
In *Proc. of AAAI-88*, St. Paul, MN, 1988.

[Simmons 90a] Simmons, R., Lin, L.J., Fedor, C.
Autonomous Task Control for Mobile Robots.
In *Proc. of IEEE Symposium on Intelligent Control*, Philadelphia, PA, September, 1990.

[Simmons 90b] Simmons, R.
Concurrent Planning and Execution for a Walking Robot.
Technical Report CMU-RI-90-16, Robotics Institute, Carnegie Mellon University,
July, 1990.

[Tan 90] Tan, M.
CSL: A Cost-Sensitive Learning System for Sensing and Grasping Objects.
In *Proc. 1990 IEEE International Conference on Robotics and Automation*, IEEE, Cincinatti, Ohio, May, 1990.

PLANNING & LEARNING

Integrating Memory and Search in Planning

John A. Allen* and **Pat Langley**
AI Research Branch, Mail Stop: 244-17
NASA Ames Research Center
Moffett Field, CA 94035

Abstract

In this paper we describe DÆDALUS, a case-based planner that learns from successful plans. The system uses a means-ends engine to generate plans, treats the retrieval of operators from memory as a classification task, and treats the update and organization of memory as a conceptual clustering task. This combination of methods lets DÆDALUS use abstractions to guide its planning when they are available, fall back on specific cases when they are not, and resort to traditional means-ends search on completely novel problems.

1 Introduction

In general, planning is intractable in that no algorithm can find solutions to all planning problems in all domains (Chapman, 1987). Nevertheless, one can still aim for general methods that can solve realistic planning problems in many situations. Researchers have explored several techniques, such as hierarchical planning (Sacerdoti, 1974), in an attempt to reduce the combinatorial search that plagues planning problems. Although these methods constrain search, they also require the implementer to introduce domain-dependent knowledge, and acquiring and coding such knowledge is difficult. This suggests that automated methods for acquiring domain knowledge for planning tasks would be very useful.

Much of the recent research in machine learning has directly addressed this issue, attempting to acquire domain-specific plan knowledge from experience. This research falls into two basic paradigms. One approach involves learning abstract knowledge, either in the form of search-control rules that reduce the effective branching factor or in the form of macro-operators that decrease the effective length of solution paths. Much of the work on explanation-based learning falls into this camp, but inductive variants also exist. Researchers who study learning abstract knowledge directly address problems of constraining search, but they often assume a simplistic rule-based representation of knowledge whose conditions require an exact match, thus ignoring issues of memory organization and retrieval.

An alternative approach involves storing specific planning experiences in memory and then using these "cases" in solving novel but related planning problems. Researchers in this case-based framework focus directly on the organization of memory, on the retrieval of knowledge from this memory, and on adapting the retrieved cases to new situations. This work is closely related to research on analogy, but it is often applied to specific performance tasks such as planning. Despite its advantages, research on case-based planning often emphasizes the importance of memory to the exclusion of the search issues that arise in domains where one lacks experience.

In this paper we describe DÆDALUS, a planning system that begins to bridge the gap between these two approaches to the representation, use, and acquisition of plan knowledge. As we describe in the following section, the system begins with knowledge of legal operators as its only domain expertise, so that it must search to find successful plans. However, DÆDALUS stores cases based on these plans in memory, and it uses them to constrain its future planning behavior. Eventually, the system moves beyond specific cases to store abstract plan knowledge, but it retains the ability to fall back on case knowledge or even search when necessary. After describing the basic system, we present some experimental evidence that DÆDALUS' planning skills improve with practice. We also compare the system's approach to learning and planning with alternatives from the literature, and propose some directions for future research.

2 The Dædalus System

DÆDALUS is a case-based planner that starts with a small number of simple cases, and builds a library that allows it to plan by indexing cases, rather than by search. The system accepts an initial state and a set of goal conditions as input and returns a sequence of operators that will transform the initial state into a state that satisfies the goal conditions. The planner is given an initial case library consisting solely of operators ap-

*Also affiliated with the University of California, Irvine, and Sterling Federal Systems.

plicable in the domain. The operators are organized hierarchically in a memory structure, and are indexed by the changes or differences they effect. DÆDALUS uses a variant of means-ends analysis that calculates the differences (the changes to be made) between the current state and the goal conditions, and uses these differences along with the features of the current state to retrieve operators (cases) from memory. The system uses the operators it retrieves to search for a plan in the domain space, much as Fikes, Hart, and Nilsson's (1971) STRIPS.

Learning in DÆDALUS consists of incorporating the generated plans into memory in a way that allows them to be retrieved when applicable. Upon encountering a previously unseen problem, the system retrieves a relevant part of a plan and uses it to select operators for the new task. The process of creating the indices for a new case leads DÆDALUS to generalize its stored plans, giving rise to useful abstractions while still retaining the ability to search if necessary. Below we describe DÆDALUS' representation and organization of plans, its performance and learning components, and its overall behavior.

2.1 Representing States, Problems, Operators, and Plans

DÆDALUS acts on data structures of four types: states, problems, operators, and plans. In general, a *state* consists of some description of the world, possibly including features internal to the agent. We use a simple STRIPS-like state representation (Fikes et al., 1971), with each state described as a set of objects and symbolic relations that hold among them.

A *problem* consists of an initial state and a set of goal conditions that the agent wants to achieve. Each state may be a *partial* description of the world. For instance, Figure 1 (A) presents a graphical description of a rocket world problem (Veloso, 1989)[1]. The initial state consists of three objects: an autonomous rover, a rover support satellite, and a one-way transport rocket. Each of the three objects are located on Earth; the rocket is on its launch pad, the satellite loaded inside, and the rover waiting nearby. The final state shows the rover exploring Mars, the satellite transmitting data back to Earth, and the rocket cracked and bent for lack of a landing procedure.

The box in Figure 1 (B) labeled "initial state", shows the STRIPS-like state representation of the pictorial depiction of the initial state in (A). The box labeled "goal conditions" is a partial description of Figure 1 (A)'s final state. Although the initial and final states are the formal definition of a problem, DÆDALUS uses the initial state and the *differences* between the initial state and the goal conditions as an internal representation of a problem, and uses the goal conditions to test for successful termination. The notion of representing problems as differences is central to our approach.

An *operator* in DÆDALUS has a set of preconditions, an add list, and a delete list, giving them a strong resemblance to the STRIPS operators (see Table 1). From this information one can derive a set of differences that exist between states before and after application, resulting in a description similar to that used for problems, and allowing them to be stored in the same memory structure.

The system represents a *plan* for solving a particular problem in terms of a derivational trace (Carbonell, 1986) that states the reasons for each step in the operator sequence. A trace consists of a binary tree of problems and subproblems, with the original task as the root node and with trivial (one-step) subproblems as the terminal nodes. Each node in the derivational trace has two recursively defined children. One child represents the subproblem of transforming the parent problem's current state into the preconditions of the parent problem's operator. The other child represents the problem of transforming the state that results from applying the parent problem's operator into the goal state of the parent. As well as having pointers to subproblems, each problem in the binary tree has pointers to its current state, its operator, and the state resulting from applying the operator.

The derivational trace in Figure 2 shows a plan that solves the problem presented in Figure 1. Here, ellipses represent problems, rectangles represent operators, and squares represent states. The root node, representing the problem, has links to both the starting state and the final state that satisfied the goal conditions. The root node also records the operator instance, `(unload-rover rover1)`, which the system selected to transform the top-level problem. Two children sprout sideways from the root node. The upper child denotes the problem of transforming the initial state into a state that satisfies the preconditions of the operator at the root node. Since this node, demarcated `(load-rover rover1)`, does not have an upper child, one can infer that all the preconditions of `(load-rover rover1)` were satisfied and the operator was directly applicable. The fact that it has a lower child shows that the application was insufficient for satisfying the preconditions of `(unload-rover rover1)`. In summary, upper children represent the problem of changing the current initial state into a state that satisfies the preconditions of the operator at the current node, and lower children represent the problem of transforming the state resulting from applying the operator at the current node into a state that satisfies the current goal conditions.

Although the derivation trace in Figure 2 shows how DÆDALUS found the successful plan, it does not show all the problem-solving activity. In fact, when DÆDALUS was first presented this problem, the first operator it selected was `(unload-satellite satellite1)`. However, once this operator was applied, DÆDALUS could find no way of resolving the difference `(at rover1`

[1]The examples in this paper make use of a domain slightly different from that presented by Veloso.

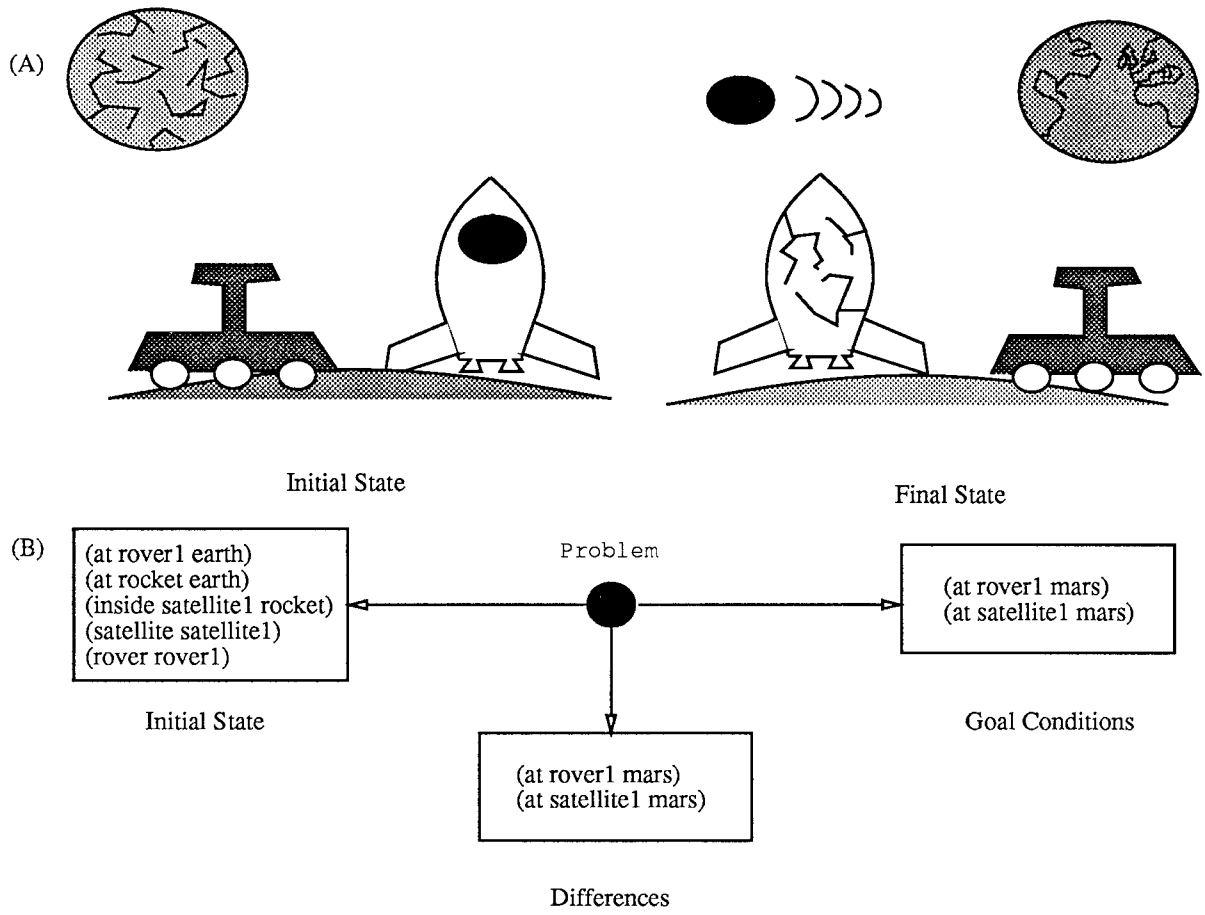


Figure 1. A simple problem in the rocket world.

`Mars)`. It was forced to backtrack over the application of `(unload-satellite satellite1)` and retrieve another operator from its memory, `(unload-rover rover1)`. Thus, problem solving can be viewed as *and/or* search with the derivational trace constituting the *and* tree resulting from that search.

2.2 The Organization of Plan Memory

DÆDALUS' memory is used to assist a means-ends style planning engine, so both the plans created by DÆDALUS and the plans given by the user need to be "indexable" by the information available to means-ends analysis. The derivational trace is designed to record the necessary information; storing the sequences of domain operators with their associated problem-solving state (PSstates), which consists of the state current at the time of selection, and the differences the operator was selected to address.

An operator and its PSstate make up a case, and DÆDALUS breaks up the derivational trace into its component cases and stores each case separately into memory. The resulting memory structures are both usable and flexible. As observed by Kolodner (1987), the information stored with an entire plan may make the plan difficult to work with, and much of that information may not be relevant to the inference at hand. Also, storing the cases separately, in addition to giving access to parts of plans, lets one reconstruct the original plan, or construct a new plan out of the pieces of several different plans.

The cases are organized in a probabilistic concept hierarchy similar to Fisher's COBWEB (1987). This memory takes the form of a tree in which each leaf describes an individual case, and each internal node describes an abstraction that covers the cases found at the leaves of the node's subtree. Every node makes up a concept describing some set of cases that are similar to each other, but different from those described by sibling nodes. The concepts are probabilistic in that they describe the likelihood of occurrences of each statement in the PSstate, as well as the likelihood of occurrence of the concept itself.

The nodes consist of two parts: the PSstate and the operator. The PSstate has a differences section and a

Table 1. The operators of the rocket world.

Name	Preconditions	Add-list	Delete-list
(launch-rocket)	(at rocket earth)	(at rocket mars)	(at rocket earth)
(load-satellite ?object)	(at ?object ?place) (at rocket ?place) (satellite ?object)	(inside ?object rocket)	(at ?object ?place)
(load-rover ?object)	(at ?object ?place) (at rocket ?place) (rover ?object)	(inside ?object rocket)	(at ?object ?place)
(unload-satellite ?object)	(inside ?object rocket) (at rocket ?place) (satellite ?object)	(at ?object ?place)	(inside ?object rocket)
(unload-rover ?object)	(inside ?object rocket) (at rocket ?place) (rover ?object)	(at ?object ?place)	(inside ?object rocket)

state section. Both sections contain a list of domain features (either goal or state) with an associated conditional probability of occurrence given membership in the concept. The operator also has an associated conditional probability. If the node is a leaf node, then all the conditional probabilities equal one. If the node is an internal node, the conditional probabilities are determined by the cases covered by the node. For example, if an internal node covers three cases, two of which have the operator `Eat-lunch` and one with the operator `Eat-dinner`, then the internal node would list two operators: one, `Eat-lunch`, with a conditional probability of two-thirds, the other, `Eat-dinner`, with a probability of one-third (another example is shown in Figure 3).

The hierarchical organization and the probabilistic information associated with the nodes are used as indices for the case information. The memory structure defines a polythetic decision tree that may be used to determine the most similar case to the case at hand. This form of indexing is used to store both cases and operators, with only a slight difference between the two: cases are indexed by PSstate (state and differences), whereas operators are indexed by their differences. This distinction turns out to be inconsequential to the retrieval process, and both cases and operators are treated identically.

2.3 Planning and Retrieval in Dædalus

In this section, we discuss DÆDALUS' performance system — its planning and memory components. We describe how a simple means-ends style planner, which solves problems through the generation of subgoals, may be guided and assisted by permitting access to a richly indexed memory of planning experience. We also discuss the mechanism of indexing plan memory and illustrate the process by way of an example.

DÆDALUS uses a variant of means-ends analysis (Newell et al., 1960; Fikes et al., 1971). In this framework, solving a problem (transforming a current state into a desired one) involves the recursive generation of subproblems. The standard means-ends approach determines all differences between the current and desired state, selects the most important difference (using some predefined criterion), and then retrieves an operator that reduces that difference. If the selected operator cannot be applied, a subproblem (called a *transform goal*) is generated to change the current state into one that satisfies the operator's preconditions, and is solved by a recursive call to the algorithm. Applying the operator produces a new state, along with a new subproblem (another transform goal) to transform this into the desired state; the algorithm is then called recursively to solve this task. The derivational trace in Figure 2 reflects the recursive nature of the means-ends engine and displays all the goals generated in the problem-solving process: upper and lower branches represent transform goals, and the central branch represents the applied operator.

DÆDALUS searches the domain space in a depth-first manner. The system continues recursively generating subproblems until it detects one of two conditions: either the differences are removed or a loop is detected. If the goals are satisfied, the system ends the recursion and proceeds with the next subgoal; if there are no more subgoals, the plan is finished. However, if a loop is detected, the planner halts its current path of enquiry, backs up, and tries another path. DÆDALUS checks for loops in transform goals, whose detection causes the system to backtrack and pursue a different plan.

Our system differs from most means-ends planners in the way it retrieves operators from memory, which is significantly different that used by either GPS (Newell et al., 1960), or STRIPS (Fikes et al., 1971). Initially, DÆDALUS is given a memory containing a set of plans that the user thinks will be useful. This may be as elaborate as the user desires, but since cases can be tedious to construct, usually the initial memory consists of ab-

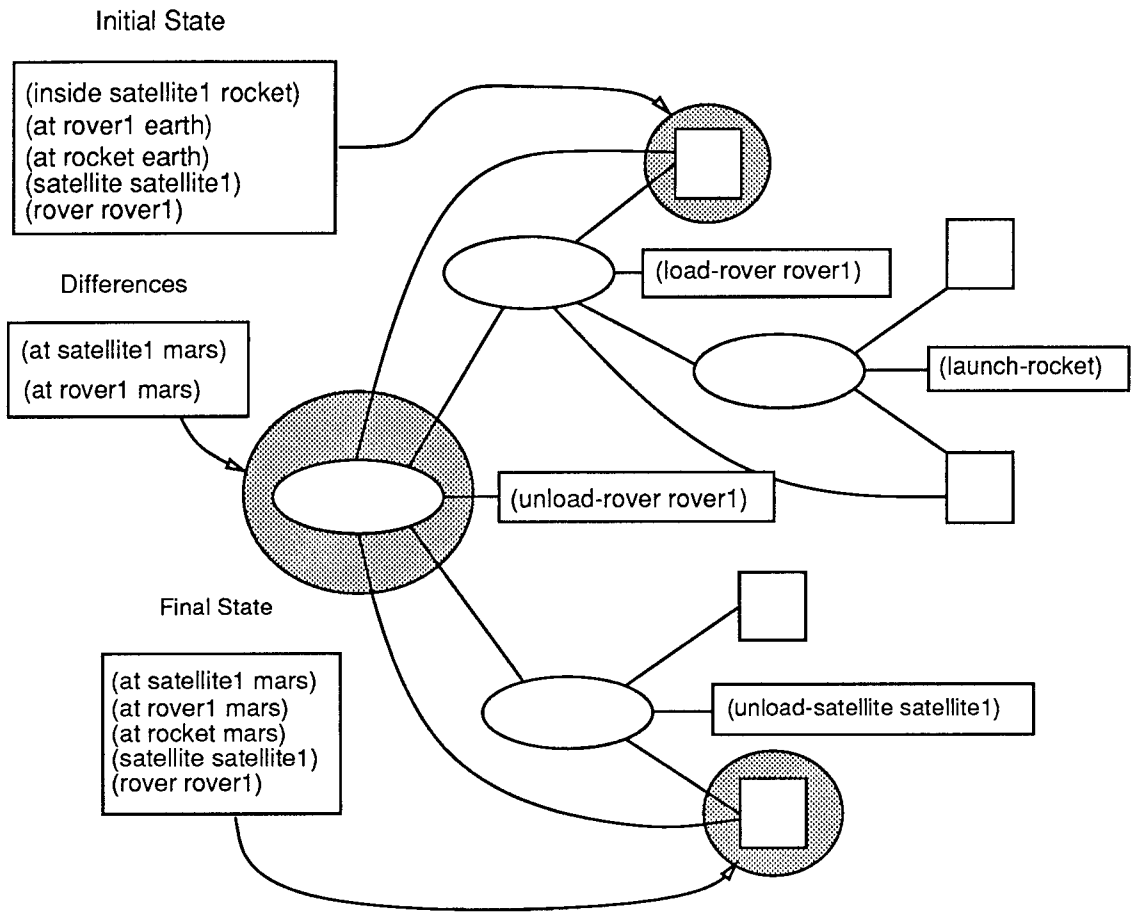


Figure 2. Derivational trace for a solution of the rocket-world example.

stract descriptions of the domain operators.

Figure 3 displays an example of the initial hierarchy for the rocket domain, which consists of five operators: `load-rover`, `load-satellite`, `launch-rocket`, `unload-rover`, and `unload-satellite`. Each node has an associated label N, a set of associated differences D_i, and a set of one or more associated operators. For example, the box in the lower left-hand side of Figure 3 shows node `N7`, with differences $\sim$`(inside ?object rocket)` and `(at ?object ?place)`, and operator `(unload-rover ?object)`. Each node has a base probability of occurrence $P(N)$, and each difference has a conditional probability $P(F|N)$ of occurrence given the concept, as does each operator. Node `N7` had a probability of $\frac{1}{2}$ (because it covers one of the two instances of node `N4`), its differences each have conditional probabilities of 1, and its operator has a conditional probability of 1 as well. Cases are kept at the leaves of the hierarchy, as indicated by the expansion of nodes `N7` and `N8`. Node `N4`, the parent of nodes `N7` and `N8`, is more general than either of its children.

As in most case-based systems, retrieval is central to DÆDALUS' operation. Retrieval from memory is done in the following manner. The means-ends engine passes the memory system a PSstate, S, to request an operator. The memory system takes S and temporarily incorporates it into the root node of the concept hierarchy. Incorporation first identifies which features in S correspond to which features in the PSstate of the root node, and then, temporarily updating the conditional probabilities of the features in the node. Finding the correspondence between features in S and PSstates in the concept hierarchy is cast as a partial-matching problem in which the domain constants of the features in S must be consistently bound to the pattern matching variables in the features of the node in the hierarchy. The second part, modifying the conditional probabilities, consists of increasing the probabilities of those features of the concept's PSstate that were matched, and decreasing the probabilities of those features that were not matched.

Once the S has been incorporated into the root node, the memory system must decide which of the root's children has the closest resemblance to the PSstate S. This decision is done by way of an evaluation

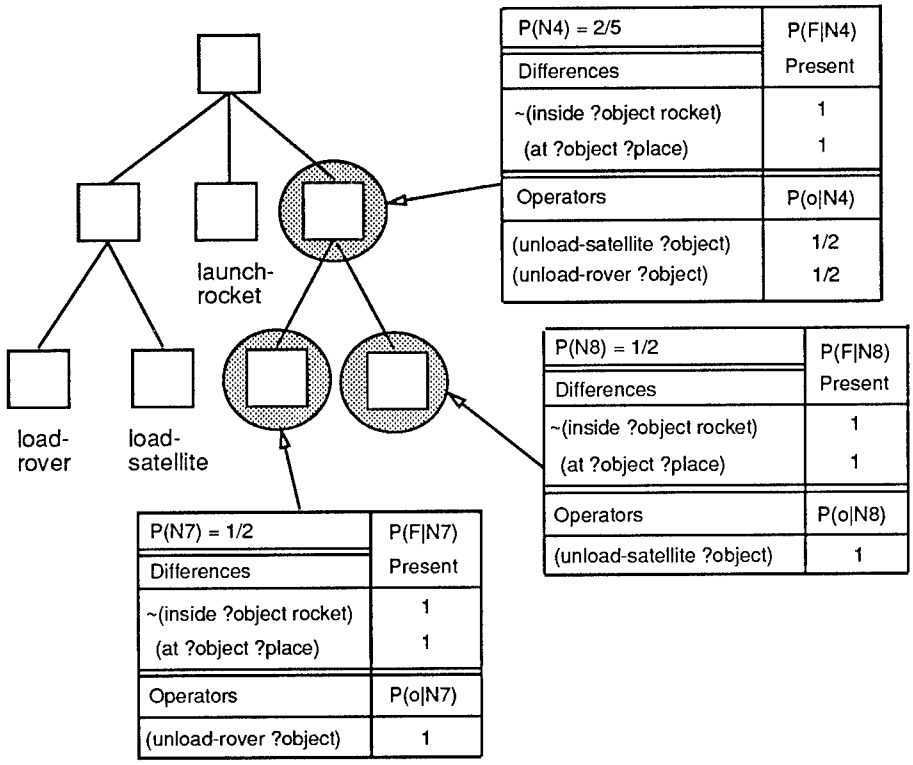


Figure 3. Initial concept hierarchy containing rocket world operators.

function adapted from Gluck and Corter (1985). The function, *category utility* = $[\Sigma_k P(C_k)\Sigma_i P(F_i|C_k)^2 - \Sigma_i P(F_i|C)^2]/N$, evaluates a *partition* — defined as a parent node and its immediate children. $P(C_k)$ refers to the *a priori* likelihood that S is a member of the child C_k. $P(F_i|C_k)^2$ is a measure of *within-class similarity*, that is, how closely to the cases summarized by C_k resemble one another. $P(F_i|C)^2$ is the *within-class similarity* of the parent node; the subtraction of this term lets category utility measure the information gained by dividing the parent into a set of classes. Dividing by N, the number of children in the partition, allows category utility scores to be compared even if the described partitions are of different size. Our modifications allow category utility to be applied to a feature-based representation rather than the attribute-value representation of the original formulation.

In choosing the best child, the memory system temporarily incorporates the PSstate, S, into each child in turn and evaluates the resulting partition. The partition resulting in the highest category utility score determines the node that most closely resembles S. Once that node is found, the PSstate is temporarily incorporated into it and the process is repeated using the selected node and its immediate children as the partition. This process continues until a leaf node is selected or the system determines that continuing down the hierarchy is inappropriate. Currently, the memory system stops at an internal node if the cases below it have been tried and did not lead to a successful plan.

Keeping track of unsuccessful cases allows the memory system to suggest a "next best" operator instance. This is a significant difference in that it places and ordering on the operator instances, rather than separating them into the relevant and irrelevant sets of more traditional means-ends planners.

In summary, DÆDALUS augments simple means-ends planning with a hierarchically organized plan memory. The resulting merger is a planner whose behavior is strongly determined by the experience encoded in the plan memory. This domain knowledge influences the planning process by controlling what operator instances are retrieved for each step in a plan.

2.4 Learning in Dædalus

DÆDALUS integrates learning into its planning process by storing the cases it obtains from derivational traces. The concept hierarchy stores information about the problems and subproblems DÆDALUS has encountered, along with the operators that led to their successful solution. Whenever a plan is found that achieves a problem or subproblem, the description of that problem is stored in the concept hierarchy. This involves storing the case (the PSstate and the selected operator) as a new termi-

nal node in the hierarchy. In addition, DÆDALUS updates the summary descriptions of the nodes (indices) by revising the probabilities on all nodes along the path between the new node and the root. The system invokes this process for each subproblem as it is solved, effectively storing (and indexing) a probabilistic 'selection rule' (Minton, 1988) for deciding among operators.

This process is almost identical to the retrieval precess, with only a few exceptions. As the memory system calculates the category utility of each partition, searching for the best match between the children of the partition and the PSstate, three other possibilities are considered: create a new sibling, merging two children, and splitting a child. At each level, category utility is calculated to determine which child of the current partition is the best candidate for the permanent incorporation of the PSstate. The system also considers putting the PSstate off by itself, creating a new class in the partition. In merging two nodes, DÆDALUS takes the two most promising nodes and checks to see if a new node, summarizing the combined cases of the original two nodes, would result in a partition with a higher category utility score (i.e., yields a greater information gain). In splitting, the system takes the best candidate and replaces it with its children, checking whether the resulting partition has a higher category utility score. If any of these exceptions have higher category utility scores than incorporating S into one of the children, the corresponding modification will be made to the hierarchy.

These hierarchy-modifying operators allow the memory system to alleviate order effects by allowing it to recover from a biased set of examples. If DÆDALUS were trained on a set of blocks-world problems, where every problem started with all the blocks on the table and ended with a single tower, then when given a problem of disassembling a tower it might recommend the `stack` operator, since it has always worked in the past. The hierarchy modifying operators can help the memory system recover from such over-commitment by splitting classes that are overly general, or merging classes that are overly specific.

The ability to merge, split, and create new siblings is a distinction between learning and retrieval. When the system is learning, the classification process makes permanent changes to the concept hierarchy and is using the hierarchy modifying operators. However, during retrieval, the classification process makes no permanent changes, and splitting, merging and creating new siblings are not considered. This distinction allows for hierarchy maintenance during learning without worrying about hierarchy maintenance during retrieval.

This storage process should give DÆDALUS more efficient future behavior. Upon encountering a new problem, the system uses its memory of past successes to select operators in a more discriminating fashion. Specific problems (described by PSstate-operators pair) are stored in the same concept hierarchy as the original operators, and the same sorting process is used to retrieve them. If a stored case matches a new problem or subproblem more closely (according to an evaluation function) than one of the original operator descriptions (because it has more in common), DÆDALUS retrieves this case and attempts to apply the associated operator.

Figure 4 displays DÆDALUS' memory after the derivational trace in Figure 2 has been incorporated. The white nodes are those found in the initial hierarchy (see Figure 3). The black nodes are those incorporated during the planning process. Node `N9` shows the resultant form of the PSstate-operator pair for `(unload-rover rover1)`. The constants in the PSstate have been replaced by variables, allowing a certain amount of generalization. The structural dependencies between the relations have been preserved by assigning the same pattern matching variable to identical constants. For instance, `(rover rover1)` and `(at rover1 Earth)` became `(rover ?x)` and `(at ?x Earth)`. Note that the argument of the operator falls under the same variabilization process. The user can specify constants where variabilization is not desired; `rocket`, `Earth`, and `Mars` have been so declared. The node `N12` is a generalization of the two cases where `(launch-rocket)` proved beneficial. The two cases were nearly identical, but in one case, `rover1` was still on `Earth`, in the other, `rover1` was inside the rocket.

The modification of the hierarchy affects the retrieval process of future queries. After having solved a problem using extensive backtracking, the system will do no backtracking while solving similar problems. At each choice point where DÆDALUS had previously retrieved an inappropriate operator (one that led to backtracking), it now has a concept describing the state and goals that characterize that choice point, and the correct operator instance to be used. When DÆDALUS sees a similar problem, it retrieves the correct operator. In this way, the system builds up a hierarchy of classes describing the problems and situations characteristic of the domain, as well as the operator instance appropriate for each. The result is a planner that does progressively less search, planning more and more by retrieval.

3 Experimental Evaluation of Dædalus

Effectively evaluating a learning planning system is an elusive and slippery task. Recent literature has pointed out shortcomings in previous forms of evaluation and proposed tentative solutions (Minton, 1988; Segre, Elkan, & Russell, 1990). In the evaluation of DÆDALUS, we have tried to avoid the pitfalls of evaluation by using the suggestions proposed by the aforementioned authors, by explaining the limitations of our approach.

3.1 Experimental Method

Our experimental study focused on planning in the blocks world domain. The domain involves a two-dimensional world that contains a table and some num-

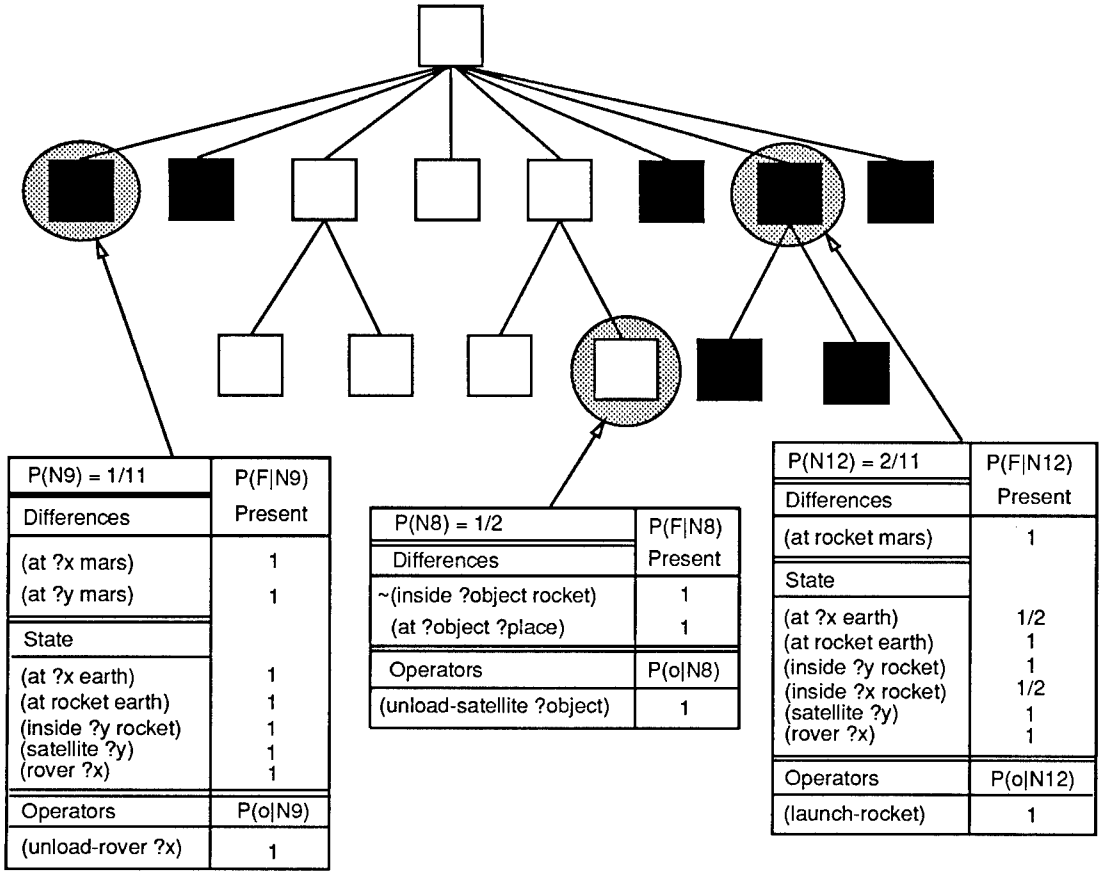


Figure 4. Concept hierarchy after incorporating the derivational trace in Figure 2.

ber of blocks. A block may be on another block, or upon the table. There is no limit to the number of blocks in a single stack or the number of blocks the table may hold. The domain has four operators that change the positions of the blocks: `(stack ?x ?y)`, `(unstack ?x ?y)`, `(putdown ?x)`, and `(pickup ?x)`.

The data was collected by averaging ten trials of forty problems each. A problem is an initial state, made out of a randomly (with some restrictions) chosen number of blocks that are randomly placed on the table and on other blocks, and a randomly generated set of goal conditions, reflecting the desired final configuration of the blocks in the initial state. Each problem was randomly generated by a problem generator, and presented to the system. If a figure compares two systems, each system was run on the same set of problems. For each run, the problems presented to the planners are initially limited to simple problems, those problems containing a small number of blocks and goal conditions. As more problems are encountered, the problem generator is permitted to produce problems having more blocks and more goal conditions. The first 10 problems consisted solely of two block problems with at most one goal condition. The maximum number of blocks and goal conditions was increased by one for every 10 problems presented. Hence, problems one through ten consisted of two block/one goal condition problems, and problems thirty-one through forty consisted of two to five blocks/one to four goals problems.

In collecting statistics for DÆDALUS with learning, the we presented the system with each problem twice. The first time, with the learning component turned off, was used to generate performance measures. The second time was done with learning turned on, and was used to train the system on the problem. This was done to avoid the potential effects of within-trial learning.

Two limitations were placed on DÆDALUS during these runs. The first was a limit on the depth of pending transform goals during backward chaining, which was set at four. This means that no path from the root of the derivational trace to any leaf may traverse more than four upper branches. This limitation was enforced primarily for expedience, since it cut down on the explosivness of the search space. Further code optimizations may allow us to relax or remove this limitation, but currently the search for a solution in the unbounded

space takes too long. The limitation is biased to favor the non-learning system in two ways. First, it restricts the amount of backtracking possible, thereby giving the non-learning system the appearance of finding a solution with little search. Second, in general, the selection rules learned from backward chaining have more transfer than those learned from forward chaining, thereby forcing the learning system to assimilate more information in order to get good performance.

The second limitation is a time limit. In these runs we allowed the system a maximum of ten CPU minutes to solve each problem. This limitation was also used for expedience. There is a difference between the transform goal limit and the time limit. The former limit will not prevent the planner from finding solutions, whereas the time limit can.

3.2 The Effect of Learning on Search

The primary goal of learning in a planner is to reduce the amount of search needed to find a solution. Figure 5 shows the amount of search done by DÆDALUS without learning (white boxes) verses that amount of search done by DÆDALUS with learning (black boxes). The metric for determining search is computed by dividing the number of nodes expanded in the domain space by the number of operators in the solution plan. This gives an indication of the amount of backtracking done by each system.

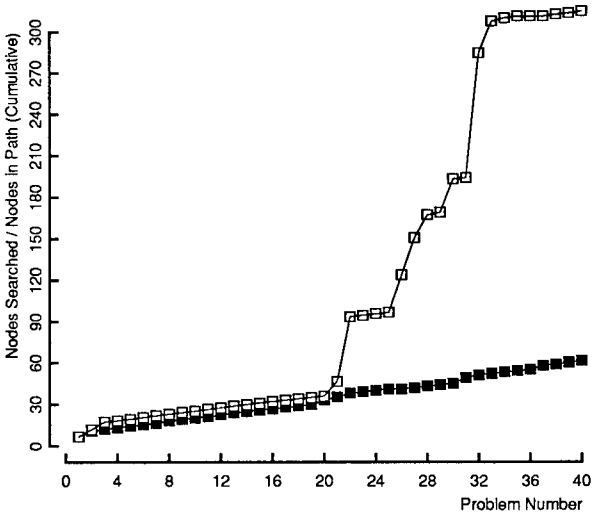


Figure 5. DÆDALUS' search with and without learning.

The graph in Figure 5 shows how *cumulative* search grows as the number and complexity of the examples increases. The cumulative search is the total problem solving search over all examples up to that point; thus, the slopes of both curves are positive because the y-axis represents cumulative search. The second derivative of the curve representing DÆDALUS without learning is positive because the relative difficulty of the problems presented is increasing (the curve levels off at the end because of the time limit). The second derivative of the curve representing DÆDALUS with learning is also positive, but only slightly. This shows that the system with learning is insensitive to problem difficulty, doing very little search on any of the problems.

3.3 The Effect of Learning on Solution Length

Traditionally, learning researchers have been primarily concerned with reducing the amount of search needed to find a plan, but have neglected to study how the learning affects the length of the solution path. There seems little benefit in reducing the effort of finding a plan if it requires the execution system to traverse circuitous or redundant routes. Ideally, a learning planner should at least produce plans that are comparable to those produced by the performance system.

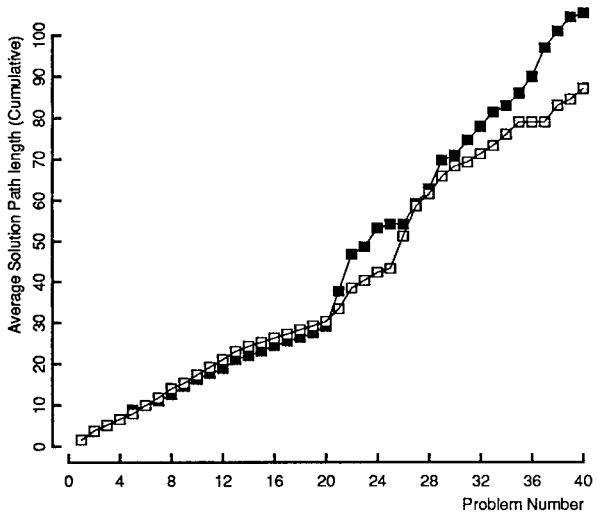


Figure 6. Solution path length with and without learning.

Figure 6 compares the solution paths produced by DÆDALUS without learning and DÆDALUS with learning. The solution paths are averaged over the same ten trials used to generate Figure 5; again, problems were not included in the average. The graph is cumulative, showing the sum of the lengths of all solution paths to that point. From this graph, one can determine that DÆDALUS with learning did not reduce the average path length as compared to the system without learning. However, it does not greatly increase the length of the solution path either. This is due to the nature of the learning mechanism, which builds its concepts solely on the problem-solving behavior of the performance system, and not on an in-depth analysis of the produced plan.

3.4 Testing Other Dimensions

The utility problem, described by Minton (1988), was based on the observation that learning systems, while reducing the amount of search, may actually give rise

to an increase in solution time. In most planning systems, this is characterized by a trade-off that brings about a reduction of search in the domain space but increased operator retrieval time. In DÆDALUS on the blocks world, the effect of this trade-off is an overall reduction in the work done by the system, but the decrease is marginal. This is an important direction for future work.

We have started preliminary testing using the tower of hanoi, rocket world (as shown by our examples), and scheduling domains. We chose the blocks world as our initial testing domain because its explosive problem space lets us study the effectiveness of operator selection. The tower of hanoi domain was chosen for the same reason, but in our formulation there is only one operator, which allows us to study the effectiveness of the selection of variable bindings. The rocket world was chosen because it contains irreversible operators, and gives the opportunity to demonstrate the power of means-ends analysis in conjunction with learning (see section 4.1 for further discussion). The scheduling domain was chosen as a first step towards a real-world domain.

These domains have been coded and some preliminary testing has been done with each. The simple runs have shown a reduction in search without compromising the solution path length. As of yet, we have not determined DÆDALUS' overall performance with these domains, but we hope to report on them as the testing and development of DÆDALUS continues.

4 Discussion

4.1 Generality

Traditionally, means-ends systems required an explicitly defined set of goal conditions that are used to generate differences. They also assume that the generated differences are in the same language as the add and delete lists of the operators. DÆDALUS is somewhat different in this respect. As previously mentioned, DÆDALUS is capable of retrieving operators that are not relevant to the current differences, and can retrieve operators without differences at all. This ability lets the system plan under these conditions, and to learn relations between the differences and the operators, or, if differences are absent, to learn relations that occur between operators and domain states.

For instance, suppose the task were to navigate a maze using three operators: `turn-left`, `turn-right`, and `move-forward-one-step`. If one were to specify `(at mouse position1)`, and `(mouse-orientation north)` as the initial state, having a goal condition `(at mouse position43)` would give no indication what operator should be chosen. However, DÆDALUS will choose an operators at random until it has found a path from `position1` to `position43`. When it learns, it will store each operator with its associated PSstate into the hierarchy, learning the relationship between the goals, states and operators. With sufficient training, DÆDALUS should be able to map out the maze, and be able to navigate it with reasonably little search.

This property allows DÆDALUS to run in domains that are not traditionally thought of as means-ends domains. It also makes the system less dependent on the choices of representation for any particular domain. Another property stems from DÆDALUS' strategy of operator selection, which differs from earlier methods it *orders* operators, rather than dividing them into relevant and irrelevant sets. One result is that DÆDALUS prefers operators that reduce multiple differences in the current problem, which should make it more selective than traditional techniques. More important, although DÆDALUS prefers operators that reduce problem differences, it is not restricted to this set. If none of the 'relevant' operators are successful, it falls back on operators that match none of the current differences. This gives it the potential to break out of impasses that can occur on 'trick problems'.

This ability makes DÆDALUS less susceptible to some of the criticisms made about linear planners. There are two classes of 'trick problems' for such planners. The first class, which includes the famous Sussman anomaly (1973), consists of problems whose strong goal interaction prevent linear planners from finding an optimal solution. DÆDALUS provides little for improving this class of problems; if a solution can be found, the system behaves like many other linear planners. The second class of problems are those in which a combination of irreversible domain operators and goal interaction prevent *any* solution from being found by a linear planner. This second class can be characterized in the following way: if one expands a domain with irreversible operators into a graph, where the nodes represent each possible state and the links represent the operators that transform one state into another, the resulting graph can be partitioned into at least two subgraphs in which the subgraph A has links into subgraph B, but subgraph B has no links into subgraph A. A 'trick problem' in such a domain has an initial state in subgraph A, and two or more interacting goals whose satisfying states are found in subgraph B. In solving the first goal, the planner must travel from subgraph A into subgraph B. As the linear planner attempts to solve the second goal, it will have to return to subgraph A in order to undo the solution of the first goal; however, there is no way to do this. To solve this problem a planner must recognize that certain subproblems must be solved before it leaves subgraph A. Non-linear planning is one way to achieve this, DÆDALUS's technique of learning the interactions provides a different solution.

4.2 Related Work on Learning and Planning

DÆDALUS is a learning planner with several unique properties, but it is only one of a set of planners produced by the machine learning field. With this in mind, it seems beneficial to compare and contrast our system with other planners that address similar issues.

Of the more traditional planners, DÆDALUS is most similar to the PRODIGY/EBL (Minton, 1988) system. Both systems rely on means-ends analysis for the generation of plans, and both systems learn information in the form of preference rules (probabilistic in DÆDALUS's case). The most significant differences between the two systems lie in what is learned, and how it is learned. PRODIGY/EBL acquires several types of control rules: node preference, node rejection, operator selection, operator preference, operator rejection, goal preference, goal rejection, binding preference, and binding rejection. These rules are separately learned and stored, using an explanation based learning technique with different domain knowledge for each type of rule. DÆDALUS only learns a subset of those rules (goal select, operator select, and variable select), and is acquires them using an incremental conceptual clustering technique.

The retrieval process returns an operator and its candidate set of bindings, an obvious mapping to selection rules for operators and bindings. Less obvious is the goal selection. As a PSstate is being sifted down the concept hierarchy, it becomes associated with nodes that make stronger and stronger claims about which goals are relevant, that is, which goals have a high probability of occurrence in the PSstate. Although this does not specify which goals are to be worked on, it does show which goals are being considered.

The PLEXUS system of Alterman (1988) is another planning system that makes extensive use of nonepisodic knowledge to eliminate search during plan repair. One marked similarity of DÆDALUS and PLEXUS is the storage and use of both general and specific plan information. In both systems, the retrieval and use of specific plans over general plans is preferred. However, both systems may make use of general plans if necessary. Another similarity is in plan repair. If a plan step fails for PLEXUS, it assumes that the failed plan step is "representative of the category of action" and will use background knowledge to find a new plan step. The repair process uses abstraction (moving up its categorical hierarchy from the failed plan step), and specialization (moving down its categorical hierarchy) to search for a suitable replacement. DÆDALUS operates in much the same way. If a plan step fails, DÆDALUS will query its memory for an alternative solution, which is usually taken from a sibling or an abstraction of the plan step initially retrieved.

The planning/execution system PLOT (Yang & Fisher, 1990) is based on the same observation as DÆDALUS, that *operator selection can be viewed as classification.* As a consequence, both systems use means-ends analysis to generate plans and probabilistic hierarchies to store plan knowledge. There are two significant differences between the PLOT and DÆDALUS: what they learn and their response to the utility problem.

DÆDALUS creates a plan in the form of a derivational trace, then breaks up the trace into its PSstate-operator pairs. PLOT also creates its plans in the form of a derivational trace, but then creates a macro-operator out of the operator sequence and stores it. These macros are indexed by the preconditions and the differences of the macro itself; all PSstate information that made up the derivational trace is discarded. As a result, PLOT balances between reactive behavior (by retrieving operators whose preconditions are satisfied) and traditional planning, whereas DÆDALUS learns rules that reduce the amount of search needed to plan.

In addressing the utility problem, PLOT borrows from Minton (1988) in keeping statistics that measure the utility of learned information. In Yang's system, a monitor watches the base rate associated with the concepts of PLOT's plan memory. If a base rate drops too low the concept is judged to be of low utility (since it is used rarely), and is pruned from the tree. DÆDALUS does no pruning, but relies on the speed of its heuristic partial-matching function and tree-shaped memory to address the utility problem.

4.3 Directions for Future Research

As it currently stands, DÆDALUS is much too simple to plan effectively in a real-world setting, but future research may remedy many of its limitations. For instance, the system learns only from successful plans and never from failures. No matter how many times the system fails to solve a problem, it cannot use knowledge of those failures to constrain future search. Consider the following scenario: DÆDALUS is controlling an autonomous robot, and it stands on the curb of the killer street of the reactionary planners (Hendler & Sanborn, 1987), but suppose now that the robot is sufficiently armored to resist four or five collisions. As DÆDALUS is currently implemented, it would step into the street, see a car coming, start planning and get run over. Now, due to its protective shell, it would have survived the collision, but since it did not successfully achieve its goal of avoiding the car, it learned nothing. Having not learned anything, DÆDALUS would do exactly the same thing every time a car came at it until it was finally obliterated. However, if DÆDALUS could learn from failure, it could learn bad operator choices as well as good operator choices, it might be able to avoid collisions after a few tries.

The natural response is to store not only those PSstate-operator pairs that led to a successful plan, but also those that failed to produce a solution. The simplest version of this approach would store both types of cases, marking one desirable and the other undesirable. If the extended DÆDALUS retrieved only undesirable operators, it would select some untried operator first. However, this approach ignores the fact that some failures are less undesirable than others, just as some some successes are less desirable than others.

A more sophisticated approach would associate a desirability score or *affect*, ranging from ∞ to $-\infty$, with each final state in a plan (such as being across the street or being hit by a car). Using a technique similar to Sut-

ton's (1988) method of temporal differences, the system would propagate affective scores back to the intermediate states (such as being halfway across the street) that are stored in memory. States close to desirable states would acquire some of the latter's positive affect, while those close to undesirable states would acquire negative affect. Upon encountering a new problem, the system would retrieve a number of plausible operators, as in Jones' (1989) EUREKA, and then select the one that is expected to produce the state with the highest affective score. Over time, the temporal-difference method might produce cases and abstractions that accurately predict the desirability of the states to which they lead, letting one learn from varying degrees of success and failure.

5 Conclusion

This paper has described DÆDALUS, which we designed in an attempt to bridge the gap between planners that learn abstract knowledge and those that learn by creating indexes to specific cases. We evaluated the system in terms of its ability to reduce the amount of search without adversely affecting the solution path lengths in the blocks world domain. Experiments in the rocket world domain have shown that DÆDALUS' combination of means-ends analysis with a memory in the form of a probabilistic concept hierarchy allows the system to successfully plan in domains not normally accessible to linear planners.

Although the system has proved successful in capturing planning knowledge in toy domains, further testing is needed to examine the full generality and limitations of the system. Future work will be needed to extend DÆDALUS' capabilities before it can be applied to real-world domains.

Acknowledgments

We would like to thank members of the ICARUS project – John Gennari, Kevin Thompson, Wayne Iba, Kate McKusick, and Deepak Kulkarni – as well as Mike Pazzani, Doug Fisher, Hua Yang, and Randy Jones for useful discussions that led to many of the ideas in this paper. We also thank Marilyn Bunzo, John Bresina and Steve Minton for their comments on an earlier draft.

References

Alterman, R. (1988). An Adaptive Planner. *Proceedings of the DARPA Workshop on Case-based Reasoning* (pp. 37–49). Morgan Kaufmann: Clearwater Beach, FL.

Carbonell, J. G. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence, 32*, 333-377.

Fikes, R. E., Hart, P. E., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence, 2*, 189–208.

Fisher, D. H. (1987). Knowledge acquisition via incremental conceptual clustering. *Machine Learning, 2*, 139–172.

Gluck, M., & Corter, J. (1985). Information, uncertainty and the utility of categories. *Proceedings of the Seventh Annual Conference of the Cognitive Science Society* (pp. 283–287). Irvine, CA: Lawrence Erlbaum.

Hendler, J., & Sandborn, J. (1987). Planning and reaction in dynamic domains. *Proceedings of the DARPA Workshop on Planning.*

Jones, R. M. (1989). *A model of retrieval in problem solving.* Doctoral dissertation, Department of Information and Computer Science, University of California, Irvine.

Kolodner, J. L. (1987). Extending problem solving capabilities through case-based inference. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 167–178). Irvine, CA: Morgan Kaufmann.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 564–569). Morgan Kaufmann: St. Paul, MN.

Sacerdoti, E. D. (1974) Planning in a hierarchy of abstraction spaces. *Artificial Intelligence, 5* 115-135.

Segre, A., Elkan, C., & Russell, A. (1990). *On valid and invalid methodologies for experimental evaluations of EBL* (Technical Report 90-1126). Ithaca, NY: Cornell University, Department of Computer Science.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning, 3*, 9-44.

Thompson, K., & Langley, P. (in press). Concept formation in structured domains. In D. Fisher & M. Pazzani (Eds.), *Computational approaches to concept formation.*

Veloso, M. M. (1989). *Nonlinear problem solving using intelligent casual-commitment* (Technical Report CMU-CS-89-210). Pittsburgh, PA: Carnegie Mellon University, School of Computer Science.

Yang, H., & Fisher, D. (1990). *Improving planning efficiency by conceptual clustering* (Technical Report CS-90-07). Nashville, TN: Vanderbilt University, Department of Compute Science.

Planning to Address Uncertainty: An Incremental Approach Employing Learning Through Experience

Scott Bennett
Beckman Institute for Advanced Science and Technology
University of Illinois at Urbana–Champaign
405 North Mathews Avenue, Urbana, IL 61801
bennett@cs.uiuc.edu

1. Introduction

No model, no matter how complex, can exactly mirror the real world. Consequently, any planner with a static model will inevitably be handicapped in dealing with a complex real–world domain, unable to deal with failures which are a result of an inadequacy of the model. For example, suppose we are developing plans which employ a gantry–arm in a warehouse to lift and place crates and boxes. In this complex environment, shown in Figure 1, we don't have perfect knowledge about the

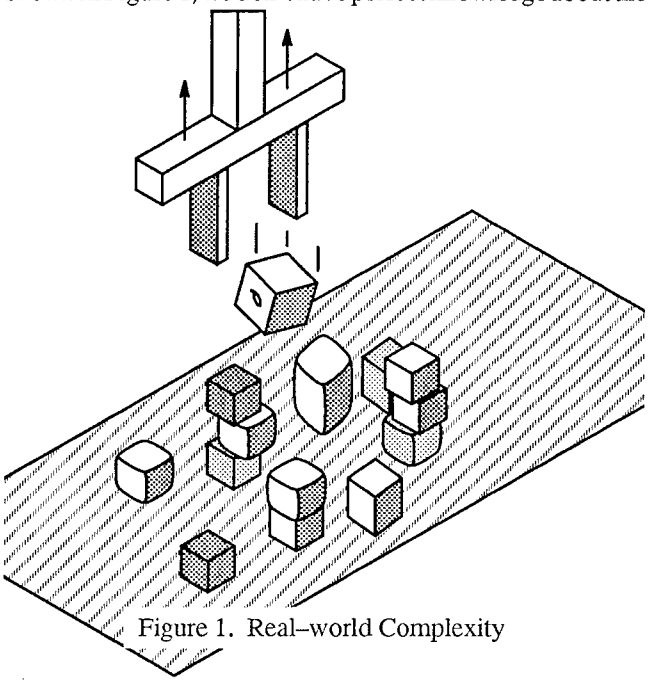


Figure 1. Real–world Complexity

effects of operators we might employ. The gantry–arm is not perfectly accurate. It also is a complicated mechanism which might fail for many reasons. The planner also cannot have a perfect model of the world objects. Boxes are not perfect rectangular prisms. Even if a more complex model were used, it is always possible that some detail, beyond the resolution of the model, could lead to an unexpected failure. A small hole in the side of one carton may catch on the sharp corner of the gripping device causing the box to slip and fall as it is lifted. In such a domain, no plan can be perfect. This is the qualification problem. More qualifications can always be added as to when a plan should not be attempted because it could fail.

One of the major differences between artificial domains and complex real–world domains is the presence of uncertainty. Most simplified models assume facts are known with certainty while in the real world often they are not. Actions carried out in the world may often have uncertain effects. Simplified representations also introduce uncertainty because they trade off accuracy for ease of use.

The primary approach to developing plans in the presence of uncertainty has followed the theme: "If we can represent and reason about uncertainty, we can develop plans that are guaranteed to succeed anyway." This theme has been pursued strongly in the robotics community. In 1982, Brooks made use of numeric error bounds which could be propagated back through a plan to specify the conditions under which the plan could achieve its goal [Brooks82]. This analysis also indicated when additional operators should be added to the plan to reduce the restrictions on uncertainty in the plan preconditions. Many planners followed which employed explicit uncertainty representations and built on these techniques including the LMT Planner [Lozano–Perez84], EDR [Donald90], and SPAR [Hutchinson90].

Generally, in these approaches, numeric error ranges are assigned to the sensors and effectors. During planning, possible error ranges are continually re–calculated as new operators are introduced into the plan. In several approaches, if the accumulated error exceeds some preset limit, operators are introduced specifically for reducing the error. Guaranteed uncertainty–tolerant plans are sought, so all possible modes of potential failure (due to uncertainty) must be considered during planning. Producing such guaranteed plans can be very difficult. This should not be surprising considering that the already complex domain representation has been made more complex through the addition of uncertainties. For instance, in Donald's EDR [Donald90] approach, generating plans when model error is possible involves generating and navigating through a large number of *slices* of configuration space, each corresponding to a different constant model error. Each point in a configuration space corresponds to one possible position and orientation of the piece being manipulated. Even if we restrict ourselves to a planar object with position and orientation, each slice of configuration space would be three–dimensional. In general, another dimension is added for every

uncertainty which is to be considered. In fact, when model uncertainties are introduced it is not always possible to find a guaranteed solution. Donald recognized this and uses an error detection and recovery (EDR) approach relaxing the guaranteed criteria to allow the system to try a plan if it can be guaranteed that the possible resulting failure or success states can be recognized. If a failure occurs, it continues planning to achieve the goal from the recognized failure state. However, recognizing when a guaranteed plan is not possible, in order to apply EDR, still involves the same expensive techniques used when finding a guaranteed plan.

Despite the fact that small uncertainties abound in many dimensions considered by these planning systems, the uncertainties may not be equally likely to cause errors with a given set of tasks. Furthermore, if a plan is modified to tolerate some particular uncertainty, it may well be tolerate other uncertainties as well. Consequently, reasoning about all the uncertainties at planning time may involve a considerable waste of resources in comparison to an approach which deals with uncertainties if they cause difficulty during plan execution. If the domain is one where failures can be tolerated during a *training phase*, a system can be employed which learns from its failures, introducing uncertainty–tolerance into plans in an on–demand fashion.

In our system, explicit approximations can be declared as deal with quantitative facts about the world. Rules employed by the system in constructing plans include tunable parameters. These parameters can be tuned in response to failures to improve the uncertainty–tolerance of generated plans. When the systems generates plans, no explicit reasoning about the approximations takes place. Only when recovering from execution failures does reasoning about the approximate nature of the data take place and then only in a limited qualitative manner. The planner can therefore quickly construct unguaranteed plans which can be refined as necessary to deal with uncertainties as they cause problems. Furthermore, plans which are constructed are generalized using *explanation–based learning* [DeJong86, Mitchell86]. These generalized plans can be employed in a class of similar situations meeting the same causal constraints as seen in the specific instance for which planning took place. This eliminates the need for the planner to generate separate plans for these similar situations.

First, we will discuss the types of approximations the system employs for modelling world facts. Next, parameter–based rules are discussed whose parameters can be tuned to add uncertainty–tolerance to operations planned using them. An architecture is then presented for generating and refining approximate uncertainty–tolerant plans. This includes a discussion of the role of explanation–based learning, execution and monitoring, and plan refinement. A sample task domain of robotic grasping is then described along with an example of system operation and some initial empirical results. Lastly, we discuss the future directions for the work.

2. Data Approximations

Data approximations can be either *external* or *internal*. An external data approximation is used to represent the uncertainty of data in the world. The system employs internal data approximations to simplify complex sets of data to make reasoning more tractable. First, let us consider external data approximations.

2.1. External Data Approximations

An external data approximations involves a set of quantities for which the system is given approximate values. Let Q be a vector $\{q_1,q_2,q_3,...,q_n\}$ of quantity variables and A be a vector of their corresponding approximate values $\{a_1,a_2,a_3,...,a_n\}$. A data approximation asserts that:

$$\bigvee_{i=1}^{n} P(q_i = x) < P(q_i = y) \text{ iff } |x - a_i| > |y - a_i|$$

This gives a very simple qualitative view of uncertainty distributions like that shown in Figure 2. It specifies that the probability of one encountering the true value of the quantity diminishes as one moves away either above or below the approximate value.

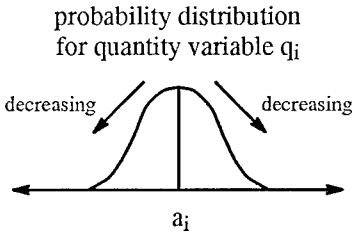


Figure 2. Characterizing a Data Approximate Quantity

In the case of external data approximations, the value vector A is the best information the system has about the values of the quantity variables Q. The only way to improve this information is to interact with the world. For purposes of planning, Q=A. The qualitative definition of a data approximation is never employed during planning, only when analyzing failures so as to tune rule parameters.

In robotics, external data approximations can be used to represent values read from sensors, which are inherently uncertain. For instance, the position of a block, as sensed by a visual system, would be represented with an external data approximation.

2.2. Internal Data Approximations

With an internal data approximation, the system chooses the values A of the quantity variables Q with a data approximation procedure. This is often motivated by the need to simplify the representation so reasoning can be performed more efficiently. Internal data approximations can be adjusted through the system's reasoning alone. The qualitative view of a data ap-

proximation applies to both external and internal data approximations.

In robotics, geometric object models are examples of internal data approximations. A simplified geometric representation can be far more efficient to reason about than the complex raw data returned by a vision system. However, in seeking a simplified representation, accuracy is sacrificed. The system has thus introduced uncertainty.

3. Parameter–Based Rules

Parameter–based rules are employed when the system plans how a goal can be achieved. The parameters are tunable and if tuned correctly can be used to improve the uncertainty–tolerance of generated plans. We make use of the following types of parameter–based rules:

constraints

Suppose that a system, in achieving some goal, must choose the best from a set of candidates. Each of the candidates differ along one or more continuous dimensions. A constraint rule is used for indicating a preference for candidates based on their value along one of these dimensions. Each constraint rule functions as part of a multi–constraint rating rule for evaluating a set of candidate choices. The parameter on which the constraint rule is based, when tuned, has an effect on how the candidates are rated by that rule. In the robotic grasping domain, parameter–based constraint rules are used in choosing the best faces to use in achieving a grasp. Currently implemented constraint rules include *opening–width–constraint* and *contact–angle–constraint* for constraining the choice of faces to those with a realizable opening width and a contact angle within the friction angle.

controls

These rules directly choose the value for some controllable quantity. The parameter to the rule directly effects this choice. Controls can be either *external* and *internal*. External controls are directly associated with some controllable parameter in the outside world. Internal controls affect the system's internal reasoning. For example, in the robotic grasping domain, an external control is *chosen–opening–width* which chooses the amount by which the gripper should open for achieving the grasp. Weights which are used in combining constraints are examples of internal controls. They affect the overall rating which some set of constraints give a candidate by determining how constraints are weighted with respect to each other. For example, *opening–width–constraint* has an associated weight *opening–width–constraint–weight* used in evaluating it relative to *contact–angle–constraint*.

3.1. Control Parameters

Figure 3 gives a *constraint diagram* for the *chosen–opening–*

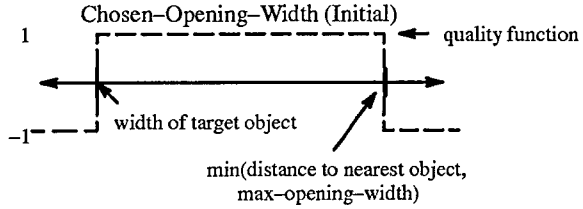


Figure 3. Initial Constraint Diagram for the Chosen–Opening–Width Parameter

width parameter employed in the robotic grasping domain. This continuous parameter, corresponding to opening–width of the gripper for grasping the object, has an upper and lower bound specified. These are specified by general expressions which return the correct bounds for the current grasping situation. In this case, even in the most approximate model, the gripper must be open at least as far as the width of the target object at the current orientation and position. It also must not exceed the maximum possible gripper opening limit of the robot and must not open so wide as to collide with a nearby object. Of course, representations for the objects and their positions and orientations are known to be data approximations. The initial belief of the system regarding opening width is that anything lying between the bounds on the value of opening width are acceptable values. The dashed line in Figure 3 is called the quality function and gives the system's current evaluation of the various values which could be chosen for the opening width parameter. The initially flat function indicates no preference as long as the preconditions for the bounds are met. However, in choosing an opening width, minimal motion of the control from its current value is preferred, reducing movement viewed as extraneous. Therefore, initially, the constraints on the *chosen–opening–width* parameter cause three rules to be generated. In general, the number of rules depends on the number of peaks and plateaus of the quality function. One rule is generated which prefers that the control be set to the lower bound when the current value is less than the lower bound, one leaves the control at its current value if that value is between the bounds, and one sets the control to the higher bound if it is greater than the higher bound. One of these rules is shown in Figure 4. A declarative specification for the constraint on the parameter is translated into a set of rules employed in planning. When control parameters are tuned in response to failures, their corresponding rules are revised and will take on the new desired behavior in planning. Tuning of control parameters amounts to posting preferences in the region between the bounds in Figure 3. These have generalized conditions which calculate the numeric location of the preference and have an effect on the quality function for the parameter. For instance, after posting a preference for opening widths greater than the lower bound, the new quality function appears as in Figure 5. The new constraint on the parameter value

```
INTRA-RULE: R190 ← one of three rules defined by the initial constraints on the opening width parameter
    FORM:   (CHOSEN-OPENING-WIDTH ?GRIPPER ?X ?Y ?ANGLE ?OBJECT ?RETURN)
    ANTS:   (GRIPPER-OPENING ?GRIPPER ?LOP187)                          find minimum required
            (GRIPPER-PERP-WIDTH ?GRIPPER ?SPAN)                         opening so fingers don't
            (MIN-SPAN-FOR-OBJECT ?OBJECT ?X ?Y ?ANGLE ?SPAN ?LEFT ?RIGHT) ← collide with object in
            (SUM ?LEFT ?RIGHT ?RETURN)                                  approximate model
            (MAX-GRIPPER-OPENING ?GRIPPER ?MAX-OPEN)
            (<= ?RETURN ?MAX-OPEN) ← can't realize it even in approximate model if too wide for gripper
            !
            (< ?LOP187 ?RETURN)
    APPROX: CHOSEN-OPENING-WIDTH ← indication that this rule is based on the opening width parameter
```

Figure 4. One of the Rules Generated from the Constraints on the Chosen–Opening–Width Parameter

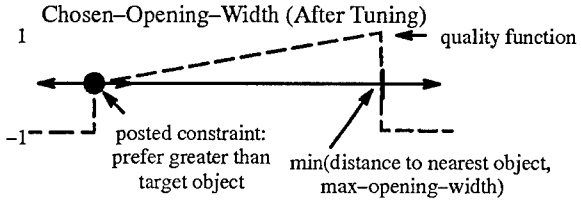


Figure 5. Constraint Diagram for the Chosen–Opening–Width Parameter After First Tuning

causes the corresponding rules to prefer an opening width corresponding to the new peak of the quality function at the maximum opening width.

3.2. Constraint Parameters

Constraint parameters work similarly to the control parameters except they don't choose a value directly but rather use their quality function to give a value some rating. If a constraint rule were evaluating one particular dimension of a set of candidate choices, the value of the quality function of the constraint parameter would be consulted for each candidate's value along that dimension. For that particular dimension, preference would be given to the candidate with the best quality function value for that dimension. Of course, candidates are generally evaluated using several constraint rules which have their resulting ratings combined using weights. These weights were previously mentioned as examples of internal controls. They can be tuned in response to failures to prefer certain constraint rules over others.

In the next section, we introduce the architecture which makes use of these types of approximations in conjunction with parameter–based rules for performing learning and planning.

4. An Architecture for Learning and Planning with Approximations

The system is organized as illustrated in Figure 6. There are three modes of operation. Input to the system is through the approximate explanation–based learning component shown in Figure 7. In the simplest mode, a goal is presented to the system which already corresponds to a plan in the knowledge–base whose preconditions are satisfied in the current state of the world. In this case, that plan is passed directly to the execu-

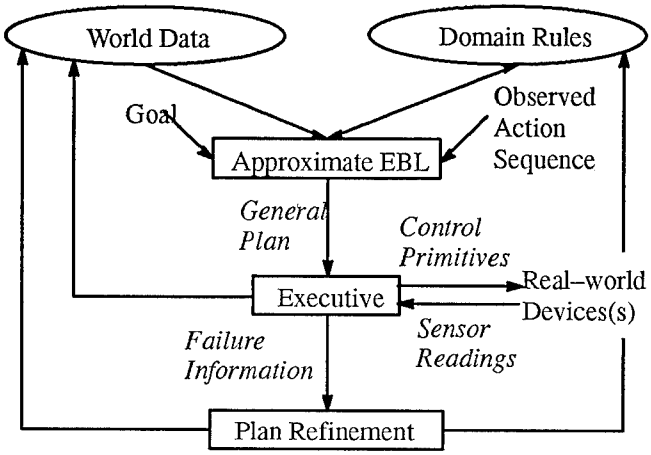


Figure 6. Approximation Architecture

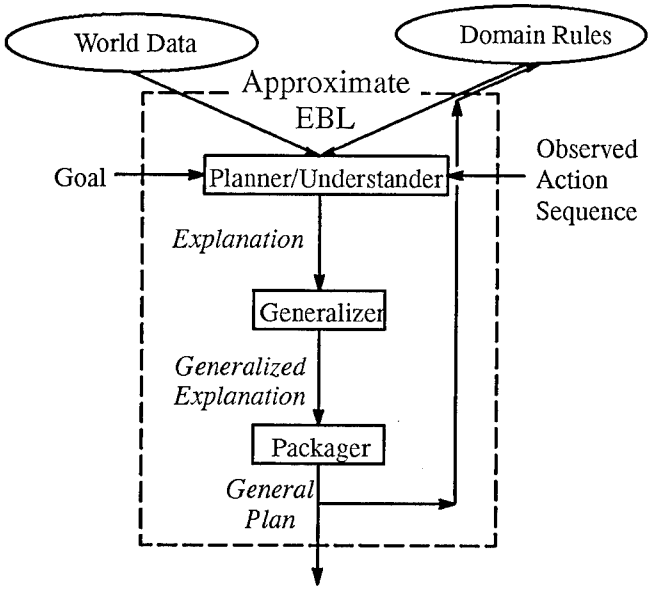


Figure 7. Approximate Explanation–based Learning

tive to be carried out. Secondly, a goal could be presented to the system which doesn't correspond to any known plans. In this case, an explanation is generated for how the goal can be

achieved in the current state. The explanation is then generalized and packaged into a general plan which is then saved and passed on for execution. Lastly, a goal and observed action sequence can be given to the approximate EBL component which then generates the explanation from both the goal and the observed actions. This is the preferred mode of operation of explanation–based learning systems because it can make explanation construction an easier task. In this last mode, the explanation is then generalized, packaged into a plan, saved, and the plan is passed on to the executive.

The executive instantiates the execution sequence associated with the plan with the appropriate bindings obtained from evaluation of the plan's preconditions in the current state. Many of these actions will be monitored, having sensor expectations associated with them that define success or failure. Should a failure occur, as defined by the expectations, the plan refinement module is called to perform tuning of rule parameters so as to decrease the chance of future failures. If no errors occur, the system is ready for the next goal or goal/observation pair.

We will now consider the components performing approximate EBL, execution and monitoring, and parameter tuning in more detail.

4.1. Approximate Explanation–based Learning

The explanation–based learning component of the system is largely the same as those in other EBL systems which require perfect world models, although the data includes declared approximations. The explanation–based component operates on these as if they were data from a perfect world model. The planner thus constructs plans treating the approximate data like they were certain, reducing planning cost in comparison to techniques that reason explicitly about the uncertainty. Of course, because of the use of tunable parameter–based rules, uncertainty–tolerant plans may be created. If the system is being used to learn from observation, an explanation is constructed (planning) from both the goal and the observed actions. Consequently, only aspects of the observed actions which are supported by the rules and data approximations become part of the resulting explanation for goal achievement. For instance, in opening a robotic gripper to surround an object, the human operator of the robot opens slightly wider. The systems starts with initial constraints on the parameter for gripper opening width which require only that the gripper be open just as wide as the object. This coupled with the bias for minimal movement causes a preference for opening only as wide as the target object if the gripper is initially open less wide. Consequently, the increased opening of the gripper, beyond the minimal opening width required, is perceived as extraneous and is not part of the explanation for goal achievement and hence does not appear in the resulting general plan. This is the desired behavior of an approximate EBL system. On one extreme, the action sequence could have been repeated literally in the plan. This would be highly non–general. On the other extreme, extensive reasoning could have been done using the available domain knowledge to construct the perfect general plan from the observed actions. This would be intractable. The purpose of an approximate model and tunable parameter–based rules is to reduce this sort of inference. Naturally, this approach requires a means for improving the imperfect plans should failures occur.

4.2. Execution and Monitoring

In order for the system to improve its plans when they don't perform well in the world, the system must have a monitoring capability. It is important that the system be able to represent what actions are to be carried out, what the expected outcome of those actions is, why that outcome is expected, and when the specified actions should be terminated. Figure 8 illustrates the syntax of monitors in the current implementation. The monitor specifies one or more coordinated actions which are performed simultaneously. Expectations are specified which are evaluated continually during execution, in the case of sensor expectations, and are also checked after termination of the action, in the case of expected features of a full sensor trace. Terminations specify under which conditions the set of actions should be halted. Any monitored set of actions employed in a plan must have its expectations justified. The support field of a monitor specifies a predicate which, if proven, justifies that the expectations will hold throughout execution of the monitor. Figure 9 gives a concrete example of a monitor employed for closing the robotic gripper on an object. The single action specified is for the gripper to begin closing

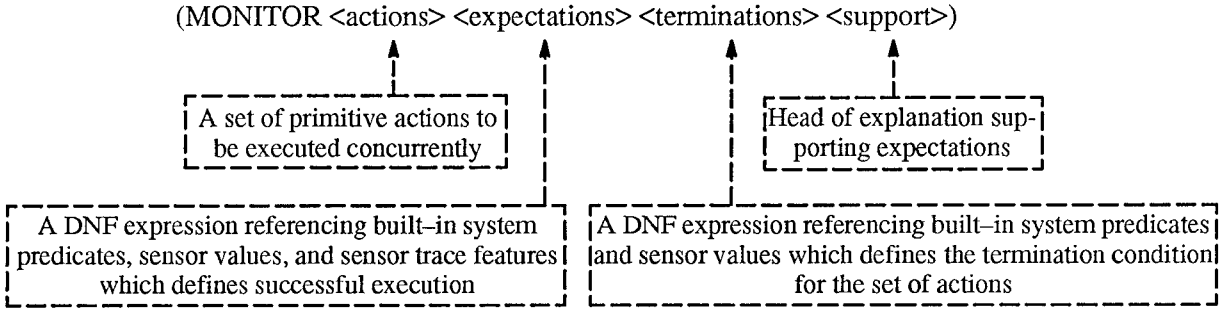


Figure 8. Syntax for Monitored Actions

```
(MONITOR   (MOVE-GRIPPER ?GRIPPER32878 CLOSE 20 64 20 POSITION)   ← Close Gripper Until Termination Met
           (AND   (FINAL-FORCE GRIPPER ?GFFORCE32731)
                  (> ?GFFORCE32731 50)                            ← Final force should exceed 50 units and final
                  (FINAL-POSITION GRIPPER ?GFPOS32732)              position should be such that the gripper
                  (> ?GFPOS32732 1))                                didn't close on itself (otherwise failure).
           (OR    (AND   (POSITION GRIPPER ?GPOS32733)
                         (PRINT (LIST (QUOTE POSITION) ?GPOS32733))
                         (QUAL= ?GPOS32733 0))                      ← Terminate action if the gripper closes on
                  (AND   (FORCE GRIPPER ?GFORCE132734)                itself or if the force exceeds 60 units
                         (PRINT (LIST (QUOTE FORCE) ?GFORCE132734))
                         (> ?GFORCE132734 60)))
           (STABLE-GRASP ?GRIPPER32878 ?OBJECT32870 ((RELATIVE-FACE ?NAME132744 ?X132762
?Y132763 ?REF-ANGLE32779 ?LEN132748) (RELATIVE-FACE ?NAME232749 ?X232764 ?Y232765 ?REF-ANGLE32824
?LEN232753))))
                                                                   ← Justification for the expectations is that a stable grasp
                                                                     has been planned by the system.
```

Figure 9. An Example of a Monitored Action

from its current position. The expectation is that the final force of the gripper on the object exceed 50 units and that the gripper not close on itself. The action terminates when the gripper exerts a force greater than 60 units on the object or the gripper closes on itself. The expectation of feeling the object between the fingers with force greater than 50 units is justified by an explanation for why the planned grasp is stable (so the object will not slip away as force is applied). The specification of expectations as well as their justification facilitates attribution of execution failures to poorly set rule parameters. The process by which parameters are tuned is discussed in the next section.

4.3. A General Algorithm for Tuning Parameters in Light of Failures

Given a goal, the system constructs an explanation for how the goal may be achieved. This can be accomplished in either an understanding mode, given an applied operator sequence, or in a planning mode where the operator sequence is derived. Rules involved in constructing the explanation include parameter–based rules as outlined above. Most of the facts employed in constructing the explanation are data–approximate having been derived from sensed values from the real world. In order for a monitored action to be achieved in the explanation, a set of expected sensor values must be justified by a further subpart of the explanation. The overall explanation is then generalized using EGGS [Mooney86] and packaged into a rule as with standard EBL systems. When the rule is executed in the real world, those sensors described in the monitored actions are observed. If the sensor readings observed violate the constraints described in the monitored actions, plan execution has failed.[1] In this case, the subpart of the original explanation which justified the expected sensor readings is suspect. Clearly, in the approximate model of the world, no error was foreseen, otherwise the explanation would not have been possible. This suspect subpart of the original explanation is the starting point for our general tuning algorithm.

The tuning algorithm has two major steps:

1) generate a qualitative explanation for how the probability of the failed expectation can be increased through tuning of parameters in the rules employed and

2) perform the actual tuning of the indicated rule parameters.

The key in accomplishing the first step is to express the relationships between generalized variables in the failing subtree as qualitative relations. This will make possible qualitative proofs which relate data–approximate quantities, rule parameters, and qualitative probabilities of success of the various predicates. The procedure is depicted graphically in Figure 10. First, the sub–tree of the overall explanation which supports the failed expectations is instantiated with the generalized binding list which EGGS produced for the original goal explanation. The predicates at the root and leaves of this sub–tree are asserted in a new situation as qualitative relations. The quantity arguments to these predicates (which are generalized variables) become quantities in our qualitative model of the sub–proof. Any data–approximate quantities or rule parameters which took part in the original explanation

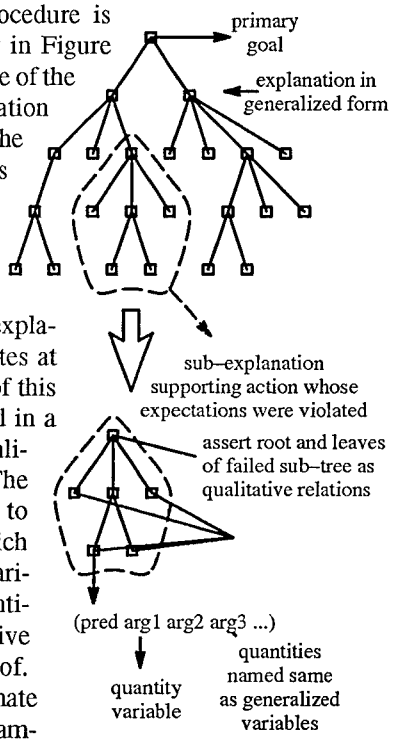


Figure 10. Generating the Qualitative Model

1. Parameter tuning in our system is driven based on *expectation failure*. This idea has long been advocated by Roger Schank [Schank82].

and whose quantity variables are members of the set of generalized quantity variables for the sub–proof are asserted as data–approximate and tunable quantities respectively in the current situation. Once these facts have been asserted which pertain to the specific proof tree, the goal of increasing the probability of the root predicate to the sub–proof can be proved using a set of domain–independent qualitative rules.

There are four classes of domain independent qualitative rules used by the system for generating the qualitative tuning explanation:

general qualitative inference rules

These are rules for inferring the effects of changes in quantities. For instance, the qualitative proportionality predicate *(Q+ ?a ?b)* asserts that the magnitude of the quantity *?b* directly influences the magnitude of the quantity *?a*. Therefore, one such inference rule states that to achieve the goal of increasing *?a* one could find a quantity *?b* for which *(Q+ ?a ?b)* holds and try to achieve the goal of increasing *?b*.

qualitative predicate definitions

These rules provide qualitative representations for the quantitative predicates employed in generating explanations. For example, the predicate *(dif ?q1 ?q2 ?r)* is used by the system for taking the difference between two quantities (*?q1* and *?q2*) and computing the result (*?r*). One of several rules which form the qualitative predicate definition for the *dif* predicate is shown on the right. It states that the magnitude of the quantity *?q1* directly influences the magnitude of the result *?r* in a *dif* predicate. These definitions and the general qualitative inference rules described above are similar to elements of Forbus' Qualitative Process Theory [Forbus84].

```
(rule   :form
        (Q+ ?r ?q1)
        :ants
        (qrelation (dif ?q1 ?q2 ?r))
```

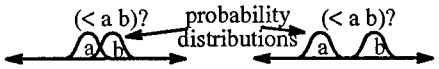
approximation definition rules

Data–approximate quantities have properties which can be expressed in a qualitative manner as discussed earlier in section 2.

qualitative probability rules

These rules relate the probabilities of success of predicates in a way similar to the general qualitative inference rules. Proportionalities can be declared between the probabilities of success of certain pairs of predicates as well as between the probability of success of a predicate and the magnitude of a quantity. Using these proportionalities, goals of achieving increases or decreases in probabilities of success can be achieved. For example, the probability of success of the antecedent to a rule is declared to have a positive influence on the probability of success of the consequent of a rule.

In order to see how the qualitative tuning explanation is constructed using these rules, it is important to understand how qualitative probabilities of success are related to tunable quantities. Quantitative predicates employed by the system have one of two basic intents. Either they are *calculation predicates*, whose purpose is to compute some value (e.g. the *dif* function discussed earlier), or they are *test predicates*, which are designed to fail for certain sets of inputs (e.g. the *less–than* function). There is no way to vary the probability of success of a calculation predicate since they always succeed. A test predicate's probability of success, is sensitive to the probability distribution of its argument quantities. In the diagram below, the less–than test on the right has a higher probability of



succeeding given the illustrated probability distributions for its arguments. While probability distributions are difficult to define and work with, recall the simpler qualitative view of the probability distribution defined for data approximations in section 2: probability density decreases as one moves either higher or lower away from the central value. The general definition for a data approximation embodies this principle. The measured quantity is taken to be the central value. Some distribution is present because of the uncertainty involved. Without knowing any details about the distribution, the definition for a data approximation states that the probability of encountering the actual value for the measured quantity decreases as we get farther from the measured approximate value. One of the approximation definition rules regarding data approximate quantities is shown below




```
(rule :form
      (PQ– (< ?test ?loc) ?test)
      :ants
      (data–approx–quantity ?loc2)
      (IQ+ ?loc ?loc2))
```

This translates to: *if a less–than is being performed between ?test and a quantity ?loc which is indirectly or directly qualitatively proportional to a data approximate quantity, the probability of the less–than succeeding is inversely proportional to the magnitude of the ?test quantity.*

Rules like this effectively translate goals to increase the probability of success of a predicate into goals to increase or decrease quantities.

In general, an explanation for positively influencing the probability of a predicate proceeds so as to:

1. relate the probability of the failing predicate to that of a test predicate involving data–approximate quantities
2. use the definition of a data approximation to relate the probability of success of a test predicate with the magnitude of a tunable quantity

To guarantee that the probability of the failing predicate will increase, all the test predicates in the rule antecedents must be examined. At least one must show an increasing probability of success and the others must be non–decreasing.

The tuning explanation, once generated, indicates only which parameters to tune and in which direction. To carry out the tuning as prescribed by the qualitative tuning explanation, new constraints are imposed on the values of the indicated parameters. Figure 11 illustrates several possible scenarios when constraints have been imposed on a rule parameter. If

the value at which the failure was suggested was originally generated from one of the constraints or bounds on the parameter, the same general predicate expression is used for calculating it but the type of constraint is changed as necessary. When constraints need to be posted between sets of existing constraints, a new general expression is created using the general expressions for the two surrounding constraints and using the ratio between their specific values in the context of the failure. Once the new constraint has been added (or the old constraint changed) the quality function is re–computed and for control parameters, the corresponding rules revised to reflect the new quality function.

With constraint parameters, another decision also must be made before tuning. When the qualitative tuning explanation indicates that the tunable quantity related to a constraint parameter is the target for tuning, it is possible that the current constraint rule had no say in the choice that failed. This is because constraint rules are combined using weights. If the quality function of the current constraint parameter did give the selected value a negative rating, the associated weight should be tuned instead of the constraint itself. This serves to increase the relative importance of a constraint which is already tuned correctly. Since weights are scaled in the range 0 to 1, this amounts to either tuning the indicated weight to be increased from the current value or equivalently tuning all the weights for the other constraints employed in the rating function to be decreased from their current values if the indicated weight is already set to 1.

Next, we introduce the robotic grasping domain which serves as the first testbed for the approach.

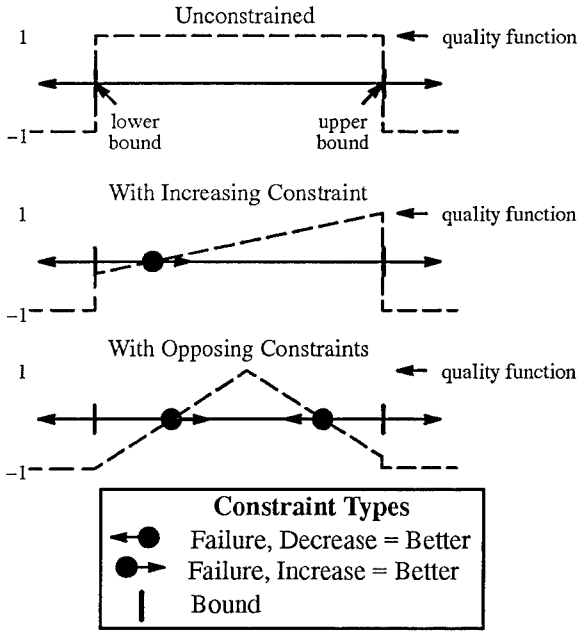


Figure 11. Three Possible Constraint Diagrams Show Constraints on a Rule Parameter

5. An Example in the Robotic Grasping Domain

We are currently using a robotic grasping domain to test our approach. Figure 12 shows the laboratory setup. The current

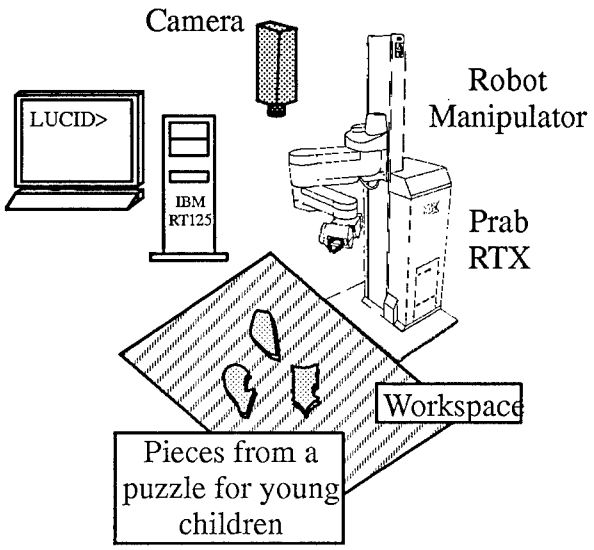


Figure 12. Laboratory Setup

implementation of the architecture is called GRASPER and is written in Common Lisp running on an IBM RT125. GRASPER is interfaced with a frame grabber connected to a camera mounted over the workspace. The camera produces bitmaps from which object contours are extracted by the system. The system also controls an RTX scara–type robotic manipulator. The RTX has encoders on all of its joint motors and the capability to control many parameters of the motor controllers including motor current. This gives the system a rudimentary capability of detecting collisions with the RTX gripper. If enough current (force) is applied to the motor to overcome friction of the joint and the position encoder indicates no movement, an obstacle must have been encountered. This type of sensing gives feedback during execution of a plan when the camera's view of the workspace would otherwise be obscured. This precise control of the manipulator is ideal for carrying out monitored actions in the world.

For the robotic grasping task, we are using plastic pieces from puzzles designed for young children. These pieces have interesting shapes and are large enough, yet challenging, to grasp. The goal is to demonstrate improving performance at the grasping task over time in response to failures. Some of the failures the current implementation learns to overcome, when using isolated grasp targets, include learning to open wider to avoid stubbing the fingers on an objects, and learning to prefer more parallel grasping faces to prevent unstable grasps.

Initially, the system uses the camera to acquire contour information about objects in the workspace. These contours are then approximated with n–gons (internal data approxima-

```
(MONITOR   (MOVE-GRIPPER ?GRIPPER32878 CLOSE 20 64 20 POSITION)
           (AND   (FINAL-FORCE GRIPPER ?GFFORCE32731)
                  (> ?GFFORCE32731 50)
                  (FINAL-POSITION GRIPPER ?GFPOS32732)
                  (> ?GFPOS32732 1))
           (OR    (AND   (POSITION GRIPPER ?GPOS32733)
                         (PRINT (LIST (QUOTE POSITION) ?GPOS32733))
                         (QUAL= ?GPOS32733 0))
                  (AND   (FORCE GRIPPER ?GFORCE132734)
                         (PRINT (LIST (QUOTE FORCE) ?GFORCE132734))
                         (> ?GFORCE132734 60)))
           NIL
           "Closing gripper for force 60"
           (STABLE-GRASP ?GRIPPER32878 ?OBJECT32870 ((RELATIVE-FACE ?NAME132744 ?X132762
?Y132763 ?REF-ANGLE32779 ?LEN132748) (RELATIVE-FACE ?NAME232749 ?X232764 ?Y232765 ?REF-ANGLE32824
?LEN232753))))
```

← Close Gripper Until Termination Met

Final force should exceed 50 units and final position should be such that the gripper didn't close on itself (otherwise failure).

← Terminate action if the gripper closes on itself or if the force exceeds 60 units

Justification for the expectations is that a stable grasp has been planned by the system.

Figure 14. The Failing Monitored Action

tions) which result in $(n^2-n)/2$ possible unique grasping face pairs. In runs performed here, n was set to 5. The data approximated object representations as well as the current information about the state of the robot manipulator are asserted in the initial situation. The target object is then selected and an explanation is generated for how to achieve a grasp of the target. Figure 13 (automatically generated by the implementation)

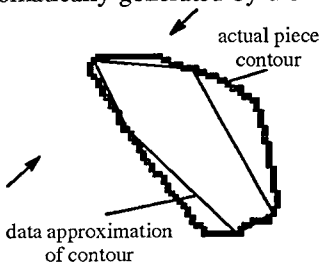


Figure 13. Grasp Target

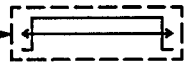
shows the selected target object with the visually sensed contour drawn with a heavy line. The light pentagon is the data approximation for the object. The object approximation employed here involves an algorithm which, in this case, tries to find the best pentagonal representation using extremes on the object contour. The arrows indicate the positions of the leading edges of the fingers for the grasp position given by the produced explanation (keep in mind that the system starts out with a very unconstrained set of rule parameters). This particular grasping operation is taking place after the system has already learned to open as wide as it can to avoid an earlier failure where the finger struck the object while moving downward.[2] It learned this by imposing a constraint on the opening–width rule parameter. The explanation for achieving *grasp–object* involves a total of about 300 nodes with a maximum depth of 10 levels. The approximate rule employed in the explanation for rating potential grasping faces on the angle between them consults the quality function of the contact–angle–constraint parameter. This rule is:

2. For a detailed example illustrating an instance of how the system learned to open wider, see [Bennett90].

rule to return a rating of the quality of the contact angle between two potential grasping faces

```
INTRA-RULE: R205
  FORM:
    (QUALITY-CONTACT-ANGLE-CONSTRAINT ?GRIPPER
                ?CALC-CA ?RETURN)
  ANTS:
    (QUALITY-CHECK CONTACT-ANGLE-CONSTRAINT
                (?GRIPPER ?CALC-CA) ?RETURN)
  CONS:
  APPROX: CONTACT-ANGLE-CONSTRAINT
```

initially flat quality function for the contact–angle–constraint parameter rates all contact angles equally



The *contact–angle–constraint* parameter is initially constrains the angle between grasp faces to be less than the arctangent of the friction coefficient (45 degrees for the approximate friction coefficient of 1 initially assigned here). All angles below this level are rated as equally good. After the explanation was generated, and its associated operator sequence executed, the monitored action, shown in Figure 14, encountered a violation of the expected sensor readings. The original explanation for the *stable–grasp* goal, indicated in the failing monitored action, is now suspect due to the violated expectations. A sketch of the specific explanation is shown in Figure 15. This explanation for why a stable grasp should have been achieved is the starting point for developing the qualitative tuning explanation. The generalized consequents and antecedents of the *stable–grasp* subproof are asserted as qualitative relations. Approximate quantities employed in the subproof are identified and asserted as such. A proof is then constructed for increasing the probability of success of the *stable–grasp* goal. Figure 16 shows the qualitative explanation for how preferring a smaller contact angle positively influences the probability that a stable grasp will be achieved. Table 1 gives the semantics for the predicates employed in the explanation. The topmost left–hand subtree establishes that the probability of the <= test predicate can influence the probability of the *stable–grasp* goal because it is an antecedent of

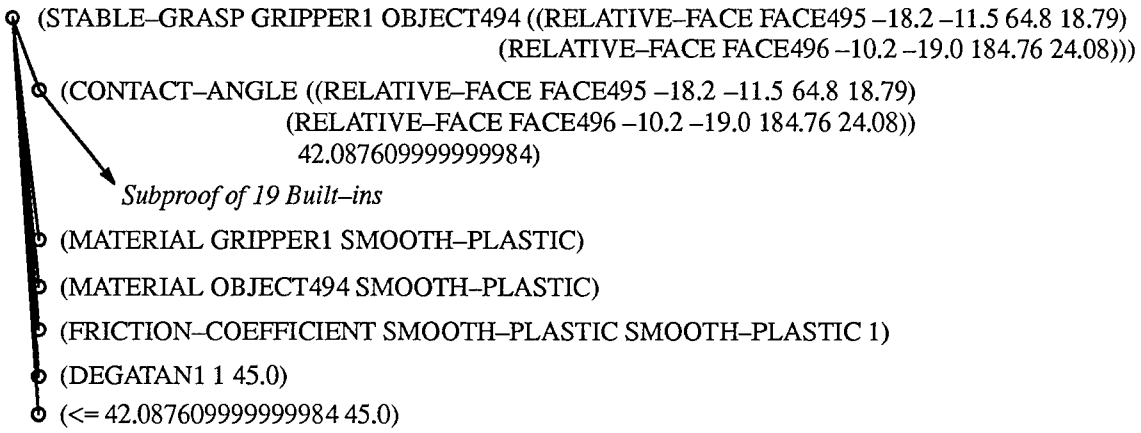


Figure 15. Explanation Specific to Failure

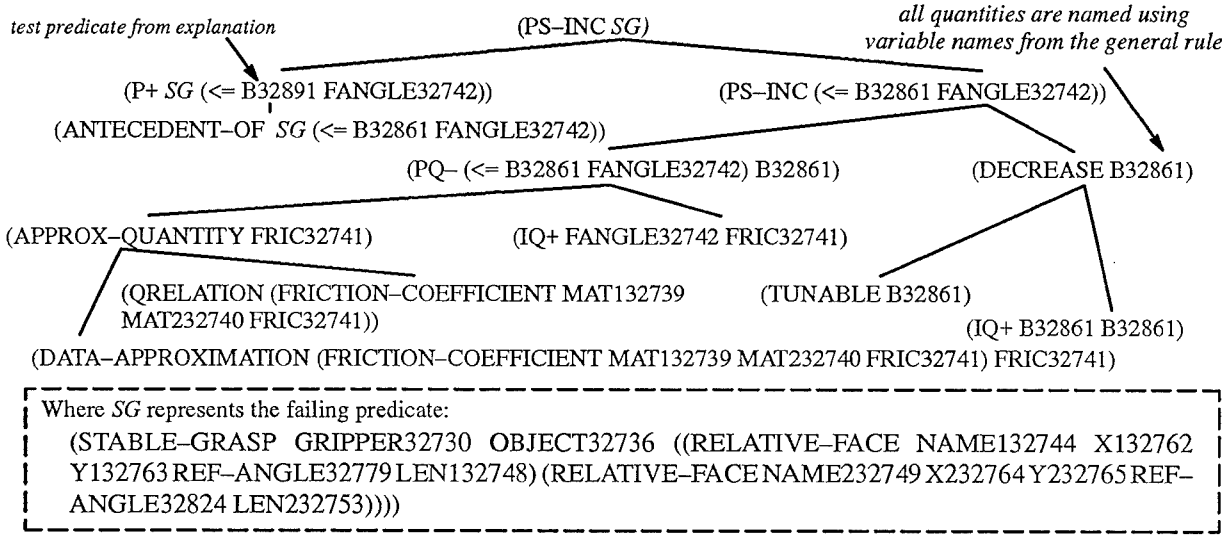


Figure 16. A Qualitative Tuning Explanation

Table 1. Predicates Employed in the Tuning Explanation of Figure 16

(PS–INC ?pred)
the probability of success of ?pred is influenced positively
(P+ ?pred1 ?pred2)
the probability of success of ?pred2 influences the probability of success of ?pred1 positively
(ANTECEDENT–OF ?pred1 ?pred2)
?pred2 is an antecedent of ?pred1 in the rule being analyzed
(PQ+ ?pred ?quant)
the magnitude of the quantity ?quant influences the probability of success of ?pred positively
(INCREASE ?quant)
the magnitude of the quantity ?quant is influenced positively
(APPROX–QUANTITY ?quant)
?qant is an approximate quantity from a data approximation (not controllable by the system)
(IQ+ ?q1 ?q2)
the magnitude of quantity ?q2 indirectly influences the magnitude of quantity ?q1 positively
(TUNABLE ?quant)
the magnitude of quantity ?quant is a tunable rule parameter

the rule. The right–hand subtree establishes that the probability of the <= can be positively influenced through a decrease in the contact angle between faces. The IQ+ predicate is a built–in predicate for establishing transitive relations between quantities. It consults the body of qualitative proportionalities which hold in the current situation

The qualitative tuning explanation indicates that a smaller contact angle should be preferred in choosing grasping faces. Figure 17 illustrates the contact–angle–constraint parameter's constraint diagram before (top) and after (bottom) tuning has occurred. After tuning, the associated rule rates contact faces using the new quality function for the contact–angle–constraint parameter and chooses a grasp position such that the two faces to be contacted are closest to parallel.

Figure 18 shows results from the first test of our current version of the parameter tuning algorithm. All 12 pieces from one puzzle were used in the experiment. Piece orientations and the order of presentation were chosen at random. Grasping targets were presented in isolation from other pieces. On the left is performance without parameter tuning on failure. Finger stubbing failures occurred on most of the pieces because the gripper only opened as wide as the model of the piece dictated.

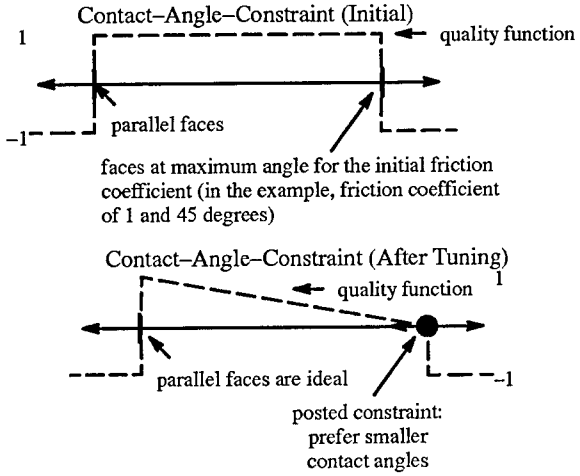


Figure 17. The Contact–Angle–Constraint Approximation Before and After Tuning

Therefore, various errors in positioning of the piece and gripper as well as the approximation of the piece shape contributed to failures of this variety. Two of the twelve pieces presented were successfully grasped. One grasp failed because the piece slipped away when the gripper was closed, indicating that a stable equilibrium grasp was not chosen. The results when parameter tuning was employed were much better. Since the pieces were presented in isolation, after the system learned to open wider, that was sufficient to prevent stubbing fingers on the object in all of the remaining trials. Vertical slipping failures did occur on the second and third trials, however. The system doesn't have a complete model of the piece beyond its contour and has insufficient knowledge to explain a failure in the vertical dimension. These failures occur when the piece pops up when the gripper squeezes because either of tapering of the sides or flex of long narrow sides of the piece. We plan to add a more complete model of the piece, using model–based vision, to permit these types of failures to be explained. In the fifth trial, the piece slips out of the gripper fingers during closing and the system learns to prefer the most parallel grasping faces it can find. This strategy succeeded on the rest of the pieces in trials seven through twelve.

6. Future Directions

For a planning system to able to function effectively in a domain where uncertainties arise, it is important that it be able to generate uncertainty–tolerant plans. It is also important that in complex domains it be possible for the planner to be able to generate these tractably. Our architecture for planning with approximations addresses both of these important problems. The most important areas we are now addressing involve the development of a theory of when parameter tuning, in general, is worthwhile, when tuning a particular parameter can be detrimental, and studying the performance and generality of our approach.

One important theoretical problem to be investigated involves when tuning should take place. Tuning a parameter for one particular observed failure, despite the rarity of that type of failure, may unnecessarily compromise performance on later encountered, but much more likely, failures. There are many desirable qualities for a real–world plan. A system performs better with plans which are uncertainty tolerant, whose preconditions are efficient to check, whose actions are efficient to carry out, and several other factors. These are all part of a real–world plan's *operationality* [Bennett89]. The current tuning algorithm assumes all failures are worth having the parameters tuned, consequently affecting the revised plan's operationality. A more intelligent algorithm would tune a parameter only if it believed there was a high likelihood of the tuned parameter preventing other failures in the future. This would prevent possible adverse effects on plan operationality

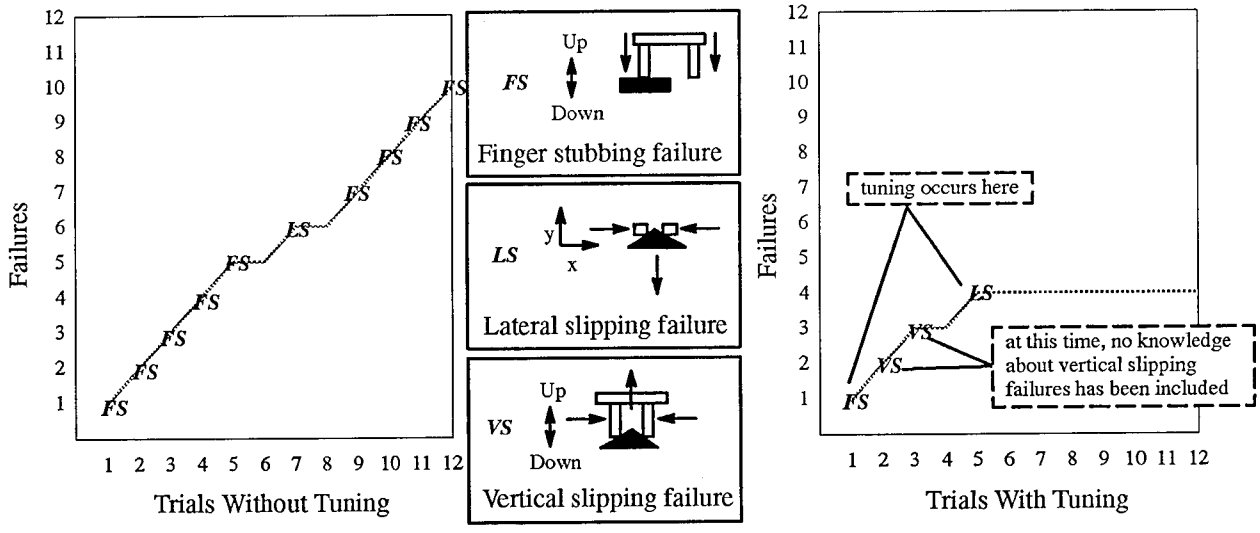


Figure 18. Comparison of Tuning to Non–tuning in Grasping the Pieces of a Puzzle

if the improved payoff in error tolerance was minimal. There are several possible approaches to recognizing the significance of a gain in error tolerance. One approach is strictly empirical whereby data is obtained on the likelihood of different types of tasks and situations in the domain. If a failure is encountered with a task and situation of low likelihood, the system would decide if it was below some threshold and simply not perform the tuning. Another interesting possibility is to have a weak theory of the types of errors likely to manifest themselves in the domain. Furthermore, some types of errors are more likely to be specific to a certain object or type of object than all the objects as a whole. Such a theory could represent this as well and help to explain when the failure was significant enough to trigger tuning of the parameters.

Another important problem we are working on involves the possibility of *overtuning* one particular parameter. Even if the observed failure is determined to be worthwhile tuning, because it presumably is a common failure type, there may be an adverse effect in tuning the parameter which must be tuned to prevent that failure. This is because that parameter may have already been tuned to prevent a previous failure in such a way which opposes the tuning suggested by the current failure. This is either evidence that another parameter would be a better one to tune or that there is something different in this world situation which has been ignored by the system. That is, that the parameter is really being used in a different context now. If such a situation is recognized, the goal should be to distinguish this situation from the one which lead to the previous tuning. Then, the one parameter would be split into two, one for each context. The first would retain the previous learned tuning and the second would include the proper tuning for the failure just discovered. The effect is to split what was previously considered by the system to be one context into two separate contexts. We refer to this process as *context–splitting*.

7. Acknowledgments

I would like to thank my advisor, Gerald DeJong, particularly for recent discussions on characterizing data approximations. Seth Hutchinson was also very helpful in discussions on planning for uncertainties in robotics. This research was supported by the Office of Naval Research under grant N–00014–86–K–0309.

References

[Bennett89] S. W. Bennett, "Learning Uncertainty Tolerant Plans Through Approximation in Complex Domains," M.S. Thesis, ECE, University of Illinois, Urbana, Il., January 1989. (Also appears as Technical Report UILU–ENG–89–2204, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana–Champaign)

[Bennett90] S. W. Bennett, "Reducing Real–world Failures of Approximate Explanation–based Rules," *To Appear in the Proceedings of the Seventh International Conference on Machine Learning*, Austin, TX, June 1990.

[Brooks82] R. A. Brooks, "Symbolic Error Analysis and Robot Planning," Memo 685, MIT AI Lab, Cambridge, MA, September 1982.

[DeJong86] G. F. DeJong and R. J. Mooney, "Explanation–Based Learning: An Alternative View," *Machine Learning 1*, 2 (April 1986), pp. 145–176. (Also appears as Technical Report UILU–ENG–86–2208, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana–Champaign.)

[Donald90] B. R. Donald, "Planning Multi–Step Error Detection and Recovery Strategies," *International Journal of Robotics Research 9*, 1 (February 1990), pp. 3–60.

[Forbus84] K. D. Forbus, "Qualitative Process Theory," *Artificial Intelligence 24*, (1984), pp. 85–168.

[Hutchinson90] S. A. Hutchinson and A. C. Kak, "Spar: A Planner That Satisfies Operational and Geometric Goals in Uncertain Environments," *Artificial Intelligence Magazine 11*, 1 (1990), pp. 30–61.

[Lozano–Perez84] T. Lozano–Perez, M. T. Mason and R. H. Taylor, "Automatic Synthesis of Fine–Motion Strategies for Robots," *International Journal of Robotics Research 3*, 1 (Spring 1984), pp. 3–24.

[Mitchell86] T. M. Mitchell, R. Keller and S. Kedar–Cabelli, "Explanation–Based Generalization: A Unifying View," *Machine Learning 1*, 1 (January 1986), pp. 47–80.

[Mooney86] R. J. Mooney and S. W. Bennett, "A Domain Independent Explanation–Based Generalizer," *Proceedings of the National Conference on Artificial Intelligence*, Philadelphia, PA, August 1986, pp. 551–555. (Also appears as Technical Report UILU–ENG–86–2216, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana–Champaign.)

[Schank82] R. C. Schank, *Dynamic Memory*, Cambridge University Press, Cambridge, England, 1982.

Explanation–Based Control: An Approach to Reactive Planning in Continuous Domains*

Gerald DeJong
Computer Science Department and the Beckman Institute
University of Illinois
Urbana IL 61801

ABSTRACT

This research grew from frustrated attempts to design an AI planning system to solve a number of every–day real–world planning problems. Beneath these failures lurked a concern that current symbolic (i.e., non–connectionist) AI planning approaches could never lead to adequate solutions. Applying current techniques to continuous domain problems feels a bit like trying to change a light bulb with a screwdriver; the tool, though fine for other applications, seem singularly inconvenient for this one. Neurally–inspiried approaches face equally daunting problems. Primary among these are the great size of the network with the concomitant high training costs (both in time and number of training instances required), and their rather limited ability to benefit from existing background knowledge, especially background knowledge of a symbolic form.

Instead, the approach is inspired by Control Theory, which has a long and successful history of dealing with continuous real–world problems. The approach can be seen either as extending AI planning with plausible inference and a control theory–inspired ontology, or as providing an automated AI solution to the *identification* problem [Truxal61] of control theory resulting in a kind of intelligent adaptive control. The research reported here is not proposed as a general substitute for current AI planning approaches. Rather, it suggests a new direction by which symbolic AI may extend its war chest of planning techniques.

Planning in continuous domains presents difficult problems for conventional planning approaches. Not the least of these is the need for reactivity. This paper investigates a direction in planning research that takes its inspiration from control theory. There are no "operators" as such with effects and preconditions that transform world states. The approach involves *plausible* explanation–based learning using a background domain theory of qualitative descriptions. Direct experience with the world refines and calibrates the generalized plausible explanations. The approach offers a natural integration of qualitative with quantitative reasoning and also explanation–based learning with empirical learning to acquire effective strategies for achieving goals in continuous domains.

The research reported in this paper was supported by the Office of Naval Research under grant number N0001486–K–0309

1. Introduction

Planning in AI has traditionally been treated through some kind of calculus for articulated actions. A plan is viewed as a set of actions that transform a given initial world state into a world state satisfying some goal. Thus, a planning problem is a triple of an initial state, a goal relation, and a set of operators. The process of planning consists in discovering a set of operators together with sufficient constraints to guarantee that the goal is satisfied in the final world state [Chapman87, Genesereth87].

In one guise or another this view is nearly universal in conventional AI approaches to planning. The underlying notion of actions that are executed to alter the world has pervaded work in classical planning [Fikes71, Laird86, Wilkins88], as well as reactive planning [Agre87, Firby87, Gervasio89, Schoppers87], opportunistic planning [Hammond88], incremental planning [Chien89], and multi–agent planning [Georgeff86].

Traditional operators are characterized by their preconditions and effects (which may be conditional e.g., [Pednault88]). It is understood that an operator may be applied to any world state in which its preconditions are satisfied, resulting in a different world state. Often this transition is modeled as timeless or instantaneous [Fikes71, Wilkins84] although increasingly it has been found that temporal reasoning is needed to adequately describe the world [Allen83, Dean83, McDermott82, Shoham86a, Vere83]. For some domains even this sort of temporal reasoning does not go far enough. In particular it is insufficient for *continuous domains*. An exception is the work of [Sandewall89] and [Dean90]. Their approach involves including integral and differential calculus constructs in an otherwise–conventional temporal reasoning system.

Nearly all domains have some continuous attributes. Such attributes, or at least their continuity, can often be

neglected. Unfortunately, forcing the continuous facets of a domain into a discrete action mold can result in spurious complications. These often surface in the guise of the frame problem [McCarthy69]

The problem is in large part due to what we will term the *discrete action* assumption which states that changes in the world are due to temporally bounded applications of operator instantiations (actions). While often adequate, the view of changing the world through actions can introduce unnecessary (and sometimes insurmountable) obstacles.

To illustrate, consider the following scenario: An airplane is on the correct glide slope to land but with an airspeed that is 10 knots too fast.

> The pilot gently begins to close the throttle. A moment later he eases back on the control yoke (the pilot's steering wheel) deflecting the elevator up and increases the plane's angle of attack. He stops closing the throttle and then holds the yoke steady once again. The plane slows while maintaining its current glide slope.

In this example gentle and continuously coordinated changes in throttle and elevator controls are essential. Closing the throttle slows the plane but also reduces the amount of lift that the wings are pruducing. Alone, this would steepen the glide slope. The undesirable side effect is counteracted by pulling back on the yoke which increases the angle of attack and produces more lift. These actions cannot be modeled as instantaneous. A temporal reasoning approach can help some but at a rather substantial cost. A temporal system would correctly recognize that the two intervals defined (one for closing the throttle and one for moving the elevator) in fact result in three important intervals: 1) throttle changing alone, 2) both throttle and elevator changing, 3) elevator changing alone. However, temporal reasoning does not ameliorate the cost of managing what happens during the second interval in which both the throttle and elevator controls change. A separate operator must exist that specifies how the combination of these two actions affects the world. Compound effects cannot in general be deduced from the effects of the simple constituent operators. The domain theory implementor must have anticipated the need for the combination operator. Indeed, a separate combination operator is needed for all such possible composite actions. In worst case this results in an exponential increase in the number of operators to be defined (the power set of the primitive operators). The planning problem with all of the combination operators is much more difficult since many more operators will appear to be relevant to accomplishing any given goal.

Even if all relevant combination operators could be anticipated and defined, planning is problematic. Many of the parameters (e.g., current engine oil viscosity) are difficult or impossible to know. Thus, an individual operator's effect may not be precisely predicted, nor can its preconditions be guaranteed. This greatly complicates planning. Somehow, people are able to successfully find solutions in these kinds of situations. Given a little practice, they have no trouble coordinating actions; they effortlessly ignore irrelevant parameters; they appear to assume reasonable values for unknowable parameters while implicitly relegating their planning behavior to those situations in which their approximations hold.

Control theory offers an intriguing alternative model for world change. Instead of operators and effects, the world is seen as an on–going process to be controled through the manipulation of certain inputs. The world imposes inter–relatioinships among the values of a collection of real–valued quantities. The planning problem, in this light, is seen as a strategy for manipulating input quantities in such a way as to bring about desired changes in other world quantites. It is this approach to world change that we adopt.

2. A Representation for Continuous Changes

The primary techniques used to wed artificial intelligence and control theory are qualitative reasoning and explanation–based learning. Qualitative representations are used to mediate between numerical values and the symbolic domain knowledge of how they relate. Plausible explanation–based learning is used to automatically conjecture control analyses from the observation of an expert's behavior and later to refine these control strategies when deficiencies are observed in the course of exercising the planning system.

The continuous aspects of the world are called *quantities*. (The term has a similar meaning in both qualitative reasoning [Forbus84] and control theory [Kuo87]. Examples of quantities are the position of the aircraft's throttle, the fuel flow through the intake manifold, the wing's angle of attack, and the speed of the plane. Quantities a) take on real values, b) are continuous, and c) may have limited extreme values. In general many world quantities are changing simultaneously. Some quantities can be directly manipulated by the planner (for example, the aircraft's throttle position or the setting of a radio's volume control). We call these quantities *controllable parametric quantities* (or *controllable parameters*). The

corresponding control theory term is *input variables* (which has a rather different meaning in computer science.) Thus, we use the new more descriptive term. As the controllable parameters are varied, values of other world quantities react in accordance with the laws of nature. Quantities which can only be indirectly manipulated are called *internal quantities*. Examples of these are the speed of the aircraft, the wing's angle of attack, the descent rate, etc. In control theory these values are usually encoded in the state of the system. One other kind of quantity is termed a *non–controllable parametric quantity*. These are quantities which, like controllable parameters, determine the values of internal quantities, but, unlike controllable parameters, cannot be manipulated by the planner. An example of a non–controllable parametric quantity in the aircraft domain is the air density. The density of the air surrounding the aircraft cannot be manipulated either directly or indirectly, but it significantly alters the aircraft's flying characteristics.

Continuous world changes are more appropriately represented as a graph of simultaneous quantity values rather than as a sequence of discrete world states. We call these graphs *quantity profiles*. They are similar in spirit to event shape diagrams [Borchardt84, Waltz82]. Example quantity profiles for a simplifies automobild domain are shown in figures 1, 2, and 3.

The planning problem, viewed in this way, is as follows: given an initial state of the world (initial values of quantities), goal values for some *internal* quantities, and knowledge about how quantities interact, find a consistent profile of changes of the *controllable parameters* which when combined with the values of the non–controllable parameters brings about the desired internal quantity changes. The aircraft example above falls into this mold: given the initial positions of the aircraft controls, the heading, the altitude, etc. of the aircraft, find a manipulation of the controls that (for the current air temperature, air density, wind conditions, etc.) reduces the speed by 10 knots while preserving the current angle of descent.

3. *The Approach*

The approach we take is to rely on Explanation–Based Learning (EBL) over a *plausible* and *qualitative* domain theory to learn about and exploit domain characteristics. Learning produces new planning constructs which are the continuous analog of EBL–acquired schemata [Mooney88, Segre87, Shavlik88] (generalizations of macro–operators).

Briefly, the learning algorithm involves 1) observing an expert who solves a problem currently beyond the system's capabilities, 2) constructing a plausible qualitative explanation for why the expert's actions result in the desired effect, 3) generalizing the explanation in standard EGGS fashion [Mooney86], and 4) fitting observed quantitative points to the resulting general qualitative concept. The new concept can be used efficiently in planning the achievement of similar future goals.

More formally, at any instant in time, the current state of the continuous world is given by a point in **N**–dimensional space, one dimension for each of **N** quantities in the world. As the world changes, the values of the quantities change in a continuous fashion, and so the point traces out a continuous trajectory in **N**–space. The trajectory is constrained to lie on a (possibly rather complicated) hyper–surface which characterizes the laws of Nature. Suppose there are **P** parameters. We can more conveniently characterize the constraints inherent in the complicated **N**–space world hyper–surface as **N** – **P** different hyper–surfaces each in **P**+1 dimensional space where each of these latter hyper–surface characterizes how a single one of the **N** – **P** internal quantities depends on the **P** parameters. The relationship of the internal quantity to the parametric quantities is functional. That is, for each combination of values assigned to the parameters there is at most one value that each internal quantity can have. This derives from the assumption that the world is deterministic. Thus, each internal quantity is precisely characterized by a **P** dimensional functional surface in a **P**+1 dimensional space.

In general, a planning goal is the specification of a profile of continuous coordinated values for a subset of the world's internal quantities over some time interval. A plan is a profile over a time interval of continuous coordinated values for a subset of the world's controlable parametric quantities. It is seldom necessary or desirable to generate a full plan prior to execution. Rather it is better to make the final decisions about contolable parameter adjustments during execution. This reduces the need to accurately predict the values that relevant non–controlable parameters will have during execution. Instead, currently observed values of the non–controlable parameters can be used. The system's reactivity derives from this use of execution–time monitored values of the world in selecting what to do next.

Before discussing planning in this formalism, consider the characterization of a particular value for a chosen internal quantity. Such a value corresponds to the intersection in **P**+1 dimensional space (with **P**

dimensions for the parameters and one for the internal quantity) of the chosen function and a **P**–dimensional hyperplane intercepting the internal quantity axis at the desired value and parallel to each of the parameter coordinate axes. If the goal hyperplane does not intersect the function, then the internal quantity cannot take on that value. If there is an intersection, it will, in general, be a **P**–1 dimensional surface capturing the constraints imposed by both the hyperplane and the function. The goal hyperplane constrains the internal quantity to the desired value; the surface enforces consistency between the internal quantity value and the parameter values. In a world with more than one parameter, the "solution" can be underdetermined; any point on the intersection suffices. Since the number of parameters can be very large indeed, the solution may be extremely underdetermined.

Now consider the problem of achieving a particular value for a single internal quantity. Further, suppose we do not care what intermediate values the internal quantity takes on. The current state of the world is fully specified and specifies a point on the surface of the selected internal quantity's function (corresponding to the current values for all **P** parameters.) The goal, if achievable, is the **P**–1 dimensional contour described above. Any countour along the function's surface that connects the current world point to the goal contour is a candidate plan.

Mathematically, the comitment to solve a simple problem of the above form (achieving a particular value for an internal quantity) adds a constraint to the underdetermined system which specifies the world's possible futures. The resulting system will in general still be (grossly) underdetermined. As the goal is made more complex (e.g., by requiring intermediate values for the internal quantity or by insisting on the coordinated behavior of other internal quantities), additional constriaints are imposed on the system.

After taking into account all of the goal's constraints, any remaining underdeterminism can be resolved arbitrarily. One obvious strategy is to select, at each time point, changes to the controlable parameters within the remaining constraints that give the greatest decrease in the distances (measured along the various internal quantity functions) to the goal.

In fact, underdetermination is a great advantage for reactive planning. For if the system were determined, there would be only one future that achieves the goal. This means values for the non–controlable parameters have been perscribed. If it happens that Nature chooses other values for the non–controlables the planned solution fails. As long as the system is underdetermined there is hope for effectively reacting to unanticipatable changes in the values of non–controlable parameters.

3.1. *Plausible Qualitative Explanations*

The plausible explanation is constructed from the system's domain theory. The domain theory is loosely based on Forbus' Qualitative Process Theory [Forbus84]. We use the qualitative predicates INCREASING, DECREASING, and CONSTANT which denote a relation between a quantity and a time interval. They are true in just those cases in which the value of the quantity is monotonically increasing, monotonically decreasing, or constant respectively over the entire time interval. The qualitative predicates GREATER–THAN, LESS–THAN, and EQUAL are relations between two quantities and a time interval. They are true if the value of the first quantity has the mathematical relation >, <, = respectively to the value of the second quantity over the entire specified interval. Finally, and most interestingly are the two qualitative proportionality predicates Q+ and Q–. The names and inspiration for these are from Qualitative Process Theory, but the semantics of Q+ and Q– are somewhat different here.* In Qualitative Process Theory Q+ and Q– have meaning only after the world of qualitative relations has been closed by assumption. That is, drawing and inference using any is contingent on knowning all fo them. We are making no such assumption. (Q+ *q1 q2 i*) is to be thought of as the conjunction of:

(INCREASING *q2 i*) ⇒ (INCREASING *q1 i*) and
(DECREASING *q2 i*) ⇒ (DECREASING *q1 i*)

while (Q– *q1 q2 i*) is short for

(INCREASING *q2 i*) ⇒ (DECREASING *q1 i*) and
(DECREASING *q2 i*) ⇒ (INCREASING *q1 i*)

The symbol "⇒" denotes *plausible* (not logical) implication. Its new semantics raises many issues that are only beginning to become clear but in any case cannot be treated here. Think of the consequent of a plausible implication as not logically entailed, but only plausibly supported. Domain knowledge is coded in the form of *processes*. Each process has preconditions and a body. The body specifies a set of qualitative proportionalities among quantities. Over intervals in which the preconditions of a process are met, the process is active and its plausible qualitative proportionalities are available for explanation construction. Multiple processes may be active at once. An explanation for an internal quantity specifies a set of qualitative

proportionalities from the active processes which justify the observed qualitative behavior of that *internal* quantity in terms of observed qualitative behavior of *parameters*. There may be many different explanations possible for an observed behavior.

In the example below there are two processes. One, **ENGINE–RUNNING**, has as a precondition that the engine be running, which may be inferred if there is gas in the tank, ignition switch on, etc. It specifies plausible knowledge such as that the engine **REV**'s are positively qualitatively proportional to the gas pedal position. The second process, **CAR–MOVING**, is when the car's speed is greater than zero. It specifies things like how the brakes work (we assume no power assist in braking in our simple automobile) and what happens going up or down hills. If we observe the expert's car slowing down while the expert simultaneously lets up on the gas, depresses the brake, rolls up the window, and coasts up a hill, the system may plausibly attribute the slow down to the hill, or the gas, or the brake, or any combination of them. There is no plausible inference chain linking rolling up the window to the slow down. If the expert had killed the engine (say he turned off the ignition) prior to slowing, the gas pedal manipulation would no longer be included in a plausible explanation since some crucial proportionalities are derived from the **ENGINE–RUNNING** process which is not active.

Explanations are generated in a most–plausible–first ordering. The *a priori* plausibility of of an explanation is derived from the plausibility ratings of the component qualitative proportionalities. In the current implementation, all qualitative proportionalities are considered equally plausible and "most–plausible" becomes the same as "simplest". Given an observation of an expert's planning, such as in figure 1, the system constructs the simplest explanation that accounts for the qualitative behavior of the automobile's speed. This explanation is generalized and used for planning.

Of course, the first explanation may not be the correct explanation. If it is incorrect or incomplete, the solution it proposes to solve some later planning problem may fail. At that time the explanation is refined to be consistent with both the old and new observations. It is generalized into a replacement planning concept.

This refinement–on–demand of the planning concepts means that the system cannot guarantee that its plan solves the problem it is given. Any solution may be a failure signaling the need for further refinement. Indeed, there would seem to be the possibility that refinement will continue indefinitely with the planner never approaching any level of competence. It can be shown that for a problem distribution, the learning algorithm converges although to a set of adequate (not necessarily correct) planning concepts.

3.2. *Quantifying the Concept*

A purely qualitative representation is not easily used to solve the planning problem. (Although see [Hogge87] for an interesting but computational intensive approach.) A planner must produce a solution qualitatively precise enough to execute in the world.

The quantitative planning concept is an approximation to the internal quantity function Nature uses in the world to compute the quantity's value from the parameter values. It is approximated by numerical interpolation from observed points.

The qualitative explanation provides two extremely important constraints on the quantitiative interpolation function. It A) identifies the parameters that are relevant to achieving the specified goal and B) assures a strong constraint on how the internal quantity value may change with parameter changes: the internal quantity's function must be monotonic in each of the identified parameters. The monotonicity is computed through transitivity of qualitative proportionalities (Q+ and Q–) in the explanation.

Provided the qualitative explanation is correct, the system has identified all of the relevant arguments to the internal quantity function and also determined the function's asymptotic behavior. The original observation of the expert provides a number of quantitative sample points. About 50 such samples make up figure 1. Each sampled point contains numerical values of all the quantities at each instant. Values for the internal quantity of interest together with just the relevant parameters (as identified by the qualitative explanation) are extracted from each observed point. These tuples must fall on the function's surface and they must obey the function's asymptotic constraints. If any does not, the conjectured qualitative explanation is not adequate to describe the planning situation. The original explanation is rejected and the next–most–plausible explanation consistent with the new observation is substituted.

At every planning instant the actual achieved value (from the finite difference simulator) is compared to the value predicted from the approximated surface. The surface is refined if the difference is beyond a pre–specified noise level ε. This means that within trial learning occurs and, more importantly, that the remainder of the plan is

constructed reactively from the *actual* value rather than the *predicted* value of the internal quantity.

The current implementation employs piece–wise linear interpolation among points. Observed data points that are within ε of the interpolation surface do not contribute.

4. *An Example*

The domain that will concern us in the remainder of the paper (one of three that the implemented system has been tested on) is planning to achieve different speeds in a simplified single–gear manual transmission automobile. The "real world" automobile is, in fact, a finite difference numerical model whose difference equations directly use the values of controllable parametric quantities.

The planner can directly manipulate the setting of the controllable parameters (gas, clutch, brake, window position, air conditioner setting, etc.). However, the settings cannot be changed instantaneously. There is a small maximum rate of change of the controllable parameters specified to the system. Any rate of change (positive or negative) up to that amount may be selected at any time point. The automobile can be "driven" by the system or an expert using keyboard and mouse input. Both are subject to the same maximum–rate–of–change constraints.

The system is implemented in LUCID CommonLisp on an IBM RT. There is but a single explanation interval so the time interval arguments are omitted for simplicity. The background domain theory contains two processes. The **ENGINE–RUNNING** and **CAR–MOVING** processes are:

ENGINE–RUNNING
PRECONDITIONS: (RUNNING ENGINE)
BODY:
(Q+ GAS–FLOW GAS)
(Q+ REVS GAS–FLOW)
(Q+ SPEED REVS)
(Q+ AIR–MOVEMENT FAN)
(Q– AIR–TEMP A/C–SETTING)
(Q+ ENGAGEMENT CLUTCH)
(Q+ SPEED ENGAGEMENT)
(Q– REVS ENGAGEMENT)
(Q+ TEMP SPEED)
(Q+ REVS TEMP)

CAR–MOVING
PRECONDITIONS: (GREATER–THAN SPEED 0)
BODY:
(Q– SPEED BRAKE)
(Q– SPEED GRADE)
(Q– REVS GRADE)
(Q+ AIR–MOVEMENT WINDOW–SETTING)
(Q+ AIR–MOVEMENT SPEED)

GAS, CLUTCH, and BRAKE represent the position of the gas, clutch, and brake pedals respectively. These are controllable parameters. The zero positions of GAS and BRAKE are fully up, the zero position for CLUTCH is fully depressed. ENGAGEMENT is the percentage of power transmitted to the wheels through the clutch. REVS is the rotational speed of the engine. SPEED is the speed of the car. GRADE is the hill gradient, a non–controllable parameter. TEMP is the engine temperature.

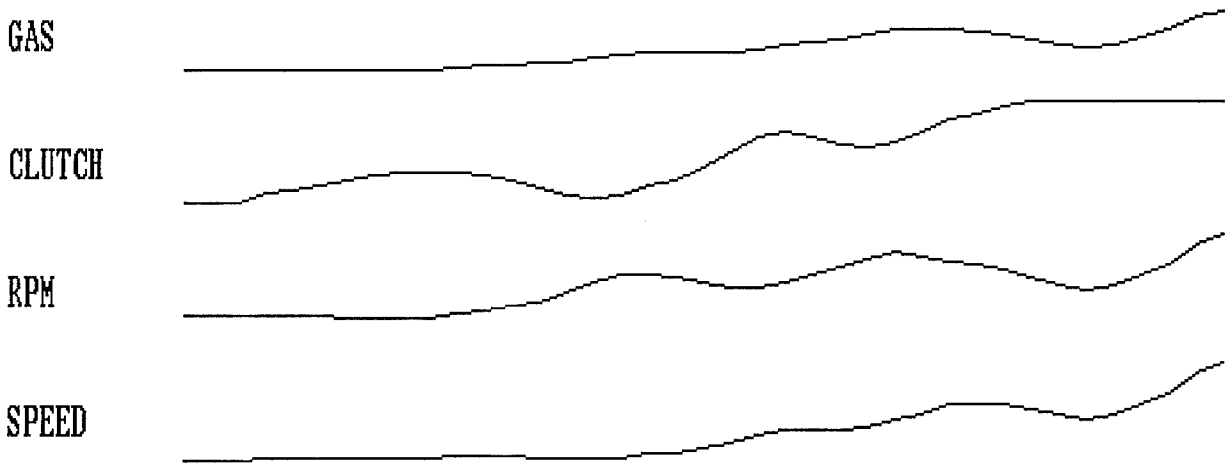


Figure 1: The Expert's Solution

These two processes say that while the engine is running increasing GAS makes REVS go up and SPEED go up; letting out the clutch makes SPEED go up and REVS go down; the engine heats up at higher speeds and allows more efficient combustion (REVS go up), and also a steeper grade causes SPEED and REVS both to go down, while the BRAKE decreases SPEED.

It is important to note that the processes given above are not the only ones that can be used to describe the workings of our automobile. One of the strengths of this approach is that it is relatively insensitive to how the domain knowledge is crafted. In particular one might write a simpler **CAR–RUNNING** process if one does not know about engine RPM's or clutch engagement. On the other hand one could write a much more complicated set of processes specifying qualitative relations among the carburetor butterfly valve position, venturi flow, manifold pressure, vacuum advance, etc. Either of these alternative domain theories would work as well as the one above for our chosen task. The more complex theory would support the acquisition of additional planning concepts that ours does not, just as ours supports concepts that the simpler alternative would not.

The system is given the goal of achieving a speed of 64 MPH from a dead stop with the engine on and idling. It currently has no problem–solving concepts and thus cannot solve the problem alone. It asks for an expert's solution which is shown in figure 1.

The simplest plausible explanation consistent with the expert's solution is:

```
(EXPLANATION (INCREASING SPEED)
    ((Q+ SPEED ENGAGEMENT)
     (Q+ ENGAGEMENT CLUTCH)
     (INCREASING CLUTCH)))
```

This explanation conjectures that the speed is increasing because the speed is qualitatively positively proportional to the clutch engagement which is qualitatively positively proportional to the clutch pedal position which is observed to be increasing.

A 2 dimensional linear quantitative interpolation function is created and the observed numerical values for SPEED, and CLUTCH are asserted. Not all observed points are recorded in the interpolation function. If a point is already correctly interpolable by existing points, it is not used. The interpolation function contains 5 points.

Next another acceleration problem is given to the system. It is to accelerate from 0 to 20 MPH. The system selects the newly constructed planning concept, but the interpolation surface cannot accept the third new data point without violating the qualitative montonicity constraints.

The explanation is, in fact, not adequate. An observation during planning is inconsistent with the qualitative explanation. The system searches for the next most plausible explanation of the data that is not contradicted by the new point. The following plausible explanation is constructed:

```
(EXPLANATION (INCREASING SPEED)
    ((Q+ SPEED REVS)
     (Q+ REVS GAS–FLOW) (Q+
     GAS–FLOW GAS) (INCREASING
     GAS)))
```

In planning with this concept the system also meets with nearlyimmediate failure. Finally, a more adequate qualitative explanation is generated:

```
(EXPLANATION (INCREASING SPEED)
    ((Q+ SPEED REVS)
     (Q+ REVS GAS–FLOW) (Q+
     GAS–FLOW GAS) (INCREASING
     GAS) (Q– REVS ENGAGEMENT) (Q+
     SPEED ENGAGEMENT) (Q+
     ENGAGEMENT CLUTCH)
     (INCREASING CLUTCH)))
```

This explanation indicates that the SPEED is a function of both the GAS and CLUTCH controllable parameters. Notice that the explanation is not, in fact, correct. For example, it misses the effect of GRADE upon SPEED. There is an implicit assumption that GRADE can be disregarded when solving acceleration/deceleration problems . This turns out to be true in the Champaign/Urbana area where the only two hills are man–made for children's sledding pleasure in the winter. It is one of the great strengths of the approach that the solution found can take advantage of systematic eccentricities in the distribution of planning problems given to the system. Why burden a planning system with reasoning about driving on a hill when this information will never be needed? This feature insures that the planner will not become bogged down in planning details that do not matter for the problems it is given. If another planning problem depends on the missing GRADE or BRAKE parameters, the above concept also would be eliminated in favor of one more faithful to reality.

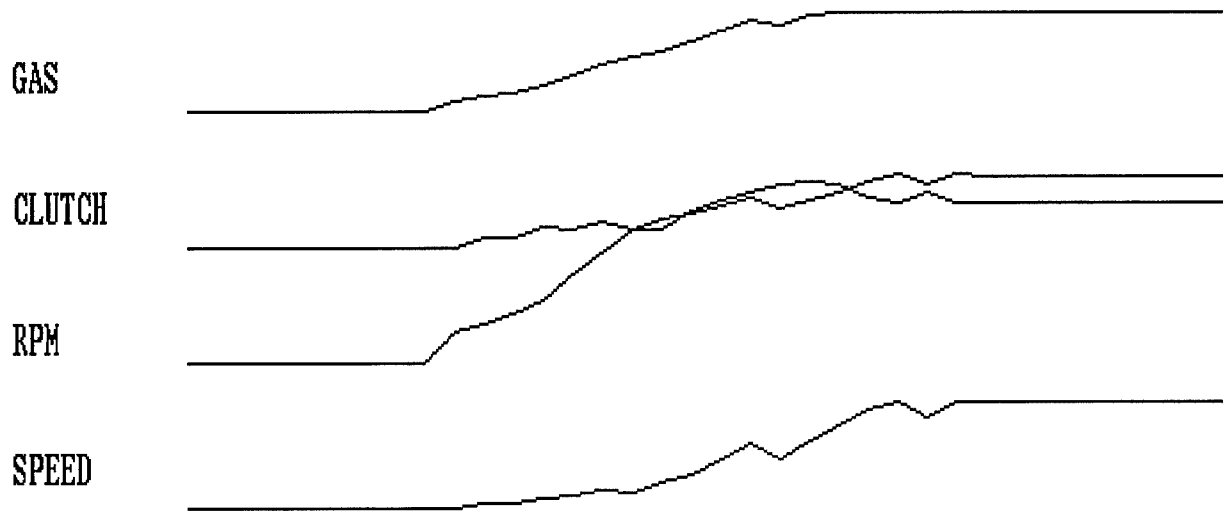


Figure 2: The Planner's First Solution

Nonetheless, this time it is sufficient to include CLUTCH and GAS as controllables. The system generates the adequate solution shown in figure 2.

Notice that the speed is not smoothly increasing; twice it actually decreases. The controllable parameters' values, particularly the CLUTCH, also exhibit a fair amount of "hunting" for the right value. This is due to interpolation error. The trajectory traced along the surface to solve this problem differs from the trajectory of the observed expert's solution. The approximated surface is most accurate near the observed trajectory.

Of the 50–odd points in the observation, the system finds that just 12 are sufficient to interpolate the others. There are many possible function surfaces that contain the observed points and respect the qualitative monotonicity constraints. The system chooses one: linear interpolation between the 12 selected points. Approximation error is due to the interpolation surface predicting one value for the internal quantity SPEED, while the finite difference model in fact yields another. The discrepancies are small enough and occur sufficiently far from observed points that there is no violation of the qualitative monotonicty constraints. Instead, new points are integrated into the interpolation approximation so that the approximate surface once again agrees with reality at all observed points. As it happens, 5 additional points are asserted to the interpolation function during this first successful solution. The function is then approximated by a total of 17 points. After several additional acceleration and deceleration problems the interpolation function contains 21 points and produces the solution shown in figure 3 to a different problem.

As can be seen, the speed is increased more smoothly. The system's manipulations of the control parameters are also smoother. The solution in figure 3 compared to the

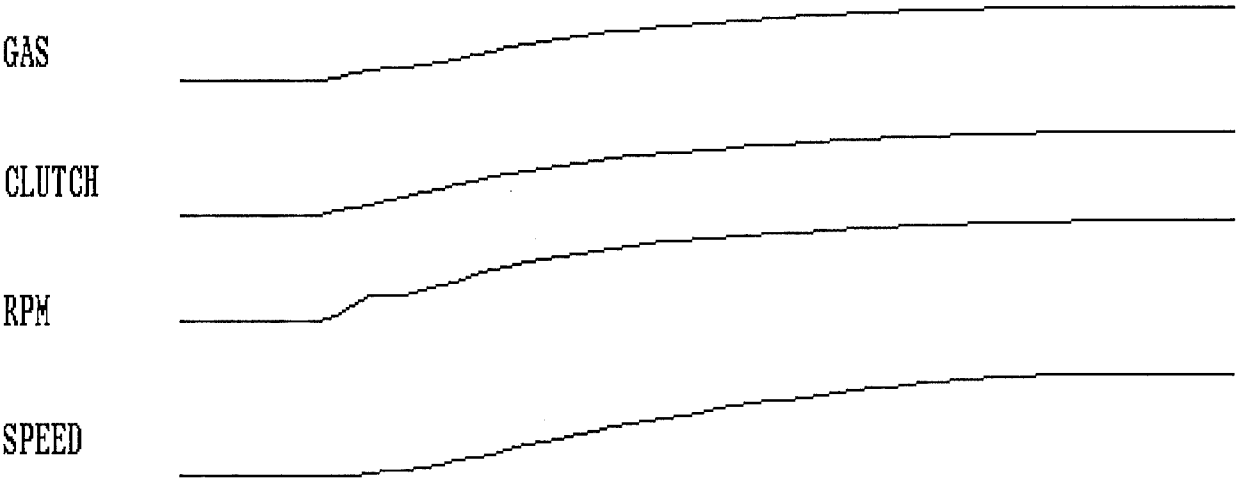


Figure 3: A Later Solution by the Planner

experts shows significantly higher REVs. As it happens the controls, especially the clutch, are more effective at higer REVs so smaller changes are required. This unintended effect is interesting froma planning point of view although not particularly desirable while driving real cars.

8. Empirical Analysis

In practice for few the domains examined convergence to competent problem–solving appears to be fast. Furthermore, planning time with the acquired constructs decreases with experience. The reason is that the cost of refining the interpolating approximation function is high compared to the cost of employing the function with no refinement. As experience with the concept increases, the approximation function converges to an acceptable function so fewer refinements are performed.

Figure 4 shows a typical planning run of 10 randomly generated acceleration and deceleration problems. Each is successfully solved by the system and each represents between 20 and 100 planning points. Fewer planning points are needed when the randomly generated goal speed is near the car's current speed. Figure 2 shows the results from the first problem in this test sequence; figure 3 is from the last.

The graphs show cumulative average information. The heavier line plots the CPU time in seconds used (up to and including the xaxis labeled problem) divided by the total number of planed points processed. The lighter line plots the cumulated error in miles per hour for all planned points (up to and including those in the problem labeled on the x–axis) divided by the number of such points.

As can be seen both CPU time and error decrease significantly with experience in just 10 problems. This graph is representative of such runs. If the graph were extended with additional problems, the lines would continue downward slightly. But this effect is due to the cumulative nature of the data collected – the high expense and error of early problems being diluted by later ones. Interpolation refinement is increasingly infrequent. In the 10th and later problems with no errors, planning continues at about 3 planned points per second with no improvement.

9. Discussion and Conclusions

The major significance of this work is A) providing a new avenue to pursue planning in continuous domains, B) providing a model that supports efficient planning with simultaneous, overlapping, and coordinated actions without the need for the implementor to anticipate such interactions, and C) providing a new model of plausible inferencing which, incidentally, is not confined to continuous domains. The cost for the planning benefits is that learning is now inextricably bound up with planning. Conversely, in this model, planning cannot be viewed as providing a guaranteed solution at planning time whose

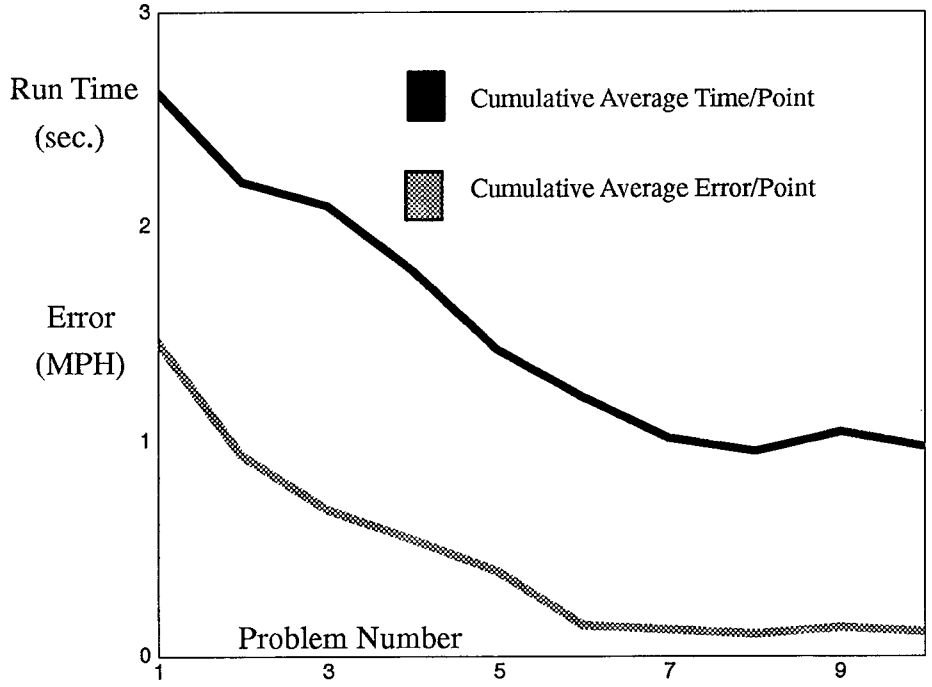


Figure 4: Emperical Performance on a Sequence of 10 Random Problems

execution is superfluous since success is logically entailed.

A major strength is in the planner's ability to acquire the minimal sufficient planning concept. The ability to control speed accurately while driving up and down hills is of little use in Urbana. In San Francisco, however, this skill may be crucial. Likewise, compensating for the effects of wind and air density while driving is unnecessary in a Detroit–built automobile, but may be crucial in a sun–powered ulta–light vehicle. Many many factors influence the speed of an automobile, or any internal quantity. Through experience a planner such as the one outlined tailors its planning concepts to environmental needs. The trick in planning must be to avoid ever thinking about most of such influences. They must be ruled out implicitly. If a planner must enumerate them all, if only to post non–monotonic assumptions that they are irrelevant, it cannot survive in the complexities of the real world. This is the well–known *qualification problem* [Genesereth87, McCarthy69, Shoham86b].

Of significance to the machine learning community is the knowledge level behavior [Dietterich86, Newell81] of the EBL system. Due to the semantics of plausible implication, every plausible conclusion that is drawn changes the knowledge level of the system. The acquired concepts are not in the *logical* transitive closure of the system. Of course, they are in the *plausible* transitive closure of the system. However, this is a weak statement. Nearly every behavior is in the plausible transitive closure of the domain theory. Yet, the system cannot acquire all such concepts. There is a learning bias [Utgoff86] precluding it. The bias guiding concept acquisition are the experiences in the real world mediated by the need to plausibly explain them. Also, theexpert's training example plays a much larger role in plausible EBL than in standard EBL.

Refining the qualitative explanation can be computationally intensive, but its convergence is guaranteed. Refining the approximate surface is less expensive but it too is guaranteed to converge. Planning without refinement is efficient for achieving simple goals with single planning concepts because hillclimbing–like approaches can be used. The monotonicity constraints guarantee that we cannot be trapped at local extrema.

REFERENCES

References

[Agre87] **P. Agre and D. Chapman, "Pengi: An Implementation of a Theory of Activity,"** *Proceedings of the National Conference on Artificial Intelligence*, **Seattle, WA, July 1987, pp. 268–272.**

[Allen83] **J. F. Allen and J. A. Koomen, "Planning Using a Temporal World Model,"** *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, **Karlsruhe, West Germany, August 1983, pp. 741–747.**

[Borchardt84] **G. C. Borchardt, "A Computer Model for the Representation and Identification of Physical Events," M.S. Thesis, Department of Computer Science, University of Illinois, Urbana, IL, May 1984. (Also appears as Technical Report T–142, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana–Champaign.)**

[Chapman87] **D. Chapman, "Planning for Conjunctive Goals,"** *Artificial Intelligence 32*, **3 (1987), pp. 333–378.**

[Chien89] **S. A. Chien, "Using and Refining Simplifications: Explanation–based Learning of Plans in Intractable Domains,"** *Proceedings of The Eleventh International Joint Conference on Artificial Intelligence*, **Detroit, MI, August 1989, pp. 590–595.**

[Dean83] **T. Dean, "Time Map Maintenance," Technical Report 289, Yale University, New Haven, CT, October 1983.**

[Dean90] **T. Dean and G. Siegle, "An Approach to Reasoning about Continuous Change for Applications in Planning," Working Paper, Computer Science Department, Brown University, February, 1990.**

[Dietterich86] **T. G. Dietterich, "Learning at the Knowledge Level,"** *Machine Learning 1*, **3 (1986), pp. 287–316.**

[Fikes71] **R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving,"** *Artificial Intelligence 2*, **3/4 (1971), pp. 189–208.**

[Firby87] **R. J. Firby, "An Investigation into Reactive Planning in Complex Domains,"** *Proceedings of the National Conference on Artificial Intelligence*, **Seattle, WA, July 1987, pp. 202–206.**

[Forbus84] **K. D. Forbus, "Qualitative Process Theory,"** *Artificial Intelligence 24*, **(1984), pp. 85–168.**

[Genesereth87] **M. Genesereth and N. Nilsson,** *Logical Foundations of Artificial Intelligence*, **Morgan Kaufmann, Palo Alto, CA, 1987.**

[Georgeff86] **M. P. Georgeff, "Representation of Events in Multiagent Domains,"** *Proceedings of the National Conference on Artificial Intelligence*, **Philadelphia, PA, August 1986, pp. 70–75.**

[Gervasio89] M. T. Gervasio and G. F. DeJong, "Explanation-Based Learning of Reactive Operators," *Proceedings of the 1989 International Machine Learning Workshop*, Ithaca, NY, June 1989.

[Hammond88] K. Hammond, T. Converse and M. Marks, "Learning from Opportunities: Storing and Re-using Execution-Time Optimizations," *Proceedings of the Seventh National Conference on Artificial Intelligence*, St. Paul, MN, August 1988, pp. 536-540.

[Hogge87] J. Hogge, "Compiling Plan Operators from Domains Expressed in Qualitative Process Theory," *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, July 13-17, 1987, pp. 229-233.

[Kuo87] B. Kuo,in *Automatic Control Systems*, Prentice Hall, 1987.

[Laird86] J. E. Laird, P. S. Rosenbloom and A. Newell, *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies*, Kluwer Academic Publishers, Hingham, MA, 1986.

[McCarthy69] J. McCarthy and P. J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence,"in *Machine Intelligence 4*, B. Meltzer and D. Michie (ed.), Edinburgh University Press, Edinburgh, Scotland, 1969.

[McDermott82] D. McDermott, "A Temporal Logic for Reasoning About Processes and Plans," *Cognitive Science 6*, 2 (1982), pp. 101-155.

[Mooney86] R. J. Mooney and S. W. Bennett, "A Domain Independent Explanation-Based Generalizer," *Proceedings of the National Conference on Artificial Intelligence*, Philadelphia, PA, August 1986, pp. 551-555. (Also appears as Technical Report UILU-ENG-86-2216, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.)

[Mooney88] R. J. Mooney, "A General Explanation-Based Learning Mechanism and its Application to Narrative Understanding," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, IL, January 1988. (Also appears as UILU-ENG-87-2269, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.)

[Newell81] A. Newell, "The Knowledge Level," *Artificial Intelligence Magazine 2*, (1981), pp. 1-20.

[Pednault88] E. Pednault, "Extending Conventional Planning Techniques to Handle Actions with Context-Dependent Effects," *Proceedings of the Seventh National Conference on Artificial Intelligence*, St. Paul, MN, August 1988.

[Sandewall89] E. Sandewall, "Combining Logic and DIfferential Equations for Describing Real-World Systems,"in *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, R. Brachman, H. Levewque, R. Reiter (ed.), Morgan-Kaufman, 1989, pp. 412-420.

[Schoppers87] M. J. Schoppers, "Universal Plans for Reactive Robots in Unpredictable Environments," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, August 1987, pp. 1039-1046.

[Segre87] A. M. Segre, "Explanation-Based Learning of Generalized Robot Assembly Tasks," Ph.D. Thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, January 1987. (Also appears as UILU-ENG-87-2208, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.)

[Shavlik88] J. W. Shavlik, "Generalizing the Structure of Explanations in Explanation-Based Learning," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, IL, January 1988. (Also appears as UILU-ENG-87-2276, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.)

[Shoham86a] Y. Shoham, "Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence," PhD. Thesis, Yale University, Dept. of Computer Science, New Haven, CT, 1986.

[Shoham86b] Y. Shoham, "What is the Frame Problem?," *Proceedings of the 1986 Workshop on Reasoning about Actions and Plans*, Timberline, Oregon, June, 1986, pp. 83-98.

[Truxal61] J. Truxal, "Identification of Process Dynamics,"in *Adaptive Control Systems*, E. Mishkin and L Braun (ed.), McGraw-Hill, 1961, pp. 51-90.

[Utgoff86] P. E. Utgoff, "Shift of Bias for Inductive Concept Learning,"in *Machine Learning: An Artificial Intelligence Approach, Vol. II*, R. S. Michalski, J. G. Carbonell and T. M. Mitchell (ed.), MORGAN, 1986, pp. 107-148.

[Vere83] S. A. Vere, "Planning in Time: Windows and Durations for Activities and Goals," *IEEE Transactions on Pattern Analysis and Machine Intelligence 5*, 3 (May 1983), pp. 246-267.

[Waltz82] D. L. Waltz, "Event Shape Diagrams," *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh, PA, August 1982, pp. 84-87. (Also appears as Working Paper 33, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.)

[Wilkins84] D. E. Wilkins, "Domain–Independent Planning: Representation and Plan Generation," *Artificial Intelligence* 22, (1984), pp. 269–301.

[Wilkins88] D. E. Wilkins, *Practical Planning: Extending the Classical Artificial Intelligence Planning Paradiigm*, Morgan Kaufman, San Mateo, CA, 1988.

A Framework for Evaluating Search Control Strategies

Jonathan M. Gratch and Gerald F. DeJong

Artificial Intelligence Research Group
Beckman Institute for Advanced Science and Technology
University of Illinois at Urbana–Champaign
405 North Matthews Avenue
Urbana, IL 61801

gratch@cs.uiuc.edu

Abstract

This paper provides a framework for describing systems which learn how to plan. In particular we view planning as search through a totally ordered space of possible plans. A control strategy describes the behavior of a planner by defining a mapping from problems to ordered search spaces and the goal of learning is to modify this control strategy to reduce the combinatorics of search. We conclude that this framework, even in this early stage, provides a useful perspective for analyzing performance learning systems. Given this characterization, it is clear that such algorithms are engaging in a search through the space of possible control strategies. It is also clear that these systems make strong assumptions about the topography of the search space, like guaranteed ascent, which we argue are violated. While our focus is on learning control strategies, the issues are relevant to the study of control knowledge in general.

1 INTRODUCTION

There has been widespread interest in applying machine learning techniques to enhance strategies of automated planning. Considerable effort has been devoted to the subproblem of performance learning. In this task a system is provided with a correct but intractable domain theory with which it must learn to solve problems efficiently. Progress in this approach has been challenged by the realization of trade–offs between the power of knowledge and the cost of using it, both in storage and time. Resolution of this challenge lies in part on clarifying the nature of the problem. The goal of this research is to provide a formal characterization of the planning process and the mechanisms by which this process may be modified. Given such a framework we can concisely describe the impact of knowledge and the rigorously evaluate competing strategies for acquiring control knowledge. This paper presents a first step towards this goal.

The focus of this paper is on learning how to plan. In particular we view planning as search through a totally ordered space of possible plans. A control strategy describes the behavior of a planner by defining a mapping from problems to ordered search spaces and the goal of learning is to modify this control strategy to reduce the combinatorics of search. These modifications are judged with respect to some measure of efficacy the learning module is trying to maximize. Thus the methods by which the learner can modify control knowledge are naturally viewed as operators in a meta–space, the space of possible control strategies where the efficacy measure defines the topography of this space. We believe this view of learning as a search in a control space provides an appropriate framework for analyzing competing learning strategies. Some important questions from this perspective include: what is the complexity of this space; what search strategy is employed; what information is available to guide search; how are control operators specified. While our focus is on learning control strategies, the issues are relevant to the study of control knowledge in general. For example, similar issues are faced when a knowledge engineer is designing or modifying a control strategy.

The research reported in this paper was supported by the National Science Foundation under grant NSF–IRI–87–19766

We will first describe a declarative representation for a simplified model of planning and discuss how current techniques of control strategy learning modify this representation. Next we describe the utility problem within this framework, describe how proposed solutions impact the search through control space, and discuss limitations of these strategies. Finally we will step into the realm of meta–planning and describe some of the issues to be faced by a system which plans to learn.

2 CONTROL STRATEGIES

This section takes a view of planning as a mapping from problems to ordered search spaces. The character of this mapping is defined by a body of knowledge called a *control strategy*. We develop a framework for describing control strategies which facilitates a discussion of strategy modification. The framework is introduced and then used to characterize properties of performance learning systems. Section 3 then uses this framework to evaluate competing learning strategies.

2.1 Planning

The goal of a Planning is: given a *problem* in terms of a goal, an initial state, and a set of action descriptions, find a sequence of actions which will transform the initial state into a state satisfying the goal. One can measure the efficacy of a Planner (from some perspective) through the use of an *efficacy function*. Such a function can be based on the percentage of problems a Planner can solve from a set of problems or the average time it takes the Planner to solve problems in a set. By associating a learning module with a planner we can hope, through experience, to improve the efficacy of a planner with respect to a particular efficacy function.

The framework we are describing is sufficiently abstract to apply to any search–based view of planning. However to draw clear connection between this framework and existing learning systems we will cast our presentation in terms of what is considered *classical planning*. This formalism uses a logic to represent states and actions and simple chaining as a rule of inference. Classical planning is akin to producing a proof that a goal can be realized from an initial world state. The process of planning is then one of search through possible proofs, guided by a control strategy.

2.2 Planning Knowledge

Given this characterization, we can view the planning process as consisting of two information sources. First there is a *structural component* which defines a possibly infinite potential search space through the interaction of a particular problem and the inference procedure embodied by the planner. For a typical backward–chaining system this space takes the form of a tree where nodes are partial state descriptions, links are partial operator descriptions, and the goal description is the root (see figure 1). Structural information can be viewed as a mapping from problems to potential search spaces.

Restriction 1: For the remainder of this discussion we will assume that a search space takes the form of a tree of partial state descriptions with a finite branching factor where the goal description is the root. Thus identical descriptions will be represented as different nodes if they are reached by distinct paths from the root.

Next there is a *ordering component* which defines how the potential space is explored for a node meeting a halting criteria. For example PROLOG explores its search space in a depth–first order where choice points are resolved by the order of rules in the knowledge base and the order of preconditions in those rules. To represent the effects of the ordering component we introduce the notion of an ordered search space. This can be viewed as the presence of ordering links between nodes in the potential search space. An incomplete ordering strategy would then specify a partial ordering among the nodes. A complete ordering strategy would impose a total ordering over the nodes. Given a total ordering we can then restrictively represent a potential search space as an order list of nodes – namely the order in which they will be visited by the planner

Restriction 2: We assume that search must be deterministic.

The PROLOG example highlights that the distinction between structure and order is not necessarily reflected in the implementation of these systems. It is typically not possible to encode an inference procedure without implicitly defining an ordering strategy. However, as since we are striving for a mathematical characterization we can entertain the fiction that these processes are independent.

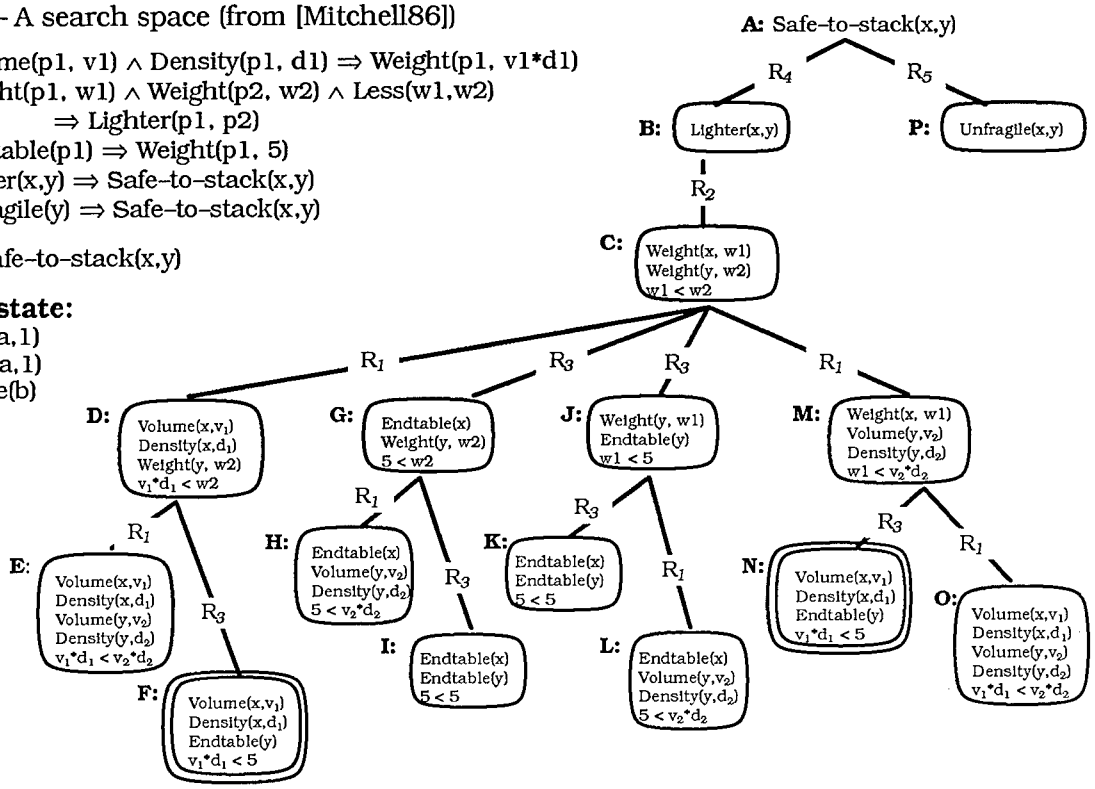
Since our focus is learning we will make an additional distinction which is meaningful to this task. A planner's behavior is described by its mapping from problems to ordered search spaces and determined by its structural

Figure 1 - A search space (from [Mitchell86])

R_1: Volume(p1, v1) ∧ Density(p1, d1) ⇒ Weight(p1, v1*d1)
R_2: Weight(p1, w1) ∧ Weight(p2, w2) ∧ Less(w1,w2) ⇒ Lighter(p1, p2)
R_3: Endtable(p1) ⇒ Weight(p1, 5)
R_4 Lighter(x,y) ⇒ Safe-to-stack(x,y)
R_5 Unfragile(y) ⇒ Safe-to-stack(x,y)

Goal: Safe-to-stack(x,y)

Initial state:
Volume(a,1)
Density(a,1)
Endtable(b)



and ordering knowledge. Learning changes the behavior of a planner by modifying or overriding this knowledge. Knowledge which cannot be modified, which is inherent to the planner, will be termed the *native strategy* of the planner. Knowledge which can be modified and new information which is acquired through learning will be termed the *learned strategy*. Information in either of these strategies can be characterized as modifying the structural or ordering component of the planner. The native strategy is simply the default control strategy implicit in the planning system (typically opaque to the learning module) while the learned strategy is a body of acquired knowledge which determines the final behavior of the system.

2.3 Planning Modification

We have sketched the basic organization of control strategies. We will now describe three basic classes of control strategy modifications. This will then provide a vocabulary to discuss our model of control strategies.

The goal of a learned control strategy is to allow rapid discovery of successful plans within a search space. Following from our dichotomy in planning knowledge there are two natural approaches for modifying planning behavior. One obvious approach is to limit the size of the potential search space defined by the structural component. Thus if we are doing a depth–first search, not searching below a node will effectively skip the subtree rooted in that node. We will call a modification to the structure of the potential search space a *structural modification.*

Learning can be exploited to alter the structure of a search space in many ways. For example acquiring new domain theory operators or modifying the definition of existing ones will change the set of reachable states. For the scope of this paper we will limit ourselves to a subset of structural modifications termed *reduction modifications*. These are modifications which prune regions of a potential search space. A reduction modification is *complete* if is guaranteed not to prune any solutions.

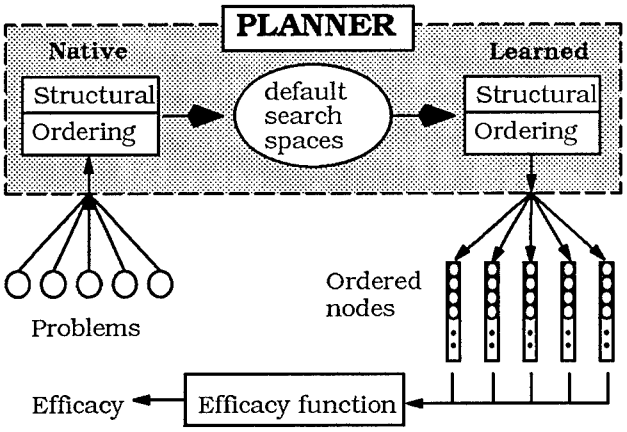
In that knowledge is rarely sufficient to make these irrevocable modifications without disrupting completeness we may simply prefer exploring some nodes before others, so if the preference is wrong the alternatives will be eventually reached. Information which modifies the

ordering component of planning is termed an *ordering modification.*

There is a third class of modification which we will not address in this paper. Recall that planners are resource limited and that transitions in the ordered search space consume resources. *Resource modifications* are ways of changing the pattern of resource use. Examples include truth–preserving simplifications of control knowledge [Minton88] and non–truth–preserving techniques [Cohen, Keller]. However, in that resource modifications also serve to change structural and ordering components, and that we have yet to develop an adequate vocabulary for describing them, we will not describe them further.

Given that the search may incorporate rejection modifications, a subset of the nodes in the search space will be visited. Given that the search is deterministic these nodes will be visited in a well defined order. Thus we can view planning as a mapping from problems to ordered lists of nodes. Information such as where solution nodes lie in this ordering and the resources expended at each node can be used as parameters to an efficacy function which in turn can serve as a measure of planning success. This is summarized in figure 2.

Figure 2 - simplified model of planing



2.4 Control Axioms

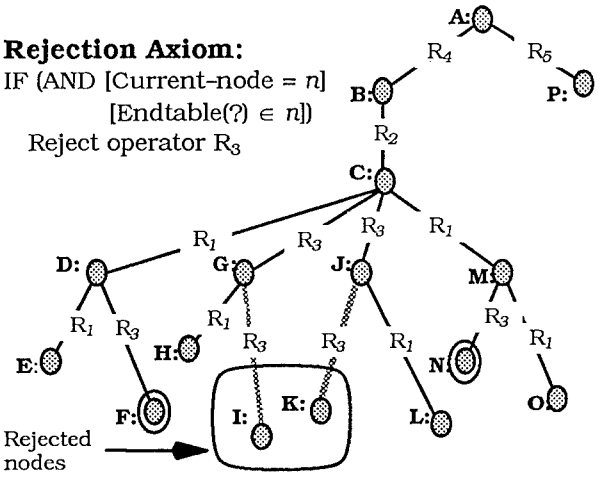
The previous sections introduce a view of planning in terms of structural and ordering knowledge. Learning is then seen as modifications to these knowledge sources. In particular, we view performance learning as a search through a space of possible search control strategies. To support this perspective we must precisely define the notion of control strategy and how a learning module can transition between alternatives. For this purpose we will adopt a declarative representation of a control strategy based on the notion of declarative control packets used in [Minton88].

Our general approach is to view a control strategy as arising from the interactions of individual declarative packets of control information termed *control axioms.* Each axiom describes a particular search space modifier and the circumstances where it applies. A state in the space of possible control strategies is then defined by a set of control axioms. Thus a set of control axioms define a mapping from problems to ordered nodes. Movement through this space is through the addition, deletion, or modification of these axioms.

As mentioned above we are restricting attention to two basic modification classes: reduction modifications and ordering modifications. A *decision procedure* contains the criteria for determining when a modification applies. A control axiom is then a decision procedure/modification pair, and can be a reduction axiom or an ordering axiom.

It is natural to think of a control axiom as a condition–action rule axiom. Figure 3 illustrates this with a rejection axiom acting on the search space in figure 1. In this case the decision procedure is defined by precondition satisfaction and the control modification is a rejection procedure which removes individual links. Any instantiation of this rule removes one link from the search and indirectly discards all nodes in the subtree pointed to by the link.

Figure 3 – Example reduction axiom

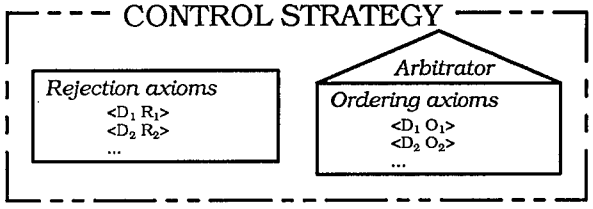


To complete our description we must make one more distinction. Ordering axioms can conflict while reduction axioms cannot. For example a control strategy may simultaneously recommend that node **B** be explored before node **P** and that node **P** be explored before node **B.**

This property requires us to maintain an arbitration procedure whenever potentially contradictory ordering modifications are allowed.

We can now provide a first cut at defining a control strategy. A control strategy C is composed of two partitions: 1) a set of reduction axioms; 2) a set of ordering axioms and an arbitration procedure (figure 4).

Figure 4



To briefly summarize, planning is composed of two information sources. There is structural knowledge which defines the structure of a potential search space and control knowledge which defines the order in which this space is search. A control strategy emerges from the synergy of individual control axioms. There are then three questions we can ask of a control strategy:

1) What is the vocabulary of control modifications – what are the mechanisms available to alter the search space.

2) What is the vocabulary of decision procedures – what information is available and appropriate to constrain the modifications.

3) How do local decisions interact to form a global strategy.

2.5 Vocabulary of Modifications.

First let us consider the vocabulary of modifications. modifications are the actions available to the learning system. To describe these modifications we must consider both their local effects (what direct impact do they have on the search space) and their global effects (how do the direct effects interact with the structure of the space and other modifications to impact the space). This section defines the local effects available to systems.

As defined, the notion of a control modification is far too powerful. It allows such control modifications as *delete every node except a solution node* and decision procedures like *prefer a path if it leads to the best solution.* We will begin by specializing the notion of control modification. Thus while arbitrary control modifications are possible, we will for now restrict consideration to the three control axioms utilized by explanation–based learning research: *rejection rules*[1], *preference rules*, and *macros.*

The search space view we have adopted from restriction 1 defines a tree of partial state descriptions joined by partial operator descriptions. The primary restriction imposed by these strategies is that reduction modifications can only directly impact the connections between nodes. Ordering modifications tend to be more subtle and are described below. Although a control modification might only affect a single link in the search space, figure 3 illustrates a single control axiom may have several distinct instantiations and thus influence a set of links in a particular search space.

2.5.1 Rejection and Preference Rules

The PRODIGY/EBL system [Minton88] is the prototype for rejection and preference rules. Rejection control axioms encode the further restriction that the connections affected must be between parent and child. The strategy implemented in PRODIGY/EBL permits a wide range in flexibility in specifying connections. As Minton observed, constructing a link from parent to child in this space can be viewed as three control modifications: choose a parent's unsatisfied state descriptor, choose an operator relevant to the descriptor, choose a specialization of the operator. PRODIGY/EBL can take advantage of these distinctions to subtly specify links but we will disregard this for the purpose of this discussion. The main point is that the rejection modifications are restricted in scope to deleting connections connecting a particular parent to some set of children.

PRODIGY/EBL allows two classes of ordering effects. Goal, operator, and bindings preferences result in ordering modifications which obey the above constraint, changes are only made in the ordering of links between a parent and its children. The node preference axiom however allows arbitrary nodes to be preferred to others as long as one is not a descendent of the other.

2.5.2 Macro Operators

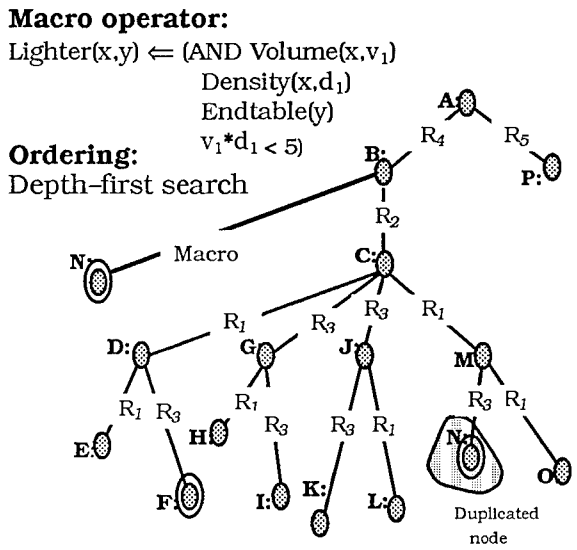
Traditional explanation–based learning systems have relied on the notion of a macro operator to increase their efficacy. Macros can be viewed as a form of ordering modification. The body of a macro encapsulates a particular path through the search space from which it was learned. This sequence is then generalized to repre-

[1]Minton [Minton88] distinguishes between selection and rejection rules but for our purposes they are treated as one. The difference is one of descriptive convenience. Selection rules say discard everything but X while rejection rules say discard X.

sent a set of possible paths. Macros alter the behavior of a planner by presenting themselves as operators. In the case of backward chaining the planner can now regress a goal through a sequence of operators in one step. If macros are expanded before other operators, this means that the node defined by the preconditions of the macro will be reached sooner than it would have otherwise been. Thus macros may order a descendent of a node before one of the nodes immediate children.

Unfortunately macros impose ordering modifications at the expense of redundancy. As is clear in figure 5, macros effectively copy nodes in the search space and change their position in the ordering. Figure 5 illustrates the effect of a macro on the search space from figure 1. In this case a redundant link has been created between node **B** and **N**. A depth–first strategy would then search node **N** earlier than before and then search it again in its original position. See [Greiner89] for a related analysis on the impact of redundant links.

Figure 5 – Direct macro effects



2.6 Decision Procedures

A decision procedure determines when a particular control modification will be realized. While control modifications provide the vehicle for defining a control strategy, the decision procedures embody the expertise. There is no reason to prefer one alternative to another unless there is some information which suggests this is a correct modification.

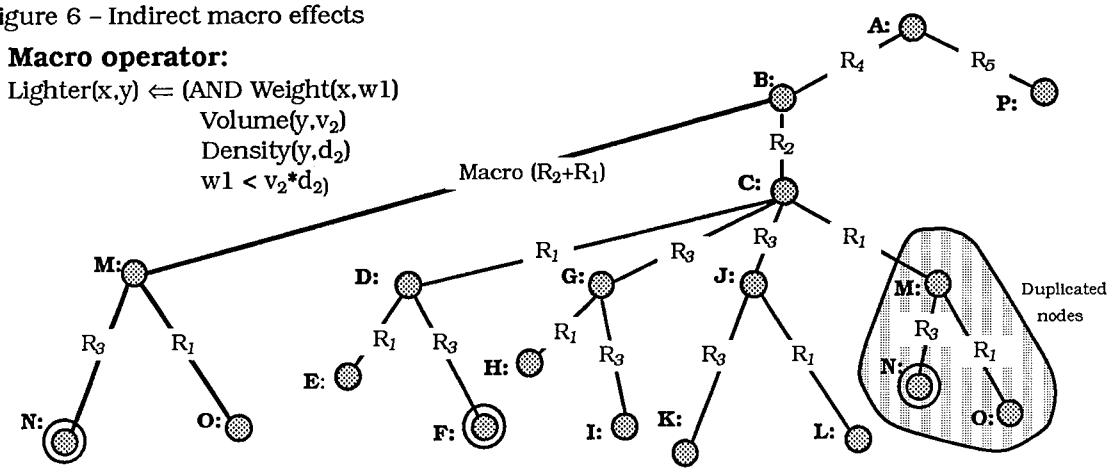
The purpose of a decision procedure is to constrain a control modification to apply to contexts where it is appropriate. So while arbitrary rejection of nodes is unlikely to benefit a system, a more informed rejection may be. In the case of the input resolution example we can guarantee the decision is appropriate given the restriction of horn–clause theories. There are also tradeoffs involved in determining appropriateness. Thus a control axiom may be appropriate in that it trims only redundant paths but efficacy reducing in that it increases the resources required to achieve a goal. Determinations of appropriateness will be discussed at length within the section on the utility problem.

There are two distinctions to make about decision procedures: their *heuristic power* [Nilsson80p. 72] and their *discriminability*. The first distinction centers on their properties on convergence (how quickly can a solution be found, are we guaranteed to find a solution, are their local maxima). The second describes what information is available to make a decision (simply the goal, the initial state, tokens on the goal stack, ordering information). A decision is *heuristic* if under certain circumstances it may lead to decreased efficacy. For example a decision procedure associated with a reduction modification will be heuristic if it sanctions the modification in circumstances which prune solution nodes.

Heuristic decisions have been typically acquired by inductive techniques. Thus LEX2 acquires preference heuristics by inducing the conditions under which an operator leads to success. PRODIGY/EBL on the other hand uses logical proofs to guarantee a control modification is correct with respect to the discriminability of the procedure.

The view of macros as control axioms leads to an interesting conclusion when we question what decision procedure they use. Macros have traditionally used a very simple decision procedure: *apply a macro iff it has an effect which unifies with the current goal.* As we will argue later we believe that much of the utility problem with macro learning can be attributed to the unsatisfactory nature of this decision procedure. Indeed it is important that many of the suggestions for improving macro performance can be viewed as adding more information to the decision procedure associated with a macro. Thus the inductively acquired utilization filters of [Markovitch] are directly mapped into this representation. [Mooney89] also suggests limitations of the use of macros. In this case he makes the strong suggestion that one should not chain on the preconditions of macros. From our viewpoint this imposes a decisions procedure which is guaranteed to visit the state characterized by a macro's preconditions iff that state is a solution node.

Figure 6 - Indirect macro effects



2.7 Indirect Control Modification Effects

While section 2.4 limited the direct effects of control modifications, it is clear from figure 6 that other nodes can be effected. In this case not only is a copy of node **M** moved earlier in the ordering but also copies of its descendents. These indirect effects arise from interactions between control modifications and the current ordering strategy used by the planner. This means that besides the node directly affected by the control modification, the entire subtree beneath that node is impacted in some way. This is obvious in the case of rejection modifications since the subtree is completely pruned from the search. In the case of a preference or macro modification the effects can vary wildly depending on the other ordering modifications it must interact with.

Figure 7 illustrates the interaction of a preference and macro modification with the traditions ordering strategies of depth–first and breadth–first search. In the case of depth–first search an entire subtree is moved as a block before other nodes in the search. For breadth–first the pattern is more complex. A whole subtree is reordered but this is broken across levels in the search. Along with reordering, macro modifications have the effect of moving subtrees to a higher level in the search. For an empirical analysis of the interaction of ordering strategies with macro modifications, see [Mooney89].

Macros are distinct from preference rules in that they may also engage in interactions with reduction modifications. This is because macros serve to insulate paths from other control axioms. In figure 6 a macro is created which corresponds to the path from **B** to **C** to **M**. If a rejection rule is later acquired which deletes **C** from the search space the macro maintains a connection to **M**. Thus a reduction axiom which effects intermediate nodes in the path captured by a macro will not impact the nodes duplicated by the macro.

2.8 Resource Bounds

The above discussion is in terms of potential problem spaces but only a finite subset of such a space can actually be explored by an implemented system. This is because actual planning is resource limited and the act of creating and traversing the search space consumes resources. In that resource bounds serve to limit the potential search space, these too can be described as reduction axioms. For our purposes it suffices to say that resource bounds map potential search spaces into *realizable search spaces*. A realizable search space consists of the set of nodes actually visited by the planner. For an in depth description of the impact of different resource bounds, see [Segre].

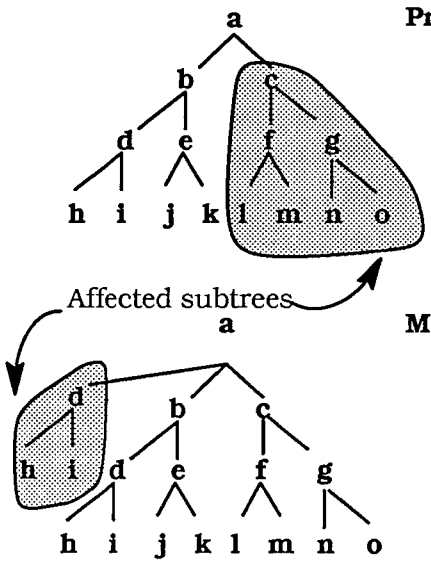
3 EBL AS SEARCH

Section 2 described how we can represent a planner by a set of control axioms. We have discussed how these axioms interact to specify a control theory and that the goal of learning is to modify this control theory in such a way as to maximize a measure of efficacy. The methods by which the learning module can modify control knowledge are thus naturally viewed as operators in a meta–space, the space of possible control strategies. We now turn to describing prior learning strategies in terms of the search methods they employed. This will then allow us to characterize their utility in terms of our model.

3.1 Guaranteed ascent assumption

Early EBL strategies make many implicit assumptions about the character of the control–space to simplify their search for a control strategy. The primary assumption is that any action in the control–space serves to in-

Figure 7 – Indirect modification effects



Preference modification: prefer **c** over **b**

Depth–First:

Without: a b d h i e j k c f l m g n o

With: a c f l m g n o b d h i e j k

Breadth–first

Without: a b c d e f g h i j k l m n o

With: a c b f g d e l m n o h i j k

Macro modification: prefer copy of **d** over **b** and **c**.

Depth–First:

Without: a b d h i e j k c f l m g n o

With: a d h i b d h i e j k c f l m g n o

Depth–First:

Without: a b c d e f g h i j k l m n o

With: a d b c h i d e f g h i j k l m n o

crease the output of the efficacy function, a *guaranteed ascent* assumption. With this strong assumption the efficacy function can be disregarded and backtracking is unnecessary.

Early systems thus employed a single operator in the control–space: *learn a macro*. This operator was then instantiated to a particular macro or macro set which was concatenated to the existing control strategy. The ultimate effect of such actions is then determined by aspects of the native strategy, primarily the default search strategy, and any decision procedure assigned to the learned macros. The typical decision procedure is *apply if macro has a consequent which unifies with the current subgoal*, which is sometimes specialized to include no chaining on macros or other ubiquitous restrictions.

Unfortunately empirical results demonstrate the failing of such a simplistic learning method [Minton85, Mooney89]. There have been four main approaches to address this failing: 1) define a native strategy which produces guaranteed ascent; 2) choose a vocabulary of control axioms which produces guaranteed ascent; 3) allow heuristic search; 4) allow backtracking. We will elaborate this briefly.

We can imagine a control–space which is the power set of all possible control axioms. As was mentioned previously, we can view an overall search strategy as composed of a fixed native component and a modifiable control component. The native component defines a starting point in control–space and the operators available to the learner define possible transitions beginning at this point. One possible approach is to identify a starting point such that we have guaranteed ascent within the space of possible learning actions. The recommendation to use a breadth–first search strategy with macro–learning [Mooney89] is a step in this direction. Such recommendations, however, rely on properties which remain constant across different domain theories, problem distributions, and efficacy functions. In that changes to any of these parameters can have dramatic consequences to the topology of the control–space we must await strong theoretical analysis before placing confidence in the generality of these recommendations.

An alternative is to change the actions available to the learner in the hope of establishing guaranteed ascent. In the framework of our decision/modification representation of control axioms this corresponds to changing the constraints on decision procedures and control modifications. A fair amount of work has centered on the former. Thus limiting the use of macros through chaining restrictions [Mooney89] can be viewed as adding more conditions to decision procedure of each macro modification. The utilization filters of [Markovitch] also specialize decision procedures and in that these filters can be acquired through learning, the control space is effectively enriched. The approach to "killer chunks" in SOAR [Tambe] and the emphasis on non–recursive decisions [Etzioni] again restrict the class of decision procedures. To evaluate these methods one must have a sophisticated domain independent understanding of how they alter the topography of the control–space, an understanding which is still forthcoming.

The above two methods avoided use of state evaluation functions to judge their progress. An alternative is to

use the efficacy function or an approximation to it to guide search through the control–space. This the approach taken by PRODIGY/EBL [Minton88] where estimates of the utility of individual rules guides the search for a global solution. Only rules with high estimated utility are incorporated into the learned strategy. Thus PRODIGY/EBL performs a hillclimbing search for an effective control strategy.

The above methods suggest irrevocable search for a control strategy. Given that a learning system incorporates a state evaluation function, it becomes useful to consider backtracking. Any strategy which incorporates selective forgetting of learned rules could be viewed as a form of backtracking. In this sense, PRODIGY/EBL has implemented a hillclimbing search with backtracking.

3.2 Efficacy Estimation

Guaranteed ascent insures every action in control space increases efficacy but it seems difficult to ensure this property. If we relax this restriction such that *some* action increases efficacy then we are forced to specify or approximate some action evaluation procedure or else abandon convergence. In this section we will discuss how efficacy can be approximated to guide search.

An efficacy function maps sets of control axioms to efficacy values for a given distribution of problems. Without further knowledge, a this function can only be implemented by table look–up and acquired through rote learning (execute a representative problem set and remember the performance). Furthermore, this table must be as large as the power set of all control axioms to obtain full coverage. Instead it is profitable to take advantage of local properties of control axioms and to understand how this local information combines to produce a global measure.

PRODIGY/EBL exemplifies the only efficacy estimation strategy suggested by the explanation–based learning community. This local strategy relies on the observation that all else being equal the contribution of one control axiom, its utility, is determined by the average savings it provides (an estimate) minus the average cost of its decision procedure (empirically derived). A strong assumption is then made that efficacy can be maximized solely by maintaining axioms of positive utility.

3.3 Interactions

Empirical studies have demonstrated the difficulty in obtaining guaranteed ascent. In this section we describe properties of control operators which shed light on this difficulty. We will also show how these same properties call into question local efficacy estimation strategies like that used in PRODIGY/EBL.

We have presented a view of control axioms operators in a control–space. To search this space effectively using the simple search strategies proposed we require strong constraints on the transitions between control strategies – either guaranteed ascent or no local maxima and effective evaluation functions. The previous section on indirect control axiom effects was the first warning that these restrictions may not hold. We now expand on that line of analysis.

What is difficult about control operator effects is that efficacy is dependent on factors which are not domain independent, calling *a priori* biases into question. One of these factors is the distributions of problems that will be seen. This is not an extreme problem – we could require that distributions are provided from the onset or assume that future distributions reflect past experience, the later requiring variable biases. More problematic is that efficacy is also dependent on the distribution of decision points within the search spaces of problems (this is elaborated below). This is problematic because this distribution is dependent on the particular control strategy in use. Thus to correctly predict the effects of adding new control knowledge we must understand how this changes the distribution of choice points within each problem space for every problem in the expected distribution of problems. A hefty task. We will now describe why efficacy is so dependent on choice point distribution and why distribution changes are so hard to predict.

Appropriateness of decision procedures: As was illustrated in figure 3, a control axioms can be instantiated in multiple ways, both within and across problems. Thus a particular control axiom corresponds to a set of instantiated control axioms. An instantiated control axiom is termed *appropriate* if its application increases the efficacy measure, all other things being equal. An interaction occurs when this axiom is appropriate for some members of its set of instantiated control axioms, but not for other members. This interaction is *realized* if nodes which would lead to inappropriate decisions are actually encountered during the search. This last statement means that local utility of a control axiom is dependent on factors which determine membership in the set of instantiated control axioms. These factors include the distribution of problems presented to the planner and other control axioms. The later is because control axioms alter the realizable search space for an particular problem and

thus may alter the class of possible instantiations for other control axioms.

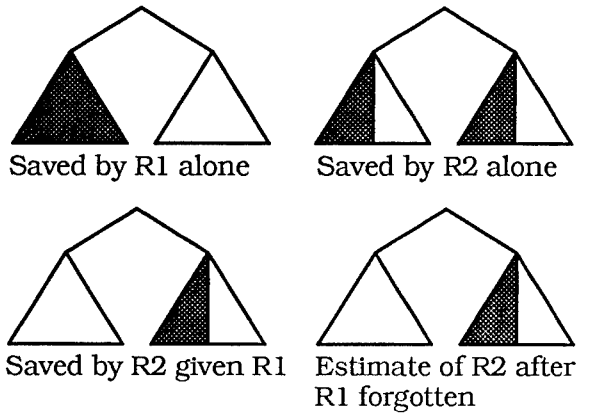
Pedanticness of decision procedures: One way to alleviate the problem of appropriateness is to increase the discriminability of decision procedures. Thus if we can guarantee that all of the instantiations of a control axiom are appropriate, we have simplified determination of local utility. PRODIGY/EBL rejection rules only trim nodes which are provably irrelevant. Markovitch's utilization filters attempt to discriminate when a macro will lead to a solution. Unfortunately increasing the discriminability of a decision procedure typically requires more resources for its evaluation and thus lowers its local utility. Thus PRODIGY/EBL learns rejection rules which suggest appropriate modifications but whose introduction increases the cost of the control strategy. This occurs when the information required to determine the appropriateness of a search modification is more expensive than the savings realized by it.

Additionally in is often not realistic to derive a decision procedure which guarantees appropriateness. An example of this arises in the context of conjunctive goals. Satisfying one subgoal in a particular way might preclude the simultaneous satisfaction of other subgoals. Thus a control axiom which is locally appropriate may be globally inappropriate. To be appropriate such a decision procedure must have available all the constraints which arise from the rest of the plan, information which may not be available.

Sociopathic control strategies: A set of control axioms is *sociopathic* [Ma] if axioms are individually judged to be good but the global behavior is bad. If this property holds, any efficacy measure in terms of local utility can be complex indeed. Appropriateness trade–offs are one property which may lead to the sociopathic property. We will also illustrate another example which impacts the PRODIGY/EBL system.

As illustrated in figure 8 the effects of reduction axioms can overlap in the areas of search they avoid, in this case two rejection rules, R1 and R2, trim overlapping nodes. PRODIGY/EBL computes the local utility of a rule base on its match time and estimated search savings. Which rule is actually credited for the savings is determined simply by the order in which they were learned. In the case where R1 is learned before R2, R1 is credited with all of the savings under the left subtree. This interaction makes R1's estimate overly optimistic and R2's pessimistic and R1 may be incorrectly retained over R2. Furthermore, the information is not available to update the savings of R2 if R1 is eventually forgotten. One unfortunate consequence of this property is that PRODIGY/EBL can be stuck on local maxima in its search for the best control strategy.

Figure 8 – sociopathic control rules



In summary we feel that the utility problem is arising from complex and subtle properties of the control space. We have shown that the search methods learning systems have available are quite simple and in our opinion inadequate to the task. Our hope in future research is to provide a better understanding of the control space which can suggest more appropriate learning operators or a better characterization of operator effects.

4 PLANNING TO LEARN

In that we are admitting that learning is a search through a meta–space, what information do we want available at this level. One obvious possibility is to provide a richer vocabulary of control–space operator effects. It is currently not possible to even express control axiom interactions much less reason about their impact on efficacy. While it is not clear such a vocabulary could be operationally specified, it seems a prerequisite to informed control strategy modification.

There are also several other issues which have been generally avoided by the learning community and which might appropriately be handled through meta–reasoning. One important issue is when to learn. Resources consumed during learning have not generally been incorporated into empirical evaluations of these systems. The excuse has been that learning time can be amortized over all problems the system will solve. However in that no system has shown convergence and that recent learning strat-

egies are resource intensive, this excuse may not be reasonable.

The reinforcement learning there has been the observation that when the search space is ill–behaved a system can achieve better overall performance by performing a broader search [Sammut]. This can be viewed as a form of experimentation, trading off current performance in the hope of a future gain. Decisions to experiment could thus be reasoned about at this level.

5 CONCLUSIONS

We have presented a search space view of planning where the efficacy of a planner is determined by properties of the ordered search spaces produced by it on a distribution of problems. A control strategy is a set of individual control axioms which determine a planners mapping from problems to ordered search spaces. The power set of all possible control axioms then defines a space of all possible control strategies and efficacy determines the topography of that space.

We conclude that this framework, even in this early stage, provides a useful perspective for analyzing performance learning systems. Given this characterization, it is clear that such algorithms are engaging in a search through the space of possible control strategies. It is also clear that these systems make strong assumptions about the topography of the search space, like guaranteed ascent, which seem to be violated. Instead we have argued that the operators used to transition in this space engage in complex and subtle interactions, resulting in a space topography which is ill–suited to the search strategies currently emploied.

In that current approaches to the utility problem are not directly addressing control interactions we feel they are not well motivated. Instead we suggest characterizing these interactions and either reasoning about them during learning or utilizing them in restricting the class of control axioms. The later should also be of interest to control strategy engineers. In that control interactions are complex, progress will likely result from strong restrictions upon the vocabulary of decision procedures.

Acknowledgements

We would like to thank Steve Chien for helpful discussions on this topic as well as comments on the draft of this paper. Thanks also to Scott Bennett and Michael Barbehenn.

References

[Cohen] W. W. Cohen, "Learning Approximate Control Rules Of High Utility," *ML90,*, pp. 268–276.

[Etzioni] O. Etzioni, "Why Prodigy/EBL Works," *AAAI90,*.

[Greiner89] R. Greiner and J. Likuski, "Incorporating Redundant Rules: A Preliminary Formal Analysis of EBL," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, AUG 1989, pp. 744–749.

[Keller] R. M. Keller, "Concept Learning in Context," *ML87,*, pp. 91–102.

[Ma] Y. Ma and D. C. Wilkins, "Sociopathicity properties of evidential reasoning systems," Technical Report KBS–90–002, Department of Computer Science, University of Illinois

[Markovitch] S. Markovitch and P. D. Scott, "Utilization Filtering: a method for reducing the inherent harmfulness of deductively learned knowledge," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, Michigan,, pp. 738–743.

[Minton85] S. Minton, "Selectively Generalizing Plans for Problem–Solving," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, August 1985, pp. 596–599.

[Minton88] S. N. Minton, "Learning Effective Search Control Knowledge: An Explanation–Based Approach," CMU–CS–88–133, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA, MAR 1988.

[Mitchell86] T. M. Mitchell, R. Keller and S. Kedar–Cabelli, "Explanation–Based Generalization: A Unifying View," *Machine Learning 1*, 1 (JAN 1986), pp. 47–80.

[Mooney89] R. J. Mooney, "The Effect of Rule Use on the Utility of Explanation–based Learning," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, AUG 1989, pp. 725–730.

[Nilsson80] N. J. Nilsson, *Principles of Artificial Intelligence*, TIOGA, 1980.

[Sammut] C. Sammut and J. Cribb, "Is Learning Rate a Good Performance Criterion for Learning," *ML90,*, pp. 170–178.

[Segre] A. Segre, C. Elkan and A. Russell, "On Valid and Invalid Methodologies for Experimental Evaluations of EBL," *Machine Learning Journal (to appear),*.

[Tambe] M. Tambe and P. Rosenbloom, "Eliminating Expensive Chunks by Restricting Expressiveness," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, Michigan,, pp. 731–737.

§9

Competition-Based Learning for Reactive Systems

John J. Grefenstette

Navy Center for Applied Research in Artificial Intelligence
Naval Research Laboratory
Code 5514
Washington, DC 20375-5000
Email: GREF@AIC.NRL.NAVY.MIL

Abstract

Traditional AI planning methods often assume a well-modeled, predictable world. Such assumptions usually preclude the use of these methods in adversarial, multi-agent domains. This paper describes our investigation of machine learning methods to learn reactive plans for such domains, given access to simulation model. Particular emphasis is given to the task of assessing the effects of differences between the simulation model and the environment in which the learned plans will ultimately be tested. Methods for utilizing existing partial plans are also discussed.

1. Introduction

The goal of this work is to explore the application of machine learning techniques to reactive planning problems arising in adversarial, multi-agent domains. In such domains, traditional AI planning approaches are usually infeasible, because of the complexity of the multi-agent interactions and the inherent uncertainty about the future actions of other agents. One approach to such problems is to develop a plan expressed as a set of condition/action rules that specify appropriate responses to any given situation. The behavior of a plan can be monitored in a simulation to discover any weaknesses or inadequacies. This information can be used to modify the rules, which can then be re-evaluated in the simulation. Such a generate-and-test cycle can be repeated until a satisfactory plan is found, which can then be released for application in the real world (see Figure 1). In many applications, off-line learning is the only realistic alternative for evaluating the performance of hypothetical plans, since testing plans on the "live" system is too difficult, too costly, or too dangerous. The current system was designed with off-line learning in mind. The overall objective is to reduce the manual effort involved in the generate-and-test cycle in evolving high-performance reactive plans.

The reactive systems we consider here may be characterized by the following general scenario: The decision making agent interacts with a discrete-time dynamical system in an iterative fashion. At the beginning of each time step, the agent observes a representation of the current state and selects one of a finite set of actions, based on the agent's decision rules. As a result, the dynamical system enters a new state and returns a (perhaps null) payoff. This cycle repeats indefinitely. The objective is to find a set of decision rules that maximizes the expected total payoff.[1] Several tasks for which reactive systems are appropriate have been investigated in the machine learning literature, including pole balancing (Selfridge, Sutton and Barto, 1985), gas pipeline control (Goldberg, 1983), and the animat problem (Wilson, 1987). For many interesting problems, including the one considered here, payoff is delayed in the sense that non-null payoff occurs only at the end of an episode that may span several decision steps.

2. Overview of the Approach

SAMUEL is a system that uses competition-based machine learning to develop reactive plans. SAMUEL incorporates several assumptions selected to make the system broadly applicable to real-world problems. First, the system's perception facilities are limited to a fixed set of discrete, possibly noisy, sensors.[2] There is also a fixed set of control variables may be set by the decision making agent. The system's decision rules are limited to simple condition/action rules of the form

```
if   (and c1  . . .  cn)
then (and a1  . . .  am)
```

where each c_i is a condition on one of the sensors and each action a_j specifies a setting for one of the control variables. A *reactive plan* in SAMUEL comprises a set of such decision rules.

The knowledge base in SAMUEL can be initialized with plans that provide a minimal level of competence

[1] See (Barto, Sutton and Watkins, 1989) for a good discussion of broad applicability of this general model.

[2] See (Whitehead and Ballard, 1990) for a discussion of the problem of *perceptual aliasing* under such conditions.

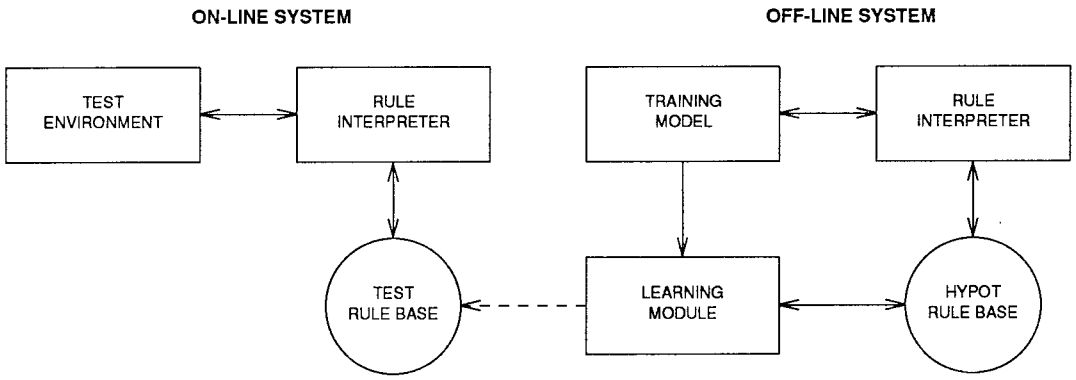


Fig. 1. Learning from a Simulation Model

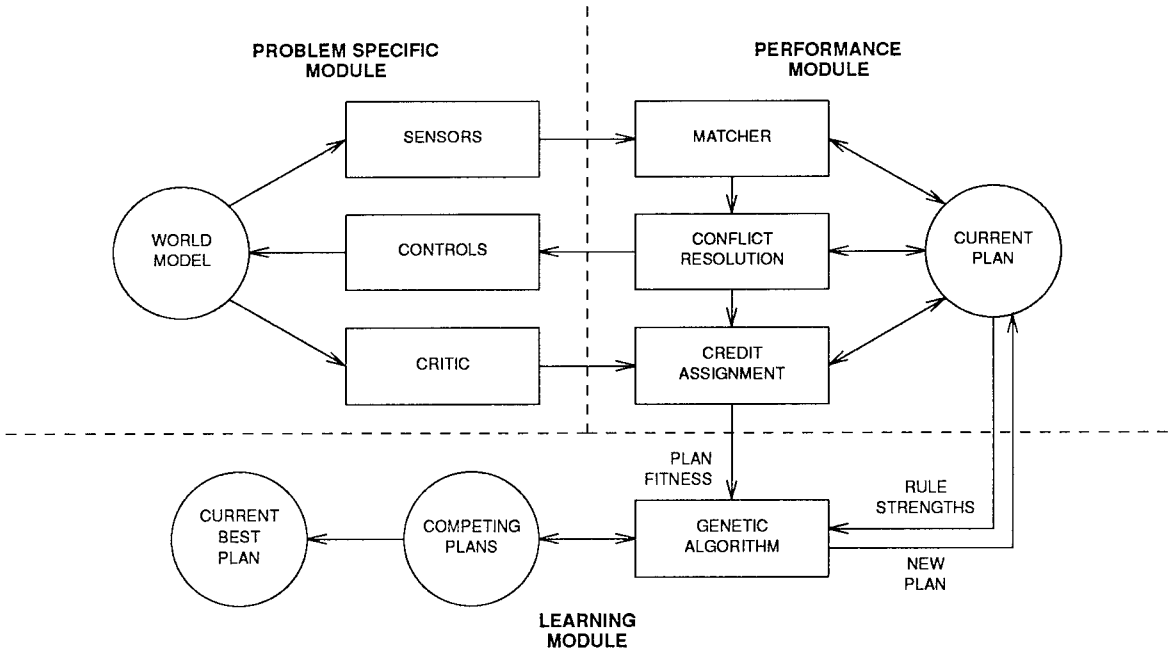


Fig. 2. SAMUEL: A System for Learning Reactive Plans

on the performance tasks. SAMUEL improves its reactive plans through the application of *competition* at two levels (Figure 2). At the rule level, each rule is assigned a *strength* that estimates its utility on the basis of its record of past payoff (Grefenstette, 1988). Conflict resolution is implemented as a probabilistic competition among rules based on rule strength. SAMUEL maintains a population of alternative plans. These plans compete with one another using a genetic algorithm (Holland, 1975): Each plan in the current population is evaluated on a number of tasks from the problem domain (typically, 20 tasks in the experiments described here). As a result of these evaluations, plans with high performance are selected and recombined, using genetic operators such as CROSSOVER and MUTATION, producing plausible new plans for the next iteration.

This learning system has been tested on a sequential decision problem first discussed by Erikson and Zytkow (1988), called Evasive Maneuvers (EM). In EM, there are two objects of interest: a prey and a pursuer. The decision maker controls the actions of the prey to evade the approaching pursuer. The pursuer can track the motion of the prey and steer toward the prey's anticipated position. Six sensors give information about the current state: the current turning rate of the

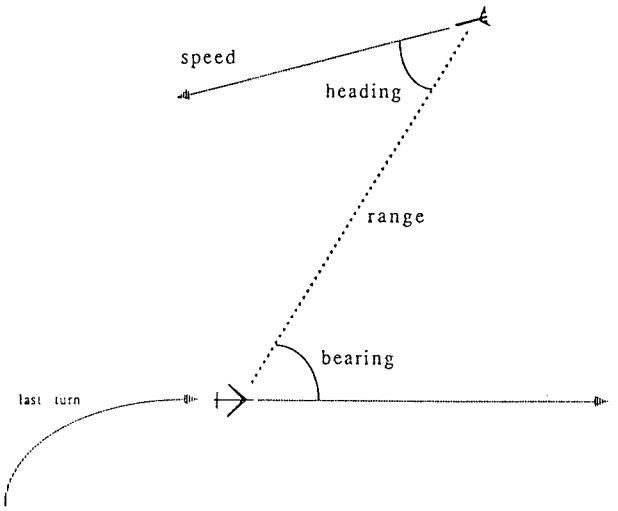


Fig. 3. The Evasive Maneuvers World

prey, a clock, the pursuer range, the pursuer bearing, the pursuer heading, and the pursuer speed (Figure 3). There is a single control variable, the turning rate of the prey. The process is divided into *episodes* that begin with the pursuer approaching the prey from a randomly chosen direction. The pursuer initially travels at a far greater speed but is less maneuverable than the prey (i.e., the pursuer has a greater turning radius than the prey) and gradually loses speed as it maneuvers. The episode ends when either the pursuer captures the prey or the pursuer's speed drops below a threshold and it loses maneuverability. This requires between 2 and 20 decision steps, depending on how many turns the pursuer performs while tracking the prey. At the end of each episode, the critic provides a payoff defined by the formula:

$$payoff = 1000 \quad \text{if prey escapes pursuer}$$

$$= 10t \quad \text{if prey is captured at time } t$$

A plan for EM consists of a set of decision rules. A sample rule follows:

```
if   (and (last-turn 0 45) (time 4 14)
          (range 500 1400) (bearing 3 6)
          (heading 90 180) (speed 50 850))
then (and (turn 90))
strength 750
```

Each condition on the left-hand side of a rule specifies a range of values for a sensor, and each action specifies the value for a control variable. The strength is an estimate of the rule's utility and is used for conflict resolution (Grefenstette, 1988).[3] For more details, see (Grefenstette et. al, 1990).

3. Case Studies

This section presents a summary of a number of empirical studies of the performance of SAMUEL on the EM problem. Because SAMUEL employs probabilistic learning methods, all graphs represent the mean performance over 20 independent runs of the system, each run using a different seed for the random number generator. When two learning curves are plotted on the same graph, a vertical line between the curves indicates that there is a statistically significant difference between the means represented by the respective plots (with significance level $\alpha = 0.05$) at that point on the curves.

3.1. Accuracy of Simulation Model

One important topic concerns the inevitable differences between the simulation model in which the knowledge is learned and the target environment in which the learned knowledge will be used (see Figure 1). We have performed a number of experiments in which the rules learned in one environment were tested in an slightly different environment. These experiments give some feeling for the robustness of the rules learned.

In one experiment, two environments were defined that differed by the initial conditions selected for the start of each episode (Grefenstette, Schultz and Ramsey, 1990). In the environment with *fixed initial conditions*, the pursuer's initial speed, distance to the prey, and relative heading were always the same, but the bearing (direction) from which the pursuer approached the prey was selected as random for each episode. In the environment with *variable initial conditions*, the pursuer's initial speed, distance and heading were randomly selected from a range of values.

Figure 4 shows the result of learning plans under fixed initial conditions (dashed curve), and the results of testing those plans in the environment with variable initial conditions (solid curve). Not surprisingly, performance of the plans degraded in the less restricted environment. Figure 5 shows the results when learning occurs in the environment with variable initial conditions (solid curve), and the results of testing those same plans in the environment with fixed initial conditions

[3] The strength of a rule is reduced by an estimate of the rule's *inconsistency*, measured by the variance in the payoff obtained by the rule. A similar method is used by Whitehead and Ballard (1990) in their Q-learning system.

(dashed curve).

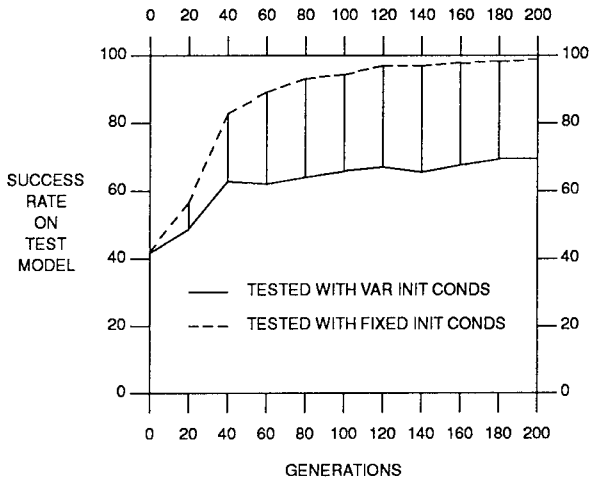


Fig. 4. Learning with Fixed Initial Conditions

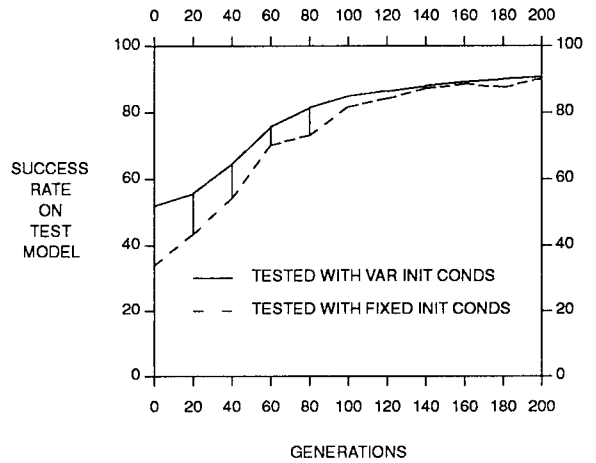


Fig. 5. Learning with Variable Initial Conditions

The interesting result here is that there was no significant degradation of performance when the plans final learned in the variable environment were tested in the simpler environment. The evidence is that the system learned robust plans that perform well regardless of the initial conditions of the pursuer. Comparing the two graphs, it is obvious that one pays a price for learning the more robust plans, in that a longer period of learning is required to reach equivalent levels of performance.

Figure 6 illustrates some aspects of this trade-off, comparing the best plans learned in each of the two training environments. Each plan was tested on 11 environments with differing initial conditions, ranging from the conditions in the fixed-initial-conditions case to the conditions in the variable-initial-conditions case.

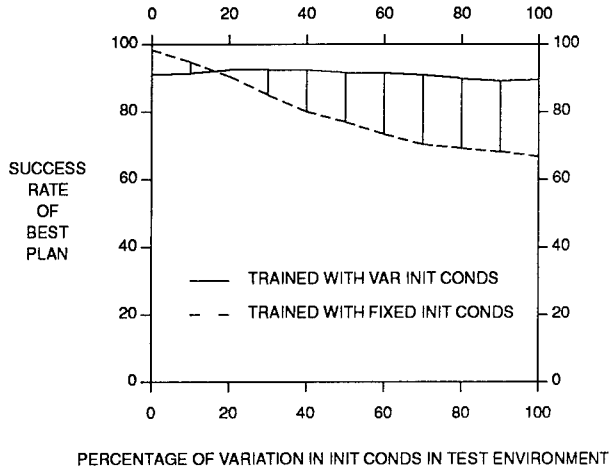


Fig. 6. Performance of Learned Plans Tested Against Gradually Variable Initial Conditions

The plan that was learned in the variable-initial-conditions environment performed uniformly well in all the tested environments. The plan learned in the fixed-initial-conditions environment degraded steadily as the environment included a greater variety of initial conditions. One interesting interpretation of Figure 6 is that the penalty for assuming an overly predictable environment is far greater that the penalty for assuming the environment less regular than it is.

Similar results were obtained in another set of experiments in which the difference between the two environments concerned the amount of noise in the sensors (Ramsey, Schultz and Grefenstette, 1990). Figure 7 shows the result of learning plans with noise-free sensors (dashed curve), and the results of testing those plans in an environment with noisy sensors (solid curve). Figure 8 shows the results when training occurs in the environment with noisy sensors (solid curve), and the results of testing those same plans in the environment with noise-free sensors (dashed curve). Again, there was no loss of performance when the plans learned in the noisy environment were tested in the noise-free environment. These results suggest strongly that, for systems like SAMUEL, it pays to make the simulation of the environment more challenging that the actual test environment is likely to be (see Figure 9).

3.2. Improving Existing Plans

One of the features of SAMUEL is that, unlike many previous genetic learning systems (Smith, 1980; Goldberg, 1983; Holland, 1986), the knowledge representation consists of symbolic condition-action rules, rather than low-level binary pattern matching primitives. The use of a high level language for rules

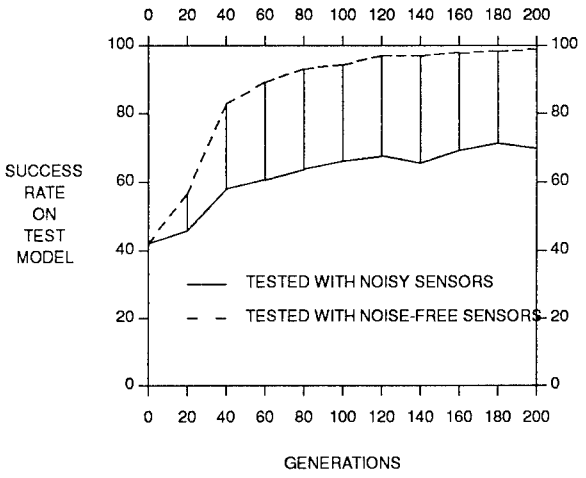


Fig. 7. Learning with Noise-Free Sensors

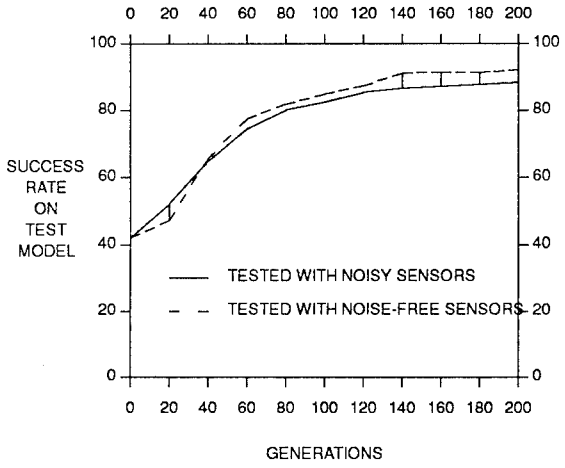


Fig. 8. Learning with Noisy Sensors

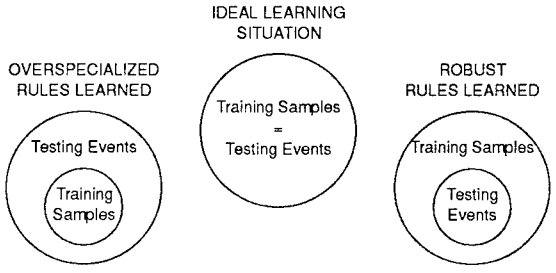


Fig. 9. Effects of Simulator Accuracy

offers several advantages. First, it is easier to transfer the knowledge learned to human operators. Second, it makes it possible to combine empirical methods such as genetic algorithms with analytic learning methods that explain the success of the empirically derived rules (Gordon and Grefenstette, 1990). Finally, it makes it easier to incorporate existing knowledge. A recent study (Schultz and Grefenstette, 1990) addressed this final point by comparing two mechanisms for initializing the knowledge structures in SAMUEL. The results presented here show that genetic algorithms can be used to improve partially correct plans, as well as to learn plans given no initial knowledge.

First, a tactical plan for EM was manually developed:

If the pursuer is far enough away, turn so that it is behind the prey. When the pursuer is closing in, make hard turns such that the pursuer losses velocity. If the pursuer is heading away from the prey and going slow, ignore it and continue in the current direction.

This plan can be expressed naturally in the rule representation language of SAMUEL. The manually generated plan successfully evades the pursuer about 75% of the time. Three methods for initializing the population of competing plans were compared. In the *adaptive initialization* method, each plan in the initial population consisted of a set of maximally general rules, which are then specialized according to the systems early experiences (Grefenstette et. al, 1990). In the *homogeneous population* method, each of the initial plans consisted of the heuristically generated plan, augmented by the maximally general rules. Finally, in the *heterogeneous population* method, part of the population is assigned the heuristic plan and the remainder of the population consists of the maximally general rules. A comparison of the three methods is shown in Figure 10.

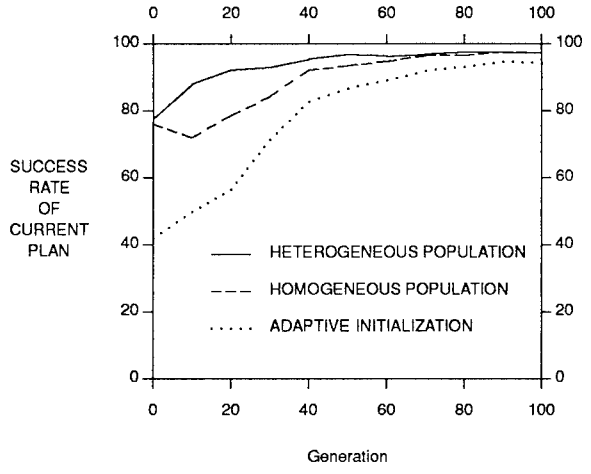


Fig. 10. Learning from Partially Correct Plans

The dotted line shows the learning curve when the no heuristic knowledge is available, i.e., the adaptive initialization method. The dashed line shows the learning curve when heuristic knowledge is incorporated in the initial population of knowledge structures using the homogeneous population method. The solid line is the learning curve for the heterogeneous population method, in which the partially correct plans compete directly with plans generated by the adaptive initialization heuristics. While a detailed analysis of the results are beyond the scope of this paper, the results indicate that, using the heterogeneous method, SAMUEL can exploit existing knowledge and learn more quickly when heuristic rules are available. This suggests that future work might explore the trade-offs between manual knowledge acquisition and machine learning.

4. Summary

These initial studies in a simple competitive environment have shown that it is possible for learning systems based on genetic algorithms to develop high performance, robust reactive plans. Once a high performance plan has been learned, it can be viewed as a source of expert behavior. In (Gordon and Grefenstette, 1990) we outline an approach to applying explanation-based techniques to reactive plans learned by SAMUEL, in order to clarify the system's performance as well as generate new reactive rules. Current efforts are also aimed at augmenting the task environment to test SAMUEL's ability to learn reactive plans for a variety of more realistic domains. Further developments along these lines can be expected to reduce the manual effort required to build systems with expert performance in complex, adversarial domains.

Acknowledgments

I want to acknowledge the contributions toward the development of SAMUEL by the members of the Machine Learning Group at NRL, including Alan Schultz, Connie Ramsey, Diana Gordon, Helen Cobb, and Ken De Jong.

References

Barto, A. G., R. S. Sutton and C. J. C. H. Watkins (1989). Learning and sequential decision making. COINS Technical Report, University of Massachusetts, Amherst.

Erickson, M. D. and J. M. Zytkow (1988). Utilizing experience for improving the tactical manager. *Proceedings of the Fifth International Conference on Machine Learning*. Ann Arbor, MI. (pp. 444-450).

Goldberg, D. E. (1983). *Computer-aided gas pipeline operation using genetic algorithms and machine learning*, Doctoral dissertation, Department Civil Engineering, University of Michigan, Ann Arbor.

Gordon, D. G and J. J. Grefenstette (1990). Explanations of empirically derived reactive plans. *Proceedings of the Seventh International Conference on Machine Learning*. Austin, TX. (pp. 198-203).

Grefenstette, J. J. (1988). Credit assignment in rule discovery system based on genetic algorithms. *Machine Learning, 3(2/3)*, (pp. 225-245).

Grefenstette, J. J. (1989). A system for learning control plans with genetic algorithms. *Proceedings of the Third International Conference on Genetic Algorithms*. Fairfax, VA: Morgan Kaufmann. (pp. 183-190).

Grefenstette, J. J., A. C. Schultz and C. L. Ramsey (1990). Simulation-assisted learning by competition. To appear in *Machine Learning*.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor: University Michigan Press.

Ramsey, C. L., A. C. Schultz and J. J. Grefenstette (1990). Simulation-assisted learning by competition: Effects of noise differences between training model and target environment. *Proceedings of the Seventh International Conference on Machine Learning*. Austin, TX. (pp. 211-215).

Selfridge, O., R. S. Sutton and A. G. Barto (1985). Training and tracking in robotics. *Proceedings of the Ninth International Conference on Artificial Intelligence*. Los Angeles, CA. August, 1985.

Smith, S. F. (1980). *A learning system based on genetic adaptive algorithms*, Doctoral dissertation, Department of Computer Science, University of Pittsburgh.

Wilson, S. W. (1987). Classifier systems and the animat problem. *Machine Learning, 2(3)*, (pp. 199-228).

Whitehead, S. D. and D. H. Ballard (1990). Active perception and reinforcement learning. *Proceedings of the Seventh International Conference on Machine Learning*. Austin, TX. (pp. 179-188).

Towards a Theory of Agency*

Kristian Hammond Timothy Converse Mitchell Marks
The University of Chicago
Department of Computer Science
Artificial Intelligence Laboratory
1100 East 58th Street
Chicago, IL 60637
ai@tartarus.uchicago.edu

1 Introduction

There is a tension in the world between complexity and simplicity. On one hand, we are faced with a richness of environment and experience that is at times overwhelming. On the other, we seem to be able to cope and even thrive within this complexity through the use of simple scripts, stereotypical judgements, and habitual behaviors. It seems that to function in the world, we have idealized and simplified it so as to make tractable our own reasoning about it. As a group and as individuals, human agents search for and create islands of simplicity and stability within a sea of complexity and change.

In general, Artificial Intelligence has ignored this tension. It has tended towards theories that either attempt to face the complexity of the world head on, or trivialize the problem through oversimplification of the world. The result of the former has been the production of "general purpose" devices that that are uniform in their ability to solve problems from different domains only in that are uniformly bad at doing so. The result of the latter has been toy programs for toy domains that do little to inform us about the true structure of intelligence.

The response to the failure of this drive towards general purpose problem-solvers has, unfortunately, been the production of domain-dependent programs that sacrifice any hope of generality in return for specialized problem-solving skills. This has been most apparent recently in the reactive movement that has been producing specialized devices that are robust in their own areas because of the skills of particular programmers rather than their own internal structure.

In this paper, we will discuss an approach to relieving this tension that rises out of the case-based reasoning movement. This approach embraces rather than avoids this paradox of the apparent complexity of the world and the overall simplicity of our methods for dealing with it. It does this by treating the behavior of intelligent agents as an ongoing attempt to discover, create, and maintain the stability that is necessary for the production of goal-satisfying action.

Our basic argument rests on the idea that general purpose intelligence is only possible within the confines of learning and planning systems that work to establish functional correspondences between the world and their conception of it. This is not to say that we advocate the position that systems should be attempting to construct perfect internal models of the world. Our view is that autonomous agents must strive towards the production of goal-satisfying behavior—not the production of internal categories that exist only to match the structure of the external world.

Our framework for the study of agency consists of four basic parts:

- Case-based planning.
- Learning from failure.
- Learning from execution-time opportunity.
- Stabilizing the environment through enforcement.

Each of these components provides a piece in the overall effort to establish the correspondence between the internal world of an agent and the complex environments in which it must function. The starting point, case-based planning, provides the framework by producing standard plans that themselves are fixed points in the world. The ability to learn from failure allows for the incremental search of the simplified space of variations that actually arise in the world as opposed to the far more complex space of those problems that might occur. Likewise, the ability to recognize and learn from execution-time opportunities provides the ability to construct and save plans for the conjuncts of goals that actually arise in a problem space while avoiding the problems of exponential search of the space itself. Finally, the ability to stabilize an environment with respect to an agent's view of the world and the plans that he has built to deal with it provides the ability to create niches in which the complexity of the world is reduced, thus making it easier to reason about and act within.

Because we are concerned with goal-satisfying behavior and with the production of actual actions in the world, we prefer not to label our work as *planning*. Instead, we prefer a label that links the work to our object of attention, the intelligent agent, and thus refer to it as the study of *agency*. We discuss our work on agency in the context of three programs: CHEF, TRUCKER, and RUNNER.

*This work was supported in part by the Defense Advanced Research Projects Agency, monitored by the Air Force Office of Scientific Research under contract F49620-88-C-0058, and the Office of Naval Research under contract N0014-85-K-010.

2 Planning and Action in an Open World

Most AI work on problems of autonomous agency has been within the context of planning research, where a fairly strict separation between planning and execution was assumed. The classical development of the theory of planning and problem-solving emphasized exhaustive preplanning, with the goal of being able to guarantee that an optimal or near-optimal plan would be found if one existed. Planners in this paradigm required certain assumptions to hold:

- The world will be stable; it will behave as projected.
- Time consumed in planning is independent of the time that can be devoted to execution.
- The information available to the planner is complete, and execution will be flawless.
- Any initially correct plan will remain correct and can in fact be carried out.

In the real world, however, these assumptions simply do not hold. As we relax these assumptions, new issues arise that researchers in the early days of planning work were able to avoid. This opening of the world leads us to a new set of constraints that apply to any agent that must produce plans and actions in the world. These constraints include:

- An agent lacks perfect information about its world and the effects of its own actions.
- An agent does not always know all of its goals in advance.
- Planning time is limited, and shared with execution-time.
- The mapping from an action in a plan to an action in the world is non-trivial.
- Projection over all possible worlds is theoretically and practically intractable.
- The goal of the agent is to act, not simply to plan.

We do not make these assumptions because we want to. We make them because we have been forced to. Such is the nature of real world domains. In general, we propose that the only way to deal with the intractability of exhaustive reasoning within complex domains is to integrate the tasks of planning and learning into a single agent architecture. Rather than suppose that it is possible to construct functional plans from scratch, we suggest that a dynamic case base of plans and their effects be produced and used incrementally. In this way, a planner can improve itself over time through the understanding of its own success and failure within a given domain.

3 Case-based Planning

One technological proposal for addressing both the issue of execution-time failure and the complexity of *de novo* plan construction comes out of emerging work in case-based reasoning [Hammond, 1989, Kolodner and Simpson, 1984, Martin, 1990, Owens, 1990, Schank, 1982]. This work has suggested an approach to problem-solving that seems to be a tractable alternative to the more traditional rule-based approaches [Fikes and Nilsson, 1971, Sacerdoti, 1975]. Case-based planning suggests that the way to deal with the combinatorics of planning and projection is to let experience tell the planner when and where things work and don't work. Rather than replanning, reuse plans. Rather than projecting the effects of actions into the future, recall what they were in the past. Rather than simulating a plan to tease out problematic interactions, recall and avoid those that have cropped up before.

3.1 A framework for case-based planning

This framework suggests seven basic case-based planning processes:

- An ANTICIPATOR that predicts planning problems on the basis of the failures that have been caused by the interaction of goals similar to those in the current input.
- A RETRIEVER that searches a plan memory for a plan that satisfies as many of the current goals as possible while avoiding the problems that the ANTICIPATOR has predicted.
- A MODIFIER that alters the plan found by the RETRIEVER to achieve any goals from the input that it does not satisfy.
- A PROJECTOR that uses cases indexed by planning solutions rather than problems to predict the outcomes of suggested plans on the basis of the outcomes of similar plans in memory.
- An INDEXER that places new plans in memory, indexed by the goals that they satisfy and the problems that they avoid.
- A REPAIRER that is called if a plan fails. It is here that we argue that causal knowledge is applied – if it is applied at all.
- An ASSIGNER that uses the causal explanation built during repair to determine the features which will predict this failure in the future. This knowledge is used to index the failure for later anticipation. As in repair, causal knowledge is useful in anticipation but not essential.

These seven modules make up the basic algorithm for a case-based planner. The RETRIEVER, MODIFIER and INDEXER make up the central planning loop that allows old plans to be modified in service of new goals. The REPAIRER is required for those situations in which plans fail. And the ASSIGNER and ANTICIPATOR provide the learning and application modules that allow the planner to avoid making mistakes that it has already encountered.

4 Learning from failure

The lack of a perfect domain model and thus complete projection means that case-based planners are open to the possibility of failure. While this is problematic in general, we argue that this possibility must be faced by any reasoning system that is functioning in an open

world. Our way of facing the issue of failure is to allow that it will happen and design systems that are able to cope with it at execution-time and, as a result of the experience, anticipate and avoid it in later planning. The product of this approach is the incremental search of the space of *actual* problems that tend to arise in any particular domain. The apparent complexity of possible worlds is reduced by allowing the planner to only concern itself with those worlds that actually exist. Our approach to the complexity of projection in planning is to learn to recognize the situations in which failures occur and use that recognition to avoid them. The planner uses its own failures to learn problematic features in a domain. These problematic features can then be used to predict problems in later planning situations so that the planner can construct its new plan knowing the problems it must avoid. By also storing plans in terms of the problems that were encountered while building them, the planner can use the prediction of a problem to find a plan in memory that avoids it.

This idea of using failures to learn the features that predict them is implemented in the CHEF planner [Hammond, 1989]. CHEF uses an *anticipate and avoid* approach to planning problems that is sharply contrasted with the *create and debug* approach taken by earlier planners [Wilensky, 1983, Sacerdoti, 1975]. CHEF attempts to predict and plan for possible failures before they actually occur rather than waiting for them to happen and repairing them once they have. The ability to learn from its own failures allows the CHEF planner to anticipate and avoid those problems that it has seen before.

It is important to note that this type of knowledge is learned through an active experimentation in new domains. In particular, it is learned when the planner's expectations fail. The case of learning to *anticipate and avoid* problems in involves expectation failures that correspond to planning errors. As we will see in the section that follows, learning about specific optimizations involves expectation failures that correspond to episodes of the planner recognizing and exploiting opportunities.

5 Opportunism and memory

Despite addressing some of the problems of traditional planning, case-based planning still shares some of its basic assumptions; the most central being a view of action as decomposable into separate planning and execution phases, where the execution phase is carrying out the dictates of the computational object called the plan. In recent years there has been a realization in the planning community that the domains that will support such a separation are much rarer than had previously been supposed.

Exhaustive pre-planning for a set of goals seems to require at a minimum that an agent be aware of all its goals at plan time and that he have complete knowledge of the physics and other agents in his environment. Unfortunately, this is simply not true of any interesting domains or situations that an agent must confront.

Just as a system that cannot fully predict the future must contend with those futures in which plans fail, it must also deal with those futures in which they succeed in unanticipated ways. In order to be effective, it must also cope with and learn from opportunities that were not predicted in much the same way that it deals with failures.

Our approach to the problem of execution-time opportunism uses episodic memory to organize, recognize and exploit opportunities. Briefly, the algorithm includes the following features:

- Goals that cannot be fit into a current ongoing plan are considered blocked and, as such, are suspended.
- Suspended goals are associated with elements of episodic memory that can be related to potential opportunities.
- These same memory structures are then used to parse the world so that the planner can make routine execution-time decisions.
- As elements of memory are activated by conditions in the world, the goals associated with them are also activated and integrated into the current processing queue.

In this way, suspended goals are brought to the planner's attention when conditions change so that the goals can be satisfied. Because the recognition of opportunities depends on the nature of its episodic memory structures, we call the overall algorithm presented here *opportunistic memory.*

5.1 Opportunistic planning

Our approach to opportunism builds on two views of opportunism in planning—that of Hayes-Roth and Hayes-Roth [Hayes-Roth and Hayes-Roth, 1979], and that of Birnbaum and Collins [Birnbaum and Collins, 1984].

Hayes-Roth and Hayes-Roth presented the view that a planner should be able to shift between planning strategies on the basis of perceived opportunities, even when those opportunities are unanticipated. Their model, which they called *opportunistic planning*, consisted of a blackboard architecture and planning *specialists* that captured planning information at many levels of abstraction. The planner could jump between strategies as different specialists "noticed" that their activation conditions were present. In this way, the planner could respond to opportunities noticed at planning time.

More recently, Birnbaum and Collins [Birnbaum and Collins, 1984] presented a view of opportunism that included a role for execution. Under their model, goals are viewed as independent processing entities that have their own inferential power. When a goal is suspended because of resource constraints, it continues to examine the ongoing flow of objects and events that pass by the agent. If circumstances that would allow for the satisfaction of the goal arise, the goal itself recognizes them and asserts itself. We share Birnbaum's and Collins' philosophical stance of trying to explain complex opportunistic behavior. However, we disagree that this behavior results from goals constantly monitoring the world. We believe that indexing suspended goals is a better explanation.

5.2 An example of opportunism

It is important to understand the type of behavior we want to capture. We will do this by looking at a simple example:

> On making breakfast for himself in the morning, John realized that he was out of orange juice. Because he was late for work he had no time to do anything about it.
>
> On his way home from work, John noticed that he was passing a Seven-Eleven and recalled that he needed orange juice. Having time, he stopped and picked up a quart and then continued home.

There are a number of interesting aspects to this example. First of all, the planner is confronted with new goals during execution, making complete preplanning impossible. Secondly, the planner is able to stop planning for a goal before deciding exactly how to satisfy it. In effect, he is able to say "I don't have all the information or the time to completely integrate a plan for this goal into my current agenda." Using Schank's vocabulary, we call this the ability to *suspend* a goal [Schank and Abelson, 1977]. And third, although the goal is suspended, the planner is able to recognize the conditions that might lead to its satisfaction.

There is another more subtle element to this example: in order to suspend planning for the goal to possess orange juice, John has to do some reasoning about what a plan for that goal entails. That is, he has to see that the goal is blocked by lack of time to go to the store. As a result, he has a clear idea at planning time what an execution-time opportunity would look like.

In this example, our opportunistic memory algorithm translates into the following:

- John's goal to possess orange juice is blocked by lack of time to run the default plan. He decides, on the basis of the preconditions he knows about, that being at a store would constitute an opportunity to get the orange juice. As a result, he links the suspended goal to the condition of being near a store.
- While coming home, he sees and recognizes a Seven-Eleven. This activates the goal to obtain orange juice that he associated with this condition earlier in the day.
- He then tests the preconditions on the plan and merges it into his current agenda.

In our example, having money, being at a grocery store, and having time are all preconditions for buying orange juice. But there is a difference between them, in that having money is a normative condition and as such does not constitute an opportunity, while being near a store is a non-normative precondition and as such does constitute an opportunity.

First, the planner suspends the blocked goals by associating them with the elements of memory that describe potential opportunities. This requires that the planner have access to a vocabulary that differentiates between the different types of planning problems.

Next, the planner executes the plans for its active goals. During execution, it has to monitor the ongoing effects of its plan as well as the effects of the plans of others in its world. The representational elements used to do this parsing are the same elements with which suspended goals have been associated. As a result, the planner's general recognition of a situation that constitutes an opportunity can immediately activate any goals that have previously been associated with that situation.

Finally, any activated goals are integrated into the current set of scheduled steps, and the plan is executed. This requires reasoning about resources and protections, as well as the effects of actions.

5.2.1 Suspending blocked goals

In general, opportunities to run plans can be derived from the preconditions on each of the steps of a plan. A planner could, given time, move through a plan step by step and collect the preconditions that have to obtain at that point in the plan. But this would require the examination of many conditions that are not particularly useful in the context of opportunism. Some preconditions for obtaining orange juice—having money, having time, and being able to carry the carton—are not useful if we are looking for the features that will allow us to recall the suspended goal at the appropriate time. For example, having money is a strong precondition for buying orange juice, but it is also a normative condition. As a result, it is a bad predictor of an opportunity to satisfy the goal to have orange juice. If the suspended goal is tied to having money, the planner will be reminded of the goal far too often.

Rather than test all preconditions of a plan for these constraints, we propose a taxonomy of *opportunity types* to derive the conditions that will serve as opportunities to satisfy the plan. For a further discussion of this taxonomy, see Hammond *et al* [1988].

In this example, it is possible to associate the blocked goal to possess orange juice with the location, GROCERY-STORE, and the object itself, ORANGE-JUICE. Associating the goal with the least likely conditions with the belief that most of the other conditions will obtain when the suspended goal is activated.

5.2.2 Recalling suspended goals

An agent usually has a wide variety of planning options for any one goal. In our example, John can pick up orange juice at *any* grocery store, not just a particular one. It is necessary, then, to be able to recognize a wide variety of situations as opportunities for goal satisfaction.

To deal with this, we have been using a version of Martin's DMAP parser [Martin, 1990] a general purpose recognition system. DMAP uses a marker-passing algorithm in which two types of markers are used to *activate* and *predict* concepts in an ISA and PART-OF network. *Activation markers* are passed from primitive features up an abstraction hierarchy. When any PART-OF a concept is active, *prediction markers* are spread to its other parts. When a predicted concept is handed an activation marker, it becomes active. Likewise, when all parts of a concept are activated, the concept itself is activated. For our uses, we have added a new type of link to the basic memory stuctures. This link associates suspended goals with concepts that represent opportunities

to achieve them. Pointing from concepts to goals, this SUSPEND link is traversed by any activation marker that is placed on the concept. So, the activation of a concept also activates any suspended goals associated with it.

5.2.3 Exploiting the opportunities

Once a suspended goal is reactivated, it has to be evaluated for integration into the current execution agenda.

In our orange-juice example, the steps required to get the planner to a store can be ignored, in that being *at the store* is the condition that activated the goal in the first place. But the planner can also ignore other steps. In particular, the steps that are used to "recover" from the precondition of being at the store once the plan is over can be ignored. The remaining steps—going into the store, buying the orange juice, and exiting—might be integrated in a fairly traditional way. The planner checks the preconditions not set by the activation conditions, and it notes the use of resources and their interactions with existing protections. The final product is a small change in the overall plan that takes the planner into the store for a moment before resuming his trip home.

5.3 Learning from opportunities

As in the case of execution–time failure, an unexpected opportunity may indicate a chance for learning to improve future performance. Just as failure-driven learning is an alternative to complete projection in debugging of conjunctive goal plans, learning from encountered opportunities presents a method for constructing such plans without complete search of the space of possible plans.

This is best understood in the context of an elaboration of our example: going to the store to buy orange juice. The basic plan for this goal is simple: go into the store, find the orange juice, buy it, and go home. During the execution of this plan a planner will have to move through the store looking for the juice. As he does so, he may see a bottle of milk and recall the need for it. He also may recall that he was out of aluminum foil as well.

At this point he does what any optimizing planner should do: he merges the separate plans for obtaining milk, orange juice and aluminum foil into a single plan for the conjunct of goals. He buys them all at once while at the store rather than buying them one at a time and returning home with each.

We want a planner that will take this experience and use it to form a new plan to pick up milk when it is at the store getting orange juice—without also picking up aluminum foil each time as well. The rationale for this choice of items to be included in this plan is clear. Given the rate of use of orange juice and milk, there is a good chance that at any given moment you may be out of either. Given the rate of use of aluminum foil, however, there is little chance that at any one time you will be out of it.

To do this the planner must face a two-fold task. It must evaluate the likelihood that a similar conjunction will ever arise again - *i.e.*, determine if the plan is worth saving at all and which goals in the initial conjunct should be included. Then it must determine the set of features that predicts the presence of the conjunct. In the language of case-based planning, it must determine how to index the plan in memory. This determination can be made empirically, by trying the new plan when any one of the goals arises and removing links between it and those goals that do not predict the presence of the other goals. Or it can be done analytically, using explanation-based learning methods to construct explanations for why the goals should or should not be expected to arise in concert. Regardless of the reasoning involved in deciding what conjoined plan to save, though, the crucial point is that the possibilities for conjunction are suggested by the world, not by projection.

5.4 An implementation of opportunistic memory: The TRUCKER program

Our first experiments with an implementation of opportunistic memory were in the TRUCKER program. TRUCKER's domain is a UPS-like pickup and delivery task in which new orders are received during the course of a day's execution. Its task is to schedule the orders and develop the routes for its trucks to follow through town. A dispatcher controls a fleet of trucks which roam a simulated city or neighborhood, picking up and dropping off parcels at designated addresses. Transport orders are "phoned in" by customers at various times during the business day, and the planner must see to it that all deliveries are successfully completed.

TRUCKER optimizes its planning for multiple goals only when it notices an opportunity to do so during execution. If TRUCKER notices an opportunity to satisfy a goal that is scheduled later in its agenda, it stops and reasons about the utility of merging the later plan with the steps it is currently running. If it is able to construct a plan that is significantly better than one which treats the plans independently, it uses the new plan. It also stores the new plan in memory, indexed by each of the separate goals. When either goal reoccurs, TRUCKER searches its action queue for for the partner goal and uses the plan that it has created for the pair. Even when a goal is placed on its action queue, TRUCKER treats it as though it were blocked. That is, it establishes the conditions that would allow TRUCKER to satisfy the goal and then associates the goal with the memory structures that would be active during the recognition of those conditions.

TRUCKER's apporach to this task is a serious departure from conventional approaches. Conventional approaches to planning, however, would be inadequate to this task, not just because of the intractability of an optimal solution, but because TRUCKER does not even know all of its goals before it must begin to act. TRUCKER must plan opportunistically, recognizing and acting upon opportunities for goal satisfaction as they arise. We argue further that since planning time is limited, and plan construction is costly, plans should be stored and re-used as much as possible. Finally, patterns of opportunity that are recognized once should be learned, and should be easier to recognize again if they recur.

5.5 Opportunism in TRUCKER

TRUCKER controls its fleet of trucks by deciding which trucks should receive given pickup-and-delivery orders, retrieving or calculating routes for the trucks to follow, and actually monitoring the progress of the trucks (the trucks are better viewed as effectors of the planner than as autonomous agents). TRUCKER's central control structure is a queue-based executor, reminiscent of Firby's RAP system [Firby, 1989], with planning and monitoring actions sharing space on the queue. In addition to planning the agendas for the trucks, and constructing routes for them to follow, the planner must react to new goals as they come in on the "telephone".

```
7:58:00 AM   Planner Action: (ANSWER-TELEPHONE)
7:58:00 AM   Planner Action:
     (HANDLE-NEW-REQUEST REQUEST.41)
- Planner relating request REQUEST.41 to memory -
7:58:00 AM   Planner Action:
(TRY-ASSIGN-TO-IDLE-TRUCK REQUEST.41)
PLANNER assigning request REQUEST.41 to truck #1
```

In absence of good reasons to the contrary, the planner hands pickup and dropoff orders to trucks based on availability in the order they come in.

```
Starting INTEGRATE-REQUEST  REQUEST.41  #1  IDLE
Resulting new plan:
 ((GOTO (800 E-61-ST))
         (PICKUP PARCEL.42 (850 E-61-ST))
         (GOTO (6200 S-COTTAGE))
         (DROPOFF PARCEL.42 (6230 S-COTTAGE)))
```

When TRUCKER receives a new request for a pickup and delivery, it attempts to satisfy the order using a variety of methods. First it checks all active requests on its truck's agendas for one that has a known positive interaction with the new request. If this fails, TRUCKER attempts to find a truck that is currently idle to take up the order. If this also fails, TRUCKER searches for a suspended request that might be usefully combined with the new order. If all else fails, TRUCKER is forced to place the request on a queue of orders waiting for idle trucks and must construct a new route for the truck using its map and current information about the available trucks.

```
7:58:39 AM
   *** Truck #2 making delivery at 1450 E-62-ST ***
   - Planner searching memory for route from
             (6100 S-WOODLAWN) to (800 E-61-ST) -
    Search unsuccessful.
   ========   PLANNER consulting map
  to build route  ========
8:03:12 AM   *** Truck #1 has a new route:   ***
             ((START N 6100-S-WOODLAWN)
               (TURN W E-61-ST)
                       (STOP 800-E-61))
8:03:12 AM   *** Truck #1 is starting new route at
 6100 block of S Woodlawn ***
8:05:48 AM   *** Truck #1 making pick-up at
 850 E-61-ST ***
*** Truck #2 done with delivery ***
```

Whenever TRUCKER is forced to construct a new route from scratch, considers the goal that is planned for to be blocked, and thus suspends it. To suspend a goal, TRUCKER marks its representation of the goal's pickup and delivery points with an annotation that there is a goal related to those locations. TRUCKER ties execution of actions to locations, landmarks and addresses that they recognize in the world. It must parse and interpret the objects in the world. It is during this parse that TRUCKER recognizes and recalls previously suspended goals. A typical TRUCKER plan, when fully expanded, is a route in the form of a list of the turns that have to be made, described in terms of street names and compass directions. So the plan step `(GOTO (920 E-55th))` after a pick-up at `(5802 S-WOODLAWN)` expands into:

```
(START NORTH (5802 S-WOODLAWN))
(TURN EAST E-57TH)
(TURN NORTH S-CORNELL)
(TURN EAST E-55TH)
(STOP (920 E-55TH))
```

As TRUCKER moves through its world, it parses the objects at its current location and responds to any changes that the tokens it has recognized suggest: turning, for example, when it recognizes the 5700 block of Woodlawn. It also checks the token for any annotation of a goal that might be associated with it. If one is found, TRUCKER activates the suspended goal and attempts to integrate it into the current schedule. This allows TRUCKER to easily and effectively activate suspended goals when the opportunities to satisfy them arise. The same memory for places and landmarks that is used to tell the trucks when to turn and where to stop is annotated with the delivery goals that have not yet been satisfied. When such a location is recognized in the course of executing another delivery, the possibility of opportunistically satisfying the goal is suggested.

```
8:17:12 AM   *** Truck #1 is starting new route at
               800 block of E 61st Street ***
*** Truck #1 has noticed an opportunity to make the
    pickup for request REQUEST.49 ***
*** Request REQUEST.49 is assigned -- Truck
     #1  inserting reassignment request in
     planner's agenda.
*** Noting combination opportunity in memory ***
8:17:45 AM   Planner Action:
   (REASSIGN-BY-NOTICED-OPPORTUNITY  REQUEST.49 #1)
- Reassignment of request REQUEST.49 means that
    truck #2 need not continue to its destination.
PLANNER assigning request REQUEST.49 to truck #1
Starting INTEGRATE-REQUEST
Current plan:
  ((GOTO (6200 S-COTTAGE) #{Structure ROUTE 2})
   (DROPOFF PARCEL.42 (6230 S-COTTAGE)))
Request-plan to integrate:
  ((GOTO (6100 S-COTTAGE))
   (PICKUP PARCEL.50 (6150 S-COTTAGE))
   (GOTO (900 E-63-ST))
   (DROPOFF PARCEL.50 (925 E-63-ST)))
8:17:45 AM   *** Truck #1 is stopping in
         6100 block of S Cottage Grove ***
Finishing INTEGRATE-REQUEST
Resulting new plan:
  ((GOTO (6100 S-COTTAGE))
   (PICKUP PARCEL.50 (6150 S-COTTAGE))
   (GOTO (6200 S-COTTAGE))
   (DROPOFF PARCEL.42 (6230 S-COTTAGE))
   (GOTO (900 E-63-ST))
   (DROPOFF PARCEL.50 (925 E-63-ST)))
```

Once a suspended goal is reactivated, it has to be evaluated for integration into the current execution agenda.

Here TRUCKER uses special-purpose techniques tailored to the domain. When a suspended goal is recalled by the planner, it attempts to find the best placement in the current route for the awakened request. Scheduling the pickup is trivial, in that a truck is at the pickup location. The difficulty lies in scheduling the delivery. TRUCKER does this by stepping through each location already scheduled and finding the section of the route that will be the least altered by the insertion of the delivery. In this way the full planner is invoked only when the recognition of an opportunity to satisfy a pending goal suggests that the combination of delivery orders may be fruitful.

5.6 Learning in TRUCKER

There is considerable regularity and repetition in the orders that the world simulation hands to TRUCKER. TRUCKER exploits this in a case-based manner, by saving particular constructed routes, remembering conjuncts of requests that have been profitably combined, and remembering the particular interleavings of steps that these conjuncts produced. When the conjuncts of goals reoccur, TRUCKER recognizes them as a known conjunct of goals for which it has a plan and uses the plan for that conjunc that it has saved in memory. When TRUCKER receives a request, the request's long-term record is inspected to see if there are any notations about combinations with other requests. If so, the planner looks to see if the other requests are currently active. If it finds the requests that it has previously been able to merge with the current one, the plan for the conjunction is added to the agenda rather than those for the individual requests. It is important to note that plans for combination of requests are only activated when all the requests are active.

```
[Day#2]
8:17:00 AM   Planner Action: (ANSWER-TELEPHONE)
- Planner relating request
    REQUEST.73 to memory -
8:17:00 AM   Planner Action:
(TRY-ASSIGN-TO-USEFUL-TRUCK REQUEST.73)
 - Truck #1 is pursuing a request that has been
previously associated with route of request
 REQUEST.73
 PLANNER assigning request REQUEST.73 to truck #1
Starting INTEGRATE-REQUEST
Current plan:
  ((GOTO (6200 S-COTTAGE))
   (DROPOFF PARCEL.72 (6230 S-COTTAGE)))
Request-plan to integrate:
  ((GOTO (6100 S-COTTAGE))
   (PICKUP PARCEL.74 (6150 S-COTTAGE))
   (GOTO (900 E-63-ST))
   (DROPOFF PARCEL.74 (925 E-63-ST)))
Resulting new plan:
  ((GOTO (6100 S-COTTAGE))
   (PICKUP PARCEL.74 (6150 S-COTTAGE))
   (GOTO (6200 S-COTTAGE))
   (DROPOFF PARCEL.72 (6230 S-COTTAGE))
   (GOTO (900 E-63-ST))
   (DROPOFF PARCEL.74 (925 E-63-ST)))
```

This automatic combination of requests that have been joined in the past will obviously not necessarily lead to an optimal assignment of requests to trucks, at least in a sense of optimality that ignores the cost of the work done in arriving at the assignment. There is an side benefit to this standardization, however, which is that using step interleavings that were previously calculated means that the routes between internal points of the schedule can also be reused.

```
========   PLANNER consulting map
           to build route  ========
8:17:12 AM   *** Truck #1 has a new route:   ***
             ((START W 800-E-61)
              (TURN S S-COTTAGE)
              (STOP 6100-S-COTTAGE))
8:17:12 AM   *** Truck #1 is starting new route at
               800 block of E 61st Street ***
8:17:39 AM   *** Truck #1 making pick-up at
6150 S-COTTAGE ***
   - Planner searching memory for route from
        (6100 S-COTTAGE) to (6200 S-COTTAGE) -
    Search successful.
   ========   PLANNER knows route from
6100S-COTTAGE to 6200S-COTTAGE ========
```

Given some regularity in the orders it receives, TRUCKER builds a library of planned routes, and a library of conjoined plans for groups of requests that have occurred together. Learning from these encountered opportunities advances TRUCKER's background goal of learning about interesting regularities in its environment, and helps amortize the complexity over the long term of satisfying its goals.

Our discussion up to now has focussed on how an agent can learn about regularities in its world, both favorable and unfavorable, and change its long-term behavior in response to them. In the next section we turn to the question of how an agent can *create* such regularities and profit from them.

6 Enforcement and the stabilization of environments

There is a direct relationship between the overall stability of an environment and our ability to predict and plan within it. The greater the stability, the more certain our predictions; and the more certain our predictions, the more powerful our plans.

Both as individuals and as societies, we respond to this by trying to increase the stability of our world. We segment our schedules of work, play and relaxation so that each day will tend to look very much like the last. We organize our homes and workspaces so that objects will be in predictable places. We even organize our habits so that particular conjuncts of goals will tend to arise together. In all aspects of our lives, we make moves to stabilize our different worlds.

In this section, we discuss this concept of *enforcement* and examine a few of the different forms that it takes. In particular, we outline a basic taxonomy of classes of stability and presents the strategies for increasing overall stability that are associated with each class. We examine its relationship to learning and argue that both learning and enforcement are strategies for building up a correspondence between an agent's mental model of the world and the actual physical reality. We also dis-

cuss the learning and planning trade-offs that have to be made when stability is optimized.

6.1 Opportunism and enforcement: An example

Recall for a moment the example we discussed in the last section involving an agent going to the grocery store to pick up a quart of orange juice and noticing that he needs milk as well. One element of this process that interests us is the notion that the more likely it is that that goals will show up in conjunction with each other, the more useful the plan will be. In this example, the utility of saving and attempting to reuse the plan to buy both the orange juice and the milk is maximized when the two goals are guaranteed to show up in conjunction whenever either of the two recurs. This suggests the idea that one of the steps that an agent could take in improving the utility of his plans would be to force the recurrence of the conjuncts of goals over which these plans are optimized. In terms of the orange juice and milk example, this means making sure that the cycles of use of each resource are synchronized. This can be done by either changing the actual use of the resources to bring them into synchronization or by changing the amounts purchased such that they would be used up at the same time. In either case, the idea is to alter circumstances in the world such that the long term utility of a plan that already exists is optimized. This is done by stabilizing the world with regard to the relative use of the two resources. This type of enforcement is aimed at controlling what we call RESOURCE CYCLE SYNCHRONIZATION in that its goal is to stabilize the use cycles of multiple resources with respect to one another.

Adjusting the amount of orange juice purchased so makes cycle of use match the cycle of use of the milk. This increases the utility of the plan to buy the two together in three ways: optimization of planning, optimization of indexing, and optimization of execution.

- In terms of planning optimization, the agent now has available a plan for a conjunct of goals that he knows will recur so he never needs to recreate it. This means never having to reconstruct the GET-ORANGE-JUICE-AND-MILK plan again.
- And in terms of indexing optimization, the plan can be indexed by each of the elements of the conjunct—rather than by the conjunct itself—thus reducing the complexity of the search for the plan in the presence of the individual goals. This means that the plan will be automatically suggested when either the HAVE-MILK goal or the HAVE-ORANGE-JUICE goal arises even when the other element of the goal conjunct does not.
- In terms of execution optimization, the agent can decide to commit to and begin execution of the new plan when either of the two goals arises. It can do this because it is able to predict that the other goal is also present, even if it is not explicitly so. This means that the agent can begin to run the GET-ORANGE-JUICE-AND-MILK plan when he notices that he is out of either milk or orange juice without being forced to verify that the other goal is active.

One way of viewing enforcement is as an extension of planning itself. As in planning, the conditions that are enforced are fixed in the world using the same sorts of actions that result in the satisfaction of goals. The difference is that the actions associated with enforcement result in changes to the actual structure of a domain.

Likewise, enforcement can be seen as an active cousin of learning. Just as learning techniques in planning are designed to build up an effective set of plans and operators for a domain, enforcement techniques are designed to do so as well. The difference here is that learning attempts to satisfy this goal by changing the learner and enforcement attempts to do so by changing the world.

6.2 Stability and enforcement

While RESOURCE CYCLE SYNCHRONIZATION was one of the first instances of stability we encountered, it is by no means the only kind. In the sections that follow, we present two other basic types of stability and related enforcement strategies.

The question is, is it possible to explicate this taxonomy of stability in a way that would allow a system to actually recognize and enforce the different types? The following sections outline this taxonomy with respect to this question by breaking each type down in terms of the following issues:

- **What types of stability are useful in and of themselves?**
- **Over what goals do they allow optimization?**
- **What strategies can be formed to enforce them?**
- **How can opportunities to apply these enforcement strategies be recognized?**

6.2.1 Stability of location

The most common type of stability that arises in everyday activity is that of location of commonly used objects. Our drinking glasses end up in the same place every time we do dishes. Our socks are always together in a single drawer. Everything has a place and we enforce everything ending up in its place.

Enforcing STABILITY OF LOCATION serves to optimize a wide range of processing goals. First of all, the fact that an often used object or tool is in a set location reduces the need for any inference or projection concerning the effects of standard plans on the objects or the current locations of objects. Second, it allows plans that rely on the objects locations to be run without explicit checks (e.g., no need to explicitly determine that the glasses are in the cupboard before opening it). Third, it removes the need at execution-time for a literal search for the object.

The final question in terms of STABILITY OF LOCATION, then, is the issue of when to attempt enforcement. As in many instances of standard learning, failure is a good indicator. Here, the problem will take the form of an execution-time failure to actually find an object that is both known to exist and is a object essential to a plan being run.

6.2.2 Stability of cues

One effective technique for improving plan performance is to improve the proper activation of a plan rather than improve the plan itself. For example, placing an important paper that needs to be reviewed on his desk before going home, improves the likelihood that an agent will see and read it the next day. Marking calendars and leaving notes serves the same sort of purpose.

One important area of enforcement is related to this use of visible cue in the environment to activate goals that have been suspended in memory. The idea driving this type of enforcement is that an agent can decide on a particular cue that will be established and maintained so as to force the recall of commonly recurring goals. One example of this kind of enforcement of STABILITY OF CUES is leaving a briefcase by the door every night in order to remember to bring it into work. The cue itself remains constant over time. This means that the agent never has to make an effort to recall the goal at execution-time and, because the cue is stabilized, it also never has to reason about what cue to use when the goal is initially suspended. The advantage of this sort of enforcement is that an agent can depend on the external world to provide a stable cue to remind it of goals that still have to be achieved. This sort of stability is suggested when an agent is faced with repeated failures to recall a goal and the plan associated with the goal is tied to particular objects or tools in the world.

These are only three of the types of stability and enforcement that we have uncovered. For a broader discussion of these and other instances of stability, see Hammond [1990].

6.3 The point

In order to plan at all in an environment, it must at least be stable with respect to its basic physics. In order to reuse plans in any interesting way at all, the environment—including the agent—must be stable with respect to other aspects as well. In particular, it must be stable with regard to the physical structure of the environment, the goals that tend to recur and the times at which events tend to take place. While many environments have this sort of stability, it is often the product of the intervention of agents attempting to stablize it so as to increase the utility of their own plans. The goal of this enforcement parallels the goal of learning—the development of a set of effective plans that can be applied to satisfy the agent's goals. The path toward this goal, however, is one of shaping the world to fit the agent's plans rather than shaping the agent to fit the world.

7 A Model of Agency

These four elements, case-based planning, learning from failure, learning from opportunity and enforcement of stability are now coming together in an overall model of planning and action. This model of agency rises out of three pieces of work: Schank's structural model of memory organization [Schank, 1982] our own work in case-based planning and dependency directed repair [Hammond, 1989], and the work of Martin and Riesbeck in Direct Memory Access Parsing [Martin, 1990]. The model is currently under development in the RUNNER program.

During execution, the agent performs an ongoing "parse" of the world in order to recognize conditions for action execution. Following DMAP [Martin, 1990], this parse takes the form of passing markers through an existing episodic memory. Because suspended goals are indexed in the memory used for understanding the world, the goals are activated when the conditions favoring their execution are recognized. Once active, the goals are then reevaluated in terms of the new conditions. Either they fit into the current flow of execution or they are again suspended.

In TRUCKER, and later in RUNNER, we tried to address the specific problem of recognizing execution-time opportunities. We now use the term *agency* to comprise the spawning of goals, selection of plans, and execution of actions. Our process model of agency is based on Martin's DMAP understander [Martin, 1990]. DMAP uses a memory organization defined by part/whole and abstraction relationships. Activations from environmentally supplied features are passed up through abstraction links and predictions are passed down through the parts of partially active concepts. Subject to some constraints, when a concept has only some of its parts active, it sends predictions down its other parts. When activations meet existing predictions, the node on which they meet becomes active. Finally, when all of the parts of a concept are activated, the concept itself is activated. The architecture provides a computational mechanism for specifying and applying domain-dependent information to a general memory search process. (For an exposition of the DMAP architecture, see [Martin, 1990]).

To accommodate action, we have added the notion of PERMISSIONS. PERMISSIONS are handed down the parts of plans to their actions. The only actions that can be executed are those that are PERMITTED by the activation of existing plans. Following McDermott [Mcdermott, 1978], we have also added POLICIES. POLICIES are statements of ongoing goals of the agent. Sometimes these take the form of enforcement goals, such as "Glasses should be in the cupboard." or "Always have money on hand." The only goals that are actively pursued are those generated out of the interaction between POLICIES and environmental features. We would argue that this is, in fact, the only way in which goals can be generated.

7.1 Goals, plans, and actions

Goals, plans, and actions interact as follows:

- Features in the environment interact with POLICIES to spawn goals.

  For example, in RUNNER, the specific goal to HAVE COFFEE is generated when the system recognizes that it is morning. The goal itself rises out of the recognition of this state of affairs in combination with the fact that there is a policy in place to have coffee at certain times of the day.

- Goals and environmental features combine to activate plans already in memory.

  Any new MAKE-COFFEE plan is simply the activation of the sequence of actions associated with the

existing MAKE-COFFEE plan in memory. It is recalled by RUNNER when the HAVE-COFFEE goal is active and the system recognizes that it is at home.

- Actions are **permitted** by plans and are associated with the descriptions of the world states appropriate to their performance. Once a set of features has an action associated with it, that set of features (in conjunct rather than as individual elements) is now predicted and can be recognized.

  Filling the coffee pot is **permitted** when the MAKE-COFFEE plan is active; it is associated with the features of the pot being in view and empty. This means not only that the features are now predicted but also that their recognition will trigger the action.

- Actions are specialized by features in the environment and by internal states of the system. As with Firby's RAPs [Firby, 1989], particular states of the world determine particular methods for each general action.

  For example, the specifics of a GRASP would be determined by information taken from the world about the size, shape and location of the object being grasped.

- Action level conflicts are recognized and mediated using the same mechanism that recognizes information about the current state of the world.

  For example, when two actions are active (such as filling the pot and filling the filter), a mediation action selects one of them. During the initial phases of learning a plan, this can in turn be translated into a specialized recognition rule which, in the face of a conflict, will always determine the ordering of the specific actions.

- Finally, suspended goals are associated with the descriptions of the states of the world that are amenable to their satisfaction.

  For example, the goal HAVE-ORANGE-JUICE, if blocked, can be placed in memory, associated with the conjunct of features that will allow its satisfaction, such as being at a store, having money and so forth. Once put into memory, this conjunct of features becomes one of the set that can now be recognized by the agent.

7.2 The study of agency

We do not see this model as a solution to the problems of planning and action. Instead, we see this as a framework in which to discuss exactly what an agent needs to know in a changing world. Advantages of this framework include:

1. A unified representation of goals, plans, actions and conflict resolution strategies.
2. Ability to learn through specialization of general techniques.
3. A fully declarative representation that allows for meta-reasoning about the planner's own knowledge base.
4. A simple marker-passing scheme for recognition that is domain and task neutral.
5. Provision for the flexible execution of plans in the face of a changing environment.

The basic metaphors of action as permission and recognition, and planning as the construction of descriptions that an agent must recognize prior to action, these fit our intuitions about agency. Under this metaphor, we can view research into agency as the exploration of the situations in the world that are valuable for an agent to recognize and respond to.

8 An Implementation of Agency: RUNNER

Most of our work in studying this architecture has been within the context of the RUNNER system. The RUNNER project is aimed at modeling the full spectrum of activity associated with an agent—goal generation, plan activation and modification, action execution, and resolution of plan and goal conflict—not just the more traditional aspect of plan generation alone.

8.1 RUNNER's world

The agent in RUNNER currently resides in a simulated kitchen, and is concerned with the pursuit of such goals as simulated breakfast and coffee. Such commonplace goals and tasks interest us in part because they are repetitive and have many mutual interactions, both negative and positive. We are interested in how plans for recurring conjuncts of goals may be learned and refined, as part of view of domain expertise as knowledge of highly specific and well-tuned plans for the particular goal conjuncts that tend to co-occur in the domain [Hammond *et al.*, 1988]. We are also interested in the issue of exactly how these plans can be used in the guidance of action.

8.2 RUNNER's representation

The knowledge and memory of the agent is captured in the conjunction of three types of semantic nets, representing knowledge of goals, plans and states. Nodes in these networks are linked by abstraction and packaging links, as in DMAP. In addition, we propose an additional SUSPEND link, which connects suspended goals to state descriptions that may indicate opportunities for their satisfaction. For example, the goal to have eggs would be suspended in association with the description of the agent being at a grocery store. In addition to being passed to abstractions of activated concepts, activation markers are always passed along SUSPEND links.

In general, the only other way in which these nets are interconnected is via *concept sequences.* A node may be activated if all of the nodes in one of its concept sequences is activated – a concept sequence for a given node can contain nodes from any of the parts of memory. The following is a partial taxonomy of the types of concept sequences we currently allow:

- Activation of a goal node can activate a node representing a default plan.
- Activation of a plan node and some set of state nodes can activate a further specialization of the plan.

- Activation of a goal node and some set of state nodes can activate a further specialization of the goal.
- Activation of any state node that has a SUSPEND link will activate the associated goal.

8.3 An example: Making coffee

The above discussion of representation may make more sense in the context of an example, currently implemented in RUNNER, of how a particular action is suggested due to conjunction of plan activation and environmental input.

One of the objects in RUNNER's simulated kitchen is a coffeemaker, and one of the plans it has available is that of making coffee with this machine. This plan involves a number of subsidiary steps, some of which need not be ordered with respect to each other. Among the steps that are explicitly represented in the plan are: fetching unground beans from the refrigerator, putting the beans in the grinder, grinding the beans, moving a filter from a box of filters to the coffeemaker, filling the coffeemaker with water from the faucet, moving the ground beans from the grinder to the coffeemaker, and turning the coffeemaker on.

The subplans of the coffee plan are associated with that plan via packaging links. In this implemented example, the agent starts out with a node activated which represents knowledge that it is morning. This in turn is sufficient to activate the goal to have coffee (this is as close as the program comes to a theory of addiction). This goal in turn activates a generic plan to have coffee. This turns out to be nothing but an abstraction of several plans to acquire coffee, only one of which is the plan relevant to our kitchen:

"Visual" input, in terms of atomic descriptions of recognizable objects and their proximities, is passed to memory. For example, the agent "sees" the following visual types:

```
countertop, white wall, box of filters
```

Among sets of possible visually recognized objects are concept sequences sufficient for recognition that the agent is in the kitchen. The recognition of the white wall and the countertop completes one of these sequences. The "kitchen" node in turn passes activation markers to its abstractions, activating the node corresponding to the agent being at home:

The activation of this node in conjunction with the activation of the generic coffee goal completes the concept sequence necessary for the goal for making coffee at home, which in turn activates the default plan for that goal. In this way a specialized plan is chosen in response to a conjunction of a recognized state and a more generic goal:

```
MEMORY:
concept sequence
    ([GOAL: drink-coffee] [at-home])
for node
     [GOAL: drink-coffee-at-home] is completed.
sending activation marker to
     [GOAL: drink-coffee-at-home]
Activating concept:
     [GOAL: drink-coffee-at-home]
Asserting new goal:
     [GOAL: drink-coffee-at-home]
sending activation marker to
     [PLAN: make-coffee-at-home]
Node [PLAN: make-coffee-at-home]
has both permission and activation:
     ((MARKER [GOAL: drink-coffee-at-home]))
     (TOP-LEVEL-PLAN)
Activating concept:
     [PLAN: make-coffee-at-home]
```

The activation of the coffee-plan causes permission markers to be sent down packaging links to the nodes representing the parts of the plan. The activation of the other object concepts from the "visual" input in turn have sent activation markers to the actions that contain them in their concept sequences. Among these is the plan step for taking a filter from the box and installing it in the coffeemaker, which is activated by seeing box of filters itself. In this way a sub-plan is suggested by the intersection of permission from its parent plan and cues from the environment that indicate that it is easily satisfiable:

```
Asserting new plan:
  [PLAN: make-coffee-at-home]
Sending permissions to steps of plan
Sending permission markers from
  [PLAN: make-coffee-at-home]
to steps
  FILL-CARAFE  PUT-BEANS-IN-GRINDER
  MOVE-GROUNDS-TO-COFFEE-MAKER
  TURN-ON-COFFEE-MAKER  GRIND-BEANS
  PUT-IN-FILTER  GET-COFFEE-BEANS
concept sequence
  ([filter-box]
   [PLAN: make-coffee-at-home])
for node [PLAN: put-in-filter] is completed.
sending activation marker to
  [PLAN: put-in-filter]
Node [PLAN: put-in-filter]
has both permission and activation:
  ((MARKER ([filter-box]
            [PLAN: make-coffee-at-home])))
  ((MARKER [PLAN: make-coffee-at-home]))
Activating concept:
  [PLAN: put-in-filter]
Asserting new plan: [PLAN: put-in-filter]
Sending permissions to steps of plan
Sending permission markers from
  [PLAN: put-in-filter]
to steps
  PUT-FILTER-IN-COFFEEMAKER GET-FILTER
concept sequence
  ([filter-box] [PLAN: put-in-filter])
for node [PLAN: get-filter] is completed.
sending activation marker to
     [PLAN: get-filter]
Node [PLAN: get-filter]
has both permission and activation:
     ((MARKER ([filter-box]
               [PLAN: put-in-filter])))
     ((MARKER [PLAN: put-in-filter]))
Activating concept: [PLAN: get-filter]
```

After another level of passing permission markers to sub-plans, the process "bottoms out" in the suggestion of the primitive action of picking up the box of filters. With no suggestions to the contrary, the action is taken:

```
Asserting new plan:
```

```
     [PLAN: get-filter]
Sending permissions to steps of plan
Sending permission markers from
     [PLAN: get-filter]
to steps
   TAKE-OUT-FILTER    PICK-UP-BOX
   LOOK-FOR-FILTER-BOX
concept sequence
     ([filter-box] [PLAN: get-filter])
for node [PLAN: pick-up-box] is completed.
sending activation marker to
     [PLAN: pick-up-box]
Node [PLAN: pick-up-box]
has both permission and activation:
   ((MARKER ([filter-box] [PLAN: get-filter])))
   ((MARKER [PLAN: get-filter]))
Activating concept: [PLAN: pick-up-box]
Suggesting action: (GRASP 'FILTER-BOX)
-----------------------------------------------
ACTION:
Performing action: (GRASP 'FILTER-BOX)
-----------------------------------------------
To the left is a countertop, up close
To the right, there's a countertop, up close
Straight ahead, there's a countertop, up close
Result of action: I'm holding onto a filter-box
-----------------------------------------------
```

The final action is chosen both on the basis of active plans and goals, and in response to the immediate circumstances in which the agent finds itself. Given a change in either the top-down guidance or the bottom-up recognition, the selection of plan and action will change in response.

9 Conclusion

Intelligence is an ongoing process. It does not begin and end with one example. It is not exercised through tricks or puzzles. It is instead a constant battle to sometimes find, often establish, and eventually exploit the order that lies within the rich complexity of the natural world.

Our study of agency rests on this idea that intelligent behavior is a long-term activity and that much of it is aimed at learning and enforcing order within a domain. Our central premise is that the stability of the world includes stability over the collections of goals that we will be called upon to satisfy, the types of difficulties we will encounter and the kinds of conditions that we will be forced to overcome. It is only by learning or enforcing these, that an agent can develop a true expertise in any domain. And it is only through an ongoing attempt to satisfy goals in a dynamic world that opportunites for learning and enforcement can ever be encountered. In this paper, we have tried to extend the core idea of case-based plannning to include learning from execution-time failure and opportunity, in an effort to also extend the kind of order that a system can learn and utilize. Likewise, we have proposed the idea of enforcement as a means to impose order in those situations where it can help the system in the later application of its own plans. In all, we have tried to present a picture of an agent that is able to cope with the complexity of its environment by learning and using the order that lies within it.

References

[Birnbaum and Collins, 1984] L. Birnbaum and G. Collins. Opportunistic planning and freudian slips. In *Proceedings of the Sixth Annual Conference of the Cognitive Science Society*, Boulder, CO, 1984.

[Fikes and Nilsson, 1971] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, 1971.

[Firby, 1989] R. J. Firby. Adaptive execution in complex dynamic worlds. Research Report 672, Yale University Computer Science Department, 1989.

[Hammond *et al.*, 1988] K. J. Hammond, T. Converse, and M. Marks. Learning from opportunities: Storing and reusing execution-time optimizations. In *Proceedings of the Seventh Annual Conference on Artificial Intelligence*, pages 536–40. AAAI, 1988.

[Hammond, 1989] K. Hammond. *Case-Based Planning: Viewing Planning as a Memory Task.* Academic Press, 1989.

[Hammond, 1990] K. Hammond. Learning and enforcement: Stabilizing environments to facilitate activity. In B. Porter and R. Mooney, editors, *Proceedings of the Seventh International Conference on Machine Learning*, Austin, Texas, 1990. Morgan Kaufmann Publishers, Inc.

[Hayes-Roth and Hayes-Roth, 1979] B. Hayes-Roth and F. Hayes-Roth. A cognitive model of planning. *Cognitive Science*, 3(4):275–310, 1979.

[Kolodner and Simpson, 1984] J. Kolodner and R. Simpson. Experience and problem solving: A framework. In *Proceedings of the Sixth Annual Conference of the Cognitive Science Society*, Boulder, CO, August 1984.

[Martin, 1990] C. E. Martin. *Direct Memory Access Parsing.* PhD thesis, Yale University Department of Computer Science, 1990.

[Mcdermott, 1978] D. Mcdermott. Planning and acting. *Cognitive Science*, 2:71–109, 1978.

[Owens, 1990] C. Owens. *Indexing and Retrieving Abstract Planning Knowledge.* PhD thesis, Yale University Department of Computer Science, 1990. In preparation.

[Sacerdoti, 1975] E. D. Sacerdoti. A structure for plans and behavior. Technical Report 109, SRI Artificial Intelligence Center, 1975.

[Schank and Abelson, 1977] R. C. Schank and R. Abelson. *Scripts, Plans, Goals and Understanding: An Inquiry into Human Knowledge Structures.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1977.

[Schank, 1982] R. Schank. *Dynamic memory: A theory of learning in computers and people.* Cambridge University Press, 1982.

[Wilensky, 1983] R. Wilensky. *Planning and Understanding: A Computational Approach to Human Reasoning.* Addison-Wesley Publishing Company, Reading, MA, 1983.

Learning Steppingstones for Problem Solving

David Ruby*
Dennis Kibler
Information & Computer Science
University of California, Irvine
Irvine, CA 92717 U.S.A.
druby@ics.uci.edu

Abstract

One goal of Artificial Intelligence is to develop and understand computational mechanisms for solving difficult real-world problems. Unfortunately, domains traditionally used in general problem solving research lack important characteristics of real-world domains, making it difficult to apply the techniques developed. Most classic AI domains require satisfying a set of Boolean constraints. Real-world problems require finding a solution that meets a set of boolean constraints and performs well on a set of real-valued constraints. In addition, most classic domains are static while domains from the real world change. In this paper we demonstrate that SteppingStone, a general learning problem solver, is capable of solving problems with these characteristics. SteppingStone heuristically decomposes a problem into simpler subproblems, and then learns to deal with the interactions that arise between the subproblems. In lieu of an agreed upon metric for problem difficulty, we choose significant problems which are difficult for both people and programs as good candidates for evaluating progress. Consequently we adopt the domain of logic synthesis from VLSI design to demonstrate SteppingStone's capabilities.

1 Introduction

Problem solving research in AI attempts to discover and understand general computational mechanisms for solving problems. Traditional planning research has focused on finding general mechanisms and demonstrating them in classic AI problem domains. This approach has been used to develop the general techniques of nonlinear planning [Sacerdoti, 1977, Tate, 1977, Vere, 1983, Wilkins, 1984], hierarchical planning [Sacerdoti, 1974, Rosenschein, 1981, Stefik, 1981], and learning from planning experience [Fikes *et al.*, 1972, Porter and Kibler, 1986, Minton, 1988, Ruby and Kibler, 1988, Laird *et al.*, 1986]. Unfortunately, these traditional problem domains lack important characteristic of real-world problems. Real-world problems require solutions that are optimized for real-valued performance constraints as well as satisfying Boolean constraints. In addition, real-world problems have large search spaces and subgoals with strong interactions. Traditional domains have only required meeting Boolean constraints. These differences in the character of the problems have made it difficult to transfer research results to real-world problems.

While general problem solvers have largely ignored real-world problems, applications research has produced domain-specific systems for solving these types of problems [Ow *et al.*, 1988, Lin and Gajski, 1988, Zanden and Gajski, 1988]. The success of application systems depend on encoding large amounts of domain-specific knowledge. One approach for acquiring this knowledge is to encode it by hand. This approach is costly both in time and human resources. A better approach is to use a learning problem solver. A learning problem solver acquires the appropriate knowledge by abstracting from its problem solving experience.

In this paper, we demonstrate that SteppingStone is a general learning problem solver for real-world problems. We show that by decomposing a problem into subproblems and learning to deal with interactions that arise between them, SteppingStone scales to difficult real-world problems. In the following sections we outline the SteppingStone approach and examine how it applies to important real-world problems, such as logic synthesis from VLSI design. We demonstrate SteppingStone's capabilities with empirical results.

2 Steppingstones for Problem Solving

SteppingStone [Ruby and Kibler, 1989] uses a means-ends analysis component to break a problem into subgoals. SteppingStone initially assumes that the ordered subgoals are independent and can be solved without undoing them once solved. Although an entire problem cannot usually be solved with this assumption, some problem subgoals usually can be solved.

An *impasse* occurs when a subgoal is encountered that cannot be solved under the independence assumption. SteppingStone then searches its knowledge base for a new sequence of subgoals we call *steppingstones*. Steppingstones allow the problem solver to pursue a different set of subgoals then that suggested by means-ends anal-

*This work was partially supported by a grant from the Hughes Artificial Intelligence Center.

ysis. These steppingstones comprise the domain-specific knowledge that the system learns. If steppingstones for resolving an impasse do not exist, the system resorts to a localized brute-force search. This search is anchored at the impasse state and is only used to resolve the current impasse. SteppingStone generalizes the solution found for the impasse to generate additional steppingstones for resolving similar impasses.

The problem-solving control knowledge acquired in SteppingStone is organized as sequences of subgoals (*steppingstones*) for resolving impasses. A sequence of subgoals consists of an ordered set of partial state descriptions, or subgoals. Means-ends analysis uses these subgoals as steppingstones to lead it through the impasse. These steppingstones are indexed by the subgoal difference they reduce and the previously solved subgoals that are undone and resolved. After following a sequence of steppingstones, any previously solved subgoals remain solved and the subgoal difference generating the impasse is reduced.

Steppingstones reduce the distance between problem states and goal states by introducing intermediate steps. Since the steppingstones are only used when impasses are encountered, they do not increase the branching factor of the problem. Since steppingstones are domain-specific sequences of subgoals that lead from one partial state description to another partial state description, they are not tied to either the initial state or the final goal. Consequently steppingstones naturally apply to different problems. This flexibility allows steppingstones to achieve generalizations that are impossible for fixed sequences of operations.

An initial version of SteppingStone was implemented and applied to the classic tile-sliding domain [Ruby and Kibler, 1989]. Although, this is a difficult and classic domain in AI, it lacks many important characteristics of real-world problems. This paper describes extensions to SteppingStone that allow it to solve more realistic problems. In the following sections we outline some characteristics of real-world problems and describe how SteppingStone will operate on this new class of problems. We then describe how the logic synthesis task of VLSI design is an example domain for this new class of problems.

2.1 Steppingstones for Optimization

Real-world domains are characterized by large search spaces, many subgoal interactions, and a variety of constraints. Real-world domains are difficult for people because there are no general heuristics that allow solving them. They require experience with the domain to become proficient. In addition, real-world problems change over time and any approach for solving them must be able to adapt to these changes.

Constraints used in real-world problems fall into two classes. Hard constraints must be met and usually outline key aspects of the problem. Soft constraints measure the quality of a solution. Soft constraints are often real-valued performance measures and meeting them is an optimization task. SteppingStone learns to optimize soft real-valued constraints as well as solve hard constraints.

We view problem solving as moving to states where

```
Order Problem Subgoals g(1),...,g(n) using Openr
For i=1 to n do
  While g(i) unsolved do
    Apply MEA without undoing g(1),...,g(i-1)
      If MEA fails to solve g(i)
        Then Apply Steppingstones from Memory
        ElseIf Steppingstones Fail
          Then Apply Local Search
            If Local Search finds Improvement
              Then Learn New Steppingstones
              ElseIf Search Fails and g(i) is Sc
                Then Assume g(i) Solved
                Else Fail
  End While
End For
```

Figure 1: Pseudo-code for SteppingStone

the goal is successively closer to completion, or *improved*. For an unsolved Boolean constraint, the only way to improve it is by finding a state where the constraint is solved. Soft real-valued constraints can be improved by finding a state with a better value for the constraint. Problem solving with both hard and soft constraints requires finding a solution that meets all of the hard constraints and optimizes the soft constraints. Figure 1 provides pseudo-code for the SteppingStone approach to problems with both hard and soft constraints.

Soft constraints are treated as subgoals by SteppingStone and are ordered by an openness heuristics [Ruby and Kibler, 1989] along with the other problem subgoals. Means-ends analysis initially attempts to solve these subgoals without undoing any of the previously solved subgoals. When this approach fails SteppingStone switches to a knowledge-based approach. This approach operates by searching memory for a sequence of subgoals (steppingstones) for improving upon the current subgoal. Once indexed, means-ends analysis is used to follow these steppingstones from the current state to a new state. If in this new state the current subgoal is not closer to being solved or all of the previously solved subgoals do not remain solved, the current state is kept and memory is searched for additional steppingstones. If in this new state the current subgoal is improved and the previously solved subgoals are still solved then this state becomes the new current state and problem solving continues.

When memory has no knowledge for reducing the current subgoal, SteppingStone falls back on local search. If this search produces further improvement, the sequence of moves used to generate the improvement is generalized into new steppingstones. Problem solving continues until the subgoal is solved. For optimization subgoals it is often impossible to determine when the subgoal cannot be improved. In these cases, problem solving continues until no further progress within the current resource limitation is possible. At this point the subgoal is considered solved.

During the search for a problem's solution, the irrel-

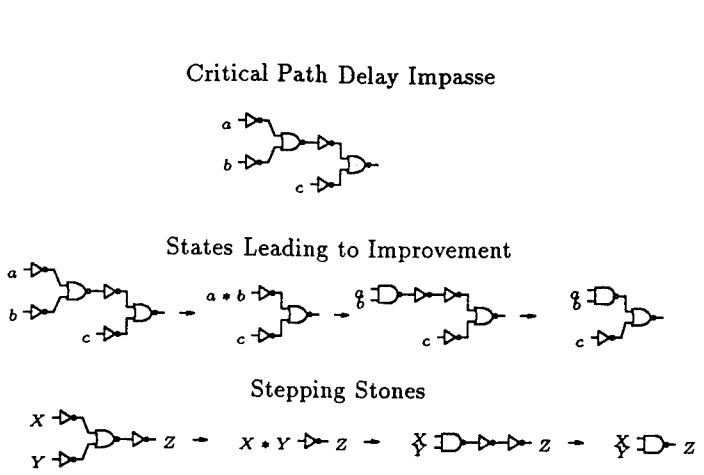


Figure 2: Steppingstones for Optimizing Critical Path

evant aspects of the state can greatly increase its difficulty. For example, adding more blocks to a blocks world problem can make the problem more difficult even if the blocks are independent of the problem solution. The local search procedure used by SteppingStone mitigates this problem by beginning the search within the region most likely to have a solution. Initially, only those parts of the state involved in the subgoals generating the impasse are allowed to be changed. If the solution cannot be found in this space, it is enlarged until the search space includes a solution or the search is terminated.

When a sequence of moves is found to improve an impasse, the sequence is generalized to generate a new sequence of subgoals or steppingstones in the following way. The sequence of moves generates a corresponding sequence of states. This sequence of states can be regarded as a very specific sequence of subgoals. Since these subgoals are used to reduce an impasse, those parts of the state uninvolved in the impasse are removed. This is determined by the parts of the state involving the subgoal being solved, and those parts of the states involved in the previously solved subgoals that were changed during the sequence of moves. From this generalized sequence of subgoals repeating subgoals are removed, yielding the final set of steppingstones [Ruby and Kibler, 1989].

In the past, problem-solving knowledge has primarily been represented as macro-operators or control rules. Macro-operators reduce the distance between states, but also increase the branching factor. Unless these macros occur often between problem states and goals states, their cost can outweigh their benefit. Control rules reduce the branching factor but leave the distance between problem states and goals states fixed. Control rules must also be evaluated at every problem-solving step. Unless they are applicable often, or result in large search reductions, their cost can also outweigh their benefit [Minton, 1988].

2.2 Logic Synthesis

One important domain that requires optimizing real-valued constraints as well as meeting a set of Boolean constraints is that of integrated circuit design. A large portion of integrated circuits consists of combinational logic. Optimizing the performance of this circuitry is a difficult and time-consuming task. Manual optimization is often applied to only the most critical portions of a design because of the cost and time required. This results in circuits that are larger and slower than necessary.

In logic synthesis, a functional specification of a circuit is mapped into combinational logic using a library of available components. These components are taken from a technology-specific library. These libraries vary depending upon the technology and particular manufacturer chosen. The synthesized circuit is optimized for a set of constraints which vary with the application.

Operating SteppingStone on the logic synthesis task requires a state space representation of the problem. Logic synthesis can be represented with a start state defined by a functional description of a circuit, along with a set of constraints. Boolean algebra provides a good language for the functional description of a circuit. The goal is a *realizable* circuit, using components from an available library, that satisfies a set of hard constraints and optimizes a set of soft constraints.

Operators for this domain map parts of the functional description to components from the technology-specific library. These mappings are well-defined and ensure the correctness of the resulting design. Mapping a functional description to a realizable design is a simple task. Finding a realizable design that satisfies a set of hard and soft constraints is much more difficult. To ensure global optimality requires an exhaustive enumeration of the design space, which is computationally intractable.

Figure 2 gives an example from the logic synthesis domain. The initial state for this problem is the boolean expression $a * b * c$. The goal is a circuit that is realizable and optimized for critical path delay time. The library of components consists of nand-gates, nor-gates, and inverters. For each component there are two types of operators, those mapping a boolean expression to that component and those mapping the component into its associated boolean expression. Although it is always possible to map from boolean expressions to components and from components back to boolean expressions, the domain does not have to be invertible. It may not be

possible to map a component back to all of the possible boolean expressions that can be mapped into it. Thus, it may not always be possible to apply a sequence of operators to move to a previous state.

Given the problem specified by the boolean equation $a * b * c$, SteppingStone produces the realizable circuit at the top of Figure 2. This is an impasse for SteppingStone since it cannot improve the critical path without undoing the goal of realizing the circuit. Local search is used to find a circuit with an improved critical path delay time. The states shown are those generated by the sequence of moves leading from the impasse state to the improved state. The steppingstones are generated by removing from these states all but those portions involved in the previously solved subgoal (realizable) that were modified while generating the improved state. These final steppingstones appear at the bottom of Figure 2.

Note that the steppingstones presented in Figure 2 are goals and can match many states. The only requirement is that the bound variables remain consistently bound through the steppingstones. Since steppingstones are used heuristically and only if grounded operations can achieve them, this type of generalization is effective.

2.3 Macro-operators and Steppingstones

Macro-operators are a common representation for problem solving knowledge. By combining a useful sequence of operations into a new operator, problem solving performance can be improved. Steppingstones differ from macro-operators in several ways.

Some types of generalizations are difficult with macro-operators. For example, the sequence of operations applied in Figure 2 map a nor-gate with inverters on the inputs and output to a nand-gate. Unfortunately, these operations will only work when the nor-gate is a 2-input nor-gate. This same transformation may also work with a 3-input nor-gate, but this requires a different set of operations. To achieve this generalization with a macro-operator requires generalizing outside of the knowledge closure, creating a new operator that cannot be deduced from the original sequence of operations. Furthermore, this form of generalization can introduce illegal and incorrect states. In most situations this cannot be tolerated. For example, in logic synthesis the final circuit must meet the functional specification and this cannot cannot be guaranteed if there are illegal transformations.

Steppingstones do not have the same generalization difficulty as macro-operators. A sequence of subgoals only partially specify a path between states, and the sequence does specify the particular operators required. The operators are selected through search after the subgoals have been instantiated. For example, the sequence of subgoals given in Figure 2 can also be used when the nor-gate is a 3-input nor-gate. Given a 3-input nor-gate with inverted inputs a, b, c, two of the inputs (a, b) would bind with the variables X and Y. SteppingStone would then search for an operator that could map the 3-input nor-gate with the inverted inputs to the conjunction $a*b$. This is possible by mapping the 3-input nor-gate and inverters to $a * b * c$. The remaining transformations are achieved in similar fashion. If there wasn't an operator for mapping this 3-input nor-gate and inverters to $a * b * c$, or any of the other subgoals, the subgoal sequence would fail. The subgoal sequence would also fail if after achieving all of the subgoals the final state did not improve the original difference being solved. In any case, the knowledge gained by SteppingStone can never generate an incorrect or illegal circuit since this knowledge is only used heuristically to guide the search process. The actual generation of the circuit is done using the basic operators whose correctness is assumed.

Macro-operators must also be tested for applicability on each problem solving cycle. Unless the macro-operator is used often, or provides a particularly large search savings, its cost may outweigh any benefit it provides. Steppingstones are only tried when the initial domain definition and search procedure prove unable to solve a subgoal. By reducing the need to test the applicability of learned knowledge, its cost is greatly reduced.

3 Steppingstones for Logic Synthesis

To demonstrate SteppingStone's capabilities in real-world domains we conducted a series of experiments in the domain of logic synthesis. These experiments were designed to demonstrate that SteppingStone could learn to optimize soft constraints in circuit design. We also explored SteppingStone's ability to adapt to different optimization tasks and changes in the domain. To estimate the quality of the solutions found by SteppingStone we built a custom program for solving the problems and compared the performance of SteppingStone to our custom approach. Furthermore, we applied SteppingStone to the solution found by our hand-crafted approach and to determine if further improvements were possible. This demonstrates that SteppingStone could be used as a post-processor to an externally derived design, whether human or machine generated, and still yield performance improvements.

3.1 Learning Space Optimizations

In our first test of SteppingStone's ability to learn optimization knowledge we began with a domain used by CPS design system [Tong and Franklin, 1989]. The design system was provided with a functional specification using a Boolean equation. The system synthesized a circuit that met the specification and was optimized for the space required. The components used to build the circuits were *inverters*, two input *and* gates, *or* gates, and *nand* gates. The space required for each of these components was: *and*=5, *or*=5, *inverter*=3, *nand*=1.

SteppingStone was initially trained on problems small enough for local search to produce optimal designs. Small random Boolean equations with independent inputs were used. These equations used the connectives *and*, *or*, and *not*. There are 2^{2^n} different equations of this type with n inputs. With the given library of components, there are three distinct ways of implementing an *and* gate or an *or* gate, and two distinct ways of implementing a *not*. Thus, for a problem of size n there are of order 3^{n-1} different possible solutions. This large search space makes this problem difficult for brute-force methods. The subgoals described earlier are both highly

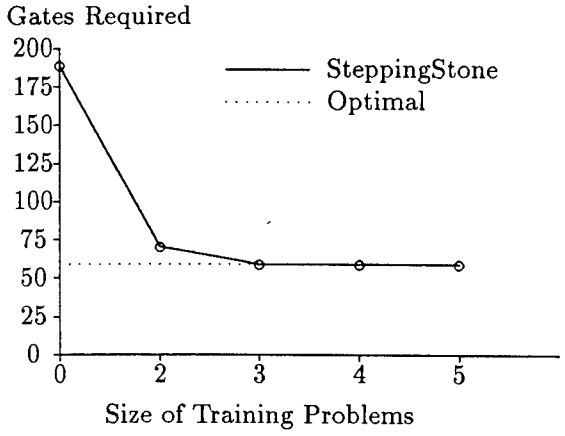


Figure 3: Average Performance on Problems of Size 30

interacting and different in character from those traditionally used, making the problem difficult for goal-based approaches.

SteppingStone was trained on five successive sets of problems with an increasing the number of inputs. The first training set had 2-input problems. The number of inputs increased until the last training set had 6-input problems. Training in a set finished after ten successive random problems were solved without any learning. Testing was done after finishing each set of training problems. The system was tested on three sets of twenty-five random problems. These sets were drawn from problems with 10, 20, and 30-inputs respectively. Learning and local search were turned off during testing. Note, the random test problems with 30-inputs were actually larger than example problems we were given from local industry.

With this particular library of components, it is easy to determine if a solution is optimal. Figure 3 plots the average optimal value for the random 30-input problems along with the other data. Note, SteppingStone quickly converges on the knowledge needed to generate the optimal solutions for these problems. These same results occurred with the test set of 20-input problems and 10-input problems.

SteppingStone learned five subgoal sequences for finding the optimal solution to any problem in this domain. This is less than half the number of problem decomposition rules learned by SCALE [Tong and Franklin, 1989] in this same domain. SCALE finds a set of rules for decomposing any problem into non-interacting subproblems, a difficult and often impossible task. In contrast, SteppingStone learns heuristic decompositions. In addition, unlike SCALE, the generalizations made by SteppingStone could be overgeneral. Although SteppingStone's knowledge is heuristic, it is still effective.

To demonstrate that learning greatly reduced the amount search required to find the optimal solution, a second test was conducted. SteppingStone was tested on the 30-input problems with learning turned off and local search turned on. Using only localizing search, SteppingStone's found the optimal solutions, but averaged 28,695 nodes expanded. The most difficult problem required 82,827 nodes expanded. The amount of search required after learning to find optimal solutions averaged only 1,806 nodes expanded.

3.2 Real-World Circuit Design

Like most real-world domains, the specifics of a logic synthesis task can vary. For example, the components available to synthesize a circuit will vary depending upon the technology chosen. These changes affect how a circuit should be synthesized. In addition, the set of constraints to be met or optimized can vary from problem to problem. To demonstrate that SteppingStone could learn to synthesize high quality circuits regardless of the constraints or library of components chosen, we created a second component library for experimentation.

An important performance constraint neglected from the previous problem was the critical path delay time. It is usually more important to optimize the critical path delay time of a circuit then the space required. The second set of problems were defined with the goal of first optimizing critical path delay time and then space (number of gates) required. This second task demonstrates that SteppingStone learns to optimize multiple constraints simultaneously. Note there were no fixed set of rules for decomposing this new problem into non-interacting subproblems.

The space required for the components from the previous library were artificial. The performance characteristics for this second library were taken from the components available from the LSI Logic Corporation. The following list contains the components chosen for this second library and their critical path delay time/gates required: 3-input *nand*=4.2ns/2 gates, 2-input *nand*=2.9ns/1 gate, 3-input *nor*=2.4ns/2 gates, 2-input *nor*=2.2ns/1 gate, *inverter*=2.9ns/1 gate.

SteppingStone was trained on this second task in the same way as used for the first task. It was given small problems that it could solve optimally with local search. It was then tested on the same three sets of test problems. Figure 4 shows how the average critical path delay time of the circuits synthesized decreased as learning increased for the random thirty-input problems. Figure 4 also shows how the space required for the circuits decreased with learning. Similar results were found for the other test problems.

With this second library it was not possible to generate the optimal circuit. In order to judge the difficulty of these problems and the quality of the solutions generated by SteppingStone, we developed a custom approach for generating solutions to these problems before beginning any experimentation. Note that we did not hand-code knowledge for SteppingStone, but actually wrote a custom program for solving problems in this domain.

Our best approach for finding a good design operated by first generating a design using the components available. The program preferred to use the largest components possible. It then used a special procedure to improve this design. This procedure *pushed* inverters through the design towards the inputs. Inverters are pushed through components by defining rules for switch-

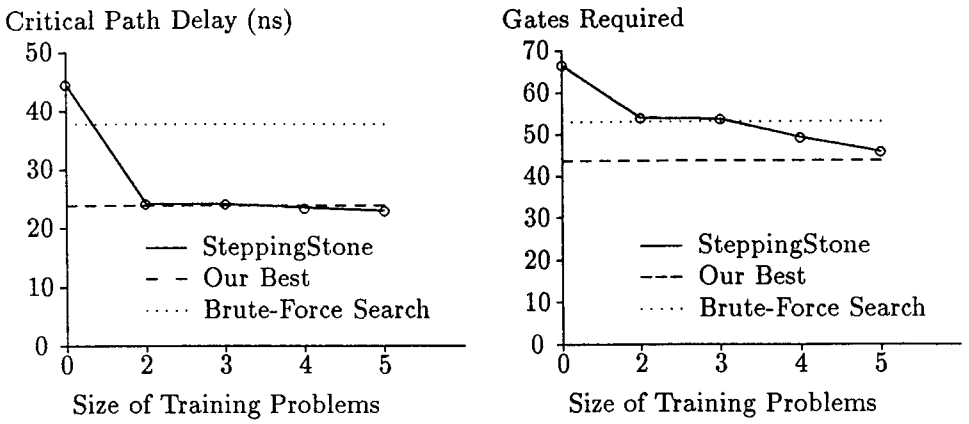


Figure 4: Average Performance on Problems of Size 30

ing from an inverter into a component to some other component with the inverters on the inputs. These inverters could then be eliminated whenever two appeared next to each other. Heuristics were used to attempt to determine situations where pushing an inverter through the design would not provide improvement because of the other components that would be changed. Although a relatively simple approach, it provided a good base for comparison. The results of our best approach are plotted in Figure 4 along with the SteppingStone results. Note that SteppingStone quickly learns enough to produce circuits close to that of our best approach and, for the critical path delay, eventually improves upon it.

To further estimate the difficulty of these problems a simple brute-force approach was also tried. The best solution found using the brute-force approach with a cutoff of 500,000 search tree nodes was recorded for each of the test problems. The averaged results are also plotted in Figure 4. Note that although SteppingStone initially produces solutions that are much worse than that possible with an exhaustive approach, it soon outperforms exhaustive search.

With this second set of components SteppingStone did not converge upon a set of knowledge for always generating the optimal solution. This does not seem unusual since application systems for doing logic synthesis are unable to generate optimal solutions to arbitrary random problems and resort to heuristic knowledge to generate good solutions. SteppingStone generates its own heuristic knowledge by recognizing recurring features of impasses and using local search to learn steppingstone for resolving the impasses.

Instead of converging upon a set of steppingstones for this domain, SteppingStone continued to acquire new knowledge as the size of the training problems increased. After training on problems up to size five, 34 subgoal sequences were acquired. Given that the number of random Boolean functions with n inputs is 2^{2^n}, or 2^{32} for problems of size five, the amount of learning is extremely small. As with the tile-sliding domain [Ruby and Kibler, 1989], this small amount of learning is due to SteppingStone's decision to learn only when reaching an impasse. After learning these 34 steppingstones, the amount of search required to find the solutions to the thirty input problems averaged 2,841 nodes expanded. Even after expanding over 30,000 nodes without learning, SteppingStone was unable to generate solutions as good as those found quickly after learning.

This set of experiments clearly demonstrates the generality of the SteppingStone approach. Learning provides the mechanism to easily adapt to changes in the domain. This provides the system with the flexibility needed for real-world problems.

3.3 Learning to Improve Solutions

In our final set of experiments, we tested SteppingStone's ability to take the solution from our best approach and improve upon it. This experiment was designed to show that the knowledge acquired by SteppingStone was different from the knowledge we encoded in our hand-coded approach. It also demonstrates that Steppingstone can be usefully combined with other means for generating initial solutions.

For this second set of results we used the library of components and goal specification from experiment two. The problems were given to our best approach and the solutions generated. The solutions were then given as initial states to SteppingStone using the knowledge it had acquired after being trained on problems with up to five inputs. Table 1 shows the averaged performance results for the solutions generated to the test set of random problems with thirty inputs. Note that SteppingStone was able to improve upon both the average critical path delay time and the amount of gates required. Improving upon both critical path delay and space required is especially surprising since improving one often comes at the cost of a decline in the other. This clearly demonstrates the knowledge acquired by SteppingStone was different from that we derived and encoded in our best approach.

Table 1 also includes the average quality of the solutions generated by SteppingStone alone. Here the results show that beginning with a good solution did allow SteppingStone to generate better final solutions. The improvement was not large for the critical path delay time, but was more significant for the space required. Learning space optimization was difficult for SteppingStone since

	Critical Path Delay	Gates
Our Best	23.75	43.68
SteppingStone	22.97	45.76
Best+SteppingStone	22.70	43.04

Table 1: SteppingStone Improvements on Our Best

they only occurred in larger training problems. Yet, by beginning with a solution that was already good, the knowledge acquired improved upon this solution.

This experiment again demonstrated the generality and power of SteppingStone. Not only was SteppingStone able to acquire knowledge and generate high quality solutions, it was able to take solutions that we generated and improve upon them. This type of flexibility is critical to any problem solver that will operate in a real-world setting.

4 Related Work

To better understand SteppingStone we compare it with other learning problem solvers. Like SOAR [Laird *et al.*, 1986] and Prodigy [Minton, 1988], SteppingStone learns on failure. However these systems are more dissimilar than similar.

SOAR learns new rules, or chunks, whenever overcoming an impasse. For SOAR an impasse occurs whenever it cannot make an unambiguous decision. SteppingStone also learns when overcoming an impasse, but its definition of an impasse is different from that of SOAR. For SteppingStone, an impasse occurs when its means-ends analysis component is unable to solve a subgoal while maintaining all of the previously solved subgoals. Means-ends analysis serves to funnel large numbers of problems states into a few impasse states. This means that learning will occur rarely. While SOAR is an eager learner, perhaps because it attempts to match human learning, Steppingstone is a lazy learner, only learning when forced to.

SOAR incorporates its learned knowledge into its problem solving rules. Consequently SOAR will try all learned rules on every elaboration cycle, firing them whenever appropriate. This prevents the impasse from ever occurring again. SteppingStone separates its learned knowledge from the rest of its problem solving knowledge. After learning to resolve an impasse, that impasse will continue to be encountered. The system ignores its learned knowledge until the impasse occurs. When an impasse is encountered SteppingStone searches its learned knowledge to see if it has encountered a similar impasse in the past. If it has, it uses this knowledge to try to resolve the current impasse. In this way, an impasse operates like an exception for the system with the steppingstones behaving as exception handlers. Since the learned knowledge is only used when an exception is encountered, its cost is reduced.

When SOAR encounters an impasse it changes to a new problem space and searches for the solution to the impasse. A new problem space is generated in response to an impasse and SOAR uses weak-methods to search this problem space. When SteppingStone encounters an impasse it does not know how to solve, it does not change its problem space. It changes its search method to a localized forward search method. It remains in the same state-space with the same domain operators.

SOAR learns a new rule when it solves an impasse. This new rule generalizes the sequence of rules required to resolve the impasse. Thus, the new rule is essentially a macro-operator. SteppingStone learns a new sequence of subgoals. Some of the distinctions between macro-operators and subgoal sequences are discussed in Section 2.3. Briefly, subgoal sequences appear to allow more flexibility than macro-operators and do not increase the branching factor.

Korf [Korf, 1985] showed that if macros are learned for solving each subgoal of a problem and these macros depend only on the value of that subgoal and the previously solved subgoals, problems can be solved easily. Unfortunately this approach only works if a complete set of these macros can be learned. This requires that the problem state space be *operator decomposable* and that states are represented as a vector of discrete state variables. SteppingStone uses a related approach but adopts a heuristic view. SteppingStone does not require complete knowledge, since it is able to use search to bridge the gap between the abstract form of generalized knowledge and its specific application on a particular problem. Since it uses search to determine if a piece of knowledge is applicable, it does not need to index its knowledge on the particular value of a subgoal. This eliminates the need for a state consisting of a vector of discrete variables. In addition, rather than learning macro-operators, SteppingStone learns subgoal sequences. On the negative side, by adopting a heuristic approach the strong analytical results no longer hold.

Prodigy [Minton, 1988] learned control rules for a STRIPS style problem solver. These control rules can suggest differences to reduce, operators to reject, or operators to try. Unfortunately, all control rules must be tested on each problem solving cycle. In addition, Prodigy depends upon a single search strategy, means-ends analysis. When subgoal interactions make this a poor strategy problem solving performance suffers. SteppingStone uses two different search strategies to offset the weaknesses of each.

5 Conclusions and Future Work

The goal of this research is to develop a general learning problem solver for real-world problems. Real-world problems are defined by both hard boolean constraints and soft real-valued constraints. SteppingStone operates on problems with both types of constraints by taking a different view of problem solving. We view problem solving as successively improving upon individual problem subgoals until they can no longer be improved. SteppingStone demonstrated that this approach allows it to solve problems with both hard and soft constraints by learning to optimize multiple constraints in the logic synthesis task of VLSI design.

Operating on real-world problems also requires scaling to large search spaces. SteppingStone derives much of its

power to scale by decomposing a problem into simpler subproblems and learning to treat these subproblems as though they were independent. Breaking a problem into independent subproblems provides an exponential decrease in problem difficulty [Korf, 1987]. SteppingStone demonstrate is ability to scale by solving large logic synthesis problems with two different component libraries.

SteppingStone represents the problem solving knowledge it learns as a sequence of subgoals, or steppingstones. These steppingstones reduce the distance between problem subgoals without increasing the branching factor of the problem. They provide a heuristic decomposition for portions of the problem made difficult by subgoal interactions. The steppingstones learned effectively generalize the problem solving knowledge by using a small amount of search for their application. Unlike many learning problem solvers, Steppingstone is a lazy learner – learning only when forced to. Moreover, the number of situations when learning is necessary is limited by the means-end analysis problem solver.

Steppingstone derives its problem solving ability from the integration of a problem solver that uses domain general and domain specific problem solving knowledge and a learner to generate the domain specific knowledge. In particular, means-end analysis applies general weak problem solving knowledge to solve the easy portions of the problem. On hard portions of the problem, specialized knowledge in the form of steppingstone are used to guide the problem solving process. Localized search on problem solving impasses is used to learn the domain-specific steppingstone knowledge. We now plan to explore SteppingStone's ability to solve additional VLSI design tasks and other real-world domains such as scheduling.

References

[Fikes *et al.*, 1972] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[Korf, 1985] Richard E. Korf. *Learning to Solve Problems by searching for Macro-Operators.* Pitman Advanced Publishing Program, 1985.

[Korf, 1987] Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.

[Laird *et al.*, 1986] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in SOAR: The anatomy of a general mechanism. *Machine Learning*, 1(1):11–46, 1986.

[Lin and Gajski, 1988] Youn-Long Lin and Daniel D. Gajski. Les: A layout expert system. *IEEE Transactions on Computer-Aided Design*, 7(8):868–876, August 1988.

[Minton, 1988] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* PhD thesis, Carnegie Mellon University, Computer Science Department, Pittsburgh, Pennsylvania, 1988.

[Ow *et al.*, 1988] Peng Si Ow, Stephen F. Smith, and Alfred Thiriez. Reactive plan revision. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 77–82. Morgan Kaufmann, 1988.

[Porter and Kibler, 1986] Bruce W. Porter and Dennis Kibler. Experimental goal regression: A method for learning problem solving heuristics. *Machine Learning*, 1(3):249–285, 1986.

[Rosenschein, 1981] Stanley J. Rosenschein. Plan synthesis: A logical perspective. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 331–337, Vancover, BC, 1981.

[Ruby and Kibler, 1988] David Ruby and Dennis Kibler. Exploration of case-based problem solving. In *Proceedings of the Case-Based Reasoning Workshop*, pages 345–356, Clearwater Beach, Florida, 1988.

[Ruby and Kibler, 1989] David Ruby and Dennis Kibler. Learning subgoal sequences for planning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 609–614, Detroit, Michigan, 1989. Morgan Kaufmann.

[Sacerdoti, 1974] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Sacerdoti, 1977] E.D. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier North-Holland, New York, 1977.

[Stefik, 1981] Mark Stefik. Planning with constraints (molgen: Part1). *Artificial Intelligence*, 16:111–139, 1981.

[Tate, 1977] Austin Tate. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888–893, Cambridge, MA, 1977.

[Tong and Franklin, 1989] Chris Tong and Phil Franklin. Learning a satisficing compiler for circuit design. In *submitted to Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1439–1445. Morgan Kaufmann, 1989.

[Vere, 1983] Stephen A. Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(3):246–267, May 1983.

[Wilkins, 1984] David E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.

[Zanden and Gajski, 1988] Nels Vander Zanden and Daniel Gajski. Milo: A microarchitecture and logic optimizer. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 403–408, 1988.

DARPA PI REPORTS

The Quest for Architectures for Integrated Intelligent Systems Extended Abstract

Allen Newell[1]
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Artificial intelligence (AI) is dedicated to the search for the nature of intelligence. As with all sciences, it starts not with definitions but with phenomena, clues and hunches. And the end result will not be a single thing or principle, but a rich body of scientific knowledge that will join with the rest of our scientific knowledge of the universe. Thus, the science comprises many searches, not just one.

The quest of interest to me here is for the nature of a single system capable of general intelligence. The phenomena that gives shape to that quest arises from ourselves. We both observe and experience ourselves to be capable of performing a vast and unfolding array of tasks, continuously acquiring further knowledge from our experiences, and operating autonomously without some other seemingly more intelligent oracle at hand to do our intelligence work for us. Scientifically, such knowledge, even about ourselves, serves only as clues and hunches. We also know our intellectual powers to be limited, the world of tasks we encounter to be restricted, and our dependence on our social milieu and education to be pervasive. Withal, our autonomously exercised intelligence delimits for AI a central scientific phenomena. To discover the nature of such systems is one central quest within AI. This quest does not gainsay the many other quests that comprise our science. Indeed, it is important not to make this one quest co-extensive with the field, for there are many other paths that must be followed if we are to fully understand intelligence. But this is the quest that beckons me.

To pin down somewhat the sort of system we seek, it helps to list some of our own capacities:

1. Behave flexibly re the environment
2. Exhibit adaptive (rational) behavior
3. Operate in real time
4. Operate in a rich environment:

   Perceive immense changing detail
   Use vast amounts of knowledge
   Control multiple-freedom movement
5. Use symbols and abstractions
6. Use language (natural and artificial)
7. Learn from environment and experience
8. Acquire capabilities by development
9. Live autonomously in a social world
10. Have self awareness and sense of self

These are not all distinct capabilities — coverage is more important than partition. Further, these are capabilities, not mechanisms. To believe a truly intelligent agent should be able to exhibit these characteristics, is not to understand how. A few may be so uniquely human that we might consider abstracting away from them. Development, social dependence and self awareness are candidates. However, we should not be too hasty to avoid any of them.

This particular quest is not new, neither for the field nor myself. Its roots go back to the late fifties, where it emerged as soon as indications began accumulating that the digital computer had opened a brand new approach to intelligence. Thus, the quest has a thirty year history. That history presents two faces. One is a waxing and waning of the quest itself, as AI uncovered one aspect after another of intelligent phenomena, so that various quests vied with each other for attention. From this face, each major development in AI increased or decreased interest in the quest for a general intelligent agent. The other face is a steady accumulation of the knowledge necessary to advance the question. From this face, all the major developments, whether their rhetoric aided or denigrated the quest for a general intelligence, have added to what we know about the mechanisms and component functions required for such an agent.

Thirty years of science have given substantial shape to the quest. We have good bets about the nature of a general intelligent agent, backed by substantial empirical explorations and discoveries, and with some supporting theory. Principal among these is that such an agent will be a symbol system, hence its structure will be that of an architecture with memories that contain encoded representations of the task environment and the agent's own operations. This specification narrows the quest to discovering the nature of the architecture of a general intelligent agent. Indeed, the

---

[1]This is the abstract of my talk on receiving the IJCAI Award for Excellence in Research at the 1991 Conference. Inadvertantly, it was not published in the IJCAI Proceedings. Though brief, it seems useful to publish it here, since it indicates an important direction for research. Over the years my research has been supported by the Defense Advanced Research Projects Agency (DOD).

quest has generally been formulated in these terms in recent years. Substantial progress has been made in understanding the basic mechanisms that go into such an architecture, although as yet there is no convergence to a single structure.

However, the quest is not just for the architecture, although it might seem that way to those of us engaged in trying to get the architecture right. Indeed, the rhetoric of expert systems takes the inference engine to be relatively unimportant compared to the knowledge base it services. But given candidate architectures that have some promise and stability, it is important to raise our sights to the goal of obtaining the total system that such architectures are to support. Let us call this ultimate target an integrated intelligent system. We need to inquire exactly what such a target system should comprise. In any event, only when we have such target systems will we find out many of the essential characteristics of the candidate architectures. Only then will we find out whether our proposed architectures have their requisite capabilities.

The current state in the quest in mid 1989 presents us with an interesting collection of candidate architectures, with enough experience and understanding of them so they are indeed candidates. But even though we talk of the many things we have done with these architectures, none of them has yet been put at the center of a genuine integrated intelligent system. Thus, I take the appropriate current characterization of the quest to be that of attempting to discover whether we have architectures capable of supporting integrated intelligent systems. Several years ago, the focus was almost entirely on discovering the architectures themselves. Some years hence, the emphasis may shift to being entirely on integrated intelligent systems — on how powerful or capable they are or fail to be, and on questions of scale. Our present point is poised between the two. I have selected my title accordingly.

My objective here is to characterize our present moment in this quest and to set the task that seems to me the interesting next step to attempt. The starting place, as always, is with the AI systems that capture this present moment. I will focus on three: Soar, developed by John Laird, Paul Rosenbloom, myself and our colleagues, Prodigy, developed by Jaime Carbonell, Steve Minton and their colleagues, and Theo, developed by Tom Mitchell and his colleagues. These systems, all centered or with substantial presence at CMU, are the ones I judge most ready for this next step. In part, this is objective judgment, biased somewhat by familiarity and participation. But it is also that these systems, by being located together, are developing a symbiotic relation that augers well for the next step. It is a form of close-in cooperation-competition that promises to drive these systems forward at a rapid pace, expose their limitations mercilessly, and foster the borrowing of good solutions.

Other systems are also important for defining our current position, for instance, BB* by Barbara Hayes-Roth, CYC by Doug Lenat and Icarus by Pat Langley (all with colleagues, since all such efforts involve cooperative teams). All these systems embody lessons of the last thirty years, though in different combinations and ways. This cumulation of existing science is important, and it is worthwhile to be sure we understand it. The ways these architectures formulate tasks and structure data and control to bring knowledge to bear are all late-generation versions of much-studied mechanisms. Also, these architectures (especially Prodigy, Soar and Theo) are part of the recent intensive concentration in AI on learning. In fact, the incorporation of learning as a pervasive characteristic of these architectures is what makes putting together integrated intelligent systems the right next challenge.

We need to set out what the step to integrated intelligent systems involves. In one sense, it is easy to do so, since we have the list above for guidance. But we must be guided also by what we know of the difficulty of attaining various subgoals, given the current AI art. Some difficulties we must insist upon, for to avoid them is tantamount to avoiding to attempt the next step. Other difficulties we must essentially dodge because the time is not yet ripe to confront them.

The first major concern, then, is autonomy — getting the system to live in the world, interacting with it by conventional channels, which means both robotically and linguistically, and for the latter, via natural languages, graphic figures and formal specification languages. More functionally, autonomy means the ability to acquire tasks and knowledge from the external world while interacting with it, whether via guidance, instruction, observation or experimentation. The second major concern is to be able to employ a rich repertoire of methods, both general and specialized to the occasion. It is not necessary that a system perform all methods, only those congenial to it, so to speak; but certainly method flexibility is necessary, as is the acquisition of methods from the external world, as well as tasks. The third major concern is that such a system have a substantial body of competence with respect to its world. It should know a lot about its place of interaction and what it contains, it should have number and reading skills, and it should have much experience with its world. Which actual tasks should be accomplished is less important than that there be diversity and that the tasks reflect the scale and complexity of the natural world. However, some tasks need to involve the use of external devices and facilities, such as computational simulators, and cooperative activity with other intelligent agents. It goes without saying that routine and continual learning from experience should occur.

The above paragraph of specifications is unremarkable, in being simply one way of casting what we know of our own competence, though with some priorities about what should be attained first. However, many of these specifications occur as independent items on AI's current research agenda. A standard argument says that research should occur bottom up — components should be understood before systems of components are attempted. It is certainly easy to enumerate difficulties, uncertainties and prematurities. Clearly the argument for attempting the step is not that we know we are ready to succeed. Rather, it is my assessment that we are ready to find out about the difficulties experimentally. In the world of integrated systems, we only find out about many key difficulties by forcing ourselves to put together experimental systems. But possibly more than that is at stake. At some point the total system wins from the synergy of its mechanisms, rather than just getting tangled up in itself. In part that is what it means to have an integrated system.

Merging Strategic and Tactical Planning In Dynamic, Uncertain Environments*

Piero Bonissone
General Electric CRD
Schenectady, NY 1230
bonissone@crd.ge.com

Soumitra Dutta
INSEAD
Fontainebleau, France 77305
dutta@freiba51.bitnet

ABSTRACT

This paper presents a new approach to planning in dynamic and uncertain environments. Planning is viewed as a process in which an agent's long term goals are transformed into short term tasks and objectives, given the context of planning. The developed model allows for a dynamic balance between long term strategic planning and short term tactical planning. A combination of rules and scripts is used. Rules are used for reasoning about long term strategic choices. Scripts are used for representing short term tasks and objectives. The uncertainty calculi of RUM/PRIMO [3,4] are used for supporting reasoning under uncertainty. In the proposed model, it is also possible to achieve a seamless integration of case-based reasoning into the planning process. These ideas have been implemented in a system called MARS, which plans in the financial domain of mergers and acquisitions.

1. Planning in Dynamic and Uncertain Domains

In this section, we motivate the need for dynamic planning, describe relevant prior research, emphasize the contributions of this research and outline the structure of the paper.

1.1 Introduction and Prior Research

Most of the early research in planning considered static *domains*, in which the state of the world was given at any instant and the emphasis was on devising plans (provably correct sequence of actions) to achieve certain goals. Two assumptions were common during the planning phase: the environment was static and the effects of various actions on the world were fully predictable. These assumptions ensured that the devised plan sequences could be executed successfully in the idealized world. However, such planners often generated plans that were not executable in the real world. Usually, this failure was caused by the plan's dependencies on states whose values were changed before the plans execution. In the planning literature, this planning process is usually referred to as *strategic planning.* However, within the scope of this paper, we will refer to it as static planning, since we want to use the term *strategy* in the typical military or business connotation.

Many researchers [1,19,24] have noted the limited capabilities of early planning systems [14,22] and have proposed models for planning in dynamic and uncertain environments. Such models have adopted different approaches for dealing with these environments. Some researchers [8,13] have interleaved plan formation and execution. Brooks [7] has proposed the decomposition of the problem into task-achieving units realizing distinct behaviors. Georgeff and Lansky [18] have emphasized the need for a rich vocabulary for reasoning about the intentions of the planner during reactive reasoning. Rosenchien, Kaelbling and Pack [23] have adopted a formal approach to describing reactive planning, based on the compilation of situated automata from specifications of the knowledge of these automata. Firby [17] has proposed a reactive planning model based on the concept of independent entities (RAPS) pursuing goals in competition with many others in execution time. Bresina and Drummond [6] have investigated the

*This work was partially supported by the Defense Advanced Research Projects Agency (DARPA) under USAF/Rome Air Development Center contract F30602-85-C-0033.Views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official opinion or policy of DARPA or the U.S. Government.

Entropy Reduction Engine Architecture, composed of a *reactor* (to produce reactive behavior), a *projector* (to explore possible future states and to advise the reactor) and a *reductor* (to reason about behavioral constraints and to advise the projector). Yang et. al. [30] have proposed a conceptual clustering of "similar" operators and plans to reduce replanning costs in real time.

Agre and Chapman [1], Schoppers [26] and Nilsson [21] have adopted different, but related approaches to reactive planning. Agre and Chapman use combinational logic (a "table look-up" mapping from "situations" to "actions") to select the action to take depending on the situation on hand [2]. Schoppers has introduced the notion of *universal plans* for reactive planning. A universal plan is "equivalent to a decision tree whose outcomes are names of effector actions and whose decision nodes are labeled with environmental conditions" [2]. Planning is achieved by repeatedly cycling through the decision tree. Nilsson [21] has proposed the concept of *action networks* for reactive planning. Action networks are based on combinational logic and can be represented by decision trees (as in universal planning). Action networks differ from universal plans in that they allow the formation of action hierarchies. References [2,10,22] further elaborate on these recent approaches to planning in dynamic and uncertain environments.

1.2 Highlights and Contributions of the Paper

In this paper, we view planning as the process by which the long term goals and aspirations of an intelligent agent are translated into short term tasks, given the constraints imposed by the context of planning. This process must consider both the *strategic* and *tactical* aspects of planning. Strategic planning focuses on the selection of strategies for achieving long term goals, i.e., on reasoning about alternative paths to achieve long term goals. Tactical or incremental planning emphasizes tasks/actions which achieve short term goals. Both aspects are critical for successful planning. Pure tactical planning would increase reactivity at the expense of strategic goal directed behavior. Pure strategic planning would lead to inadequate flexibility in reacting to a changing world. While tactical planning can (and should) react quickly to a dynamic environment, strategic planning should in general be more stable and change only when required by a drastically altered environment or long term goals. This behavior ensures that an intelligent planner will be able to move steadily towards a strategic goal. This paper addresses the important issue of how a planner can balance strategic and tactical planning in an uncertain and dynamic environment.

The notion of a *strategy hierarchy* is introduced to represent the varying strategic and tactical aspects of planning. A strategy hierarchy is similar to a decision tree. However, the decision tree does not represent base level situation-action pairs (as in universal plans) or a hierarchy of plan representations (as in hierarchical planning [9]). Rather, the tree represents a continuum of decision points, ranging from the *strategic* to the *tactical/incremental*. The nodes close to the root node are highest in "strategic intent", while the nodes closest to the leaf nodes have maximum "tactical details". The higher nodes represent various strategic choices, while the leaf nodes represent plan details in the form of *scripts* or *skeletal plans*. In our approach we use a production rule system. However, unlike the systems of Chapman and Agre, Schoppers and Nilsson, where the plans themselves are represented by production rule like structures, we use rules for encoding the complex reasoning associated with each node in the strategy hierarchy. This gives us a greater degree of flexibility to account for the dynamic world and the changing context of planning (see section 3 for more details).

Most models use Boolean matches for taking planning decisions (e.g., testing pre-conditions of plans). This restriction limits the capability of the system to handle uncertainty in domain knowledge. Usually, these systems have no capability for representing varying *degrees of confirmation, refutation* and *ignorance* about a given decision variable. It is also the case that these systems cannot usually aggregate the contributions of different proof paths for a single decision variable. This feature, essential to exploit the redundancy in the knowledge base, allows the reasoner to use multiple deductive paths to obtain a distribution of values (with their associated uncertainty qualification) for a given decision variable. The planning mechanism described in this

paper uses the uncertainty calculi of RUM/PRIMO [3,4] to support such a representation of uncertainty and an aggregation of multiple proof paths (see section 3.1).

Another important issue addressed in this paper is the integration of case-based reasoning into the planner. Case-based planning is a relatively new but important planning methodology that considers the effect of prior experiences while formulating new plans. In our approach, fragments of cases are interpreted and represented by rule templates. The same mechanism used to aggregate multiple deductive paths provides a seamless integration of case-based with rule-based reasoning [14] (see section 3.6).

We have highlighted the important aspects of our research and have identified our contribution to the field of planning in dynamic and uncertain environments. The domain chosen to illustrate our ideas is the financial domain of mergers and acquisitions (M&A). M&A is a rich and interesting domain and is of high importance to businesses today. The planning scenario chosen is that of the various players (*raider*, *target* and *arbitrageur* - see section 2.1) in the context of a hostile merger attempt. Our ideas have been implemented in a prototype system called MARS (see section 2.2).

1.3 Structure of Paper

The paper contains three additional sections. The next section describes the domain of M&A and introduces the MARS system. Details of the planning mechanisms in MARS are given in section 3. Section 4 summarizes the characteristics of our planning methodology and describes our future work.

2. Mergers and Acquisitions

This section introduces the domain of mergers and acquisitions (M&A) and gives a brief overview of the MARS system.

2.1 M&A: An Introduction

The structure of corporate USA has been changed dramatically by the flood of mergers and acquisitions witnessed over the past decades. Annually, these deals total tens of billions of US dollars. To lend some useful conceptual abstraction, we can consider two players of interest in simple M&A deals: the *raider* (who usually initiates a take-over attempt) and the *target* (which is the company of interest to the raider). Another player of interest who is outside the structure of the actual M&A deal, but has a keen interest in the entire process is the *professional arbitrageur* (who tries to make arbitrage profit by wisely shifting his investments during the merger process). While the actions of each of these players vary from deal to deal, it is possible to identify certain basic actions associated with their individual roles. For example, some of the representative actions of a raider are *target monitoring*, *target evaluation and selection*, *merger strategy selection*, *target response evaluation* and *attack strategy modification*. Sophisticated planning and reasoning is required by every player in the M&A domain. Even in simple M&A deals, other complicating factors, such as multiple bidders and legal complications, often arise. Although each M&A deal is special and uniquely complex, we can still identify two types of M&A deals: *friendly* (agreed to by friendly companies for mutual benefit) and *hostile* (involving forcible takeovers). In this paper, we focus on hostile takeover attempts, as the planning and reasoning requirements for such M&A are much more interesting (from a computational perspective). The reader may consult references [15,20] for more details on various aspects of M&A.

2.2 Overview of MARS

MARS (A Mergers and Acquisitions Reasoning System) is a prototype AI reasoning system that both simulates and provides expert advice regarding the actions of the raider, the target and the arbitrageur. There are four independent simulators in MARS. The global simulator generates the values and changes of the macro-economic variables affecting the M&A deal (e.g., the interest rate and the price of Treasury Bills). The other three simulators generated and execute the reasoning and planning of the raider, the target and the arbitrageur respectively. There is a fusion of different reasoning techniques in all four simulators. Each of them is capable of integrated reasoning and planning with uncertain, incomplete and time varying information. MARS is implemented in Common LISP using RUM/PRIMO, and runs on the Symbolics. More details on the structure,

implementation and use of MARS can be found in reference [5].

3. MARS: Planning Details

This section provides details on the planning mechanisms implemented in MARS. As MARS is implemented using RUM/PRIMO [3,4], the section begins with a short summary of the relevant features of RUM/PRIMO.

3.1 Uncertainty Calculi & Belief Revision in RUM/PRIMO

Both facts and rules in RUM/PRIMO can represent uncertainty. Facts are qualified by a degree of *confirmation* and a degree of *refutation*. For a fact A, the lower bound of the confirmation and the lower bound of the refutation are denoted by L(A) and L($\neg$A) respectively. As in the case of Dempster's [12] lower and upper probability bounds, the following identity holds: L($\neg$A) = 1 - U(A), where U(A) denotes the upper bound of the uncertainty in A and is interpreted as the amount of failure to refute A. Note that L(A) + L($\neg$A) need not necessarily be equal to 1, as there maybe some *ignorance* about A which is given by (1 - L(A) -L($\neg$A)). The degree of confirmation and refutation for the proposition A can be written as the interval [L(A), U(A)].

RUM/PRIMO provides a natural representation for *plausible* rules. Rules are discounted by *sufficiency* (*s*), indicating the strength with which the antecedent implies the consequent and *necessity* (*n*), indicating the degree to which a failed antecedent implies a negated consequent. Note that conventional strict implication rules are special cases of plausible rules with $s = 1$ and $n = 0$. Each rule has an associated context which represents the set of preconditions determining the rule's applicability to a given situation. This mechanism provides an efficient screening of the knowledge base by focussing the inference process on small rule subsets.

RUM/PRIMO provides an uncertainty calculus based on a set of five Triangular norms (T-norms) [4] for inference in the rule graph. Each T-norm $T_i(a, b)$ lies within the interval $[T_1(a, b), T_3(a, b)]$, where $T_1(a,b)=\max(0,a+b-1)$ and $T_3(a,b)=\min(a,b)$ respectively. Their corresponding DeMorgan dual T-conorms, denoted by $S_i(a,b)$, are defined as:

$$S_i(a,b) = 1 - T_i(1-a, 1-b).$$

For each calculus (represented by the five T-norms), the following four operations have been defined in RUM/PRIMO:

Antecedent Evaluation: To determine the aggregated certainty range [b, B] of the n clauses in the antecedent of a rule, when the certainty range of the ith clause is given by $[b_i,B_i]$:

$$[b,B] = [T_i(b_1,b_2, \ldots ,b_n), T_i(B_1,B_2, \ldots ,B_n)]$$

Conclusion Detachment (Modus Ponens): To determine the certainty range, [c, C] of the conclusion of a rule, given the aggregated certainty range, [b,B] of the rule premise and the rule sufficiency, s, and rule necessity, n:

$$[c, C] = [T_i(s,b), 1 - (T_i(n, (1-B)))]$$

Conclusion Aggregation: To determine the consolidated certainty range [d, D], of a conclusion when it is supported by m (m>1) paths in the rule deduction graph, i.e., by m rule instances, each with the same conclusion aggregation T-conorm operator. If $[c_i,C_i]$ represents the certainty range of the same conclusion inferred by the ith proof path (rule instance), then

$$[d, D] = [S_i(c_1,c_2, \ldots c_m), S_i(C_1,C_2, \ldots ,C_m)]$$

Source Consensus: To determine the certainty range, $[L_{tot}(A), U_{tot}(A)]$ of the same evidence, A, obtained by fusing the certainty ranges, $[L_i(A), U_i(A)]$, of the ith information source out of a total of n different possible information sources:

$$[L_{tot}(A), U_{tot}(A)] = [\text{Max}_{i=1, \ldots ,n} L_i(A), \text{Min}_{i=1, \ldots ,n} U_i(A)]$$

The theory of RUM/PRIMO is anchored on the semantics of many-valued logics and is possibilistic in nature. References [3,4] describe a comparison of RUM/PRIMO with other uncertainty systems, such as Modified Bayesian, Certainty Factors, Dempster-Shafer, and Fuzzy logic.

RUM/PRIMO supports a belief revision mechanism to support reasoning under dynamic environments. The belief revision mechanism detects changes in the input, keeps track of the dependency of the intermediate and final conclusions on these inputs, and maintains the validity of these inferences. For any conclusion made by a rule, the mechanism monitors the changes in the certainty measures that

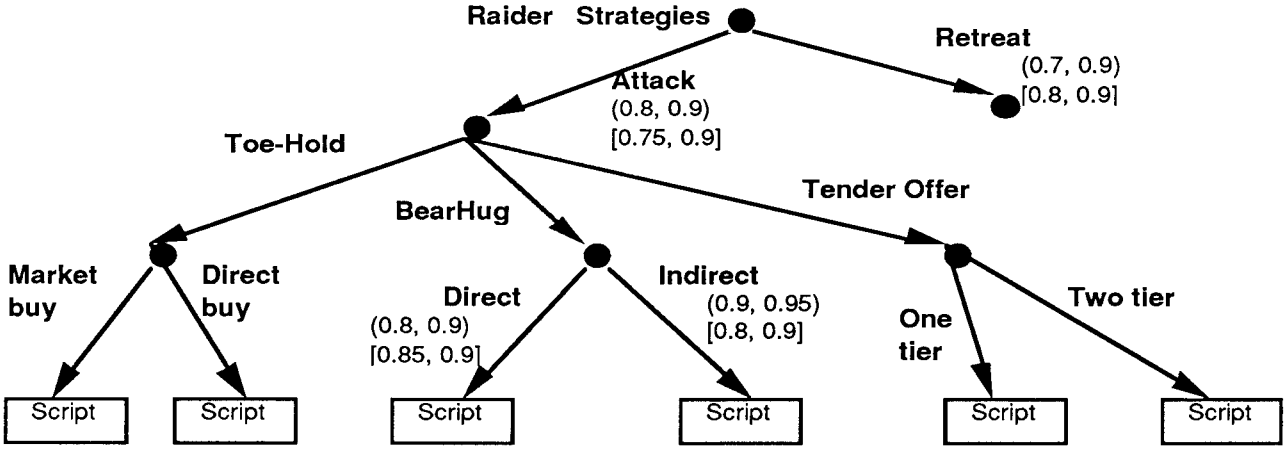


Figure 1: Simplified Strategy Hierarchy for the Raider

constitute the conclusion's support. Validity flags are used to reflect the state of the certainty. For example, a flag can indicate that the uncertainty measure is valid, unreliable (because of a change in support), too ignorant to be useful, or inconsistent with respect to the other evidence. A lazy evaluation is performed on the changes propagated by changes in the environment.

3.2 Aims & Goals of Planning

We define dynamic planning as the *process by which long term goals are transformed into short term objectives and tasks.* The planner has to continuously select and fine-tune the most desirable short-term objectives and tasks, given the long term goals, the past history of actions, the current world state and the predicted future course of events. Such a description of planning has its similarities and differences with the models considered by other researchers. It is similar to the models of Agre [1], Schoppers [26], and Nilsson [21] in stressing the need to *continuously react to a changing environment.* It is different in emphasizing the importance of strategy formulation in the planning process. Strategy formulation refers to something more than the mere determination of a goal hierarchy; it refers to the process of determining *the best way* to achieve a long term goal. The emphasis is not just on finding the sequence of steps on a path to a goal, but on choosing between different paths by reasoning about the various attributes of each solution path given the context of planning (goals, resource constraints, etc.)

The process of strategy formulation is, in general, a hierarchical process. Choices made at a given level influence subsequent strategic choices and the selection of tactical tasks. For example, a raider can adopt many different strategies in taking over a target company. His choice of a general attack strategy will influence his choice of sub-strategies for subsequent actions in the takeover attempt, as his strategy gets translated into shorter term (tactical) objectives and tasks.

3.3 Representation of Plans

The planning model developed in this paper provides for an explicit representation of various strategic choices and tactical tasks. The underlying representation is that of a *strategy hierarchy.* A strategy hierarchy is a decision tree like structure. At each node of the decision tree, certain choices have to be made relative to the context, i.e., the current goals of planning and the state of the dynamic world. These choices vary in their strategic vs. tactical content depending upon the level in the strategy hierarchy. The strategic content is highest in decisions closer to the root node and gradually decreases to the leaf nodes where the tactical content is the highest. Complex reasoning is required for evaluating the *desirability* of alternative strategic choices at a node. This reasoning is encoded by rules. Planning scripts represent tactical tasks or task sequences necessary to be executed once certain strategic choices have been made. Most scripts are located closer to the leaf nodes as they represent the tactical aspects of planning and are dependent on the strategic choices made higher up in the

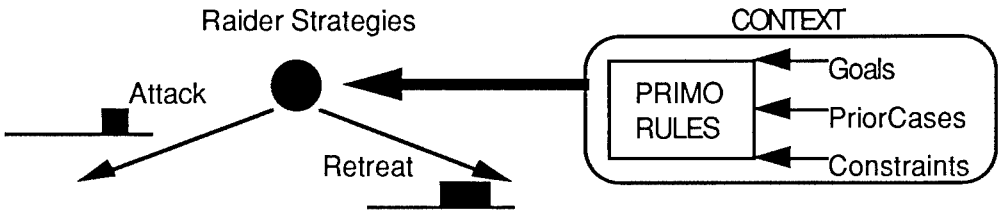


Figure 2: Strategic Choice at Node

decision tree. These concepts are explained below with the aid of an example.

Figure 1 depicts the simplified conceptual structure of the partial strategy hierarchy for the raider in MARS. At the top level, the raider can only adopt one of the two possible strategies: *attack* the target or *retreat* and concede defeat. Assuming that the raider chooses the attack strategy, there are several different sub-strategies available: *toe-hold* (slowly acquiring stock in the target), *bear-hug* (making a private merger offer to the company), or *tender-offer* (making a public merger offer). After this choice is made, the raider must select among further options. For example, the bear hug sub-strategy leads to two options: to contact the target management either *directly* or *indirectly* (through intermediate contacts). Based on his choice of appropriate strategies, the raider has to execute certain actions to achieve short term goals in consonance with his long term strategy and goals. These actions are represented by plan scripts (shown as rectangles at the leaf nodes in Figure 1).

The strategy hierarchy in Figure 1 depicts the hierarchical process of strategy formulation. Each dark circle in the decision tree indicates a point at which certain strategic choices must be made by reasoning about the current state, the goals and constraints, the known history and the predicted future states. This reasoning process has to include expert domain knowledge, constraints (on resources, goals and actions), uncertainty (in knowledge of current and predicted states of the world) and analogical knowledge (comparisons with other similar situations). This has been depicted in Figure 2 where the enclosed box to the right of the decision node shows the context in which the decision is made.

We represent the reasoning processes at each node by RUM/PRIMO rules as shown in Figure 2. The evaluation of these rules yields (on applying the uncertainty calculi of PRIMO) an interval valued measure of the *desirability* of the various strategic choices at that node (shown numerically in Figure 1 and graphically in Figure 2). Thus the choice at each node of the decision tree is no longer a simple YES/NO decision, but is decided by the degrees of "desirability" and "undesirability" of various strategies as given by PRIMO rules. Fairly complex reasoning mechanisms can be incorporated into this architecture.

The concept of a strategy hierarchy described here is different from that of a goal/sub-goal decomposition or a plan/sub-plan hierarchy as typically described in the planning literature [9]. The decision nodes in Figure 1 do not specify an order of goals and sub-goals or a hierarchical abstraction of plans/sub-plans. Rather they specify choices about how to plan and react so as to achieve the long term goals, given the context of problem solution.

The level of abstraction of strategy formulation is highest at the level of the root node and is incrementally refined as one moves towards the leaf nodes. Close to the leaf nodes, strategic choices have been refined down to a level at which they directly impact short term goals and actions. These short term goals and tasks are represented by scripts. For example, let us assume that the raider has decided to attack a particular target and has opted for a direct bear hug strategy. After this strategy selection, the raider must perform certain atomic actions in accordance with his choice: he must determine the details of an offer (based on his evaluation of the company's assets and management) and he must choose how to approach the target management with the offer. These actions are included in the script associated with the choice of the direct bear hug strategy.

Each script box (of Figure 1) has three parts: **begin *strategy* script** (actions executed when a *strategy* is initially chosen), **continue *strategy* script** (actions

executed in current planning cycle when the *strategy* has already been chosen), and **end *strategy* script** (actions executed when the *strategy* is closed, either successfully or unsuccessfully). The **begin** and **end** strategy scripts are selected for execution only once, while the **continue** strategy script can be selected for many contiguous planning cycles. Each script contains a sequence of steps or tasks which are executed once the script is selected for execution. There is no required ordering on the tasks within a script. The only requirement is that the script report back to the planner any failure in executing a particular task or task sequence. In the event of such a failure, it is the planner's responsibility to replan to avoid the failed state. The scripts that encode the tactical task details are similar to skeletal plans and scripts, as described in the planning literature [25,27].

3.4 Simulation Cycle

We have tested our planning approach within the simulation facilities of the MARS system. The MARS simulation cycle begins with an initialization phase in which attributes of the various players are entered (e.g., the assets and goals of the raider). After this initialization phase, the simulation cycle in the *autonomous* mode of operation consists of four steps:

1) the raider makes a move;
2) the target responds to the raider's move;
3) the arbitrageur shifts assets in response to the observed raider and target actions;
4) the global simulator modifies the macro-economic parameters appropriately.

This cycle is repeated continuously until the raider is either successful or concedes defeat. Each simulator plans and executes one or more tasks/actions during each simulation cycle.

MARS can operate in three different modes, all of which can be arbitrarily interleaved: *autonomous* (in which all simulators plan and operate autonomously), *I am target* (in which the user plays against the raider) and *I am raider* (in which the user plays against the target). Variable values and planning choices can be changed interactively by the user at any time. These three modes of operation produce a rich variety of planning behaviors and reactions on the part of the raider, the target and the arbitrageur.

3.5 Strategic Planning Process

The planning and execution of actions is done in a dynamic environment. The process of strategic planning requires the selection of the most *desirable* strategies, by moving down the strategy hierarchy (as illustrated in Figure 1), and the execution of the scripts associated with the chosen strategies. The complexity arises from the fact that the choice of various strategies must account for various goals, resources, constraints, and other features of the domain, such as uncertainty and prior cases. As the world is dynamic, the planner must also ensure that, at every stage of planning, its choices are in agreement with the current evaluation of the world. The dynamic nature of the world may necessitate the change of earlier choices, making the ability to replan a necessary feature. In the example below, the raider simulator is used to explain this process.

Various factors affect the choice between the *attack* and *retreat* strategies for the raider. RULE 1 is a PRIMO rule that reflects the importance of the financial strength of the raider on this decision. Leaving aside details of PRIMO's syntax (see [3,4]), RULE 1 states that, given a highly desirable target, if the raider is financially stable and has adequate financing, he should choose the *:attack* strategy to take over such a target. Lines 3 and 4 describe the context, i.e., the prerequisite for evaluating this rule. In this example, the context is the identification of a target company that is highly desirable to the raider. Line 5 represents the premise. "Adequate-financing" and "financial-stability" are two user-defined functions returning interval valued qualifications of the degree to which the raider has adequate financing and is financially stable, respectively. Lines 6 & 7 represent the conclusion. The contribution of RULE 1 to the desirability or undesirability of the *:attack* strategy is computed using the T_2 T-norm operator in accordance with PRIMO's uncertainty calculi. The degree of importance of the premises for the conclusion is expressed by the *sufficiency* and *necessity* measures (*extremely-likely* and *likely* respectively - line 7) and by the choice of the appropriate T-Norm operator (T_2 in this case).

Consider the decision node in Figure 1 at which the raider must decide whether to attack or retreat. RULE 1 is one rule

```
;;; RULE 1

(def-rule (adequate-financing-available        company-data                ;;; line #1
          (raider-strategies.rules))      (?raider ?target)                ;;; line #2
      (lb-pass-threshold                                                   ;;; line #3
           (most-desirable-target ?raider ?target) 1000)                   ;;; line #4
      (t2     (adequate-financing ?raider) (financial-stability ?raider))  ;;; line #5
      ((raider-strategy ?raider)                                           ;;; line #6
       ((:attack (d2 *extremely-likely* *likely*)) :intersect)))           ;;; line #7
```

contributing to the degree of confirmation/refutation of the desirability of the strategy *:attack*. In general, there may be other PRIMO rules contributing to the desirability of the strategy *:attack*. These rules yield, on the application of PRIMO's uncertainty calculus (section 3.1), a net interval valued measure of desirability of the *:attack* strategy for the raider. Similarly another set of rules (such as RULE 2 below) yield an interval valued measure of the desirability of the *:retreat* strategy for the raider. These desirability measures are represented graphically in Figure 2 (the black band delimits the lower and upper bounds on the desirability measures). The strategic choice made by the raider depends on the chosen selection mode. The default mode in MARS is to choose the strategy with the highest degree of confirmed desirability (i.e., with the highest lower bound).

3.6 Integrating Case-Based Reasoning in the Planning Process

The presence of adequate financing is one feature that is considered by the raider while deciding whether or not to attack the target. The possibility of a successful anti-trust move is another relevant factor, specially if the raider and target are in similar industry sectors. The relevance of this factor is represented by RULE 2. An important factor in the determination of the success of an anti-trust move is the presence of similar prior situations (cases), in which the merger move either succeeded or was blocked. The relevance of prior cases is represented by Rule 3.

When the premise of RULE 3 is evaluated, the planner accesses the case library of MARS to determine the presence or absence of similar precedents. Cases are stored in the MARS case library by rule templates (of the same form as RULEs 1, 2 and 3) and hence when accessed by the premise of RULE 3, yield an interval valued answer expressing the degree of confirmation/refutation of the presence of a similar precedent. RULE 3 expresses the degree of relevance of the presence of prior cases to the conclusion of whether an anti-trust move shall succeed (i.e., to the premise of RULE 2). In general, other rules will also contribute to the estimation of the the premise of RULE 2. RULE 2 in turn, contributes (along with other rules) to the evaluation of the desirability of the strategy *:retreat* for the raider simulator. The inference engine of the planner remains unaffected by the use of or lack of use of case-based reasoning. Whenever, rules (such as RULE 3) explicitly mention the importance of prior cases for the current conclusion, the planner accesses the case library and evaluates the rule graph corresponding to the relevant instantiated case templates. This process makes it possible to seamlessly integrate case based reasoning into the planning process. More details on case based reasoning as implemented in MARS and its integration with rule based reasoning are given in reference [14].

3.7 Tactical Planning Process

The scripts associated with nodes in the strategy hierarchy represent the tactical details of planning. As the external world is assumed to be dynamic and uncertain, the ability to detect failures and to replan is crucial. Replanning may be necessary due to two reasons: **scripts may fail** (one or more script actions may be non-executable) or **chosen strategies may be sub-optimal** (choices made higher up in the strategy hierarchy are no longer optimal). In the first situation, it is fairly simple to detect the need for replanning. Since each script associated with a selected decision node in the strategy hierarchy is executed before exploring any other decision nodes, the planner waits for the successful completion of the script before making any more strategic choices. If the script fails, then the planner selects the next best alternative path at the same level and continues. If there are no other alternative

```
;;; RULE 2 (Abridged version)
(def-rule (anti-trust-possibility .....                    ;; rule details omitted
   (t2 (value (anti-trust-success ?raider) :yes))          ;; premise
   ((raider-strategy ?raider)
      ((:retreat (d2 *very-likely* *likely*)) :intersect)))  ;; conclusion

;;; RULE 3 (Abridged version)
(def-rule (precedent-anti-trust .....                      ;; rule details omitted
   (t2 (successful-precedent ?raider ?target))             ;; premise
   ((anti-trust-success ?raider)
      ((:yes (d2 *very-likely* *likely*)) :intersect)))    ;; conclusion
```

paths at that level, the planner moves one level **up** in the strategy hierarchy and picks the next best alternative (i.e., relative to the alternatives already seen) at that level. There is no explicit repair mechanism which is invoked when a script fails. Each script contains the information to explore different methods to achieve a certain short term goal and any necessary knowledge for repairing possible failures. A script fails only if all methods and repair procedures are exhausted (similar to RAPS [17]).

The second cause of plan failure occurs due to dynamic changes in the world. Certain strategies chosen earlier (higher up in the strategy hierarchy) may no longer be the best strategy (i.e., the strategy with the highest degree of *desirability*) at a later time (when the planner has reached lower levels in the strategy hierarchy) due to a changed environment. The planner detects this change with the help of a belief revision mechanism supported in PRIMO (section 3.1). A strategic decision at any node in the strategy hierarchy is made in a given context (Figure 2). The PRIMO rules (such as RULEs 1, 2 and 3) used for evaluating the desirability of different alternatives at that node form a rule-graph. The belief revision mechanism keeps track of the dependencies of these rules and flags any changes in the values affecting these rules. At every decision node, before moving down the strategy hierarchy, the planner checks to see if the strategic choices made until now are still optimal, i.e., if the path from the root node to the current node is still optimal. This test ensures that the context under which planning is being done is still valid. If the world has changed causing an earlier strategic choice to become sub-optimal, then the planner backtracks up the strategy hierarchy to the node where the strategic choices have to be reevaluated. The planner must terminate the open strategies along the backtracking path, i.e., it must execute the *end strategy scripts* of the terminated strategies. Replanning can then proceed from that node.

The planner balances the strategic vs. tactical aspects of planning by introducing thresholds for changing strategies which vary according to the strategic importance of the choice. These thresholds are higher for strategically important choices (closer to the root node in Figure 2) than for choices more intimately related to tactical planning (closer to the leaf nodes in Figure 2). For example, consider the desirabilities of certain strategic choices during a particular execution cycle as shown by the numbers in parentheses (e.g., (.8, .9)) in Figure 1. Based on the depicted measures of *desirabilities*, the script for the *indirect bear hug attack strategy* has been selected and is in progress (the *begin indirect bear hug strategy script* has been executed). Assume that during the next execution cycle the desirability of various strategic choices for the raider has changed to the numbers shown in squared brackets (e.g., [0.8, 0.9]) in Figure 1. The degree of desirability of the *direct bear hug* strategy is now higher than the currently chosen *indirect bear hug* strategy (0.85 > 0.8). Thus the *indirect bear hug* script should be closed and the *direct bear hug* script should be started. At this point, the degree of desirability of the *:retreat* strategy is also greater than that of the *:attack* strategy. However, we may decide not to change to the *:retreat* strategy as this decision is intimately related to the long term goals of the raider and the difference in the desirabilities (0.05) is perhaps not large enough to warrant a change during this cycle. This allows the planner to achieve a balance between strategic and tactical planning. The change thresholds can either be fixed or dynamic. In the latter case, the planner can dynamically vary the strategic and tactical contents of its actions. In this example, the magnitude of change in the desirability of the two changes has been deliberately

chosen to be the same (0.05) to emphasize that different change thresholds can be adopted at different levels in the strategy hierarchy to account for varying impacts of strategic vs. tactical planning.

4 Conclusion

In this paper, we have described an approach to merging the strategic and tactical aspects of planning. We view planning as the process by which the long term goals of an agent are translated into short term objectives and tasks. We have used the term *strategic planning* to refer to reasoning about and choosing between alternative means to achieve the long term goals, given the context of planning (goals, constraints, etc.). This interpretation differs from the one common in the planning literature, in which strategic planning is often viewed as planning in a static, predictable world. Our interpretation of strategic planning reflects the use of the term in the military and business domains. The concept of a strategy hierarchy has been introduced to explicitly represent the varying strategic and tactical aspects of planning. The identification, selection and structuring of possible strategic choices is a domain dependent task. A rule based approach is used for reasoning about and evaluating the degrees of desirabilities of the various strategic choices at different nodes in the strategy hierarchy. PRIMO's uncertainty calculus provides a thorough treatment of uncertainty in the domain, which is reflected in the execution of such rules. The tactical details of planning are represented by scripts, which are usually found close to the leaf nodes of the strategy hierarchy. We have also demonstrated a methodology for incorporating case based reasoning into the strategic planning process. Our planning model emphasizes a dynamic balance between the strategic vs. tactical aspects of planning. This balance is achieved by introducing dynamic and flexible thresholds in the strategy hierarchy. These thresholds vary with the amount of strategic vs. tactical content of the planning choices. Such a balance is important for achieving coherent goal directed behavior.

We have successfully tested our planning approach in the domain of Mergers and Acquisitions (M&A), and we have implemented it in the MARS system. A characteristic of the M&A domain is the competitive nature of planning under partial and imprecise information. The raider and the management of the target company can be seen as two players in a multi-player game. Each player has a set of strategies, which are determined and conditioned by the player's assets (constraints), high level goals, attitude, and the current macro-economic environment. These strategies may be altered when drastic changes occur to the environment or to the high level goals.

The implementation of the strategies is determined by the selection of sub-strategies and by their refinements until specific tactics are executed. Because of the competitive nature of the domain, planning is followed by counter-planning, whose effects may cause the modification, interruption, or termination of the adopted tactic. These tactical changes occur much more frequently than the strategic ones.

The planning requirements described above are similar to those of many military applications. Our next experiment will be the testing and validation of our planning approach in the domain of military transportation planning, within the context of crisis management.

References

[1] Agre, P. and D. Chapman, Pengi: An Implementation of a Theory of Activity, in *Proceedings of the 6th AAAI*, Morgan Kaufmann Publishers, 1987, pp. 268-272.

[2] *AI Magazine*, vol. 9, no. 2, Summer 1988.

[3] Aragones, J. & P.P. Bonissone, PRIMO: A Tool for Reasoning with Incomplete and Uncertain Information, in the *Proceedings of the 3rd International Conference on Information Processing & Management of Uncertainty in Knowledge-Based Systems*, Paris, July, 1990, pp. 891-898.

[4] Bonissone, P.P., S. Gans, and K.S. Decker, RUM: A Layered Architecture for Reasoning with Uncertainty, in the *Proceedings of the 10th IJCAI*, 1987.

[5] Bonissone, P.P. & S. Dutta, MARS: A Mergers & Acquisitions Reasoning System, Forthcoming in *Computer Science in Economics & Management*, Kluwer Academic.

[6] Bresina, J. & M. Drummond, Integrating Planning and Reaction: A preliminary report, in *Proc. of the AAAI Spring Symposium on Planning in Uncertain,*

Unpredictable or Changing Environments, Stanford, 1990, pp. 24-28.

[7] Brooks, R. A., A Robust Layered Control System for a Mobile Robot, AI Memo # 864, AI Laboratory, MIT, 1985.

[8] Chien, R.T., & S. Weissman, Planning and Execution in Incompletely specified Environments, in the *Proceedings of the 4th IJCAI*, 1975, pp. 169-174.

[9] Cohen, P. & E. A.. Feigenbaum, *The Handbook of AI*, Vol. III, Addison Wesley, 1982

[10] *Computational Intelligence*, Special Issue on Planning, vol. 4, no. 4, Nov. 1988.

[11] Dean, T. & M. Boddy, An Analysis of Time-Dependent Planning, in the *Proceedings of the 7th AAAI*, 1988, pp. 103-113.

[12] Dempster, A.P., Upper and Lower Probabilities Induced by a Multi-valued Mapping, *Annals of Mathematical Statistics*, 38, pp. 325-339, 1967.

[13] Durfee, E., & V. Lesser, Incremental Planning to Control a Blackboard-Based Problem Solver, in the *Proceedings of the 5th AAAI*, 1986, pp. 58-64.

[14] Dutta, S. & P.P. Bonissone, Integrating Case Based and Rule Based Reasoning: The Possibilistic Connection, in the *Proceedings of 6th Conference on Uncertainty in AI*, Cambridge, MA, 1990, pp. 290-300.

[15] Ferrara, R. C., *Mergers and Acquisitions in the 1980s: Attack and Survival*, Practising Law Institute, New York, 1987

[16] Fikes, R.E., & N.J. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence*, 2, 1971, pp. 189-208.

[17] Firby, R., An Investigation into Reactive Planning in Complex Domains, in the *Proceedings of the 6th AAAI*, 1987, pp. 202-206.

[18] Georgeff, M. and A. Lansky, Reactive Planning and Reasoning, in the *Proceedings of the 6th AAAI*, 1987, pp. 677-682.

[19] Hendler, J. and J. Sanborn, A Model of Reaction for Planning in Dynamic Environments, in the *Proceedings of the DARPA Knowledge-Based Planning Workshop*, 1987, pp. 24-1-24-10.

[20] Kuhn, R. L., *Mergers, Acquisitions and Leveraged Buyouts*, Vol. IV, Dow-Jones Irwin, 1990.

[21] Nilsson, N.J., Action Networks, in the *Proceedings of the Rochester Planning Workshop*, Oct. 1988, pp. 21-52.

[22] *Proceedings of the AAAI Spring Symposium on Planning in Uncertain, Unpredictable or Changing Environments*, Stanford, March 1990.

[23] Rosenschien S., J. Kaelbling & L. Pack, The Synthesis of Digital Machines with Provable Epistemic Properties, in the *Proceedings of the Conference on Theoretical Aspects of Reasoning about Knowledge*, 1986, pp. 83-98.

[24] Sacerdoti, E., *A Structure for Plans and Behavior*, American Elsevier, 1977.

[25] Schank, R.C., Scripts, *Plans, Goals and Understanding*, Lawrence Erlbaum, 1977.

[26] Schoppers, M.J., Universal Plans for Reactive Robots in Unpredictable Domains, in the *Proceedings of the 10th IJCAI*, 1987, pp. 1039-1046.

[27] Stefik, M.J., Planning with Constraints, Report no. 80-784, Ph.D. dissertation, Stanford University, 1980.

[28] Swartout, W., DARPA Workshop on Planning, *the AI Magazine*, vol. 9, no. 2, 1988, pp. 115-130.

[29] Wilkins, D., *Practical Planning*, Morgan Kaufmann Publishers, 1988.

[30] Yang, H., D.H. Fisher, & H. Franke, Planning, Replanning, and Learning with an Abstraction Hierarchy, in the *Proceedings of the AAAI Spring Symposium on Planning in Uncertain, Unpredictable or Changing Environments*, Stanford, 1990, pp. 151-155.

Planning Under Uncertainty and Time Pressure

Thomas Dean*
Department of Computer Science
Brown University, Box 1910, Providence, RI 02912

Abstract

We are interested in the role of prediction in planning and control. Our research has focussed on the problems involved in making predictions under uncertainty and time pressure. In this paper, we consider two ideas from related disciplines that have had an important influence on our work. The first idea is from estimation and control theory and is concerned with integrating observation and prediction. The second idea is from decision theory and experimental design and is concerned with computing the expected value of information. We provide brief introductions to each of these ideas, and attempt to provide some insight into their utility for artificial intelligence applications by supplying examples from our recent research.

Introduction

Prediction and modeling are important in building planning systems. Indeed, one informal characterization of planning is in terms of predicting possible futures so as to select among them by performing certain actions. Much of our research is aimed at improving the technology available for predicting possible futures. We have developed techniques for reasoning about causal relationships and metric time [Dean and McDermott, 1987, Dean, 1988, Dean, 1989], dealing with uncertainty in the order of events [Dean and Boddy, 1988b] and the persistence of propositions [Dean and Kanazawa, 1989], and reasoning about continuously changing quantities [Dean and Siegle, 1990].

Recently, we have begun to pay greater attention to two issues that are critical in applying such techniques to real-world planning problems. The first issue concerns controlling the computational costs associated with prediction. In many applications, it is important to carefully control the time spent performing inference in order to ensure that the system responds to its environment in a timely manner. This control of inference can be compiled into the system by making tradeoffs at design time to guarantee a specific response time. Alternatively, control of inference can be made part of the run-time system to enable the system to allocate computational resources based on current demands.

The second issue concerns the role of observation and sensing in designing useful predictive systems. Predictive models are generally poor at making long-term predictions. In addition, the accuracy of both near- and long-term predictions are critically dependent on the system's estimate of the current conditions. By making frequent measurements, a predictive system can maintain an accurate estimate of the current conditions in order to make more accurate predictions. For the most part, research in planning has focussed on open-loop feedforward techniques that rely on precise models and accurate information regarding initial conditions. For many applications, closed-loop feedback techniques appear to be more appropriate.

In the following, we consider some ideas drawn from other disciplines and show how they bear on the two issues mentioned above. In particular, we look at the theory of Kalman-Bucy filters in control theory for ideas about how to treat observations that differ from current expectations. From decision theory and experimental design, we consider how to assign value to various information sources, including observations we might make and computations we might perform. In each case, the goal is to show how these ideas from other disciplines can inform research on planning in artificial intelligence.

Observation and Prediction

We begin with a very brief introduction to Kalman filtering to illustrate how observation and prediction can be combined to complement one another. We then show how to apply the basic intuitions underlying the Kalman filter to problems in mobile robotics.

In the following, we assume a discrete-time, dynamic system. Let $\mathbf{x}(k)$ be a vector representing the state of the system at time k. Let $\mathbf{u}(k)$ be the known input (or

*This work was supported in part by a National Science Foundation Presidential Young Investigator Award IRI-8957601 with matching funds from IBM, and by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Air Force Office of Scientific Research under Contract No. F49620-88-C-0132.

control action) to the system at time k. The state of the system at time $k+1$ is determined by

$$\mathbf{x}(k+1) = f(\mathbf{x}(k), \mathbf{u}(k)) + \mathbf{v}(k),$$

where f models the response of the dynamic system to a given input, and $\mathbf{v}(k)$ is a vector of Gaussian-process noise, modeling the input disturbance or process noise.

Let $\mathbf{z}(k)$ represent the (observable) output of the system at time k, so that

$$\mathbf{z}(k) = h(\mathbf{x}(k)) + \mathbf{w}(k),$$

where h models the physics of the measurement process and $\mathbf{w}(k)$ is a vector of Gaussian-process noise, modeling the measurement errors.

The objective is to maintain an accurate estimate of the current state of the system. The estimate of the system state at time k given all the measurements up until time j is denoted $\hat{\mathbf{x}}(k|j)$. At each time k, all of the past measurements are summarized by an estimate, $\hat{\mathbf{x}}(k|k)$ and an associated covariance matrix.

There are three basic steps performed in updating the estimate of the system state to reflect the measurement made at $k+1$. In the first step, called the *prediction step*, we compute what we expect to observe at $k+1$. This involves first computing an estimate of the state at $k+1$ given all the measurements up until k:

$$\hat{\mathbf{x}}(k+1|k) = f(\hat{\mathbf{x}}(k|k), \mathbf{u}(k)).$$

We then use this estimate to compute the predicted measurement:

$$\hat{\mathbf{z}}(k+1|k) = h(\hat{\mathbf{x}}(k+1|k)).$$

In the second step, called the *observation step*, we make the observation and compare the resulting measurement with what we expected. The difference between the expected and actual measurement is called the *innovation*:

$$\nu(k+1) = \mathbf{z}(k+1) - \hat{\mathbf{z}}(k+1|k).$$

In the third and final step, called the *estimation step*, we take $\hat{\mathbf{x}}(k+1|k+1)$ to be the vector sum of our estimate given the measurements up until k and a correction factor $g(\nu(k+1))$ which reflects our confidence in the measurement at $k+1$:

$$\hat{\mathbf{x}}(k+1|k+1) = \hat{\mathbf{x}}(k+1|k) + g(\nu(k+1).$$

The details concerning g are not important.[1] What is important is how the information from prediction and observation are combined.

[1] We have sketched here the basic ideas involved in Kalman filtering. In general, the information regarding the estimated state of the system is summarized by the first two moments of a statistical distribution: the mean vector and the covariance matrix for the state variables. The correction function g takes into account of the covariance information to discount measurements that deviate significantly from expectations. See Bar-Shalom [1988] or Brammer and Siffling [1989] for introductions to the theory of Kalman filters.

First, note that we have models for predicting not only the current and future states of the system, but also the current and future measurements made in observing the system. These models account for uncertainty in the underlying process by incorporating probabilistic noise models for disturbances in the dynamical system and errors in measurement. At each point in time, we compare what we expect to observe with what we actually observe in order to determine how much weight we want to attribute to each, based on what sort of errors we expect from the noise models. In the following, we provide some examples to illustrate how we are applying this basic idea.

The mobile robots in our lab use sonar as the primary means of sensing the surfaces of objects for navigation purposes. A sonar sensor consists of an ultrasonic transducer, a receiver, and some signal-processing hardware. Information about the distance from the sensor to nearby surfaces is obtained by measuring the round-trip time of flight of an ultrasonic pulse that is emitted by the transducer, bounces off an object surface, and returns to the receiver.

If the transducer is pointed along a line perpendicular to a nearby planar surface, then the sensor can be modeled as the actual distance to the surface corrupted by zero-mean Gaussian noise. However, if the transducer is not pointed perpendicular to the nearest object surface, then there is some chance that not enough of the energy from the ultrasonic pulse will be returned to the receiver to determine the true time of flight to the nearest surface. Instead, the pulse may be reflected, bouncing off possibly several objects before a signal with enough energy is detected by the receiver. In this case, the information returned by the sensor may deviate significantly from the distance to the nearest object.

Kalman filtering techniques can be used to maintain estimates of the distance separating a mobile robot from nearby walls, corners, and other environmental features that exhibit well-behaved sonar signatures [Leonard and Durrant-Whyte, 1989]. We use these estimates to update the robot's position with respect to a global map, and to track walls in negotiating corridors in the computer science building [Lee, 1990]. The navigation system identifies features and then tracks them over time using the Kalman filtering equations to discount misleading sonar measurements due to multiple reflections.

The basic Kalman filtering equations are also central to the stochastic geometric modeling techniques developed by Smith and Cheeseman [1986] and Durrant-Whyte [1988]. In [Hayahsi and Dean, 1988], we describe a method for locating an autonomous mobile robot using local observations and a global satellite map. The satellite map provides approximate elevation data for an area within which the robot is known to be located. The map consists of a coarse grid of rectangular regions

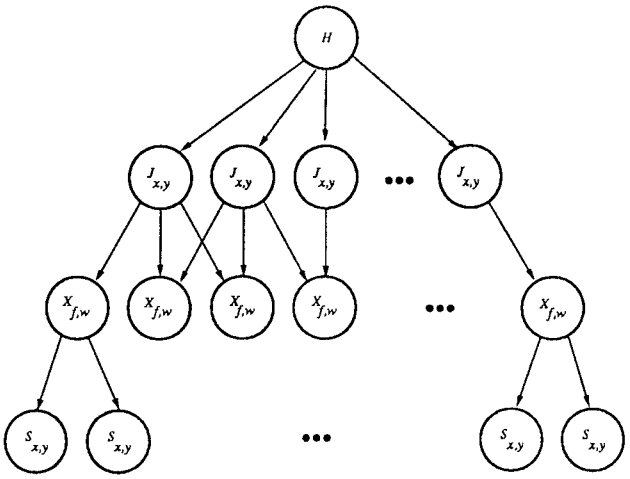


Figure 1: The probabilistic model for map learning

annotated with upper and lower bounds on the elevation within the region. In exploring its environment, the robot makes measurements to extract information about the relative position and orientation of local landmarks. These landmarks are integrated into a stochastic map which is then matched with the satellite map to obtain an estimate of the robot's current location.

We employ the Kalman filtering equations directly in a number of our navigation routines, but the basic intuitions underlying the Kalman filter are applicable to wide variety of problems including problems that do not involve reasoning about continuous quantities.

In our research on learning representations of large-scale space, we have adopted Kuipers' [1978] approach to map learning as inducing a graph that captures certain qualitative features of the environment [Basye *et al.*, 1989]. In recent work, we have cast the problem of combining measurements to support hypotheses concerning the configuration of such qualitative features in terms of Bayesian inference [Dean *et al.*, 1990a]. Our methods involve encoding the underlying decision problem as a Bayesian probabilistic network.[2]

We assume that the robot trying to learn the map can enumerate the set, $M = \{M_1, M_2, \ldots, M_m\}$, of all possible maps, where each M_i is just a labeled graph. Since the size of M is potentially quite large, we restrict it in a number of ways. In particular, we assume that the system of junctions and corridors that make up our robot's environment can be registered on a grid, so that every corridor is aligned with a grid line and every junction is coincident with the intersection of two grid lines.

The Bayesian network is defined as follows. Let H be a random variable corresponding to the actual configuration of the environment; H takes on values from M. Let $J_{x,y}$ be a random variable corresponding to the label of the intersection at the coordinates, $\langle x, y \rangle$, in the grid; $J_{x,y}$ can take on values from the set of possible junction types (e.g., T junctions and L junctions). Let $X_{f,w}$ correspond the presence of a feature, f, at a particular position, w. Let $S_{x,y}$ be a random variable corresponding to a possible measurement taken at the coordinates, $\langle x, y \rangle$, in the grid. The complete probabilistic model is shown in Figure 1.

Even after restricting M, the model shown in Figure 1 is prohibitively expensive to evaluate.[3] To ease the computational burden, we heuristically select a subset of M to use as the sample space for H. The problem with this is that the space of possible maps chosen may not include the map corresponding to the actual configuration of the environment.

To handle the possibility of excluding the real map, we add a special value, $\perp$, to the sample space for H, and make all of the $\Pr(J_{x,y}|\perp)$ entries in the conditional probability tables $1/s$ where s is the number of junction types. If the posterior probability for $H = \perp$ given the evidence ever exceeds a fixed threshold, then the system assumes that it has excluded the real map, and dynamically adjusts its decision model by computing a new sample space for H guided by the results of the exploratory actions taken thus far.

In the estimation step of the Kalman filter, we use the innovation to discount measurements that are unlikely to be relevant to the current estimate. In the map learning approach described above, we use the posterior probability for $\perp$ as an indication that the current sample space for the hypothesis is inadequate. In both cases, we use knowledge about the accuracy of our predictive model and our ability to observe the environment to reason about the weight to attach to each in making decisions. In Kalman filtering, this knowledge takes the form of the innovation and covariance. In map learning, this knowledge is condensed in the form of the threshold used to determine if the real map has been excluded from the sample space for H, and the conditional probabilities of the form $\Pr(S_{x,y}|X_{f,w})$ used to quantify the dependency between observations and features in the real world.

Value of Information

In the real world, information costs. Every time that you get operator assistance in dialing a long-distance number or consult an accountant about your income tax you are paying for information. It is often useful to be able to assess the value of information so as to make reasonable decisions regarding whether or not to pay for it. In this section, we consider a theory due to Howard [1966] concerned with quantifying the value of

[2] See [Dean *et al.*, 1990b] in this volume for a brief introduction to Bayesian probabilistic networks.

[3] See [Dean *et al.*, 1990b] in this volume for a discussion of some of the complexity issues involved in evaluating Bayesian probabilistic networks.

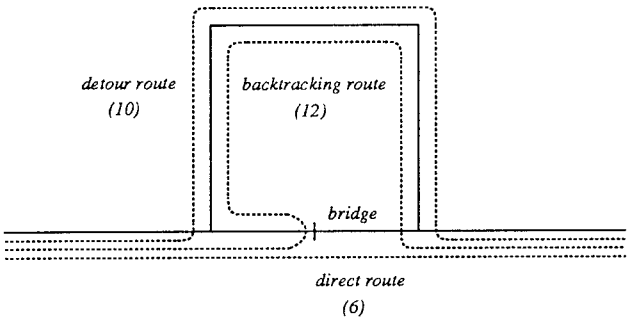


Figure 2: Alternative routes to the beach

information in decision-theoretic terms.

Suppose that you live in the city and are taking your summer vacation at a beach some distance from the city. Suppose further that there are two routes to the beach: a direct route that takes six hours and roundabout route that takes ten hours. We will call these the *direct* and *detour* routes. The direct route requires that you cross a bridge which, as luck would have it, is undergoing major repairs this summer. There is a 50% chance that the bridge will be closed at the time you wish to cross it. If you attempt the direct route and find the bridge closed, you will have to backtrack to the detour route, and your total transit time will be twelve hours.

You decision involves choosing whether to try the direct or detour route first. Figure 2 shows the three possible outcomes of your decision. If you choose the detour route, the trip will take ten hours. If you choose the direct route, the trip will take either six hours or twelve hours depending on whether or not the bridge is closed. We need to assign a value or cost to each of the possible outcomes, and, in this case, a natural measure of cost is time spent in transit. A decision-theoretic analysis would conclude that the optimal decision (one that minimizes cost) is to take the direct route with an expected transit time of 9 hours.

Now we extend the example to consider issues involving the value of information. Suppose there is a state police station located near the highway prior to the point at which we have to decide between the direct and detour routes. We will assume that the state police can provide us with information about the current status of the bridge. Suppose that stopping at the police station requires getting off the highway and traveling to a nearby town, and that the total time spent in acquiring the information about the bridge is estimated to be 30 minutes.

In this extended example, we have an additional decision to make besides simply whether to take the direct or detour route. You can think of the trip to the police station as particular type of test with two possible findings: the bridge is open or the bridge is closed. We can reason about the expected value of obtaining this information as follows. Let

$$\mathrm{E}(T|\mathcal{E}) \tag{1}$$

be the expected travel time, T, for the optimal course of action based on the background information, $\mathcal{E}$. In reasoning about whether or not to stop at the state police station, we compute the expected travel time given the additional information obtained from the police:

$$\mathrm{E}(T|I_S, \mathcal{E}), \tag{2}$$

where I_S represents the event of obtaining information from the police regarding the status, S, of the bridge, either *open* or *closed*. The expected value of the information obtained from stopping at the police station is just the difference between Equations 2 and 1

$$\mathrm{E}(\mathrm{Val}(I_S)|\mathcal{E}) = \mathrm{E}(T|I_S, \mathcal{E}) - \mathrm{E}(T|\mathcal{E}),$$

where

$$\begin{aligned}\mathrm{E}(T|I_S, \mathcal{E}) \;=\; & \mathrm{E}(T|S = \mathit{closed}, \mathcal{E})\,\Pr(S = \mathit{closed}|\mathcal{E}) + \\ & \mathrm{E}(T|S = \mathit{open}, \mathcal{E})\,\Pr(S = \mathit{open}|\mathcal{E}).\end{aligned}$$

In the example, $\mathrm{E}(\mathrm{Val}(I_S)|\mathcal{E}) = 1.0$, implying that we should be willing to spend up to one hour to obtain the information regarding the status of the bridge.

More generally, let $\mathrm{E}(V|\mathcal{E})$ be the expected value of carrying out your present plan or policy. Suppose that, prior to carrying out your present policy, someone offers to sell you information pertaining to some variable, X, used in calculating $\mathrm{E}(V|\mathcal{E})$. To be more specific, suppose that the informant is clairvoyant and knows the actual value of X. Let I_X correspond to the event of obtaining the information regarding X.

The expected value of obtaining this information is given by

$$\mathrm{E}(\mathrm{Val}(I_X)|\mathcal{E}) = \mathrm{E}(V|I_X, \mathcal{E}) - \mathrm{E}(V|\mathcal{E}). \tag{3}$$

To compute $\mathrm{E}(V|I_X, \mathcal{E})$, we evaluate the expectation given knowledge about X for each possible value of X provided by the informant, summing over these expectations weighted by our prior on X

$$\mathrm{E}(V|I_X, \mathcal{E}) = \sum_{x \in \Omega_X} \mathrm{E}(V|X = x, \mathcal{E})\,\Pr(X = x|\mathcal{E}). \tag{4}$$

It is important to note, as did Howard in the 1966 paper [Howard, 1966] in which he introduced Equations 3 and 4, that we use the prior distribution $\Pr(X|\mathcal{E})$ for X because, until the informant provides the information about X, our knowledge of X is based entirely on our background knowledge $\mathcal{E}$.

In the map learning problem described earlier, at each point in time, the robot has to decide between two alternatives, P_K and P_U, corresponding to taking paths through known and unknown territory. It is assumed that at all times the robot has some current task (e.g., to take a parcel from one office to another). The robot has to balance the demands of its current task

against the demands of future tasks. In the case of the path through unknown territory, the robot will learn something new about its environment; it may end up taking longer to complete its current task, but it may also learn something that will enable it expedite future tasks. In [Dean *et al.*, 1990a], we describe how to compute the expected value of exploration, in order to guide the actions of the robot in exploring its environment and carrying out its tasks.

In [Dean *et al.*, 1990b] in this volume, we describe how a robot might take into account the expected value of perceptual actions in tracking moving objects. In the tracking problem, the robot has to consider sequences of actions, and the prediction and modeling issues are more complicated than in the map learning problem.

In the map learning and tracking problems, the robot reasons about the value of performing physical actions, including perceptual actions, that provide valuable information, but there is a hidden cost associated with acquiring such information: the cost of processing the information once you have it. In fact, inference is never without cost, and in time-critical applications it is important that we are careful not to squander precious computational resources.

In recent years, there has been considerable interest in reasoning about the costs and benefits of computation[4] [Boddy and Dean, 1989, Horvitz *et al.*, 1989, Russell and Wefald, 1989]. In particular, researchers have focussed on the properties of algorithms and decision procedures. For instance, in time-critical applications, it is useful to build decision procedures that can be interrupted at any time to return an answer such that the answer returned gets better, in some decision theoretic sense, the more time allowed. This sort of behavior is a fundamental property of the *anytime algorithms* of Dean and Boddy [1988a] and the *flexible computations* of Horvitz [1988].

While it is important to design decision procedures that are better suited to interacting with the real world, we believe that the more interesting and important issues involve devising better ways of allocating scarce computational resources to decision making in time-stressed situations.

In [Dean and Boddy, 1988a], we define the notion of *deliberation scheduling* in terms of scheduling anytime decision procedures on a uniprocessor. The key idea is that, given a set of anytime decision procedures and expectations about their performance as a function of time spent in computation, it is possible to optimally allocate resources to those procedures in some well-defined decision-theoretic sense.

If the time spent in deliberation scheduling is negligible in comparison with the time spent in deliberation, then the cost of deliberation scheduling can be ignored. In some cases, however, the decision problem of optimally allocating computation is itself computationally complex and hence the cost of deliberation scheduling cannot be ignored. In this case, if the deliberation scheduling algorithm can also be cast as an anytime algorithm, it is still possible in certain cases to optimally allocate computational resources to both the meta-level deliberation scheduling procedure and the base-level decision procedures.

These basic methods have been applied to dynamic planning problems in which the planning system has to continually reevaluate its computational commitments as new information becomes available [Boddy, Forthcoming].

[4] See Dean [1990] for an overview of decision-theoretic techniques for controlling inference in time critical applications.

Conclusion

In this paper, we have considered two basic ideas regarding planning in uncertain domains. The first idea concerns reasoning about how to combine information from predictive and perceptive modules given expectations about the performance of each. You should not be overly influenced by what you expect to see, but neither should you always believe what you think you see. The Kalman filter provides an elegant tool for combining observation and prediction.

The second idea concerns reasoning about the value of information, including both perceptual and computational information sources. In time-critical applications in which the time and effort involved in gathering and processing information are significant, one has to make decisions regarding what to look at and what to think about. Bayesian decision theory provides a basis for making such decisions, and Howard's value-of-information theory provides insight into computing the value of such information sources.

It should be noted that Bayesian methods do not themselves provide the solutions to the problems we are interested in. We view the invocation of Bayesian decision theory as a basis for posing problems, and the use of Bayesian networks as a convenient means of analyzing the complexity issues involved in applying Bayesian decision theory. The Bayesian methods require that we enumerate and assign values to a set of possible states of affairs and that we quantify certain dependencies involving state variables. In satisfying these requirements, the combinatorial issues and potential problems involved in gathering the necessary statistics become apparent. The underlying decision method is quite simple algorithmically, as it involves exhaustively evaluating a potentially large set of possible states of affairs. The real contributions are concerned with the representations used for characterizing the possible states of affairs. In an important sense, Bayesian methods provide a discipline for systematically exploring representational issues.

References

[Bar-Shalom and Fortmann, 1988] Yaakov Bar-Shalom and Thomas E. Fortmann. *Tracking and Data Association.* Academic Press, New York, 1988.

[Basye *et al.*, 1989] Kenneth Basye, Thomas Dean, and Jeffrey Scott Vitter. Coping with uncertainty in map learning. In *Proceedings IJCAI 11*, pages 663–668. IJCAI, 1989.

[Boddy and Dean, 1989] Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings IJCAI 11*, pages 979–984. IJCAI, 1989.

[Boddy, Forthcoming] Mark Boddy. *A Framework for Time-Dependent Planning.* PhD thesis, Brown University, Providence,RI, Forthcoming.

[Brammer and Siffling, 1989] Karl Brammer and Gerhard Siffling. *Kalman-Bucy Filters.* Artech House, Norwood, Massachusetts, 1989.

[Dean and Boddy, 1988a] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings AAAI-88*, pages 49–54. AAAI, 1988.

[Dean and Boddy, 1988b] Thomas Dean and Mark Boddy. Reasoning about partially ordered events. *Artificial Intelligence*, 36(3):375–399, 1988.

[Dean and Kanazawa, 1989] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.

[Dean and McDermott, 1987] Thomas Dean and Drew V. McDermott. Temporal data base management. *Artificial Intelligence*, 32(1):1–55, 1987.

[Dean and Siegle, 1990] Thomas Dean and Greg Siegle. An approach to reasoning about continuous change for applications in planning. In *Proceedings AAAI-90*, pages 132–137. AAAI, 1990.

[Dean *et al.*, 1990a] Thomas Dean, Kenneth Basye, Robert Chekaluk, Seungseok Hyun, Moises Lejter, and Margaret Randazza. Coping with uncertainty in a control system for navigation and exploration. In *Proceedings AAAI-90*, pages 1010–1015. AAAI, 1990.

[Dean *et al.*, 1990b] Thomas Dean, Kenneth Basye, and Moises Lejter. Planning and active perception. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control.* DARPA, 1990.

[Dean, 1988] Thomas Dean. An approach to reasoning about the effects of actions for automated planning systems. *Annals of Operations Research*, 12:147–167, 1988.

[Dean, 1989] Thomas Dean. Using temporal hierarchies to efficiently maintain large temporal databases. *Journal of the ACM*, 36(4):687–718, 1989.

[Dean, 1990] Thomas Dean. Decision-theoretic control of inference for time-critical applications. Technical Report CS-90-44, Brown University Department of Computer Science, 1990.

[Durrant-Whyte, 1988] Hugh F. Durrant-Whyte. *Integration, Coordination and Control of Multi-Sensor Robot Systems.* Kluwer Academic Publishers, Boston, Massachusetts, 1988.

[Hayahsi and Dean, 1988] Akira Hayahsi and Thomas Dean. Locating a mobile robot using local observations and a global satellite map. In *Proceedings of Third IEEE International Symposium on Intelligent Control*, pages 135–140. IEEE, 1988.

[Horvitz *et al.*, 1989] Eric J. Horvitz, Gregory F. Cooper, and David E. Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings IJCAI 11*, pages 1121–1127. IJCAI, 1989.

[Horvitz, 1988] Eric J. Horvitz. Reasoning under varying and uncertain resource constraints. In *Proceedings AAAI-88*, pages 111–116. AAAI, 1988.

[Howard, 1966] Ronald A. Howard. Information value theory. *IEEE Transactions on Systems Science and Cybernetics*, 2(1):22–26, 1966.

[Kuipers, 1978] Benjamin Kuipers. Modeling spatial knowledge. *Cognitive Science*, 2:129–153, 1978.

[Lee, 1990] Jin Joo Lee. Localization with kalman filtering. M.Sc. Thesis, Brown University, 1990.

[Leonard and Durrant-Whyte, 1989] John J. Leonard and Hugh F. Durrant-Whyte. Active sensor control for mobile robotics. Technical Report OUEL-1756/89, Oxford University Robotics Research Group, 1989.

[Russell and Wefald, 1989] Stuart J. Russell and Eric H. Wefald. On optimal game-tree search using rational meta-reasoning. In *Proceedings IJCAI 11*, pages 334–340. IJCAI, 1989.

[Smith and Cheeseman, 1986] Randall Smith and Peter Cheeseman. On the representation and estimation of spatial uncertainty. *The International Journal of Robotics Research*, 5:56–68, 1986.

Extending the Partial Global Planning Framework for Cooperative Distributed Problem Solving Network Control *

Keith Decker and **Victor Lesser**
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01002

Abstract

Generalized Partial Global Planning is an attempt to extend Partial Global Planning (PGP) to other domains and provide a framework for new coordination algorithms. It is based on the observation that the relationships the PGP mechanism uses to measure increased network coordination can be defined for, and detected in, arbitrary cooperating systems. This paper describes several issues that will be important in extending the PGP mechanism to heterogeneous, dynamic, and "real-time" agents, and in developing new mechanisms such as negotiation. A scenario is described that illustrates these issues and which can be used as a basis for experimenting with solutions to them. The approach taken emphasizes the detection and classification of goal relationships between agents.

1 Introduction

The partial global planning (PGP) approach to distributed network control increased the coordination of agents in the network by avoiding redundant activities, shifting tasks to idle nodes, and providing predictive results that reduced overall problem solving time by estimating the duration of tasks to allow schedules to be built from interleaved plans [Durfee and Lesser, 1987b]. Generic Partial Global Planning tries to extend this approach by communicating more abstract information (goals, capabilities) and detecting the relationships that are needed by the partial global planning mechanisms. In the same way that contract nets [Davis and Smith, 1983] provide a domain-independent task allocation mechanism, we are proposing to build a generic network control system that will provide support for: modeling agents' capabilities, desires, and intentions; modeling the relationships between these; and building network controllers based on these models.

*This work was supported by DARPA contract N00014-89-J-1877, and partly by the Office of Naval Research under a University Research Initiative grant, number N00014-86-K-0764, NSF-CER contract DCR-8500332.

The global coherence problems we would like to address occur in many systems, such as the Pilot's Associate system [Smith and Broadwell, 1987], where situations occur that cause potentially complex and dynamically changing goal relationships to appear in goals that are spread over several agents. Each agent in the Pilot's Associate system has subgoals that other agents must fulfill, and receives subgoals from other agents that only it can fulfill.

For example, assume that we are in a tactical situation, so the tactical planner is in control (see Figure 1). It has two ordered subgoals: turn on the active sensors (request to situation assessment), and get a detailed route of the plane's movements during the tactical maneuver (request to the mission planner). Turning on active sensors causes a plane to become a transmitter, and thus become easily detected (most of the time the plane uses passive sensors). Since this is dangerous, the situation assessment agent will ask the pilot-vehicle interface (PVI) to ask for pilot confirmation of the use of active sensors. The pilot, upon seeing the request, asks the PVI to plot the escape route of the plane on the screen in case things go wrong. The PVI passes this goal to the mission planner.

Meanwhile, the tactical planner has asked the mission planner to produce the detailed route for the tactical maneuver. Which task does the mission planner act on first? From a local view, it may perhaps do the tactical planner request first because the tactical planner goals are a high priority. But from a global perspective, we see that unless the mission planner plans the escape route, which is needed by the pilot in order to authorize turning on the active sensors, which is needed for the tactical planner to do its job, the whole system goal of handling the tactical situation is in jeopardy. Hence the mission planner should do the escape route plan first.

To handle these interactions, we are developing a set of generic relationships among goals that fall into four general categories: domain relations, graph relations, temporal relations, and non-computational resource constraints. By communicating goals at different levels of abstraction we can reduce the amount of communication required between agents by communicating detail only when necessary.

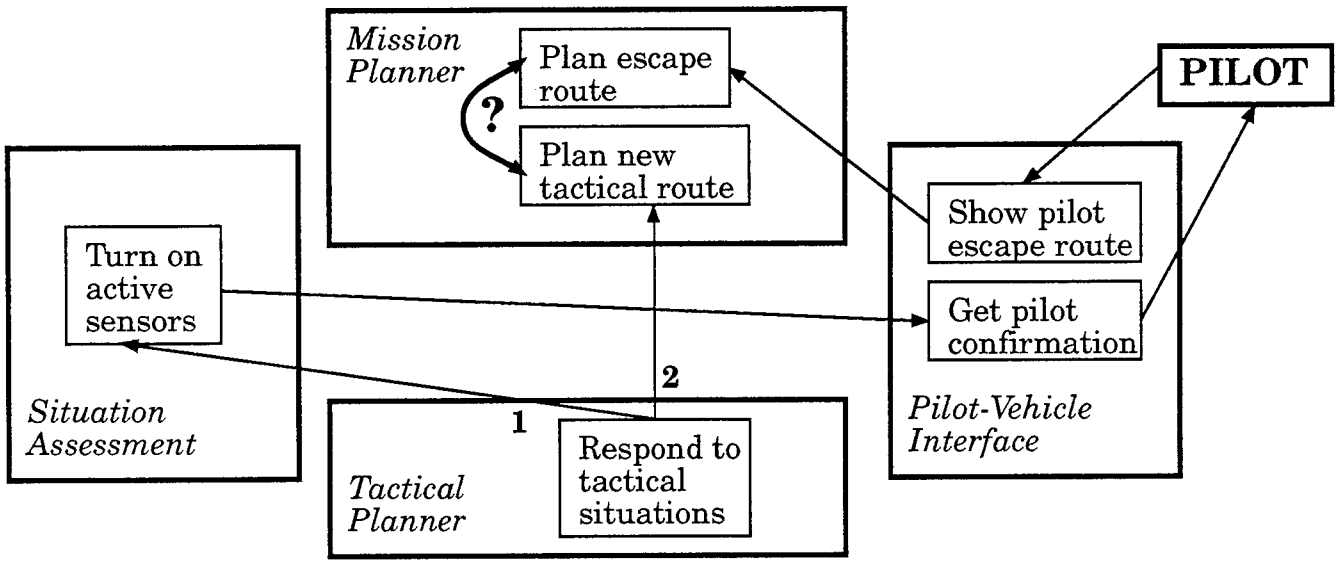


Figure 1: Dynamic Situations in Pilot's Associate

Section 2 briefly reviews the existing PGP mechanisms and why they work. However, we want not only to apply the PGP approach to other areas, but also to extend it. In developing a complete approach to distributed coordination, we found several other areas in which we would like to extend the PGP mechanisms. These issues include *heterogeneous agents* (which may have different problem solving criteria), *dynamic agents* (which have several different strategies available for problem solving and several different methods for accomplishing goals), *real-time agents* (which have hard goal deadlines), and *negotiating agents* (which are a consequence of the conflicts inherent in any and all of the other issues). Section 3 discusses these issues in greater detail, defining them and how they relate to the original PGP mechanisms. A scenario is presented in Section 4 that illustrates these issues in the Distributed Vehicle Monitoring Testbed (DVMT). Finally, an initial approach is outlined in Section 5 that describes how we intend to integrate the goal relationship mechanisms into the DVMT and how to use them to achieve a more flexible PGP mechanism.

2 Partial Global Planning

Partial global planning [Durfee and Lesser, 1987b, Durfee and Lesser, 1989] was developed as a distributed control technique to insure coherent network problem solving behavior. It is a flexible approach to coordination that does not assume any particular distribution of subproblems, expertise, or other resources, but instead lets nodes coordinate in response to the current situation. Each node can represent and reason about the actions and interactions of groups of nodes and how they affect local activities. These representations are called **partial global plans** (PGPs) because they specify how different *parts* of the network *plan* to achieve more *global* goals. Each node can maintain its own set of PGPs that it may use independently and asynchronously to coordinate its activities.

A PGP contains an objective, a plan-activity-map, a solution-construction-graph and a status:

- The **objective** contains information about *why* the PGP exists, including its eventual goal (the larger solution being formed) and its importance (a priority rating or reasons for pursuing it).
- The **plan-activity-map** represents *what* the nodes are doing, including the major plan steps the nodes are concurrently taking, their costs, and expected results.
- The **solution-construction-graph** contains information about *how* the nodes should interact, including specifications about what partial results to exchange and when to exchange them.
- The **status** contains bookkeeping information for the PGP, including pointers to relevant information received from other nodes and when that information was received.

A PGP is a general structure for representing coordinated activity in terms of goals, actions, interactions and relationships.

When in operation, a node's PGPlanner scans its current network model (a node's representation of the goals, actions and plans of other nodes in the system) to identify when several nodes are working on goals that are pieces of some larger network goal (partial global goal). By combining information from its own plans and those of other nodes, a PGPlanner builds PGPs to achieve the partial global goals. A PGPlanner forms a plan-activity-map from the separate plans by interleaving the plans' major steps using the predictions about when those

steps will take place. Thus, the plan-activity-map represents concurrent node activities. To improve coordination, a PGPlanner reorders the activities in the plan-activity-map using expectations or predictions about their costs, results, and utilities. Rather than examining all possible orderings, a PGPlanner uses a hill-climbing procedure to cheaply find a better (though not always optimal) ordering. From the reordered plan-activity-map, a PGPlanner modifies the local plans to pursue their major plan steps in a more coordinated fashion. A PGPlanner also builds a solution-construction-graph that represents the interactions between nodes. By examining the plan-activity-map, a PGPlanner identifies when and where partial results should be exchanged in order for the nodes to integrate them into a complete solution, and this information is represented in the solution-construction-graph.

The PGPlanner, as it was used for coordination in the distributed vehicle monitoring task, relied on the fact that the level of abstraction at which the node plans were communicated was a sequential sequence of intermediate goals (times and locations in which to extend a vehicle track). Each intermediate goal was an abstraction of the processing and integration work that each node planned for locally. These intermediate goals were ordered by the local node planners based on several criteria [Durfee and Lesser, 1988], but these relationships are not transmitted in the node plans. There was no representation of temporal relationships between intermediate goals. The PGPlanner reorders node activities by hill-climbing in the space of costs of the present ordering of activities. The cost of an ordering is computed from relationships like redundancy, reliability, predictiveness, and independence of the activities.

Why does partial global planning work well in the DVMT? It is because:

- It avoids redundant work among nodes by noticing interactions among the different local plans. Specifically, it notices when two node plans have identical intermediate goals, i.e., when they are working on the same time region. This occurs in the DVMT because in the interests of reliability nodes have overlapping sensors.
- It schedules the generation of partial results so that they are transmitted to other nodes and assist them at the correct time. To do this it uses the estimates of the times that activities will take and the inferred relation that if node $\mathcal{A}$ estimates that it will take less time than node $\mathcal{B}$ to complete an intermediate goal, and the goals are spatially near, that node $\mathcal{A}$ can provide predictive information to node $\mathcal{B}$.
- It allocates excess tasks from overloaded nodes to idle nodes. Node plans provide the information needed to determine if a node is overburdened or underutilized. A node is underutilized if it is either idle or participates in only low-rated PGPs. A node is overburdened if its estimated completion time of a subgoal of goal G is much later than the completion time of all the other subgoals of G [Durfee and Lesser, 1989].
- It assumes that a goal is more likely to be correct if it is compatible with goals at other nodes. In the DVMT task, a goal represented a processing task to ascertain whether a vehicle was moving in a region r at time t. This goal could, in fact, be wrong — based on noise or errorful sensor data that was the basis for the preliminary task analysis that generated the goal. Nodes choose local plans to work on based on the highly rated PGPs they have received. Thus, if the intermediate goals of a node become part of a PGP, then they are worked on before other intermediate goals in other local plans the node may have (even though the node may have rated those local plans higher in its local view).

To control how they exchange and reason about their possibly different PGPs, nodes rely on a meta-level organization that specifies the coordination roles of each node. If organized one way, the nodes might depend on a single coordinator to form and distribute PGPs for the network, while if organized differently, the nodes might individually form PGPs using whatever information they have locally. The partial global planning framework lets nodes converge on common PGPs in a stable environment (where plans do not change because of new data, failed actions, or unexpected effects of their actions). When network, data, and problem-solving characteristics change and communication channels have delay and limited capacity, nodes can locally respond to new situations, still cooperating but with potentially less effectiveness because they have somewhat inconsistent PGPs [Durfee and Lesser, 1987a]. The PGP framework does not, however, deal with conflicts in non-computational (physical) resources.

3 Issues in Extending the PGP Mechanisms

3.1 Heterogeneous Agents

How can the PGP mechanisms be extended to handle agents that have different local problem solving criteria? This can arise in several ways:

- Some agents in the system are humans with local (personal) decision criteria that cannot be adequately or fully modeled.
- Some agents in the system have different expertise, and hence different local decision criteria (cooperative design problems [Lander and Lesser, 1989], pilot's associate-style problems [Smith and Broadwell, 1987]). The PA scenario in Section 1 is a classic example of heterogeneous agents with shared global goals and differing local expertise.

The PGP mechanism assumes a shared local and global decision evaluation function (so that all agents, given the same information and enough time, will arrive at the same decisions). Conflict between agents

comes about because some agents lack data or have out-of-date data. Agents do not have to exchange or negotiate about decision criteria. While this well-documented assumption simplified the PGP mechanism, a homogeneous agent assumption (where the local decision criteria are shared) is not always appropriate. The PGP mechanism also assumes that the agents will pursue one goal at a time — the goals are ordered, and if an agent has excess capacity it can fill it with tasks from lower-rated goals. Planning for the simultaneous achievement of multiple goals is not supported.

The modularity of the PGP mechanism (which separates the local agent's incremental planner [Durfee and Lesser, 1988] from the PGPlanner) comes close to permitting heterogeneous local decision criteria. The only problem arises when the PGPlanner reorders the node plan for another agent. The PGP plan evaluation function that was used to develop a global schedule contains terms to avoid upsetting the order of another agent's plan (*independence* measured the distance of the current ordering from the original node plan ordering, *locally-predicted* measured the distance of the current ordering from regular time order). In some domains a portion of this ordering may be fixed. We have suggested marking temporal goal relationships as *hard*, *negotiable*, and *soft* (see Section 5.2). This allows the plan evaluator to rule out certain impossible orderings (hard constraints), and to avoid those that may cause replanning at the target node (negotiable constraints).

3.2 Dynamic Agents

How can the mechanisms be extended to handle agents that have a great deal of latitude in the methods that they use to solve problems? Each method may have a different effect on the characteristics of the solution, such as completion time or certainty. These agents can appear in human systems and systems where agents use approximate processing techniques [Decker *et al.*, 1990b]. In the PA scenario in Section 1, the mission planner might solve its dilemma by using different algorithms to respond to each plan request. A fast but inaccurate algorithm may suffice to give the pilot an idea of a corridor of escape, while a more complex and precise algorithm can be given the bulk of the computational resources with which to plan the near-term tactical maneuver.

Because only one method existed for accomplishing a goal (and no set of different criteria existed for determining what would be considered an acceptable solution), the PGP mechanism could equivalently exchange goals and the plans to accomplish those goals, at a single level of detail. The node plans that were exchanged indicated the goal of an agent to produce a track with certain characteristics (classes, sensed times, and regions) and a plan consisting of the ordering of the sensed times at which the agent would work (called *i-goals*), expected i-goal durations, and a mapping of the i-goal start and end times with respect to node problem solving time.

Two extensions need to be made. First, communicating goals at a single level of detail is inappropriate in more complex domains; certainly the detection of the interactions of two goals ("partial global goals") will not always be simple [Robinson and Fickas, 1990]. Secondly, many different methods may exist for accomplishing a goal, each with its own effects on duration, precision, and other goal characteristics. This makes the existing PGP node plan structure change rapidly when problem solving methods are changing dynamically (as an agent reacts to the problem being solved). The node plan structure can be modified to hold ranges as well as a best current estimate for a value, but it is also likely that agents will have to reason and perhaps negotiate about predictability versus reliability issues as well [Durfee and Lesser, 1987a]. The node plan structure could also be expanded with contingency plans for "routine expectation failures" [Dean, 1987] to allow for predictability in the face of a changing environment.

3.3 Real-time Agents

What happens when time becomes an integral part of local and shared goals? Dynamic agents will be able to modify both task durations (perhaps trading them off for other goal characteristics) *and* the goal deadlines themselves. In the PA scenario in Section 1, the mission planner's dilemma arises from the fact that it is under real-time constraints — if there were no impending deadlines for the pilot and tactical planner, the mission planner would have little reason to prefer one allocation of its computational resources over another.

While the PGP mechanism estimated the times for tasks or goals to be completed in order to spot idle processing resources, it did not handle deadlines. I-goals had expected durations; node-plans anchored (mapped) the completion of the various i-goals to a plan activity map. Experiments were conducted with the local incremental planner that did indicate the ability to plan to meet deadlines in a single agent [Lesser *et al.*, 1988, Decker *et al.*, 1990b].

In extending the architecture to so-called "real-time" problem-solving, agents may have goals with hard deadlines, which add constraints to the construction of a plan activity map. Furthermore, the addition of hard deadlines or other domain constraints changes the nature of the interaction between a node's local problem solving mechanism and the PGP mechanism — some of the local ordering will remain local preference but some may be due to hard constraints, as discussed in Section 3.1 above. From the classical perspective, real-time network control also means scheduling both periodic and non-periodic tasks to deadlines; the original PGP mechanism did not deal with periodic tasks. The existing hill-climbing algorithm for scheduling may no longer be appropriate.

Often in real time situations planning is *reactive*, where the current situation mostly controls an agent's actions (where the "current situation" may include

both local and global information), rather than *reflective*, where a sequence of actions is planned out in some detail before execution. This is because the agent must respond quickly, but more importantly, the agent may be too uncertain of the outcomes of its actions and of the changing world state to plan too far into the future. However, an intelligent agent will make use of periodic tasks, which occur in a predictable fashion, and known non-periodic tasks, to build a opportunistic planning framework that can keep an agent from painting itself into a corner with purely reactive planning techniques, or from exhaustively planning uncertain future details with reflective planning techniques.

Many new mechanisms are being put into place in the DVMT to allow the control of real-time problem solving (with hard deadlines); many of these mechanisms should integrate easily with the ones described here. For example, the original PGP mechanism had to have its own time estimation routines. We envision that the real-time mechanisms being developed for a single agent will be able to provide such services to the new coordination mechanisms we are also developing.

3.4 Negotiating Agents

A direct consequence of heterogeneous, dynamic, and real-time agents is the need for negotiation to solve conflicts. Even with a known global decision evaluation function, conflicting decisions of equal global value may have very different local value to the agents. Often the character of an early partial solution will have an impact on what style of coordination is needed. For example, if early partial results show poor data and low beliefs, the coordination mechanism may want to encourage redundant derivations of results in areas shared by more than one agent, or the parallel derivation of a result by two agents using different algorithms. The PA scenario in Section 1 probably occurs in too short a time-frame to allow negotiation between the agents, but other PA scenarios might profitably use negotiation techniques[1].

The PGP mechanism uses a shared global plan evaluation function that is parameterized. One extension is to allow the parameters (such as *redundancy* and *reliability*) to vary during problem solving. A negotiation facility could be developed to allow agents to usefully alter the global (or perhaps only semi-local — we are interested in agents that may develop only a partial view of what other agents are working on) decision criteria. Where the PGP mechanisms exchanged all local information, our extensions would allow for a multi-stage process [Kuwabara and Lesser, 1989] where agents would communicate only the information believed relevant to the issue at hand. Agents could ask for more contextual information when it is needed to resolve a conflict between agents. Agents would not automatically acquire information from other agents performing non-related problem solving activities.

In order to examine all of the above issues more closely, a scenario is described below which exhibits the issues mentioned above.

[1]For example, a sensor may overheat and be shut down by the system status module, even though it is a projected resource requirement for some tactical situation. The tactical planner and system status may negotiate over the amount of time that the damaged sensor can be used if the situation arises.

4 Scenario

The scenario is based on the basic task of the DVMT: to track vehicles moving through an area via their acoustic signatures. To describe the scenario we will describe the tasks involved, the agents who will carry out those tasks, the environment within which they are acting, and the desired interactions.

The "vehicles" in the scenario are various aquatic creatures and waterfowl. One of the tasks involves protecting fish from fish-eating ducks. This includes detecting both fish and ducks and notifying the fish of potential duck attacks in time for the fish to escape. Another task involves simply tracking any pigeons in the area and displaying the tracks accurate to within certain spatial and temporal guidelines. These tasks are specified by "system goals" (see Section 5.1).

The scenario demonstrates the issues described earlier in Section 3. Fish-protectors and pigeon-trackers are heterogeneous agents that have different utilities for the same data (see Section 5.1.1). The hard time deadlines will allow us to experiment with real-time performance and use approximate processing methods to give each dynamic agent a choice of different problem solving methods to choose from. The environmental data for the scenario will give the agents just cause for altering their plans during problem solving. Finally, the distribution and timing of the scenario provides several opportunities for the agents to negotiate. These issues are revisited in Section 4.4 after the scenario itself.

4.1 The Agents

There are 4 standard DVMT tracking agents with identical domain knowledge. Agents 1 and 3 are fish-protectors, and agents 2 and 4 are pigeon-trackers. Agent 3 has a faulty sensor, which will produce a large amount of noisy data. The four agents are shown pictorially in Figure 2. We will omit the definition of the agent's sensors and their precise coordinates. Also note that we can expand the scenario to more agents by tiling pigeon- and fish-agents in a natural way.

4.2 The Environment

We will describe the environment by describing the patterns and vehicle tracks that are given to the environment simulator. These tracks can be seen visually in Figure 3.

At times 1–10 a fish F_1 travels through the area seen only by agent 1. The fish is initially meandering.

At times 1–10 a pigeon P_1 travels through the area of agent 2. It begins in the non-shared area and

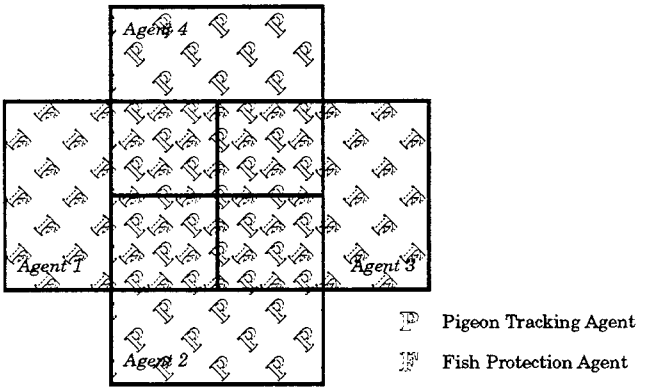


Figure 2: Four DVMT Agents: Two fish-protectors and two pigeon-trackers

crosses into the area shared by agents 2 and 3 at time 6. The pigeon is in a meandering pattern.

At times 1–10 a duck D_1 travels through the area of agent 3. It begins in the non shared area and crosses into the area shared by agents 3 and 4 at time 3, and into the area shared by agents 1 and 4 at time 9. The duck is meandering.

At times 4–10 a duck D_2 travels through the areas of agents 4 and 1. It begins in agent 4's area only from times 4–5, then in the shared area of agent 4 and 1 at time 6 and 7, and then into agent 1's non-shared area. Duck D_2 is in an attack pattern with fish F_1 as the potential victim.

At times 6–10 a pigeon P_2 travels through the area of agent 1 and 4. It remains in the area covered by agent 1.

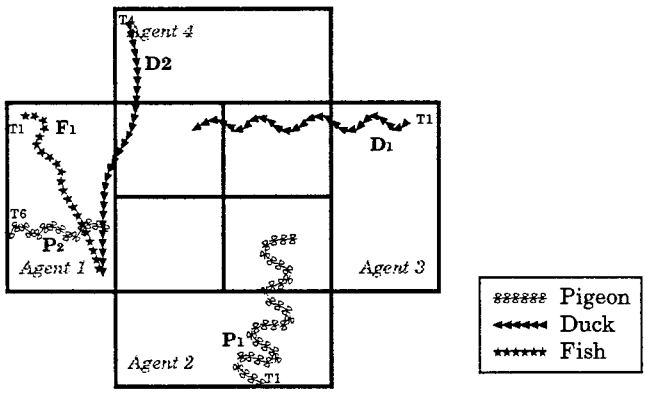


Figure 3: The CDPS scenario environment

The actual grammar specifying the characteristics of these vehicles has been omitted. Several other things have been left deliberately unspecified, most notably the mapping from "real-world time" to processor time (so that we can experiment with how tightly pressed for time the agents are).

4.3 The Intended Interactions

There are four major interactions that are brought about by this environment. These interactions also show up clearly in the goal relationship example in Figure 4, page 10.

1. Agent 4 can help agent 3 disambiguate the data from its faulty sensor by tracking the duck D_1.
2. Agent 4 can notify agent 1 about the impending duck D_2. It might provide other predictive data, for example, it might try to track the duck precisely even though it is a pigeon tracking agent. This can be accomplished through negotiation between agents 1, 3, and 4.
3. Agent 2 can help agent 3 by taking care of the pigeon data in the shared area (thus reducing the load on agent 3 and its faulty sensor).
4. Agent 1 can notify agents 4 and 2 about the impending pigeon, providing predictive information (but note that agent 1 will be under severe time pressure).

4.4 Revisiting the important issues

The scenario above expresses the issues that were listed earlier:

Heterogeneous Agents: Having two different system goals cause the agents' local views of "what is important to do next" to change. Given the same data, Agent 1 and Agent 4 would treat it differently, in isolation, because they have different local decision functions (because Agent 1 is a fish-protector and Agent 4 is a pigeon-tracker). If it were not for the communication of the goals of Agent 1 to track ducks, Agent 4 might pay little attention to the duck it detects. The attention and effort that Agent 4 puts into the duck must come from some pre-specified or dynamically constructed global decision function that describes how Agent 4 should cooperate with Agent 1. Hence the scenario forces us to deal with the issues of different local views, of exchanging views, and of dynamically constructing global views. See also *Negotiating Agents* below.

Dynamic Agents: The agents have a set of adaptable methods for reaching their goals, each of which have different characteristics. Agent 3 will be using methods very different from those used by agent 1, even though both agents have the same system goal. An agent such as agent 4 might change the processing method on some data (say duck D_1) when new data (duck D_2) appears. The overloaded Agent 1 is also likely to switch strategies as it becomes overloaded. Hence the scenario allows us the opportunity to use agents with various problem solving methods and agents that change strategies during problem solving (with the commensurate difficulties in coordination).

Negotiating Agents: Agents 1 and 3, for example, may have different initial ideas of what agent 4 should be processing (given agent 4 has nothing to do on its own), and certainly different from that of agent 4 itself. Agent 4 must develop some

partially global view in relation to Agents 1 and 3 and all of their local goals. Agents 2 and 3 also have data in their overlapping areas. The scenario presents the opportunity to dynamically modify the global view of coordination, changing parameters to fit the situations that develop between Agents 4 and 3, then 4, 1 and 3, and finally 2 and 3. The scenario also presents conflicts between local views and the developed global views — see *Real-time*, below.

Real-time: Both of the system goals require solutions by a deadline. Agent 1 must forego helping agent 4 when pigeon P_2 appears — it is too busy. If a pigeon pops up in agent 2's sensed region, it may not be able to keep its commitments to agents 1 and 3. Thus the scenario provides goals with deadlines and conflicts between local and global views.

5 Approach

The approach we will take is a refinement and extension of that used in the PGP mechanism. The basic mechanism remains the exchange of information that allows each agent to independently affirm (in the case of anticipated domain relationships or "settled questions" [Gasser *et al.*, 1989]) or discover (in the case of unanticipated "open system" interactions [Hewitt, 1986]) its relationship to other agents in the system.

Rather than exchange node plans, we exchange agent goals of various types and levels of detail. By detecting the relationships between its own goals and the goals of other agents it may interact with, an agent may locally schedule actions while taking into account non-local goals. When domain problem solving requires multi-agent interactions, node plans may still be exchanged, as well as agent *capabilities*, which describe an agent's potential long-term goals. By using hierarchical goal structures and not always exchanging plans we can reduce the amount of communication required and focus what communication does take place to reduce the number of global views developed under the PGP mechanism — we can produce (partially global) schedules rather than partial (global schedules).

5.1 Goals

In extending the PGP mechanism, a major difficulty we encounter is in how goals should be specified for the agents. Most AI programs represent goals as either satisfiable logical formulae or *ad hoc* symbols. How can agents understand each others' goals, and to what extent do they need to?

We require the ability to recognize goal relationships, and the ability to recognize to what degree a goal has been satisfied. These two abilities are sufficient for "scheduling" coordination (reordering tasks locally), but not for multi-agent domain problem solving (where we may exchange or share tasks between nodes). For the latter, goals (and plans) must be recognizable (able to be acted upon) by the underlying agent problem solving structure, or the coordination mechanism must be able to translate them as such.

In the DVMT the agents do have a shared language for domain goals, and a language for control goals is under development. The new coordination mechanism, in order to remain aloof from the specifics of the DVMT, should not take too much advantage of the shared language. Rather, it should rely on the detection of goal relationships and degree of satisfiability directly, where these will be easy to implement because of the existence of a shared goal language. The highest goal specification for an agent is called a *system goal*.

5.1.1 System Goals

A *system goal* is a high-level goal describing the desired solution characteristics for an agents' problem solution. It includes:

Completeness: What parts of a full solution are the most important? To what degree does a partial solution fulfill a goal? In the DVMT, completeness specifies the level at which a hypothesis is a solution, its length, and its event classes.

Precision: How specific must be the data contained in the solution? DVMT hypotheses have a well-defined precision measure.

Certainty: How certain must an agent be that the hypothesis it is putting forth is a solution? Since DVMT belief uses a 4-tuple, this is a bit more complex than it sounds.

Deadline: By what time must the solution be produced?

These desired solution characteristics must be considered along with their interactions — to allow agents to trade off characteristics in a controllable manner. For example, is a solution with twice the precision and half the certainty as good as one half as precise and twice as certain? A simple way to specify this is to use an evaluation function to rate the solution, but this may not allow explicit reasoning about tradeoffs between solution characteristics.

5.2 Goal Relationships

Four classes of goal relationships have been identified that will be useful; a small subset of these relationships have been used to improve the scheduling of tasks in a single agent blackboard system where tasks can be executed in parallel [Decker *et al.*, 1990a]:

Domain Relations: This set of relations is generic in that they apply to multiple domains, and domain dependent in the sense that they can be evaluated only with respect to a particular domain — inhibits, cancels, constrains, predicts, causes, enables, and supergoal/subgoal (from which many useful graph relations can be computed). These relations provide *task ordering* constraints, represented by temporal relations on the goals (see below).

Graph Relations: Some generic goal relations can be derived from the supergoal/subgoal graphical structure of goals and subgoals, for example, overlaps, necessary, sufficient, extends, subsumes, competes. The *competes* relation is used to produce *task invalidation* constraints.

Temporal Relations: These depend on the timing of goals — their start and finish times, estimates of these, and real and estimated durations. From Allen [Allen, 1984], these include before, equal, meets, overlaps, during, starts, finishes, and their inverses.

Non-computational Resource Constraints: A final type of relation is the use of physical, non-computational resources. This is the major relation in some domains, such as factory scheduling and office automation [Sadeh and Fox, 1989, Martial, 1989].

5.2.1 Domain Dependent Relations

This set of relations are *generic*, in that they apply to multiple domains, but *domain dependent* in that they can be evaluated only with respect to a particular domain.

supergoal/subgoal: Goal B is a *subgoal* of goal A if B is required for some method of achieving A.

inhibits: Goal A *inhibits* goal B if when goal A is accomplished goal B cannot be accomplished. This definition ignores the time component. A might either *permanently* or *temporarily* inhibit B.

cancels: Goal A *cancels* B if when goal A is achieved, goal B is *no longer achieved*. Notice that this is subtly different from A *inhibiting* B; Inhibition implies that B cannot be accomplished, canceling implies B is no longer achieved, but not that it cannot be achieved.

If B *cancels* B then B is a *recurring* goal, one that undoes itself. Of course any set of goals may be placed in a recurring relationship if they cancel each other in sequence.

constrains: Goal A *constrains* goal B if the two goals are related somehow at the domain level by the need to exchange information from A to B in order to solve B. This may or may not be a time constraint. This is necessary information. If A constrains B then A [>,m,o,oi,d,di] B (A is before, meets, overlaps, is overlapped by, is during, or surrounds B [Allen, 1984]). Note that the "constraint relation" does not tell you what the constraint is, but that one exists. It means that the two subgoals are interacting subproblems.

predicts: Goal A *predicts* goal B if information about the solution of A is useful for the solution of B but not necessary. This could be information used to reduce uncertainty, or to guide search.

causes: Goal A *causes* B if the completion of goal A physically entails the occurrence/completion of B.

enables: Goal A *enables* B if the completion of goal A must occur before goal B can be satisfied. Obviously this implies a strong temporal constraint.

5.2.2 Graph Relations

The generic graph relations can be derived simply from the subgoal/supergoal structure of the goal hierarchy, without using any internal information about a goal or any domain-dependent information. We can view the goal hierarchy as an acyclic AND/OR graph. Two nodes are *equivalent* if they have *equivalent* subgoals, or consist of identical primitive actions. We may also want to assume algorithmically that equivalent nodes are only represented once in the goal tree (which will then be a goal acyclic graph).

Just because a goal relation can be derived from a graph does not mean that it is trivial; when adding a new goal node to the graph a great deal of computation may need to go on to see how that goal really relates to the others. Part of this computation comes in recognizing repeated goals (that occur multiple times in the graph), so that you cannot blindly expand a goal into a set of subgoals.

overlap: Goal A *overlaps* B if there exists a goal G such that A is a supergoal of G and B is a supergoal of G.

necessary: Goal B is *necessary* for goal A if goal A cannot be accomplished without accomplishing goal B (B is an AND subgoal in an AND/OR tree).

sufficient: Goal B is *sufficient* for goal A if accomplishing goal B accomplishes goal A (goal B is either an OR subgoal or A, or is sufficient for an OR subgoal of A).

extends: Goal A *extends* B if there exists a *supergoal* G such that A and B are both *necessary* for G.

subsume: Goal A *subsumes* B if B is a *subgoal* of A or if B is *obviated* by A. B is *obviated* by A if A is *sufficient* for the *parent* of B.

competing: A *competes* with B if A and B are *n-competing* for some n. Inductively, A and B are *0-competing* if A and B are in different disjuncts for each of their parents. A and B are *n-competing* if every pair of parents of A and B are *i-competing* for $0 \leq i \leq n-1$. This captures the intuitive idea that two goals *compete* if there is no possible way that both goals *must* be fulfilled. This does not mean that they cannot both be fulfilled, or that they interfere with one another in any way. The concept is graph-theoretic, not domain dependent.

5.2.3 Time Based Relations

A third set of relations are not strictly domain dependent, but do depend on the timing of goals. There are several possible features of goals that are applicable to timing: actual start and finish times, estimated start and finish, deadlines, and (estimated, actual) lengths. Any one of these numbers alone will allow at least some limitation of the possible temporal relations between two goals. Furthermore, temporal

relations may be preferences (soft constraints) as well as absolute relations. We envision three levels of temporal constraints:

Hard: Hard constraints cannot be violated. Inability to satisfy hard constraints means that the problem is overconstrained. Overconstrained problems may result in negotiation, for example, in real-time problems where a node may have an alternate solution path that takes less time but is also less certain (approximate processing).

Negotiable: Negotiable constraints are preferences that a local node does not want violated needlessly. Inability to satisfy a negotiable constraint means that the constraining node must be part of the decision to modify the constraint.

Soft: Soft constraints are preferences that a node has but do not require negotiation in order to violate. For example, local orderings of intermediate goals in the DVMT are soft constraints.

Temporal relations have been studied before, and very precise definitions have been put forward by James Allen [Allen, 1984]. We give the graphical suggestion of a definition as presented by Allen in lieu of the formal definitions to be found in Allen's papers. Remember that each relation has an inverse and that in the presence of limited information a set of these relations may hold between any two timed goals:

before: $\frac{xxxx56789}{12345yyyy}$

temporally equal: $\frac{123xxxx89}{123yyyy89}$

meets: $\frac{12xxxx7890}{123456yyyy}$

overlaps: $\frac{123xxxx890}{12345yyyy0}$

during: $\frac{123xxxx890}{1234yy7890}$

starts: $\frac{123xxx789}{123yyyyyy}$

finishes: $\frac{12345xxx90}{12yyyyyy90}$

5.2.4 Using Goals

The new DVMT control architecture [Decker *et al.*, 1990b] uses the specification of the system goal(s) to choose a strategy for satisfying the goal. The strategy posts a set of goals whose specification in turn allows the choice of an appropriate substrategy, etc. At some point a goal can trigger the creation of a *focus* (the lowest leaf in an expanded control plan) which specifies a set of low-level control parameters and rating heuristics. These focus parameters and heuristics are associated with a *channel* in the low-level domain problem solving system, existing concurrently with other active foci and their associated channels.

It is this hierarchical set of control goals, starting with the system goal, that are communicated in our approach. Goal relationships detected between local goals and the goals received from other agents allow us to coordinate scheduling. Figure 4 shows all four agents in the scenario, and all of their control goals. Note that not all of these goals are active simultaneously, in particular, the *identify* goal for a hypothetical vehicle always comes strictly *before* the *tracking* goal for the newly identified vehicle. The example in Section 5.4 shows how Figure 4 is constructed for Agents 1 and 4.

5.3 Coordination Rules

The last item that we need to specify in our approach is the PGP algorithm itself. Initially both a PGP-like scheduling algorithm and the organizational information needed to use it will be encoded as a special set of control knowledge sources that handle when to send and receive goals and hypotheses and how to modify the local schedule given this information.

Some of these rules are generic (like "don't do redundant work") but as in the original PGP mechanism, they must be operationalized for the particular domain. Each rule should also be considered a "temporarily settled question," subject to being reopened in the domain setting [Gasser *et al.*, 1989].

The PGP algorithm orders intermediate goals according to their cost as computed from the cross-product of a vector of computable factors (such as redundancy, reliability, etc.) and global cooperation parameters that give a weight to the corresponding term in the calculation. These factors were the following:

Redundancy: The number of nodes that can perform this goal. The GPGP scheduler can use the equality and subgoal relationships to examine redundancy.

Reliability: The number of nodes that cannot perform this goal. This is the inverse of redundancy.

Duration: The duration of the goal. This measure can be used by the GPGP scheduler as well.

Predictiveness: Any goal with a duration of x can provide predictive information for a goal of duration y if $x < y$. The predictiveness measure was then the minimum activity distance (minimum number of sensed times) between the two goals. GPGP considers the predictiveness relation to be a domain-dependent relation.

Locally-predicted: The minimum activity distance between a goal and any goal to be executed before it. The effect of this measure was to keep a node from jumping around between times in constructing a track; extending an existing partial solution was preferred. The GPGP scheduler can use the hierarchical construction of the goal hierarchy to avoid reordering the steps in constructing a track.

Independence: how many goals occur before a given goal in the initial local node plan. This measure was intended to keep the PGPlanner from straying too far from the original local node plan ordering. The independence measure for each goal is constant, because it depends only on the initial local node plan, not on the position of the

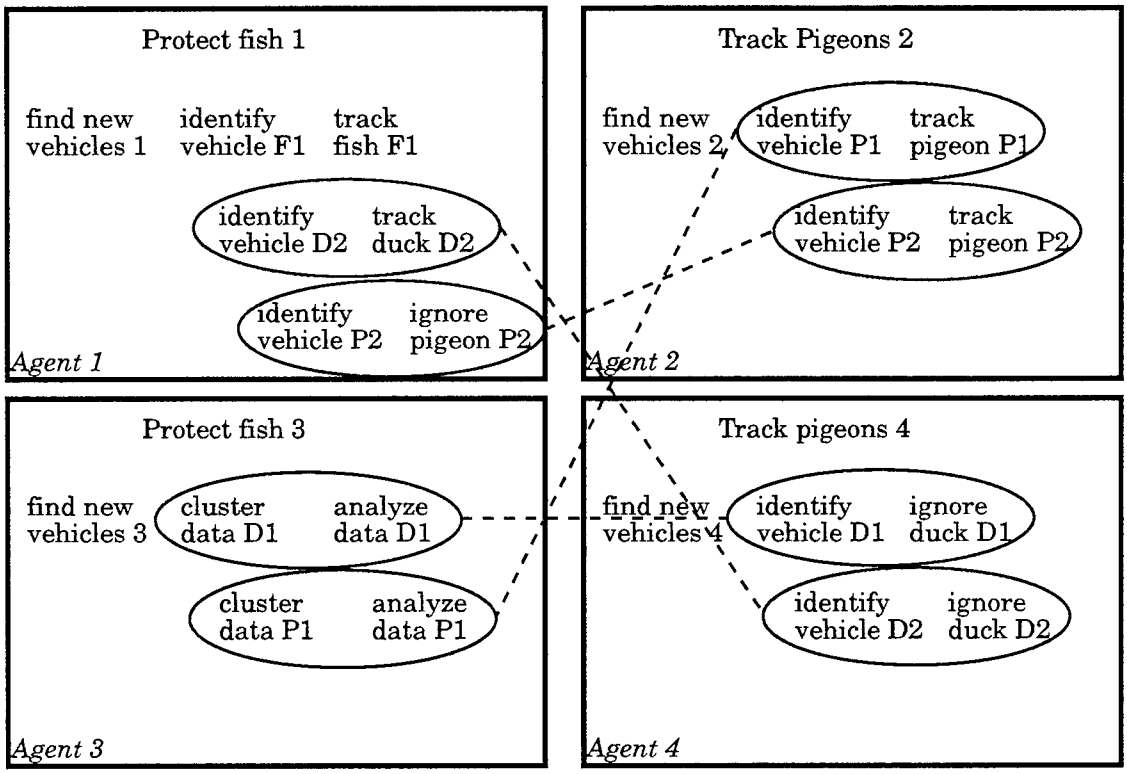


Figure 4: Goal *overlaps* relationships (dotted lines) between agents in the CDPS scenario

goal in the re-ordered plan. Goals later in the initial ordering have higher independence measures. Because the PGP algorithm used a swapping procedure to create a new ordering from the old, the higher independence measure made later goals harder to swap. The GPGP scheduler receives local ordering preferences, and so it knows what local orderings were necessary, which may be negotiated, and which are only preferences. Thus this relationship is not directly needed.

Diversity: The diversity value of a goal is 0 if it does not derive redundant information, or if all goals following it derive redundant information. Otherwise, the diversity of a goal is measured by the minimum activity distance between the goal and the later non-redundant goals. The effect is to plan to do non-redundant work before redundant work, and a GPGP scheduler can detect redundancy through goal relationships.

General coordination rules:

1. Broadcast system goals (establish long term relationships among nodes).
2. Broadcast capabilities (establish basic node capabilities).
3. Send any goal that *overlaps* another agent's goal to that agent (notify nodes of potential interactions based on perceived roles).
4. Send hyps that satisfy (partially satisfy?) another agent's goals to that agent.

PGP-like rules:

1. Avoid useless redundancy with other nodes.
2. Build reliable solutions. It is OK to be redundant in order to increase certainty in a solution. Two nodes with the same system goal might both analyze the data in an overlapping area, whereas agents with different system goals might let one or the other handle it. The PGP mechanism never really did this.
3. Minimize durations. In the case where more than one agent has an equivalent goal, then let the one with the shorter predicted duration do it.
4. Provide predictive information to other nodes.
5. Perform regular problem solving (follow the local control plan).

The following PGP relations are handled by the local control plan: keep local order if possible (independence), extend tracks in time order (locally-predicted), avoid intra-node redundancy (diversity). These deal with reordering local plans — but we are not constructing plans, but merely exchanging goals.

Finally, there is still the question of how to integrate the goals of other agents into an agent's local problem solving; when to accept goals that will cause work at the local node, and when to split goals to allow them to be shared between agents. We are pursuing these questions using the PGP mechanisms as an initial starting point.

5.4 Example 1

These coordination rules will only go part way toward our goals, but will serve for an illustrative example. This brief example shows Agent 4 providing predictive information to Agent 1 (a track hypothesis for duck D_2).

The four agents have the same domain knowledge and the same meta-control knowledge. This meta-control knowledge consists of two basic strategies: a *goal-directed strategy* and a *clustering strategy*. The goal-directed strategy consists of substrategies to find new vehicles, identify vehicles, track a vehicle, and 'ignore' a vehicle. The clustering strategy consists of substrategies for finding new areas, clustering data, tracking, and analyzing the tracks. All of the substrategies have one or more foci that implement them, and these foci may differ by the amount of time they take and the precision or certainty of their result. For example, *find-new-vehicles* can use a threshold on signal strength to ignore weakly sensed vehicles, *identify-vehicle* can do more or less work in making an identification, and so on.

First all agents broadcast their system goals and capabilities to the other agents. Each agent will begin in the default (goal-directed) strategy to satisfy its system goal, and will post a *find-new-vehicles* goal. Since *find-new-vehicles* can potentially output any type of vehicle, this goal overlaps the system goals of the other agents (indicating the potential for coordinated scheduling) and is broadcast to them. The agents also recognize that the *find-new-vehicles* goals overlap in the shared sensed areas. If this goal were made up of smaller ones, then the smaller ones would be exchanged, but in this case this goal is the lowest level. Actually reasoning about and splitting goals that overlap like this is an area for future work.

For this example we will just look at the interaction of Agents 1 and 4.

Agent 1 soon detects a vehicle (F_1) and starts a new goal to identify it. While Agent 1 has not identified the vehicle yet, it does have enough information to predict that the vehicle is not a pigeon (the fish signals do not overlap very much with bird sounds) and so it does not transmit the identification goal to Agent 4 (it could change its mind about this later). The *find-new-vehicles* goal is not retransmitted and it remains active.

When Agent 1 identifies the fish F_1, it begins tracking it. The tracking goal is not transmitted.

Later (time 4), while Agent 1 is tracking the fish, Agent 4 detects a new vehicle (D_2). When Agent 4 starts up a new goal to identify this vehicle, it sends the goal to Agent 1, for the goal is potentially a duck, which Agent 1 is interested in. At this point Agent 1 can only believe that Agent 4 has detected something that may be a duck, and it knows where it is and how strongly Agent 4 believes that it is a duck.

While Agent 4 is identifying the duck, the duck crosses into the shared sensor area of agents 1 and 4. Agent 1 detects the data as a new vehicle, and creates a goal to identify it. This goal is transmitted to Agent 4, since it could be a pigeon. Agent 4 realizes that Agent 1's *identification* goal overlaps its own as it specifies an area coincident with the identification track being developed at Agent 4. Thus redundant work is occurring. In this case having both agents work on the data should increase certainty in their conclusions, because neither agent is using a provably dominated algorithm (in fact, the algorithms are the same, the data arise from independent sensor readings). Thus the identification goals become equivalent, and this recognition is communicated to Agent 1. Agent 4 will complete its identification before Agent 1, because it started earlier.

When Agent 4 completes its identification, the resulting hypothesis (that there is a duck on a track extending from times 4 – 7) is transmitted to Agent 1. This hypothesis satisfies Agent 1's *identification* goal.

6 Future Work

We are currently building the initial implementation of the approach above, including the scenario. When this is complete we can provide details of our implementation, and we will have a testbed with which to experiment with other coordination algorithms, with negotiation algorithms, and with the interaction of our coordination mechanisms and the new hard real-time problem solving mechanisms we are also building.

References

[Allen, 1984] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[Davis and Smith, 1983] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20(1):63–109, January 1983.

[Dean, 1987] Thomas Dean. Planning, execution, and control. In *Proceedings of the DARPA Knowledge-based Planning Workshop*, December 1987.

[Decker *et al.*, 1990a] Keith S. Decker, Alan J. Garvey, Marty A. Humphrey, and Victor R. Lesser. Effects of parallelism on blackboard system scheduling. In *Proceedings of the Fourth Annual AAAI Workshop on Blackboard Systems*, Boston, August 1990.

[Decker *et al.*, 1990b] Keith S. Decker, Victor R. Lesser, and Robert C. Whitehair. Extending a blackboard architecture for approximate processing. *The Journal of Real-Time Systems*, 2(1/2):47–79, 1990. Also COINS TR-89-115.

[Durfee and Lesser, 1987a] Edmund H. Durfee and Victor R. Lesser. Planning coordinated actions in dynamic domains. In *Proceedings of the DARPA Knowledge-Based Planning Workshop*, pages 18.1–18.10, December 1987. Also COINS-TR-87-130.

[Durfee and Lesser, 1987b] Edmund H. Durfee and Victor R. Lesser. Using partial global plans to coordinate distributed problem solvers. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, August 1987.

[Durfee and Lesser, 1988] Edmund H. Durfee and Victor R. Lesser. Incremental planning to control a time-constrained, blackboard-based problem solver. *IEEE Transactions on Aerospace and Electronic Systems*, 24(5), September 1988.

[Durfee and Lesser, 1989] Edmund H. Durfee and Victor R. Lesser. Negotiating task decomposition and allocation using partial global planning. In M. N. Huhns and L. Gasser, editors, *Distributed Artificial Intelligence, Vol. II*. Pitman Publishing Ltd., 1989.

[Gasser *et al.*, 1989] Les Gasser, N. F. Rouquette, R. W. Hill, and J. Lieb. Representing and using organizational knowledge in distributed AI systems. In M. N. Huhns and L. Gasser, editors, *Distributed Artificial Intelligence, Vol. II*. Pitman Publishing Ltd., 1989.

[Hewitt, 1986] Carl Hewitt. Offices are open systems. *ACM Transactions on Office Information Systems*, 4(3):271–287, July 1986.

[Kuwabara and Lesser, 1989] K. Kuwabara and V. R. Lesser. Extended protocol for multi-stage negotiation. In *Proceedings of the Ninth Workshop on Distributed AI*, September 1989.

[Lander and Lesser, 1989] Susan Lander and Victor R. Lesser. A framework for the integration of cooperative knowledge-based systems. In *Proceedings of the 4th IEEE International Symposium on Intelligent Control*, pages 472–477, September 1989.

[Lesser *et al.*, 1988] Victor R. Lesser, Jasmina Pavlin, and Edmund Durfee. Approximate processing in real-time problem solving. *AI Magazine*, 9(1):49–61, Spring 1988.

[Martial, 1989] Frank V. Martial. Multiagent plan relationships. In *Proceedings of the Ninth Workshop on Distributed AI*, September 1989.

[Robinson and Fickas, 1990] William N. Robinson and Stephen Fickas. Negotiation freedoms for requirements engineering. Technical Report CIS-TR-90-04, Department of Computer and Information Science, University of Oregon, April 1990.

[Sadeh and Fox, 1989] N. Sadeh and M. S. Fox. Preference propagation in temporal/capacity constraint graphs. Technical report CMU-RI-TR-89-2, Robotics Institute, Carnegie Mellon University, January 1989.

[Smith and Broadwell, 1987] David Smith and Martin Broadwell. Plan coordination in support of expert systems integration. In *Proceedings of the DARPA Knowledge-Based Planning Workshop*, pages 12.1–12.6, December 1987.

Integrated Agent Architectures: Benchmark Tasks and Evaluation Metrics *

Mark E. Drummond
Sterling Federal Systems
NASA Ames Research Center
MS: 244-17, Moffett Field, CA 94035

Leslie Pack Kaelbling
Teleos Research
576 Middlefield Road
Palo Alto, CA 94301

1 Introduction

An *integrated agent architecture* is a theory or paradigm by which one may design and program intelligent agents. An intelligent agent is a collection of sensors, computers, and effectors, structured in such a way that the sensors can measure conditions in the world, the computers can process the sensor information, and the effectors can take action in the world. Changes in the world realized by the effectors close the loop to the agent's sensors, necessitating further sensing, computation, and action by the agent.

In recent years there has been a proliferation of proposals in the AI literature for integrated-agent architectures. Each architecture offers an approach to the general problem of constructing an integrated agent. Unfortunately, the ways in which one architecture might be considered better than another are not always clear.

For instance, Nilsson's (1988) *action nets* provide a means for structuring actions in terms of the individual goals they are to achieve in the environment. Rosenschein and Kaelbling's (1986, 1989) *situated automata theory* provides a new view on the role of logic, complexity, and information in situated agents, and has resulted in a new generation of software tools for building complex systems (Kaelbling, 1987a,b, 1988). Schoppers (1987) has suggested an approach to plan generation and execution based on the idea of *universal plans*. Georgeff and Lansky's (1985) *Procedural Reasoning System* provides a graphical programming environment based on the theory of augmented transition networks. *Plan nets* (Drummond, 1989; Drummond & Bresina, 1990) act as generators of possible behaviors and help explain the relationship between planned and unplanned action. Recent work on the SOAR architecture has studied the problems that arise when an integrated learning and problem-solving system interacts with an external environment (Laird & Rosenbloom, 1990).

But so what? What can systems based on these architectures really do? What can one do that another cannot?

*This work was supported by the Defense Advanced Research Projects Agency through NASA-Ames under contract NAS2-13229.

There has been a growing realization that many of the positive and negative aspects of an architecture become apparent only when experimental evaluation is performed and that to progress as a discipline, we must develop rigorous experimental methods. In addition to the intrinsic intellectual interest of experimentation, rigorous performance evaluation of systems is also a crucial practical concern to our research sponsors. DARPA, NASA, and AFOSR (among others) are all actively searching for better ways of experimentally evaluating alternative approaches to building intelligent agents.

One tool for experimental evaluation involves testing systems on benchmark tasks in order to assess their relative performance. As part of a joint DARPA- and NASA-funded project, NASA-Ames and Teleos Research are carrying out a research effort to establish a set of benchmark tasks and evaluation metrics by which the performance of agent architectures may be determined. This paper is a short report on the project's general aims and proposed methods. Possible points of debate are addressed by looking back over the transcripts of the Benchmarks and Metrics Workshop, held at NASA-Ames in June, 1990 (referred to hereafter as the BMW-I). This paper does not reproduce any statements from the BMW-I in verbatim form, but instead attempts to communicate the essence of the participants' comments.

2 Roles of Benchmark Tasks in the Research Community

Consider the following representative definition of "benchmark."

> **Benchmark** *n* – surveyor's reference mark for determining further heights and distances. (Garmonsway, 1965)

A benchmark is a reference point; a tool for determining where one stands. In this sense, a benchmark does not uniquely determine how data obtained from the study of that benchmark will be used. Data obtained from the study of benchmarks can be used in a variety of ways, and some of these are presented in this section. In summary, we feel that in order to correctly determine "further heights and distances" in the

integrated agent research community, we must have a common language for describing agent architecture performance. One way to do this is by establishing a common set of benchmark tasks and evaluation metrics.

A suite of carefully-designed benchmark tasks would serve two primary roles in the research community.

From experiments to principles. The first, and most important, role would be to provide a common frame of reference for researchers. In the literature, a wide variety of incommensurable vocabularies are used to describe intelligent systems. Someone faced with the task of understanding the relative strengths and weaknesses of different architectures must, currently, be able to grasp the relationship between "productions," "wires," "operator descriptions," "problem spaces," and many other such concepts. A set of benchmark tasks would allow the performance of systems to be directly compared; in addition, it would allow the designers of agent architectures to relate the internal characteristics of their architectures to externally observable properties of instances of those architectures. A researcher would be able to say things like "System X performs as it does on benchmark A because its representation of time is so flexible but its algorithm for Y is too slow."

The understanding gained from such experiments is critical. Consistent success or consistent failure on a specific set of benchmark problems should lead one to consider what features the given set of benchmarks share. If some particular problem feature can be implicated in the necessary success or failure of a given architecture then this source of knowledge can be fed back into the architecture's design. Without this sort of "closed-loop" experimental evaluation, corrections to any given architecture can be motivated only by abstract mathematical and computational aesthetics. The space of possible architectures is huge, and most notions of formal aesthetics are ill-defined. It makes more sense to explore the space of possible architectures driven by success and failure on particular representative benchmark problems.

Enabling technology transfer. The occasional practical application of integrated agent technology would not hurt the field. A wide variety of practical industry and government problems would benefit from better technology transfer. Technology transfer would be facilitated by benchmark problems that are representative of the intrinsic difficulty of practical applications. As it stands, a person with a practical application task in mind is given no easy access into the relevant set of technologies. For instance, given the problem of controlling a particular device in a factory, under particular constraints and resources, which architecture would be more appropriate: PRS (Georgeff & Lansky, 1985) or O-Plan (Currie & Tate, 1985)? Which system is better suited for the application at hand? Both PRS and O-Plan have been tested on representative tasks: PRS has been applied to the Space Shuttle reaction control system and O-Plan has been applied to spacecraft mission sequencing. However, not all architectures have been applied in this way, and even PRS and O-Plan must be further applied to other problems to better understand their individual strengths and weaknesses. The existence of a common set of benchmark tasks will make such comparisons possible across the entire integrated-agent research community.

As well as facilitating technology transfer to problems of practical interest, a common set of benchmarks can make new research ideas available to researchers in other fields. For instance, one might expect that research in the area of real-time operating systems would benefit from an understanding of the latest ideas in integrated agent architectures. Representative tasks of common merit would facilitate communication among various fields.

3 What Benchmarks Are Not

There are some obvious and some non-obvious worries associated with the establishment of a common set of benchmark tasks and evaluation metrics. This section briefly addresses some worries that were articulated by participants of the BMW-I.

Benchmarks will necessarily force us to worry only about numbers. A benchmark simply provides system performance indicators, perhaps expressed as numbers, perhaps not. These performance indicators can be used in a variety of ways. A particularly pedestrian use of performance indicators would be simple comparison; for instance, a question such as "which system got the highest score on task A?" with no further scrutiny as to *why* is deeply uninteresting. As mentioned above, the availability of performance indicators derived from the application of a particular architecture to a particular benchmark is a starting point for understanding the principles underlying the architecture's performance.

The benchmarks will not include all theoretically interesting problems and will also not be representative of practical, real-world problems. What if the benchmarks are so badly chosen that the performance data they engender is totally meaningless, both theoretically and practically? This is a legitimate worry, and due care must be taken when selecting benchmark tasks. Of course, it is inevitable that tasks judged inappropriate by some will be included in the common set, but each research group is free to choose those benchmark tasks that address issues of concern to them. If the set of common benchmark tasks does not provide a task of interest, then an appropriate task may be added. Community acceptance of a proposed task will come in the form of other research groups publishing performance results on that task.

An architecture can't be evaluated by a single benchmark. We do not propose to directly evaluate architectures through performance on an individual benchmark problem, but instead propose to evaluate

the performance of a specific system that is an instance of an architecture. Formal evaluation of the process by which an architecture is used to produce a solution to a specific benchmark problem is beyond the scope of this first effort. An architecture might include, but would not be limited to: design principles, programming languages, programming lore, user's manuals, existing code, etc. We cannot directly control for these architecture-related features, so the true worth of any given architecture cannot be directly evaluated by the method of benchmarks and metrics. Instead, an architecture will be evaluated in a "second-order" way: consistent success by the users of an architecture in applying that architecture to a widely-ranging set of benchmark problems will indicate the architecture's general utility.

4 Benchmark Tasks and Evaluation Metrics

4.1 Possible Task Attributes

Benchmark tasks can seen as varying across a number of dimensions. Placement in the space of possible task attributes will, in some sense, determine "task difficulty". As tasks are added to the evolving set of benchmarks, the importance of particular task attributes will become apparent.

It would be practically useful to have some families of tasks in which stress on selected task attributes could be systematically varied. This would allow researchers to experiment with a set of problems of graded difficulty and perhaps allow insight into how agent performance scales with task difficulty.

The following is an initial subset of possible task attributes.

Resource Management. Does the task have properties pertaining to metric time and continuous quantities? If so, the problem may take on the character of a classical optimization problem.

Geometric and Temporal Reasoning. Does the task involve extensive geometry or reasoning about activities over time? These kinds of reasoning may require specialized representations.

Deadlines. Does the task impose absolute deadlines for goal satisfaction, or does the utility of goal satisfaction vary continuously with time?

Opportunity for Learning. Is the task specification complete at the beginning of the agent's execution? If not, the agent must be able to acquire knowledge about its environment.

Multiple Agency. Does the task require the definition of a community of interacting agents? Such tasks might require complex communication protocols or reasoning about the internal states of other agents.

Informability. Can the agent be presented with explicit goals and facts about the world during the course of its execution?

Dynamic Environment. Does the agent's world change over time independent of the actions the agent takes? How predictable are the dynamics of the world?

Amount of Knowledge. How much *a priori* knowledge is available to be used by the agent? Some domains, such as medical diagnosis, require the assimilation of a large amount of domain knowledge.

Reliability of Sensors and Effectors. In some task domains, sensors and effectors are completely reliable; in others, the main difficulty lies in accurately integrating data from a number of highly unreliable sensors and achieving robust overall behavior through the use of unreliable effectors.

4.2 Task Specifications

There are a broad range of possible methods for specifying tasks, ranging from informal to physical. Each of these methods has associated pros and cons; for instance, natural language task descriptions, simulators, and formal specifications are all easily transmitted electronically, but physical environments are less easily duplicated.

Natural Language Task Descriptions. Natural language descriptions can be too vague for use in focused, comparative studies, but they are easy to generate, which makes them useful in certain situations. An example might be "Pick up cups using a simple mobile robot with a manipulator and an overhead vision system."

Simulators. Simulators provide a precise computational task description, which facilitates direct comparison between solutions. In addition, it is often possible to instrument a simulation to simplify debugging and evaluation.

Formal Specifications. Tasks may specified using a logical or mathematical description of the environment, its sensor and effector interfaces with the agent, and the goals of the agent. Formal specifications may allow prior analysis and provide insight into the underlying problem complexity. It is very difficult, however, to specify formally the majority of complex tasks.

Physical Environments. Some tasks are most easily specified by providing a physically embodied environment. Such specifications have the disadvantage of being difficult to replicate, because it is hard to control all aspects of the environment, such as lighting and RF interference.

A useful suite of benchmarks might include a variety of specification types for a particular general task; the resulting specifications would describe slightly different but closely related tasks. A useful example would be to have both simulation and physical specifications of a robotic control task. This would allow researchers to debug their ideas in simulation before trying them out in the real world. Although success in a simulation

is no guarantee of success in the real world, failure on a simulation will often entail real-world failure as well.

4.3 Modes of Evaluation

Part of the specification of benchmark tasks is the selection of particular evaluation metrics. Suitable metrics will become more obvious as the work progresses—some metrics will be applicable to all benchmark tasks and others will be task-specific. At this early stage however, two classes of metrics are of clear importance.

Agent Performance. How well does the agent perform? Performance will be measured using domain-specific performance metrics that are supplied in conjunction with each benchmark task.

Agent Construction. An important aspect of an integrated agent architecture is the ease with which it can be applied to a range of tasks. While it is difficult to formally measure the time to build a system, it can be informally reported in conjunction with agent performance results.

5 Conclusions

The development of a set of benchmark tasks and performance metrics for integrated agent architectures will foster both scientific progress and technology transfer. Broad coverage of the space of task attributes by the benchmark tasks will be required in order to ensure scientific and practical relevance of the benchmarks.

It is important to understand that no single research group will be able to determine a community-wide set of benchmarks by fiat. For a set of benchmarks to be accepted by a community, they must be designed by that community. The process presented in this paper, namely, that of a developing set of benchmarks that can be augmented by anyone willing to define a benchmark task, is one way to achieve such acceptance. The resulting set of benchmark tasks should be viewed as a resource that will foster progress, not as an exam for members of the community to pass or fail.

Acknowledgments

Thanks to Stan Rosenschein for help in organizing BMW-I and for insightful comments on drafts of this paper. Thanks also to all of those participated in BMW-I.

References

[1] K. Currie & A. Tate, 1985. O-Plan: Control in the open planning architecture. In *Proceedings of BCS Expert Systems '85.* Warwick, U.K. Cambridge University Press. pp. 225-240.

[2] M. Drummond, 1989. "Situated Control Rules," *Proceedings of the Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Canada.

[3] M. Drummond and J. Bresina, 1990. "Anytime Synthetic Projection: Maximizing the Probability of Goal Satisfaction," *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, Massachusetts, pp. 138-144.

[4] N. Garmonsway, 1965. *The Penguin English Dictionary.* Penguin Books Ltd, Harmondsworth, Middlesex, England.

[5] M. Georgeff and A. Lansky, 1985. "A Procedural Logic", *Proceedings IJCAI-85*, Los Angeles, Calif.

[6] L.P. Kaelbling, 1987a. "An Architecture for Intelligent Reactive Systems," *Reasoning About Actions and Plans*, M. Georgeff and A. Lansky, Eds., Morgan Kaufmann

[7] L.P. Kaelbling, 1987b. "Rex: A Symbolic Language for the Design and Parallel Implementation of Embedded Systems," *Proceedings of AIAA Conference on Computers in Aerospace*, Wakefield, Massachusetts.

[8] L.P. Kaelbling, 1988. "Goals as Parallel Program Specifications" Proceedings of the Seventh National Conference on Artificial Intelligence, St. Paul, Minnesota.

[9] J.E. Laird and P.S. Rosenbloom, 1990. "Integrating Execution, Planning, and Learning in Soar for External Environments," *Proceedings of Eighth National Conference on Artificial Intelligence*, pp. 1022-1029.

[10] N.J. Nilsson, R. Moore, and M. Torrance, 1990. *ACTNET: An Action Network Language and its Interpreter.* Draft paper, Stanford Computer Science Department.

[11] S.J. Rosenschein, & L.P. Kaelbling, 1986. "The Synthesis of Digital Machines with Provable Epistemic Properties," *Proceedings of Workshop on Theoretical Aspects of Knowledge*, Monterey, CA (March 13-14).

[12] S.J. Rosenschein & L.P. Kaelbling, 1989. "Integrating Planning and Reactive Control", *Proceedings of NASA Telerobotics Conference*, Pasadena CA.

[13] M.J. Schoppers, 1987. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proceedings of IJCAI-87.* Milan, Italy. pp. 1039-1046.

The CORTES Project: A Unified Framework for Planning, Scheduling and Control

Mark S. Fox and Katia P. Sycara

Center for Integrated Manufacturing Decision Systems
The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

We present an overview of CORTES, an integrated framework for production planning, scheduling and control (PSC). CORTES's approach to PSC problems departs from others in the hypotheses it explores: *Generality Hypothesis*: There exists a single approach that can optimize decision making across a wide variety of PSC problems. *Flexibility Hypothesis*: The same approach can be used for both planning, predictive scheduling and reactive control. *Uncertainty Hypothesis*: In order to provide the appropriate level of precision in PSC, reasoning about uncertainty must be an integral part of the PSC approach. *Scale Hypothesis*: Large PSC problems, that contain thousands of activities, resources and constraints, must be solved in a qualitatively different manner than small PSC problems. CORTES uses Constrained Heuristic Search to make PSC decisions. In this paper, we describe CORTES, its architecture, problem solving method, and functions including modeling, planning, scheduling, distributed scheduling, dispatching, and uncertainty management.

1. Introduction

Our research explores the role of constraints in solving planning, scheduling and control (PSC) problems. It is generally believed that to efficiently construct optimizing solutions to large PSC problems, a fundamental understanding of ***problem structure*** and ***properties*** is required. It is our conjecture that knowledge of domain constraints will lead to this understanding. The goal of the CORTES project is to operationalize this conjecture.

CORTES is a distributed system for production planning, scheduling and control. CORTES is designed to be composed of an integrated set of modules distributed across many workstations and connected by a communication network. The overall architecture is shown in Figure 1-1.

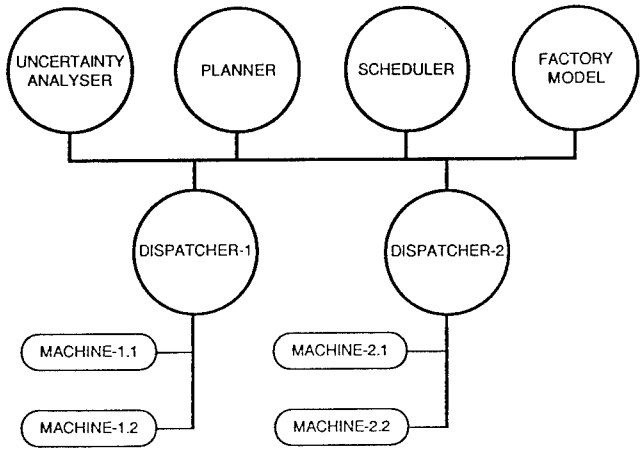


Figure 1-1: The CORTES Architecture

CORTES represents a departure from previous approaches to solving PSC problems in the hypotheses it explores:

1. ***Generality Hypothesis:*** There exists a single approach that can optimize decision making across a wide variety of PSC problems. Previously, PSC approaches were tailored to the particular production environment, with the "common wisdom" being that there does not exist a single approach, short of enumeration, that applies to all PSC problems. We believe that there does exist a single approach that may be generally applied to PSC problems, that also provides very good results and is computationally efficient.
2. ***Flexibility Hypothesis:*** The same approach can be used for both planning, predictive scheduling and reactive control. Traditionally, planning, scheduling and control approaches have tended to be separate and unrelated in approach. For example, in actual production environments,Manufacturing Resource Planning (MRP) tends to be used for

planning, scheduling approaches can range from dispatch approaches to knowledge based scheduling, and control tends to be ignored. In AI, planning algorithms tend to be generative, scheduling is constraint directed, and control is reactive.

3. ***Uncertainty Hypothesis:*** In order to provide the appropriate level of precision in PSC, reasoning about uncertainty must be an integral part of the PSC approach. Production environments contain a plethora of stochastic events that increase the uncertainty with which a schedule may be executed. For example, unexpected events such as personnel not showing up for work, machine failures, failure of resources, etc, may quickly invalidate a schedule. Consequently, PSC approachs must take a pro-active approach in mitigating the effects of uncertainty.

4. ***Scale Hypothesis:*** Large PSC problems, that contain thousands of activities, resources and constraints, must be solved in a qualitatively different manner than small PSC problems. The point is that in large PSCs, the aggregate behavior of the system be optimized, as opposed to any individual entity or job. Optimizing each decision is computationally expensive. Instead, many decisions must be made at the aggregate level using statistical summaries of underlying requirements.

CORTES is evolutionary in its approach in that it can be viewed as a continuation of the line of constraint directed scheduling systems developed at Carnegie Mellon University [Fox & Smith 84, Smith et al. 86, Fox 90]. It departs from the approach of these previous systems in its use of Constrained Heuristic Search (CHS) as its underlying problem solving paradigm [Fox 89].

In the reminder of the paper, we first review the Constrained Heuristic Search (CHS) problem solving paradigm. We then describe the functionality of each of the modules in figure 1-1 and how this is provided using CHS.

2. Constrained Heuristic Search

Our approach to both planning and scheduling is based upon a problem solving paradigm we call Constrained Heuristic Search (CHS)[1] CHS views problem solving as a constraint optimization activity. CHS combines the process of constraint satisfaction (CSP) [Mackworth 87] with heuristic search (HS). CHS retains heuristic search's synthetic capabilities and extends it by adding the structural characteristics of constraint satisfaction techniques. In particular, our model adds to the definition of a problem space [Newell & Simon 76], composed of states, operators and an evaluation function, by refining a state to include:

1. **Problem Topology**: Provides a structural characterization of a problem.
2. **Problem Textures**: Provide measures of a problem topology that allows search to be focused in a way that reduces backtracking.
3. **Problem Objective**: Defines an objective function for rating alternative solutions that satisfy a goal description.

This model allows us to (1) view problem solving as constraint optimization, thus taking advantage of these techniques, (2), incorporate the synthetic capabilities of heuristic search, thus allowing the dynamic modification of the constraint model, and (3) extend constraint satisfaction to the larger class of optimization problems. In the following, problem topology and textures are defined.

2.1. Problem Topology

We define problem topology as a constraint graph G, composed of variables V and constraints C:

V is a set of variables $\{v_1, v_2, ..., v_m\}$
C is a set of constraints $\{c_1, c_2, ..., c_n\}$

Each variable in N may be a vector of variables whose domains may be finite/infinite and continuous/discrete. Constraints are n-ary predicates over variables vertices.

We extend the topology to allow constraints to be grouped into a modified conjunctive normal form:

$[s_1 \text{ AND } s_2 \text{ AND } ... \text{ AND } s_o]$

where each s_i denotes a set of constraints where either only one or at least one constraint must be satisfied.

We distinguish between two types of problem topologies:

Definition 1: A *completely structured problem* is one in which all non-redundant vertices and edges are known a priori.

This is true of all CSP formulations and in this case CHS reduces to either a CSP or a COP (i.e., optimization problem).

Definition 2: A *partially structured problem* is one in which not all non-redundant vertices and

[1]This section is a composed of excerpts from [Fox 89].

edges are known prior to problem solving.

This definition tends to be true of problems in which synthesis is performed resulting in new variables and constraints (e.g. the generation of new subgoals during the planning process).

Operators in CHS have many roles: refining the problem by adding new variable and constraint vertices, reducing the number of solutions by reducing the domains of variables (e.g., assigning a value to a variable vertex), or reformulating the problem by relaxing constraints or omitting constraints and/or variables.

Our intent is to distinguish topologies that lead to significant changes in problem solving quality and efficiency. Examples include:

- The decomposability of constraint graphs into unconnected or lossely connected subgraphs, allowing the problem solver to focus on one set of variables and constraints before attending to another.
- Graph width which combined with arc-consistency will guarantee backtrack free search [Freuder 82].
- Contention graphs which identify the degree of contention that exists among variables for the same values.

2.2. Problem Textures

Focus of attention in search is concerned with the ability of the search algorithm to opportunistically decide where the next decision is to be made [Erman et al. 80]. In CHS, for search to be well focused, that is to decide where in the problem topology an operator is to be applied, there must be features of the topology that differentiate one subgraph from another, and these features must be related to the goals of the problem. We have identified and are experimenting with seven such features that we call problem *textures* [Sadeh & Fox 88]. Below we define these textures for CHSs where all solutions are equally preferred, i.e., the Problem Objective rates all solutions to the constraints equally acceptable.

(Variable) Value Goodness: the probability that the assignment of that value to the variable leads to an overall solution to the CHS (i.e. to a fully consistent set of assignments). This texture is related to the value ordering heuristics [Haralick & Elliott 80] which look for the least constraining values. Value ordering heuristics are meant to reduce the chance of backtracking. In the case of discrete variables, the goodness of a value is the ratio of complete assignments that are solutions to the CHS and have that value for the variable over the total number of possible assignments.

Constraint Tightness: Constraint tightness refers to the contention between one constraint or a subset of constraints with all the other problem constraints. Consider a CHS A and a subset C' of constraints in A. Let B be the CHS obtained by omitting C's constraints in A. The constraint tightness induced by C' on A is defined as the probability that a solution to B is not a solution to A. In the case of discrete variables, this is the ratio of solutions to B that are not solutions to A over the total number of solutions to B.

Variable Tightness with respect to a set of constraints: Again consider a CHS A, a subset C' of constraints, and the CHS B obtained by omitting C' in A. A variable V's tightness with respect to the set of constraints C' is defined as the probability that the value of V in a solution to B does not violate C'. In the case of discrete variables, this is simply the ratio of solutions to B in which V's value violates C' (i.e. at least one of the constraints in C') over the total number of solutions to B.

Constraint Reliance: This measures the the importance of satisfying a particular constraint. Consider a constraint c_i. We defined CHS B as being CHS A - $\{c_i\}$. Given that constraints can be disjunctively defined, the reliance of CHS A on a constraint c_i is the probability that a solution to CHS B is not a solution to A. In the case of discrete variables, constraint reliance is defined as the ratio of the number of solutions to CHS B that are not a solution to CHS A to the number of solutions to CHS B. The larger the value, the greater the reliance the problem has on satisfying the particular constraint.

Variable Tightness: Consider a variable v in a CHS A. Let C' be the set of constraints involving v and B be the CHS obtained by omitting C' in A. v's tightness with respect to C' is simply called v's tightness. Hence the tightness of a variable is the probability that an assignment consistent with all the problem constraints that do not involve that

variable does not result in a solution. Alternatively one can define *variable looseness* as the probability that an assignment that has been checked for consistency with all the problem constraints, except those involving that variable, results in a fully consistent assignment. Notice that if one uses a variable instantiation order where v is the last variable, v's tightness is the *backtracking probability*. Variable looseness/tightness can be identified with variable ordering heuristics [Haralick & Elliott 80, Freuder 82] which instantiate variables in order of decreasing tightness.

These textures generalize the notion of constraint satisfiability or looseness defined by [Nadel 86] and apply to both CHSs (and CSPs) with discrete and continuous variables. Notice that, unless one knows all the CHS's solutions, the textures that we have just defined have to be approximated. Textures may sometime be evaluated analytically [Sadeh & Fox 88]. A brute force method to evaluate any texture measure consists in the use of Monte Carlo techniques. Such techniques may however be very costly. In general, for a given CHS, some textures are easier to approximate than others, and some are also more useful than others. Usually the texture measures that contain the most information are also the ones that are the most difficult to evaluate. Hence there is a tradeoff. Each domain may have its own approximation for a texture measure.

2.3. Problem Objectives

Many problems, for example scheduling, are optimization problems and not simply satisfaction problems. The notion of what is best becomes important. Rather then defining what is best in an evaluation function or an objective function, our goal is to embed objectives directly in the constraint graph so that it can be both propagated and used to make local decisions. For example,

- Disjunctive constraint sets may have preferences associated with each disjunct.
- Start times, commonly found as a variable in scheduling constraint graphs, can have preferences associated with each alternative time.

In our work we have extended the textures to take into account the Problem Objectives, using Bayesian probabilities to approximate the likelihood that a variable results in an optimal value [Sadeh & Fox 88].

2.4. CHS Problem Solver

The CHS model of problem solving is a combination of constraint satisfaction and heuristic search. Search is performed in the problem space where each state contains a problem topology. The problem solving model we propose contains the following elements:

- An initial state is defined composed of a problem topology, i.e., the PSC activity, time and capacity constraint graph,
- Constraint propagation is performed within the state,
- Texture measures and the problem objective are evaluated for the state's topology,
- Operators are matched against the state's topology, and
- A variable node/operator pair is selected and the operator is applied, i.e., a resource or start time is assigned to an activity.

The application of an operator results in either adding structure to the topology, further restricting the domain of a variable, or reformulating the problem (e.g., relaxation).

It is our belief, which is supported by experimentation, that this approach is powerful enough to solve a variety of PSC problems. Domains in which it has been applied include, job shop scheduling [Sadeh 90], cell scheduling and transportation planning [Sathi et al. 90]. Secondly, the opportunism inherent in the approach, allows the approach to be applied to both predictive planning and scheduling, and reactive control.

2.5. Representation

The main conceptual primitives in the CORTES representation are *activities*, *resources*, *production units*, *states*, and *constraints* [Sathi et al. 85, Fox 88]. These primitives provide an extensible framework that can be used to represent the relevant aspects of manufacturing environments. In addition, these primitives are represented at various levels of conceptual abstraction depending on the granularity of knowledge. For example, an *operation* is a specialization of an activity. An important component of the representational framework is the *relations* that connect the primitives and their instantiations. The main types of relations are temporal and causal.

In general, there are five types of constraints that a scheduler should take into consideration. These five types are domain-independent and help structure constraints in many kinds of scheduling domains (e.g., factory scheduling, transportation scheduling).

- Physical constraints. Physical constraints include, number of machines, fixtures, setup and run times for each operation.

- Organizational constraints. Examples of organizational constraints include meeting due dates, reducing Work in Process, increase machine utilization, and enhance throughput.
- Preferential constraints. Examples of preferential constraints include preference for using a particular machine for an operation (perhaps because of its speed or accuracy), or using a particular human operator (perhaps because of his skill).
- Enablement constraints. These refer to constraints, the fulfillment of which creates a state that enables the execution of an activity. For example, a process plan embodies enablement constraints.
- Availability constraints. These constraints refer to the availability of particular resources at scheduling time. For example, a machine may become unavailable because of breakdown, the assignment of a third shift makes extra resources available for scheduling.

In the model, we treat explicitly two types of constraints, *required constraints* and *preferential constraints* [Fox 83]. The degree of satisfaction of a preferential constraint is expressed by a *utility function* ranging between 0 and 1. A value of 0 utility is non-admissible; a value of 1 is optimal. Variables can be constrained by more than one constraint. The utility value associated with a variable is calculated by taking the weighted sum (with constraint importance as the weight) of the utilities of all the constraints that affect the variable.

Constraints differ in *importance*. A particular constraint could have different importance depending on the context in which it is applied. The importance of a constraint is specified by a value between 0 and 1. An importance of 0 implies that the constraint should not be considered, and 1 signifies maximum importance. The actual level of importance is relative to the importance of the other constraints under consideration. The measure of importance of a constraint may be viewed as a weight that can be combined with a constraint's utility value to form a weighted combination of utilities. Constraints also differ in *relevance*. Depending on the context, a constraint may be more relevant than others.

3. Scheduler

The detailed scheduler is an activity-based scheduler [Sadeh 90], where the activities are the operations that must be scheduled according to a process plan that specifies a partial ordering among these operations. Each operation requires one or several resources for each of which there may be one or several alternatives. Scheduling is viewed as a constrained heuristic search problem whose solution is a schedule that satisfies the many technological, temporal, organizational, and preference constraints that are imposed both by the characteristics of the job shop itself and the environment.

The scheduler models a problem as a constraint graph, where there are two types of nodes: activities and resources. An activity is an 4-tuple defining its start time, duration, and resources it is to use. With each activity, we associate utility functions that map each possible start time and each possible resource alternatives onto a utility value (i.e. preference). These utilities [Fox 83, Sadeh & Fox 88] arise from global organizational goals such as reducing order tardiness (i.e. meeting due dates), reducing order earliness (i.e. finished good inventory), reducing order flowtime (i.e. in-process inventory), using accurate machines, performing some activities during some shifts rather than others, etc. A resource is a 3-tuple defining its total capacity, available capacity over time, and the activities that are scheduled to use it.

We distinguish between two types of constraints: activity temporal constraints and capacity constraints. The activity temporal constraints together with the order release dates and latest acceptable completion dates restrict the set of acceptable start times of each activity. The capacity constraints restrict the number of activities that a resource can be allocated to at any moment in time to the capacity of that resource. For the sake of simplicity, we only consider resources with unary capacity in this paper. Typically the limited capacity of the resources induces interactions between orders competing for the possession of the same resource at the same time.

The schedule is built incrementally by iteratively selecting an activity and assigning a start time and resource(s) to it, propagating temporal and capacity constraints and checking for constraint violations. If constraint violations are detected the system backtracks. Search is focused via a set of *variable* and *value* ordering heuristics so as to minimize backtracking and optimize schedule quality.

The variable ordering heuristic assigns a *criticality measure* to each unscheduled activity; *the activity with the highest criticality is scheduled first.* The value ordering heuristic attempts to leave enough options open to the activities that have not yet been scheduled in order to reduce the chances of backtracking. This is done by assigning a *goodness* measure to each possible reservation of the activity to be scheduled. Both activity criticality and value goodness are composed of *texture measures.* The

next two paragraphs briefly describe both of these measures[2].

A critical activity is one whose resource requirements are likely to conflict with the resource requirements of other activities. [Sadeh & Fox 88, Sadeh 90] describes a technique to identify such activities. The technique starts by building for each unscheduled activity and for each appropriate time interval a probabilistic *activity demand* that denotes the probability that the activity will require a resource at that time interval. Clearly activities with many possible start times and resource reservations tend to have smaller demands at any moment in time, while activities with fewer possible reservations tend to have higher ones. In a second step, the activity demands for each resource are aggregated over time to form a demand profile for a resource. The demand profile expresses likely contention for the resource over time. The percentage contribution of an activity's demand to the aggregate demand for a resource over a highly contended-for time interval is the *activity reliance*.

To choose the next activity to schedule, the scheduler focuses on the resource/time interval with the highest aggregate demand. The activity with the the highest reliance on the resource is picked to be scheduled next, since it is the activity that is most likely to be involved in contention for the resource.

The particular start time assigned to the chosen activity is picked using either of two strategies:

1. **A Least Constraining Value Ordering Strategy (LCV)**: This heuristic attempts to select the reservation that is the least likely to prevent other activities to be scheduled.
2. **A "Greedy" Value Ordering Strategy (GV)**: At the other extreme, a reservation can be chosen that maximizes the preference of the activity for the resource/time interval.

Experimental results have demonstrated the effectiveness of this approach for problems where resource contention is an issue [Sadeh 90].

4. Distributed Scheduling

As part of the CORTES project, we are investigating how to manage scheduling when distributed across multiple schedulers [Sycara 90]. In particular, we are investigating how schedulers, which possess their own resources, coordinate their decisions when they require resources possessed by others. Due to the size of the scheduling problem, we distinguish between coordination at the strategic level versus the tactical level. Our approach assumes that each scheduler develops schedules using Constrained Heuristic Search. At the strategic level, the coordination of large numbers of activities requiring resources outside of a particular scheduler is performed by communicating statistical summaries of aggregate demand textures. These demands are used to bias a scheduler's reservations so that it does not require another's scheduler's resources during a period of high demand. At the strategic level, we expect that coordination problems will be reduced but not removed. It is the role of the tactical level to negotiate resource allocations that could not be handled strategically.

The textures play four important roles in distributed search: (1) they focus the attention of an agent to globally critical decision points in its local search space, (2) they provide guidance in making a particular decision at a decision point, (3) they are good predictive measures of the impact of local decisions on system goals, and (4) they are used to model beliefs and intentions of other agents. The development of the presented texture measures is the result of extensive experimentation in the single agent setting. We have completed the implementation of a distributed testbed and are currently performing experiments involving multiple agents. Experimental results from the distributed scheduling testbed are presented in [Sycara 90].

5. Planning

We are currently investigating the integration of planning with scheduling[3]. In previous planners, planning has been an end unto itself. Any feasible plan is considered a success, with only very inflexible criteria for plan quality, such as minimizing the total number of actions. In the context of the CORTES project, the planner will be producing process plans to be used by the scheduler. The quality of these plans is defined by the quality of the schedules that the scheduler can produce using them. Thus, there is a strong need for a constraint language to use in communicating with the scheduler to determine what sorts of plans would be good.

Current state-of-the-art planners are constraint-directed, domain-independent, hierarchical, nonlinear, and support replanning [Wilkins 88]. We intend to include these capabilities, and extend them where appropriate.

[2]For a more complete description of these measures, the reader is referred to [Sadeh & Fox 88, Sadeh 90].

[3]See [Frederking & Chase 90] for more details.

Planners already exist that use constraints on planning variables to increase the power of their representation and to reduce arbitrary decisions that can lead to unnecessary backtracking. In addition to making wider use of constraints, we will make this planner be truly constraint-*directed* by developing measures of criticality for goal ordering and operator selection. This will provide a domain-independent representation for the domain-dependent heuristics that focus attention in the search for a plan.

The planner will always support planning at different levels of abstraction, and the re-use of plans in support of reactive planning.

6. Uncertainty Analyzer

Uncertainty is a fact of life in most job shop scheduling environments. Sources of uncertainty include: Demand change (seasonal, forecast error, cancel orders, expedition), Inventory Policy (raw material arrival pattern, safety stock policy) Machine failure, Change of time duration (transit, set-up, processing), Yield, and Quality (Tool wear, precision). Uncertainty increases as the planning horizon is extended, and its the amount and sources of uncertainty change over time.

The presence of uncertainty means that it is very unlikely that a detailed predictive schedule that assigns precise start and finishing times on resources for activities is going to be adhered to. This characteristic imposes two requirements on schedulers: (a) A scheduler should be able to represent and reason about degrees of uncertainty, and (b) a scheduler should be able to react to unexpected events on the factory floor. The inability of a scheduler to reason about uncertainty almost always results in a schedule being invalid at the time it is released to the production floor.

CORTES manages uncertainty in three stages. In the first stage, the Uncertainty Analysis module monitors and records the stochastic events. It develops over time a model of the sources and characteristics of uncertainty. Once a valid model is constructed, the Uncertainty Analysis module passes the information to the Scheduler. In the second stage, the scheduler uses the uncertainty models to reduce the precision of its schedules. Precision can be reduced by increasing the durations of activities, overlapping activity temporal intervals, or assigning activities to resource aggregates rather than to specific resources. In the third stage, the Dispatcher control module, is able to react more flexibly to stochastic events by taking advantage of the imprecision inserted in the schedule by the scheduler; it can start an activity earlier or later or assign an activity to another resource in an aggregate (i.e., work center). The Dispatcher's task is to dispatch jobs to machines and monitor machine and job execution status. The Dispatcher notes deviations from the schedule and resource unavailability and communicates this information to the Scheduler, Uncertainty Analyzer and factory floor.

Two approaches have typically been utilized to address the problem of temporal uncertainty. One approach is based on the idea of dividing the time horizon into time zones using progressively coarser time units to describe events in the future. For example, a time unit of one hour may be used to project a schedule over a one week horizon; a time unit of a day mey be used to project a schedule from a one-week to a one-month horizon and so on. Although this approach recognizes the fact that events that are further int the future are less accurately predictable, it has been criticized [Kerr 89] as suffering from the presence of discontinuous boundaries between time zones and the difficulty of handling orders whose processing crosses a zone boundary. A second approach to handling uncertainty is the use of probability distributions to describe schedule parameters. This approach has the disadvantage [Kerr 89] that probability is concerned with the combination and manipulation of independent random variables whereas many of the probabilistically described scheduling parameters are not independent (e.g., processing times of different jobs on a particular machine could depend on some characteristic of the machine)[4].

The CORTES uncertainty analyzer represents uncertainty in terms of fuzzy logic [Zadeh 85, Kaufmann 85, Prade 79]. The present version [Chiang & Fox 90] focuses on uncertainty concerning machine failures. The mean time between failure and mean duration of the failure are assumed known. It is also assumed that once a machine is fixed after a failure, processing resumes at the point of interruption with no rework necessary. In other words, machine failure causes a variation in processing time only and not in scheduling order. The time between machine failures and the failure duration are used to express uncertainty in processing time. Instead of being random variables of known distribution, the duration of failure and time between failures may be only approximately known. This approximate information on the procesing time bounds is expressed in terms of fuzzy numbers of *Type-1*, where a real number that is approximately known is expressed as a confidence interval of upper and lower bounds. Fuzzy bound values may be ther result of subjectively known processing characteristics

[4]Handling variable dependence through the use of conditional or joint probability distributions poses severe estimation problems.

described by a shop operator, or of known distributions described by shop statistics. An extension of type-1 fuzzy representation of uncertainty in operation duration is *Type-2* representation where the lower and upper bounds of a confidence interval, instead of being ordinary numbers are fuzzy numbers that themselves have intervals of confidence.

Uncertainty bounds work in a similar manner as earliest start time/latest start time and earliest finish/latest finish time. The bounds can be viewed as slack to protect against uncertainty. The mean processing time is reserved for the operation and the slack time is reserved for protection against uncertainty. Once an operation is ready for processing, a dispatcher should follow the schedule within the prescribed bounds. We ran experiments [Chiang & Fox 90] to compare various cost measures, such as tardiness, work-in-process, and delayed orders, under various processing duration representation schemes (type-2 fuzzy representation, fixed processing time given in the process plan, mean processing time considering machine failure duration and time between failures) and under different cost structures and shop loads. The general result is that type-2 bounds give sufficient protection against uncertainty in processing time with less investment in the planned cost (planned cost= planned tardiness+planned work-in-process+ planned lateness)[5]. For a more detailed description of the experiments, see [Chiang & Fox 90].

7. Conclusion

In this paper, we have given an overview of the CORTES integrated framework for production planning, scheduling and control (PSC) system. CORTES's approach to PSC problems departs from others in the hypotheses it explores:

1. Generality Hypothesis: There exists a single approach that can optimize decision making across a wide variety of PSC problems.
2. Flexibility Hypothesis: The same approach can be used for both planning, predictive scheduling and reactive control.
3. Uncertainty Hypothesis: In order to provide the appropriate level of precision in PSC, reasoning about uncertainty must be an integral part of the PSC approach.
4. Scale Hypothesis: Large PSC problems, that contain thousands of activities, resources and constraints, must be solved in a qualitatively different manner than small PSC problems.

The CORTES project is investigating all four assumptions in parallel. We have experimental data across a variety of PSC problems that support the generality assumption. The flexibility assumption is currently being tested by our integration of PSC functions. The uncertainty assumption is supported by the ease with which we have adapted CHS to account for uncertainty. The Scale assumption remains to be tested.

[5]To protect against uncertainty the planned operation duration is longer, more planned work-in-process exists and orders are planned to arrive late. Hence, the more protection we design into the bounds, the higher the planned cost.

References

[Chiang & Fox 90] Chiang, W-Y., and Fox, M.S. Protection Against Uncertainty In a Deterministic Schedule. In *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management*, May, 1990. Submitted for publication.

[Erman et al. 80] Erman, L.D., Hayes-Roth, F., Lesser, V.R., and Reddy, D.R. The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Computing Surveys*12(2):213-253, 1980.

[Fox 83] M. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1983.

[Fox 88] Fox, M., and Sycara. K. *Knowledge-Based Logistics Planning and its Application in Manufacturing and Strategic Planning.* Technical Report First Interim Report, submitted to RADC, CMU Robotics Institute, December, 1988.

[Fox 89] Mark S. Fox, Norman Sadeh, and Can Baykan. Constrained Heuristic Search. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Pages 309-315. 1989.

[Fox 90] Fox, M.S.
Constraint Guided Scheduling: A Short History of Scheduling Research at CMU.
Computers and Industry, 1990.
To Appear.

[Fox & Smith 84] Fox, M.S., and Smith, S.
ISIS: A Knoweldge-Based System for Factory Scheduling.
*International Journal of Expert Systems*1(1):25-49, 1984.

[Frederking & Chase 90]
Frederking, R.E., and Chase, L.L.
Planning in a CIM Environment: Research Towards a Constraint-Directed Planner.
In *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management*, May, 1990.
Submitted for publication.

[Freuder 82] Freuder, E.C.
A Sufficient Condition for Backtrack-free Search.
*Journal of the ACM*29(1):24-32, 1982.

[Haralick & Elliott 80]
Haralick, R.M., and Elliott, G.L.
Increasing Tree Search Efficiency for Constraint Satisfaction Problems.
*Artificial Intelligence*14(3):263-313, 1980.

[Kaufmann 85] Kaufmann, A. and Gupta, M.
Introduction to Fuzzy Arithmetic: Theory and Applications.
Van Nostrand Reinhold, New York, N.Y., 1985.

[Kerr 89] Kerr, R.M., and Walker, R.N.
A Job Shop Scheduling System Based on Fuzzy Arithmetic.
In *Proceedings of the 2nd Interational Conference on Expert systems and Leading Edge in Production and Operations Management*, Pages 433-450. Hilton Head Island, S.C., May, 1989.

[Mackworth 87] Mackworth, A.K.
Constraint Satisfaction,
In Shapiro, S., *Encyclopedia of Artificial Intelligence*. J. Wiley & Son, 1987.

[Nadel 86] Nadel, B.A.
The General Consistent Labeling (or Constraint Satisfaction) Problem.
Technical Report DCS-TR-170, Department of Computer Science, Laboratory for Computer Research, Rutgers University, New Brunswick, NJ 08903, 1986.

[Newell & Simon 76]
Newell, A., and Simon, H.A.
Computer Sciences as Empirical Inquiry: Symbols and Search.
*Communications of the ACM*19(3):113-126, 1976.

[Prade 79] Prade, H.
Using fuzzy set theory in a scheduling problem.
*Fuzzy Sets and Systems*2(2):153-165, 1979.

[Sadeh 90] Norman Sadeh, and Mark Fox.
Focusing Attention in an Activity-based Job Shop Scheduler.
In *Proceedings of the Forth International Conference on Expert Systems in Production and Operations Management*, 1990.

[Sadeh & Fox 88] Sadeh, N., and Fox, M.S.
Preference Propagation in Temporal Constraints Graphs.
Technical Report , Intelligent Systems Laboratory ,The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213, 1988.
CMU-RI-TR-89-2.

[Sathi et al. 85] Sathi, A., Fox, M.S., and Greenberg, M.
Representation of Activity Knowledge for Project Management.
*IEEE Transactions on Pattern Analysis and Machine Intelligence*PAMI-7(5): 531-552, September, 1985.

[Sathi et al. 90] Sathi, N., Fox, M.S., Goyal, R., and Kott, A.
CORAL: An Order Configuration and Resource Allocation System.
Technical Report , Carnegie Group Inc., Five PPG Place, Pittsburgh PA 15219,
1990.
In preparation.

[Smith et al. 86] Smith, S., Fox, M.S., and Ow, P.S.
Constructing and Maintaining Detailed Production Plans: Investigations into the Development of Knowledge-Based Factory Scheduling Systems.
*AI Magazine*7(4):45-61, Fall, 1986.

[Sycara 90] Katia Sycara, Steve Roth, Norman Sadeh, Mark Fox.
Managing Resource Allocation in Multi-Agent Time-Constrained Domains.
In *Proceedings of the 1990 Darpa Workshop on Innovative Approaches to Planning, Scheduling and Control*, 1990.

[Wilkins 88] Wilkins, D.E.
Practical Planning.
Morgan Kaufmann Publishers Inc., 1988.

[Zadeh 85] Zadeh, L.
Fuzzy Sets.
*Information and Control*8(338), 1985.

An Architecture for Adaptive Intelligent Systems

Barbara Hayes-Roth

Knowledge Systems Laboratory
Computer Science Department
Stanford University

Abstract

Adaptive intelligent systems perform multiple concurrent tasks requiring knowledge-based reasoning and interaction with dynamic entities in real time. Because opportunities to perceive, reason, and act typically exceed its computational resources, an agent must determine which operations to perform and when to perform them so as to achieve its most important objectives in a timely manner. Therefore, we view real-time performance as a problem in intelligent control. We propose control requirements and present an architecture to address them. We are evaluating the architecture in several experimental applications, one of which--the Guardian system for intensive care monitoring--we describe here.

1. Adaptive Intelligent Systems

Adaptive intelligent systems ("agents") perform multiple concurrent tasks requiring both knowledge-based reasoning and interaction with dynamic entities in real time. Tasks requiring such agents occur in diverse domains, such as: power plant monitoring [Touchton, 1988], process control [d'Ambrosio, et al., 1987, Pardee, et al., 1990], experiment monitoring [O'Neill and Mullarkey, 1989], student tutoring [Murray, 1989], aircraft pilot advising [Washington and Hayes-Roth, 1989], and medical monitoring [Fagan, 1980, Hayes-Roth, et al., 1989a]. To perform such tasks, an agent must possess capabilities for: *perception*--acquiring and interpreting sensed data to obtain knowledge of external entities; *cognition*--knowledge-based reasoning to assess situations, solve problems, and determine actions; and *action*--actuating effectors to execute intended actions and influence external entities. Because external entities have their own temporal dynamics, interacting with them imposes a periodic hard and soft real-time constraints on the agent's behavior.

In a complex environment, an agent's opportunities for perception, action, and cognition often exceed its computational resources. While faster hardware or software optimization may solve this problem for selected applications, they will not solve the general problem of limited resources or obviate its concomitant resource-allocation task [Smith and Broadwell, 1988]. For an agent of any speed, we can define tasks whose computational requirements exceed its resources. Moreover, we seek more from an intelligent agent than satisfactory performance of a predetermined task for which it has been optimized. Rather, we seek satisfactory performance of a range of tasks varying in required functionality, available knowledge, and real-time constraints. We seek adaptation to unanticipated conditions and requirements. Other things being equal, the broader the range of tasks an agent can handle and the wider the range of circumstances to which it can adapt, the more intelligent it is. Thus, we view real-time performance as a problem in intelligent control. An agent must use knowledge of its goals, constraints, resources, and environment to determine which potential

This research was supported by DARPA contract N00039-83-C-0136, with supplementary funding from NIH contract 5P41-RR-00785, EPRI contract RP2614-48, and AFOSR contract F49620-89-C-0103DEF, and gifts from Rockwell International, Inc. and FMC Corporation, Inc. Thanks to Ed Feigenbaum for Sponsoring the work at the Knowledge Systems Laboratory. This paper incorporates information from earlier papers [Collinot and Hayes-Roth, 1990, Hayes-Roth, 1990, Hayes-Roth, et al., 1989a].

operations to perform at each opportunity. When the operations required to achieve goals exceed available resources, the agent may have to modify goals as well. Because it is situated in a dynamic environment and faces a continuing stream of events, an agent must make a continuing series of control decisions so as to meet demands and exploit opportunities as they occur. In general, an agent should use intelligent control to produce the best results it can under real-time and other resource (e.g., information, knowledge) constraints.

Our conception of real-time performance in intelligent agents differs from other views. We do not view real-time performance as a provable, guaranteed, universal property of the agent. Nor do we seek real-time performance through engineering of the agent for narrowly specified task environments. We feel that these constructs are premature and probably unrealistic for the versatile and highly adaptive agents we envision. Rather, we view real-time performance as one of an agent's several objectives, which it will achieve to a greater or lesser degree as the result of interactions between the environment it encounters, the resources available to it, and the decisions it makes. In many cases, the agent will produce timely results for a task only at the expense of quality of result or by compromising the quality or timeliness of its performance of other tasks. As the agent's competence expands, so will its need to make such compromises.

2. Requirements for Real-Time Control in Intelligent Agents

Following [Rosenschein, 1989], we model an intelligent agent as a dynamic embedded system, modeled as a time series of states,with instants of time mapped to a state space of variable values. A change in the value of a variable is an *event*, *e*. The system's behavior is described with *measurements* defined as functions on state values. Because the system is dynamic, we describe properties of both individual states and time series of states. *Descriptive measurements* represent objective properties, for example the *importance* of an event *e1* or the *latency* of event *e2* following the occurrence of *e1*. *Utility measurements* represent valuational properties, for example the satisfaction of particular constraints on the latency of *e2*. We partition the overall system into components representing the intelligent agent, *I*, and the environment, *E*. Each component has its own dynamic state, which varies as a function of information passed among its internal components, as well as information received from the other component. We further partition the agent, *I*, into components for perception, *P*, cognition, *C*, and action, *A*, which similarly manifest events generated internally or by other components. To describe interactions between components, we refer to pairs of *trigger* and *response* events, where both events occur in one component but presumably are mediated by interaction with another component. For example, a trigger-response pair in *E* may be mediated by events in *I*. In some cases, we refer simply to a mediated event, for example an *I*-mediated event in *E*.

In the terms of our framework, intrinsic characteristics of an agent's environment may be defined as measurements on events in *E*, while characteristics of the relationship between an agent and its environment may be defined as measurements on events in *E* and *I*. For example: *Data Glut*. The agent cannot process all potentially interesting events in the environment. The average rate of events in *E* very much exceeds the maximum rate of *E*-mediated events in *I*. *Data Distribution*. Important environmental conditions may correspond to configurations of events on different state variables and over variable time intervals. This can be described as particular kinds of many-to-one mappings of events in *E* to events in *I*. *Diversity of Events*. Environmental conditions vary in importance. This can be expressed as the variability of values on an "importance" attribute of events in *E*. *Real-Time Constraints*. The values of events vary, in part, as a function of when they occur. This can be expressed in terms of utility measurements that incorporate the absolute or relative times of occurrence of events in *E*. *Multiplicity of Conditions*. We cannot enumerate all interesting conditions the agent will encounter, the set of *E*-mediated events in *I* that produce criterial values on some measurement. *Predictability*. The environment permits probabilistic prediction of some future events. This can be expressed as descriptive measurements on particular patterns of events in *E*. *Potential Interactions*. Globally coordinated courses of action are sometimes superior to sequences of locally determined actions. This can be expressed as utility measurements on particular patterns of *I*-mediated events in *E*. *Underlying Model*. Some knowledge of the environment is available, expressed as descriptive measurements on the correspondence between patterns of state values

or events in *E* and *I*. *Diverse Demands*. Multiple interacting demands for interaction with the environment include: interpretation, diagnosis, prediction, reaction, planning, and explanation. These can be expressed as utility measurements on particular types of *I*-mediated events in *E*. *Variable Stress*. The environment varies in stress over time. This can be operationalized as descriptive measures involving particular environmental variables, for example, the rate of important events or the number and types of different demands for interaction.

We define the primary objective of an agent very generally: *To maintain the utility of its behavior within an acceptable range over time.* For a given agent in a given environment, we could formalize this requirement as utility measurement on *I*-mediated events in *E* and on events in *I* to constrain resource management. Although we could use this measurement to evaluate the agent's behavior in the given context, it would provide little guidance toward the design of effective agents. We need a more specific set of requirements to constrain the space of possible agent architectures. For example: *Communications*. For *I* to interact with *E*, there must be communications involving *I*'s components, with information passing from *E* to *P*, from *P* to *C*, from *C* to *A*, and from *A* to *E*. *Asynchrony*. Given data glut and real-time constraints, the agent must function asynchronously with the environment. Event rates in *I* and *E* and in in *P*, *C*, and *A* must be independent. *Selectivity*. Given data glut and event diversity in *E*, the agent must determine whether and how to perceive, reason about, and act upon different events. Other things being equal, the conditional probability of an *I*-mediated response in *E*, given its trigger event, should be an increasing function of the trigger event's importance. The same holds for events in *P, C*, and *A*. *Recency*. An agent's sensory information is perishable, the utility of its reasoning degrades with time, and the efficacy of its actions depends upon synchronization with fleeting external events. Therefore, recency is one important selectivity criterion. This can be expressed as a sharply decreasing conditional probability of an *I*-mediated response event in *E*, given its trigger event, over time. The same holds for events in *P*, *C*, and *A*. *Coherence*. The agent should produce a globally coordinated course of action when that is preferable to locally determined actions. We impose utility measurements on certain patterns of *I*-mediated response events in *E*, as well as on mediated response events in *P*, *C*, and *A*. Other things being equal, we require a low conditional probability of mediated response events, given associated trigger events, when those response events would not fit an ongoing pattern. *Flexibility*. Conversely, the agent must react to important unexpected events in a dynamic environment. Other things being equal, we require a high conditional probability for an *I*-mediated response event in *E*, even if it does not fit an ongoing pattern, given a very important trigger event. The same holds for anomalous response events in *P*, *C*, and *A*. *Responsiveness*. Other things being equal, the more urgent a situation is, the more quickly the agent should perceive, reason, and act. That is, the latency of an *I*-mediated response event in *E*, following its trigger event, should decrease as the urgency of the trigger event increases. Similar constraints apply to response events in *P*, *C*, and *A*. *Timeliness*. Given its dynamic environment, the agent must meet various hard and soft real-time constraints on the utility of its behavior. These may be expressed as utility measurements involving latencies within *I*-mediated pairs of trigger and response events in *E*. Similar measurements could be applied to events in *P, C*, and *A*. *Robustness*. An agent must adapt to resource-stressing situations by gracefully degrading the utility of its behavior. As environmental stress increases (e.g., as event rates increase or maximum trigger-response latencies decrease), the global utility of the agent's behavior (e.g., the rate of *I*-mediated response events in *E*, weighted by importance) should decrease gradually, rather than precipitously. The same holds for interactions among *P*, *C*, and *A*. *Scalability*. In the terms of our framework, the agent's satisfaction of the requirements above (but perhaps not its absolute level of performance on any one task) should be invariant over increases in problem size. *Development*. An agent must exploit new knowledge to improve the utility of its behavior. As relevant knowledge in *I* increases, we should observe improvement in the agent's satisfaction of some of the above requirements and, therefore, in the global utility of its behavior.

3. Proposed Agent Architecture

The BB1 "dynamic control architecture" (Figure 1) [Hayes-Roth, 1985, Hayes-Roth, 1989a, Hayes-Roth, 1990] has concurrent systems for perception, action, and reasoning. A

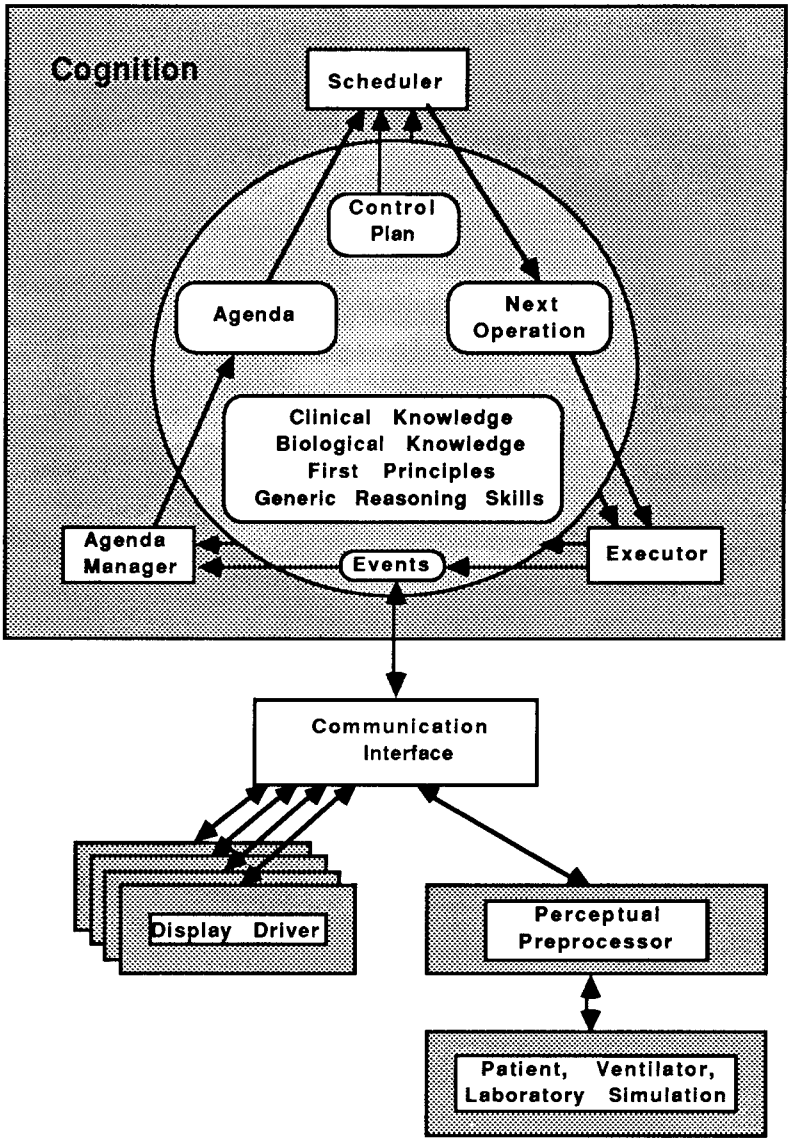


Figure 1. Proposed Agent Architecture: BB1

communications interface asynchronously relays data among their I/O buffers [Hewett and Hayes-Roth, 1989]. *Perception systems* acquire information about the environment as a basis for reasoning and action [Boureau and Hayes-Roth, 1989, Washington and Hayes-Roth, 1989]. Each *sensor* acquires signals of a certain type, transduces them into an internal representation, and holds the results in a limited-capacity buffer. The *preprocessor* retrieves these results and then abstracts, annotates, and filters them, according to dynamic instructions from the reasoning system, and places the results in its output buffer for relay to the input buffer of the reasoning system or an action system. For example, a preprocessor might abstract a sequence of numerical values of a variable into the value class "high" and the trend "rising," annotate these observations as "not relevant to ongoing tasks," but "very important" and "very urgent," and relay them to the reasoning system immediately. BB1 also adapts the global rate at which its preprocessor sends new observations, given changing reasoning activities. Thus, perception systems shield the reasoning system from data overload and maximize its vigilance within the available resources, providing the most useful information available in a compact, readily useable form. *Action systems* control the execution of actions to affect the environment

based on perception and reasoning. Each *driver* monitors its input buffer, retrieves intended actions, translates them into executable programs of effector commands, and controls their execution by sending commands to its effector at appropriate times. Each *effector* immediately executes commands in its input buffer. Thus, action systems relieve the reasoning system of the burden of managing low-level details of action execution. BB1 supports graduated reactions. Very fast *peripheral reactions* occur within a perception or action system, producing input-driven attentional shifts or feedback control of actions. Fast *reflex reactions* occur across perception-action arcs, with perceptual information directly driving action execution. Slower *cognitive reactions* involve all three kinds of systems, with cognition mediating the performance of actions in response to perceived information. Absolute response latencies at each level depend on the implementation.

In the reasoning system [Hayes-Roth, 1985, Hayes-Roth, 1990], operations occur in the context of a *global memory*, which represents *perceptual inputs, factual knowledge, reasoning knowledge,* and *reasoning results* in a conceptual graph formalism. For example, an agent might have factual knowledge of the structure and function of certain physical systems and reasoning knowledge of the operations and strategies involved in diagnosis, prediction, planning, or other tasks. Reasoning results include diagnoses, predictions, plans, etc. These are organized in an interval-based time-line representation, distinguishing phenomena that have occurred, are expected, or are intended. For example, an agent might record that "diagnosis: kinked tube occurred at 0159" explains "observation: rising PIP occurred during 0159-0200" and that "action: tube straightening occurred at 0201" will cause "observation: falling PIP expected by 0202." The global memory also contains *input buffers* for data sent by perception systems and *output buffers* for intended actions sent to action systems. Finally, it contains information regarding the agent's cognitive behavior: *cognitive events*, *agenda*, *control plan*, and *next operation*, discussed below.

BB1 performs reasoning operations that are suggested by and make changes to information in the global memory. Its *satisficing cycle* [Collinot and Hayes-Roth, 1990, Hayes-Roth, 1990] iterates three steps: (1) The *agenda manager* uses recent important *perceptual* and *cognitive events* and the current *control plan* to identify and rate a few of the most important reasoning operations, recording them on the *agenda*. (2) The *scheduler* uses the control plan to determine when to interrupt agenda management and which operation to execute, recording it as the *next operation*. (3) The *executor* executes the next operation, producing cognitive events that represent: new perceptual filters or intended actions in output buffers; new inferences or conclusions for ongoing reasoning tasks; or new control decisions that initiate, terminate, or modify strategies for reasoning tasks. By controlling resource allocation at its fundamental unit of computation, the satisficing cycle enables Guardian to guarantee real-time responses to selected events.

Dynamic control planning determines the utility--quality and timeliness--of perception, reasoning, and action [Collinot and Hayes-Roth, 1990, Hayes-Roth, 1985, Hayes-Roth, 1990]. A *control plan* is a temporally organized pattern of *control decisions*, each describing a class of operations to be performed, under specified constraints, during some time period. Control operations, which are triggered by events and, when executed, generate or modify control decisions, construct control plans incrementally. They guide the scheduling of reasoning operations. Thus, an agent can construct and follow plans, but also change its plans, given a changing environment. Control plans also modulate the speed-quality tradeoff in the satisficing cycle. They determine the order in which reasoning operations are identified and the associated interrupt conditions, so that an agent can execute a "good enough" operation as soon as possible or the "best available" operation when a deadline occurs. Finally, control plans focus the attention of perception systems. For example, given a plan to diagnose a particular problem, the agent would increase its attention to relevant data. Given a plan to perform computationally demanding reasoning tasks, it would lower its global input data rate by adjusting rates for different variables according to their relevance, importance, and urgency.

4. Satisfaction of Real-Time Control Requirements

The proposed agent architecture is addresses the requirements above: *Communications.* Information passes from the environment to perception subsystems, from perception subsystems to cognition and action subsystems, and from the cognition subsystem to perception

and action subsystems. *Asynchrony.* Parallel subsystems, with buffered communications, provide asynchronous perception, cognition, action. *Selectivity.* Limited-capacity event buffers selectively favor "high priority" inputs--those that are recent, relevant, important, and urgent. Perception/action subsystems selectively process high priority sensed data and intended actions. The agenda manager selectively triggers and schedules high priority operations. Dynamic control plans selectively favor high priority reasoning tasks and establish associated focus of attention parameters. *Recency.* Limited-capacity buffers with best-first retrieval and worst-first overflow favor recent items, as does the heuristic best-first agenda manager. *Coherence.* Dynamic control plans provide a global focus of attention to coordinate perception, cognition, and action over time. They also strategically organize reasoning operations within a task and among concurrent reasoning tasks. *Flexibility.* Exceptional events can override global focus of attention in perceptual preprocessors or the cognitive system. *Responsivity.* Graduated reactive responses--peripheral, reflex, and cognitive responses--span a range of latencies. Within cognitive responses, additional gradations are supported. The agenda manager can control cycle time. Dynamic control planning can establish deadlines and discriminate among alternative reasoning methods strategies. *Timeliness.* Satisfying each of the requirements discussed above contributes to an agent's timely response to the most important events. In addition, dynamic control planning allows an agent to reason explicitly about the time requirements of alternative operations and the time constraints on its behavior. *Robustness.* Satisfying many of the requirements discussed above entails gracefully trading amount of computation, and therefore, expected quality of response, against latency of response. *Scalability.* Several aspects of the architecture are designed to accommodate changes in scale. For example, perceptual preprocessing and focus of attention will protect the agent against increasing perceptual overload. Given a discriminating control plan, the satisficing cycle will produce stable cycle times regardless of increases in problem size. *Development.* Increases or improvements in knowledge should improve the agent's ability to meet several of these requirements. For example, improvements in its control knowledge should enable it to focus perceptual attention more effectively, improve the strategic control of its reasoning, and execute higher-priority operations more rapidly.

5. The Guardian Application

Given our research goal to develop a general architecture for intelligent agents, experimental development of agents that operate in diverse domains is a major part of our research. Each new domain tests the sufficiency and generality of the current architecture and presents new requirements for subsequent versions. We are now studying monitoring agents for semiconductor fabrication [Murdock and Hayes-Roth, 1990], power plant maintenance, and medical monitoring [Hayes-Roth, 1989a]. To illustrate how agents are implemented within the proposed architecture, consider Guardian.

Guardian monitors simulated intensive-care patients who have temporary failure of one or more organ systems, which is treated with life-support devices that assume the functions of the ailing system until it heals. For example, the ventilator is an artificial breathing machine that augments the patient's own breathing. Life-support devices are adjusted based upon frequent patient observations. Some observations are made continually and automatically, for example, measurements of air pressures and air flows in the patient-ventilator system. Other observations are made intermittently. Blood gases, for example, are measured once every hour or so, while chest xrays are usually taken once or twice a day. Based on patient observations, device settings are adjusted to vary the amount of assistance the device provides. For example, ventilator settings determine the number of breaths delivered to the patient per minute, the volume of air blown into the patient's lungs on each breath, and the amount of oxygen in the air. Other therapeutic actions might include adjusting a ventilator tube, clearing the patient's air passages, administering drugs, etc. The short-term goal of SICU monitoring is to keep the patient as comfortable and healthy as possible, to diagnose and correct unexpected problems, and to refine and execute the long-term therapy plan, while progressing toward therapeutic objectives. The long-term goal is to withdraw life-support devices gradually so that the patient eventually can function autonomously.

Guardian's task instantiates the requirements for real-time control. Because it has access to many automatically sensed patient data variables and because it can reason about and act upon these

observations in many different ways, it must selectively perceive important patient data and perform key reasoning operations that contribute to its most important actions. Because the patient embodies a dynamic physical process, Guardian must asynchronously perceive patient data, reason about the patient's condition, and perform therapeutic actions. To insure that its behavior is current, it must "forget" unrealized past opportunities for perception, reasoning, and action in favor of present opportunities. To achieve longer-term goals, Guardian must enact a coherent pattern of perception, reasoning, and action over time. On the other hand, uncertain changes in the patient's condition require flexibility and adaptation. Guardian must respond to patient conditions of varying urgency; other things being equal, the more urgent the patient's condition is, the more quickly it must perceive relevant information, perform necessary reasoning, and execute appropriate actions. Guardian must satisfy a variety of hard and soft real-time constraints on the utility of its behavior. Because it encounters situations that strain or exceed its capacity--too many signs and symptoms, interpretation, diagnosis, prediction, and planning tasks, and therapeutic actions--its performance must degrade gracefully, not precipitously. Guardian must maintain the quality of its behavior as we scale up to more realistic problems and improve the utility of its behavior with more knowledge.

Guardian currently monitors a simulated patient. A single perceptual preprocessor manages its perception of twenty automatically sensed variables, with an average overall rate of one value per second. It also perceives irregularly reported lab results and messages from human users. Each one, if passed to the cognitive system, would trigger several cognitive operations, whose execution would produce several cognitive events and trigger new operations. Although this is not a high data rate in absolute terms, it is beyond Guardian's current cognitive capacity, one operation every couple of seconds. Moreover, we anticipate that Guardian's sensory activity will increase from twenty to one hundred automatically sensed variables, each sensed at least once per second. There will be at least twenty irregularly sensed data variables. Thus, Guardian faces significant and growing perceptual overload.

To avoid falling behind real time, Guardian's perceptual preprocessor applies dynamic abstraction, filtering, and annotation parameters sent by the cognitive system. It abstracts numerical data values into value classes and trends. It assigns data values to three levels of importance: life-threatening, abnormal, and other. It distinguishes data that are relevant to ongoing reasoning activities from those that are not relevant. It distinguishes three levels of urgency: events that permit an effective response within four minutes, one hour, or longer. It filters data based on criterial value changes within deadlines. Thus, the cognitive system can bound the variability of unsent intervening values. Using these mechanisms, the preprocessor typically reduces sensed data rates by over 90%, maintaining an average overall perception rate of approximately one perceptual input every twenty-two seconds, without reducing solution quality [Washington and Hayes-Roth, 1989]. Additional selectivity is provided by the cognitive system itself.

Guardian has a wide range of medical knowledge including: knowledge of meaningful classifications and trends of the currently sensed patient variables; knowledge of a twenty-node hierarchy of respiratory disease conditions, along with their likely signs and standard treatments; knowledge of the normal structure and function of the respiratory, circulatory, pulmonary exchange, tissue exchange, and tissue metabolism systems and the ventilator; knowledge of the normal and abnormal structure and function of abstract flow, diffusion, equilibrium, and metabolic systems; knowledge of prototypical therapeutic protocols for managing a small number of evolving disease conditions; knowledge of the importance and urgency of particular observations and diagnoses; knowledge of the precondition, results, and time required to perform a number of therapeutic actions.

Guardian knows how to perform several reasoning tasks: interpretation of time-varying data, diagnosis of observed signs and symptoms, selection of corrective actions for diagnosed conditions, prediction of future patient conditions, explanation of observations, diagnoses, and predictions, and dynamic therapy planning. For most of these, it has both *associative* and *model-based methods*. Associative methods use clinical knowledge and permit quick responses to familiar situations. Model-based methods use more fundamental biological and physical knowledge and permit more thorough (and time-consuming) responses to both familiar and unfamiliar cases. Each method is implemented as a set of abstract reasoning operations that are

triggered by particular kinds of perceptual or cognitive events, along with control operations that construct resource-bounded control plans in particular contexts. The results of all reasoning activities are recorded in temporally organized episodes in the global memory.

Depending on the circumstances, Guardian may be logically capable of pursuing many different reasoning tasks with both associative and model-based methods. Given the real-time constraints on its behavior, however, Guardian typically must be quite selective about which tasks it pursues and how it allocates reasoning resources among them. Accordingly, it uses strategic knowledge to construct a dynamic global control plan that differentially favors the triggering and scheduling of executable operations involved in competing reasoning tasks.

For example, in one scenario, Guardian observes that a post-operative patient has low body temperature. It makes a global control decision to perform a sequence of reasoning tasks: diagnose the low temperature as a normal result of post-operative status; predict a spontaneous rise in temperature to normal over a period of hours; infer undesirable consequences of low temperature, low arterial CO2 rising to normal with temperature; and plan and execute a sequence of changes to breathing rate coordinated with body temperature to maintain normal arterial CO2. For each task, Guardian makes local control decisions about whether to apply associative or model-based reasoning methods and how to organize its reasoning. At the same time, its global control plan allows it to incorporate new perceptions, but not to reason about most of them as they are less important than ongoing activities. However, Guardian does respond to a request for explanation of the relationship among temperature, breathing rate, and arterial CO2 in terms of the underlying anatomy and physiology. And it subsequently interrupts all of these tasks--diagnosis, prediction, planning, and explanation--when it observes very high peak inspiratory pressure, a life-threatening condition with a four-minute deadline. Guardian makes a new global control decision to direct all of its resources to correcting this condition as quickly as possible. As a result, its perceptual preprocessor refocuses to favor data relevant to the high peak pressure. Its agenda manager adopts a shorter deadline to insure quick reasoning. And its satisficing cycle favors associate reasoning operations that quickly diagnose and correct the high peak pressure. Given these adaptations, Guardian very quickly performs a sequence of operations: diagnose the immediate problem, inadequate ventilation; increase the breathing rate so the patient will get enough oxygen and the deadline will be extended; diagnose the underlying problem, pneumothorax (hole in the lung); perform the appropriate action, insert a chest tube to release accumulated air in the chest cavity; reduce the breathing rate now that the pressure is relieved; confirm that the pressure is normal; and confirm that the blood gases are normal. Once the problem is solved, Guardian makes a new global control decision to resume its interrupted activities, adapting them as necessary to results of intervening events.

Several display drivers manage Guardian's dynamic graphical displays of: patient history; reasoning and results of diagnosis, prediction, planning, and explanation; and control reasoning.

6. Evaluation of the Proposed Architecture

Guardian's evolution through four demonstration systems shows its expanding competence. In Demonstration 1, Guardian had factual knowledge for only the respiratory system, the ventilator, an abstract flow system, and two types of flow system faults, blockage and leakage. It had reasoning knowledge for associative and model-based diagnosis and model-based explanation. It monitored eight patient data variables and diagnosed and explained kinked tube and one-sided intubation problems. As described above, by Demonstration 4, Guardian had substantially more factual and reasoning knowledge and handled a much more complex scenario. In fact, each scenario represents a class of scenarios that Guardian can handle, the breadth being determined by the extent of Guardian's medical knowledge relevant to the capabilities being demonstrated. For example, in Demonstration 4, one capability is to respond under severe time constraints, in the context of important ongoing activities, to an unanticipated critical problem. We demonstrate that by Guardian's response to a pneumothorax when it already is performing several important prediction, planning, and explanation tasks. However, Guardian actually can respond effectively to the unanticipated occurrence of any critical condition currently in its clinical knowledge, in the context of any combination of ongoing prediction, planning, and explanation activities involving its current knowledge. Demonstrating Guardian over successive

scenarios gives evidence of the essential correctness, extensibility, and scalability of its underlying approach. Demonstrations 1, 2, and 3 entailed significant changes to Guardian's architecture and knowledge representation. Demonstration 4 required additions only to its reasoning skills and factual knowledge. Although we expect to make continuing improvements at all levels of Guardian, most future improvements will be at higher knowledge levels.

We evaluated the real-time interactions of Guardian's perception, action, and reasoning systems [Hewett and Hayes-Roth, 1989]. Our experiment shows constant communication latencies among the systems over a range of activity within each system and vice versa. Absolute latencies are determined by processor speed, network speed, and program optimization. Thus, within the ranges tested, Guardian achieves true concurrency and asynchrony of perception, action, and reasoning. We also evaluated the real-time performance of Guardian's perception system [Boureau and Hayes-Roth, 1989, Washington and Hayes-Roth, 1989]. One experiment showed that adaptation of filtering thresholds, based on dynamic load and focus of attention in the reasoning system, reduced input data by 94% on average, with no reduction in the quality of the reasoning results. A second experiment showed that knowledge-based prioritizing of input data allows Guardian to meet deadlines for critical observations over a range of data rates, compared to a control condition in which critical observations are lost under high data rates. Thus, Guardian's perception system effectively shields Guardian from input variability and overload, while maintaining responsiveness to critical data.

We evaluated Guardian's reasoning [Collinot and Hayes-Roth, 1990] about four *key events* in Demonstration 4: observed low temperature requires prediction of future temperature changes and causal implications; inferred low PaCO2 requires planning of rate changes to keep PaCO2 in normal range; the user's request requires explanation of how low temperature and normal breathing rate cause low PaCO2; and observed high PIP requires diagnosis of the underlying pneumothorax and insertion of a chest tube. The high PIP event is highly critical because it is potentially life-threatening. The other events are moderately critical. We made *component measurements* of the *correctness*, *specificity*, *timeliness*, and *selectivity* of Guardian's response to each event. We measured the *global utility* of Guardian's performance by integrating component measurements according to two rules. Under rule 1, global utility is the sum of products of event criticality and response quality (correctness, specificity, and timeliness) for the four events. Rule 2 introduces a condition: If Guardian responds correctly, specifically, and within deadline to highly critical events, then global utility is as defined in rule 1; otherwise global utility is 0. The results show that Guardian responded correctly, specifically, and within deadline to all key events and responded more selectively and more quickly to the high PIP than to less critical events. As a result, the global utility of its performance over the complete scenario was quite high by both integration rules. By contrast, a comparison system having a "less intelligent" architecture, but the same factual knowledge and reasoning skills, failed to respond correctly, specifically, or within deadline to some events--in particular the highly critical event, high PIP. As a result, its global utility was lower by both integration rules, especially the second rule where its failure to respond to the high PIP gave a global utility of 0.

We made the above measurements while systematically manipulating input data rates and knowledge base size [Collinot and Hayes-Roth, 1990] . Results show that Guardian maintains high values of all component measurements and high global utility despite increases in data rates or knowledge. By contrast, the comparison system frequently failed to respond correctly, specifically, or within deadline to key events, especially the high PIP, and therefore produced lower global utility as data rates or knowledge increased. These results indicate Guardian's robustness over increasingly complex monitoring situations and scalability over increasing amounts of knowledge.

7. Limitations of the Proposed Architecture

We must acknowledge that the proposed architecture makes agents vulnerable to errors that do not occur under some other architectures. By definition, the architecture's real-time control mechanisms--its perceptual filtering, limited capacity I/O buffers, dynamic control planning, focus of attention, and satisficing cycle--allow an agent to ignore many opportunities to perceive, reason, and act and to perform sub-optimal operations. In general, the agent allocates limited computational resources among competing activities in proportion to their

urgency and importance. In many cases, this will not affect the global utility of the agent's performance. In others, it will produce acceptable degration in particular aspects of performance. In extreme cases, however, an agent might decide prematurely to perform costly, ineffective, or counterproductive operations; or it could fail to perform highly desirable operations that are well within its capabilities. Nonetheless, we hypothesize that, if we wish to build agents that function well in complex real-time environments, we must forego optimality in favor of effective management of complexity [Simon, 1969]. Allowing the possibility of occasional, more or less consequential error is a necessary concession toward that end. Formulating control knowledge that allows an agent to meet important real-time performance requirements while minimizing the impact of incompleteness and suboptimality is a primary objective of our research.

8. References

[Ash, et al., 1990] Ash, D., Vina, A., Seiver, A., and Hayes-Roth, B. Action-oriented diagnosis under real-time constraints. *Procs. of the International Workshop on Principles of Diagnosis*, 1990.

[Boureau and Hayes-Roth, 1989] Boureau, L., and Hayes-Roth, B. Deriving priorities and deadlines in real-time knowledge-based systems. *Procs. of the IJCAI89 Workshop on Real-Time Systems*, 1989.

[Collinot and Hayes-Roth, 1990] Collinot, A., and Hayes-Roth, B. Real-time control of reasoning: Experiments with two control models. Stanford Unversity: KSL Report 90-17, 1990.

[d'Ambrosio, et al., 1987] d'Ambrosio, B., Fehling, M.R., Forrest, S., Raulefs, P., and Wilbur, M. Real-time process management for materials composition in chemical manufacturing. IEEE Expert, 1987.

[Fagan, 1980] Fagan, L.M. VM: Representing time-dependent relations in a medical setting. PhD Dissertation, Stanford University, 1980.

[Goto and Stentz, 1989] Goto, Y., and Stentz, A. Mobile robot navigation: The CMU system. IEEE Expert, Volume 2, 4, 44-54, 1989.

[Hayes-Roth, et al.,1979] Hayes-Roth, B., Hayes-Roth,, F., Rosenschein, S., and Cammarata, S. Modelling planning as an incremental, opportunistic process. *Procs. of the IJCAI*, 6:375-383, 1979.

[Hayes-Roth, 1985] Hayes-Roth, B. A blackboard architecture for control. *Artificial Intelligence*, 26:251-321, 1985.

[Hayes-Roth, 1990] Hayes-Roth, B. Architectural foundations for intelligent agents. *Real-Time Systems: The International Journal of Time Critical Systems*, 2, 99-125, 1990.

[Hayes-Roth, et al., 1989a] Hayes-Roth, B., Washington, R., Hewett, R., Hewett, M., and Seiver, A., Intelligent real-time monitoring and control. *Procs. of the IJCAI*, 1989.

[Hayes-Roth, 1989a] Hayes-Roth, B. A multi-processor interrupt-driven architecture for adaptive intelligent systems. *Procs. of the IJCAI89 Workshop on Real-Time Systems*, 1989.

[Hayes-Roth, et al., 1989b] Hayes-Roth, B., Hewett, M., Washington, R., Hewett, R., and Seiver, A. Distributing intelligence within a single individual. In L. Gasser and M.N. Huhns (Eds.) Distributed Artificial Intelligence, Volume 2. Morgan Kaufmann, 1989.

[Hayes-Roth, 1989b] Hayes-Roth, B. Making intelligent systems adaptive. In K. VanLehn (Ed.), Architectures for Intelligence. Lawrence Erlbaum, 1989.

[Hayes-Roth, 1987] Hayes-Roth, B. Dynamic control planning in adaptive intelligent systems. *Procs. of the DARPA Knowledge-Based Planning Workshop*, Austin, Texas, 1987

[Hewett and Hayes-Roth, 1989] Hewett, M., and Hayes-Roth, B. Real-Time I/O in Knowledge-Based Systems. In V. Jagannathan, R.T. Dodhiawala, and L. Baum (Eds.), Current Trends in Blackboard Systems. Morgan Kaufmann, 1989.

[Hewett and Hayes-Roth, 1990] Hewett, R., and Hayes-Roth, B. Representing and reasoning

about physical systems using generic models. In J. Sowa, S. Shapiro, and R. Brachman (Eds.) Formal Aspects of Semantic Networks, Morgan Kaufmann, 1990.

[Johnson and Hayes-Roth, 1987] Johnson, M.V., and Hayes-Roth, B. Integrating diverse reasoning methods in the BB1 blackboard control architecture. *Procs. of the AAAI*, 1987.

[McTamaney, 1989] McTamaney, L.S. Mobile robots: Real-time intelligent control. IEEE Expert, 2:55-70, 1989.

[Murdock and Hayes-Roth, 1990] Murdock, J., and Hayes-Roth, B. Intelligent monitoring and control of semiconductor manufacture. *Procs. of the Fifth Annual SRC/DARPA CIM-IC Workshop*, Berkeley, 1990.

[Murray, 1989] Murray, W. Dynamic instructional planning in the BB1 blackboard control architecture. In V. Jagannathan, R. Dodhiawala, and L. Baum (eds.), Current Trends in Blackboard Systems, Morgan Kaufman,1989.

[O'Neill and Mullarkey, 1989] O'Neill, D.M., and Mullarkey, P.W. A knowledge-based approach to real time signal monitoring. *Procs. of theAAAI*, 1989.

[Pardee, et al., 1990] Pardee, W. J., Schaff, M. A., and Hayes-Roth, B. Intelligent control of complex materials processes. *AI in Engineering, Design, Analysis, and Manufacturing*, 4:55-65, 1990.

[Rosenschein, 1989] Rosenschein, S.J., Hayes-Roth, B., and Erman, L. Notes on methodologies for evaluating IRTPS systems. Procs. of the AFOSR Workshop on Intelligent Real Time Problem Solving Systems. Santa Cruz, 1989.

[Smith and Broadwell, 1988] Smith, D.M., and Broadwell, M.M. The pilot's associate - An overview. *Procs. of the Eighth International Workshop on Expert Systems and their Applications,* 1988.

[Sowa, 1984] Sowa, J. Conceptual Structures: Information Processing in Mind and Machine. Addition-Wesley, 1984.

[Stankovic,1988] Stankovic, J.A., Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, 21:10-19, 1988.

[Simon, 1969] Simon, H.A. The Sciences of the Artificial. MIT Press, 1969.

[Touchton, 1988] Touchton, R.A. Reactor emergency action level monitor. Technical Report NP-5719, Electric Power Research Institute, 1988.

[Washington and Hayes-Roth, 1989] Washington, R., and Hayes-Roth, B. Managing input data in real-time AI systems. *Procs. of the IJCAI*, 1989.

[Washington and Hayes-Roth, 1990] Washington, R., and Hayes-Roth, B. Abstraction planning in real time. *Procs. of the AAAI Symposium on Planning in Dynamic Uncertain Environments*, Palo Alto, 1990.

Specifying Complex Behavior for Computer Agents

Leslie Pack Kaelbling*
Teleos Research
576 Middlefield Road
Palo Alto, CA 94301

1 Introduction

Consider the problem of programming computer-controlled agents to behave in complex environments. These agents might be robot arms that assemble cars, household assistants that do the laundry and take out the trash, or database agents that schedule appointments and keep computer files up to date. Such agents must interact with a world that is dynamic and is predictable in some respects but highly unpredictable in others.

In recent years, a wide range of formalisms have been developed for specifying behaviors for computer agents, including the paradigms of the general notions of "classical planning" and "reactive behavior." These formalisms represent points in a complexity space that has as two of its most important dimensions

- ease of expression of complex action strategies by the human programmer
- efficiency of execution of formal behavioral specification by the agent

There is no single formalism that is most appropriate for all problems of agent behavior specification. By studying the properties of various behavior-specification formalisms and of the settings of particular problems, we can choose formalisms appropriately.

This paper will focus on three different methods for specifying behaviors for agents: direct programming, operator descriptions, and goal reduction rules. These will serve as example formalisms that will allow us to discuss ease of human programming, ease of automatic execution, and the value of compilation.

2 Framework

In order to make the following discussion precise, we must assume a concrete model of the agent's interaction with its environment. This discussion will be based on a model of computation in which an agent is seen to perform a finite transduction from a stream of input data into a stream of output data (this model also forms the foundation of *situated-automata theory* [Rosenschein, 1985, Rosenschein and Kaelbling, 1986]). The agent receives an input from the environment, updates its internal state as a function of the input and the state value, then outputs that action, effecting the world. This cycle happens at regular intervals that are timed in a way that allows the agent to keep pace with the important events in its environment (this pace may vary from 100 cycles per second in an automatic-pilot system to 1 cycle per day in a system that does inventory management in a store).

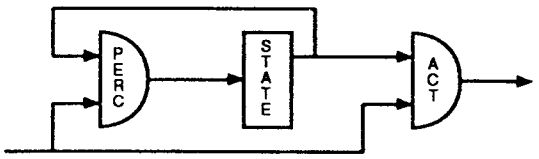


Figure 1: Framework for embedded computation, divided into perception and action functions.

The job of an agent designer, then, is to specify the state-update and output functions, which make up the agent's program. We shall refer to them as the *perception* and *action* functions, as shown in Figure 1. We require the computation time of these functions to have a finite upper bound. This bound will guarantee that the agent can react with appropriate speed to external events by having a bounded delay between the arrival of any given input and the generation of an output that depends on that input. This paper is primarily concerned with the specification of the action function.

Another popular computational model for embedded agents is one of many concurrent processes. Typically, one process runs with a guaranteed fixed cycle time, and its outputs can be influenced by the results of other processes as they are completed. This is a useful model, especially appropriate for machines with coarse-grained processor parallelism, but it makes the semantic analysis of the computation performed by the agent quite difficult. The exact meaning of the result

*This work was supported in part by the Air Force Office of Scientific Research under contract F49620-89-C-0055DEF, in part by the National Aeronautics and Space Administration under Cooperative Agreement NCC-2-494 through Stanford University subcontract PR-6359, and by the Defense Advanced Research Projects Agency through NASA contract NAS2-13229.

of any computation depends crucially on how much time has passed since the inputs to the computation were sensed by the agent; this is more difficult to measure and keep track of in a concurrent-process model than in the simple circuit model of Figure 1.

One important thing to note is that, in this framework, "perceptual actions" that are performed to gain information are not distinguished from actions in general. One reason for treating all actions uniformly is that perceptual actions may be externally indistinguishable from other actions and use the same resources: a robot may put its hand on a table to steady the table or to find out if it is clear. We must, then, consider all actions together in attempting to determine which one is most appropriate to execute.

This paper considers different ways in which programmers can specify the mapping from an agent's perceptual state (current values of the internal state and input vectors) into an action; we shall refer to this mapping as the *action map*.

3 Specifying Action Maps

There is a wide variety of formalisms that may be used by a human programmer to specify the action map for an agent. It is widely held to be easier for humans to program in formalisms that allow a modular, declarative expression of the program, rather than a direct procedural account. This point is discussed at length by Winograd [Winograd, 1985] in connection with the general knowledge-representation problem. This section will address the use of three different types of formalisms for specifying actions maps, leaving issues of executability for discussion in the following section.

3.1 Direct programming

The most traditional method of supplying the action map for an agent is to use the standard methods of computer programming. Using a functional or procedural programming language, the programmer can specify the function that should be computed to generate each new action.

In simple domains, especially those to which the methods of control theory can be applied, this approach is quite adequate. In many domains, for example, there is a simple numerical functional relationship between output values and input values, which can be easily specified in a traditional programming language.

Another situation for which this method is appropriate is when the programmer has complete information about the initial state of the world and about the effects of the agent's actions on the world. In this case, the agent's program can typically be written as a list of actions, which the agent executes one-by-one, ignoring the input values from the world. Domains that are this benevolent and understandable are rare, but the approach has been used successfully for "sequencing" unmanned space missions and for programming highly constrained robotic assembly tasks.

In most other cases, the actions that an agent should take are highly conditional on the perceptual state, requiring a large and complex computer program. Of course, any action map can be specified this way, but direct programming can become very tedious and difficult for the programmer.

3.2 Classical Operator Descriptions

The standard artificial intelligence (AI) technique for specifying an action map is to give a description of the abilities of the agent, a description of a desired goal state of the world, and a description of the initial state of the world. From this information and the assumption that the agent should act in such a way as to cause the world to satisfy the goal-state description, it is possible to derive the next action that should be taken by the agent by finding a string of actions that, if executed starting in any state satisfying the initial state description, will cause the world to be in some state satisfying the goal state description. The agent's abilities are typically described using an operator description language. In this language, each possible action of the agent is characterized by a set of preconditions and a set of postconditions. If the preconditions are true in the world and the agent performs the action, then the postconditions will be true in the world. The descriptions of the initial and goal states of the world do necessarily correspond to completely individuated perceptual states. In general, they can be arbitrarily general or specific. If the initial state description is *true*, then the agent must act without assuming anything about the initial state of the world. This process is typically referred to as "planning" and has a large related literature [Allen *et al.*, 1990].

Using operator descriptions to specify action maps is very appealing. It allows the programmer to write a declarative specification in terms of facts about the world and the abilities of the agent; this makes the task less like regular programming and (theoretically) easier for non-professionals. Another benefit is that, once the agent's abilities have been described in the operator description language, generating a new action map amounts to specifying new initial state and goal state descriptions. Finally, this finite description of the operators and the initial and goal states may engender behaviors of arbitrary complexity; there is no bound on the number of actions that can be strung together to achieve the goal.

This approach has a number of drawbacks, as well.

First, the semantics of the operator descriptions can rarely be satisfied in the real world. The effects of low-level operations, such as sending a voltage to a wheel in a mobile robot, cannot be modeled reliably at a level of abstraction for which planning is appropriate. Higher level actions that have non-deterministic results might be usefully modeled with probabilistic operator descriptions.

In addition, operator-description languages are typically oriented toward single goals of achievement, but it is often useful to supply a goal of maintenance like "don't spill the milk." Goals of maintenance could be added to such a framework by adding a third component to operator descriptions describing which conditions are maintained. Although this extension is theo-

retically possible, it gives rise to an explicit version of the frame problem [Hayes, 1990], in which a possibly infinite number of maintained conditions would have to be specified for each operation.

3.3 Goal Reduction Rules

There are many formalisms that lie between direct programming and operator descriptions on the procedural-declarative spectrum. One is goal reduction rules. Gapps [Kaelbling, 1988] is a declarative language in which the programmer writes a set of instantaneous goal-reduction rules. These rules, together with a top-level goal description, specify an action map for the agent. The goal-reduction rules specify how to take the top-level goal and, depending on the current state of the world, reduce it to another top-level goal. Eventually the reduction process bottoms out in an action that is correct to execute given the current state of the world.

The use of goal-reduction rules moves some of the burden of "plan synthesis" from the agent to the programmer, but allows easy expression of many kinds of action strategies that are difficult to encode using operator descriptions. Take, as an example, the strategy of hammering in a nail until it is flush with a board. It might be possible to describe an operator *hit-the-nail* that has as its post-condition that the nail is some fraction of an inch farther into the board than it was, or one called *probably-hit-the-nail* that 10% of the time has as its postcondition that the nail is flush with the board. Both of these uses of operator descriptions require fairly sophisticated plan synthesis methods. The simple goal-reduction rule captures the commonsense knowledge that if your goal is to have the nail be flush with the board and the nail is not yet flush with the board, you should hit the nail.

```
(defgoalr (ach nail-flush-with-board)
    (if (nail-not-flush-with-board)
        (do hit-the-nail)
        (do anything)))
```

Rather than saying exactly what the effects of an operation are, the user specifies under which environmental conditions an action is appropriately performed. In the goal-reduction approach, the initial condition of the world need not be specified; instead, the world is monitored as the agent interacts with it, and each action is selected on the basis of the currently perceived state of the world rather than on its predicted state. This makes it easy to specify action mappings for domains in which the effects of individual actions are quite unreliable.

One drawback of goal-reduction rules in comparison with operator descriptions is that the programmer must provide a reduction rule for any primitive goal that might occur. This is in contrast to the operator-description approach, in which any formula in the formal language used to specify the domain could, potentially be used as a goal.

4 Executing Action Maps

Once the programmer has specified an action map, it must be executed by the agent. The degree of difficulty of this execution depends on the nature of the language used to specify the map. It can vary from trivial to nearly impossible. This section considers the computational aspects of executing specifications written in each of the specification languages discussed above.

4.1 Direct programming

Languages used for direct programming are designed to be compiled and executed directly by the agent's computer. In using such languages, it is incumbent upon the programmer to guarantee that the computation time for the state-update and action functions is bounded. This problem can be avoided by using a language, such as Rex [Kaelbling, 1987b] or a more standard real-time programming language and operating system, that guarantees response time.

4.2 Operator Descriptions

Operator description languages are not directly executable by an agent. The standard execution model is to search for a sequence of actions that will take the agent from any state satisfying the initial state description to some state satisfying the goal state description. Having found this sequence, the agent should take the first action. In order to avoid repeating this work, this process is often divided into two phases: planning and execution. In the planning phase, the search is done and the chain of actions stored. In the execution phase, the actions are simply emitted in sequence with no regard to the state of the world. More sophisticated systems of this type perform "execution monitoring" in which the planning phase records the expected state of the world between the actions. The execution phase then monitors the execution of the plan, making sure that the world satisfies the descriptions of the expected intermediate states. If it does not, the system reverts to the planning phase with a description of the current state of the world as the initial state.

Chapman has shown that the planning phase in such a system is, in the general case, undecidable [Chapman, 1987]; for very restricted operator-description languages, it is merely intractable, with the time it takes to find a plan increasing exponentially in the number of operators. Because the agent must complete the planning phase before it takes its first action, this sort of execution of operator descriptions does not satisfy the requirements of having a constant bound on the reaction time of the agent. Additionally, an advantage that we cited of operator-description languages, that a small description could generate an arbitrarily long program, is a detriment for execution, because arbitrarily long programs take exponentially more time to generate.

Also, if it takes too long to perform the planning phase, the information upon which it was based, especially the initial state description, may change, invalidating the entire plan. A good execution monitoring system might notice this before any wrong action was

taken and cause the planning phase to be re-entered, but this kind of behavior makes the reaction time even worse.

Despite the apparent intractability of executing operator descriptions, there are at least two ways to limit the total expressiveness of the language and make the execution compatible with a requirement for guaranteed reaction time.

4.2.1 Using Space

If we are willing to limit the scope of planning to plans of a fixed length, it is possible to do the search to that depth in an amount of time that is bounded by a constant. This can be thought of as expanding the search tree in parallel to a fixed depth all at once; hence, "using space." This process can be made even more efficient (but no longer strictly correct) if a beam search is used, assuming that at each level of the search all but a certain number of candidate plans can be pruned.

In this formulation of the planning problem, just the first step of the plan is executed. The next time the action function is called, the computation is repeated and, again, the first step is executed. This planning and execution style does no caching of plans and is, therefore, not in danger of diverging from the expected execution path in the world. The disadvantage is that the time-constant required to do this computation may be too large for many systems (on current hardware) to keep up with their environment.

4.2.2 Using Time

An alternative to computing a fixed-length plan on every cycle is to express the planning process as an incremental computation, which is carried out over the course of many calls to the action function. This method of "using time" requires that state be used in the computation of the action. On each call to the action function, the planning process generates an output, but it may be one that means "I don't have an answer yet." After some number of cycles (depending on the size of the planning problem) the planner will generate a real result. This result might be cached and executed as in a traditional system, or the agent might simply take the first action and wait for the planner to generate a new plan.

One advantage of organizing the computation this way is that it allows the programmer to specify a hierarchical action map. It may be that the best actions for the agent to take are those specified in the operator description language (because it is easiest for the human programmer to come to grips with the complexity of the domain in this language), but that certain instantaneous reflexive reactions can be specified in a more direct way. The agent can then execute the planner and the reflex program in parallel, performing the action suggested by the planner when there is one, and otherwise heeding its reflexes. This need not happen on just two levels; the general organization of such a system with many levels is described by Kaelbling [Kaelbling, 1987a].

We must still take care that the plan generated by the planner, given that time has passed since it began its task, is appropriate for the situation in which it is finished. This can be guaranteed if the planner monitors the conditions in the world upon which the correctness of its plan depends. If any of these conditions goes false, the planner can begin again. This is correct behavior, with the planner continuously emitting the "I don't know" output and allowing the agent to react reflexively to its environment if necessary.

Such a planner might generate a plan in the form of a linear sequence of actions or a set of condition-action rules. It is important to be able to evaluate the validity of a plan as time passes, so that the planner may be reinvoked if the execution of the plan does not take place as expected. One useful and robust way to provide a validity test is to generate a directly executable action map for some small part of the input space that the agent is likely to find itself in as it traverses a path from the current state to a goal state. The plan becomes invalid when the world enters a state for which the plan provides no action. Triangle tables [Nilsson, 1985] were a solution of this type, but they assumed that the likely deviations of the world from the intended solution path would be, themselves, to other states on that path. More general plans could be constructed by making their scope (the number of situations for which they have a reaction) somewhat larger. How large they should be depends on the nature of the domain and how likely the operators are to do what they are expected to do. Given probabilistic characterizations of the operators' effects, it is possible to generate an action map such that if the agent were to act according to the map it would, with high probability, arrive in a goal state before finding itself outside the domain of the map (a planning algorithm with similar characteristics has been developed by Drummond and Bresina [Drummond and Bresina, 1990]).

The kind of planner discussed above is a form of an anytime algorithm [Dean and Boddy, 1988]. An anytime algorithm always has an answer, but the answer improves over time. In the example given above, the answer is useless for a while, then improves in one big jump. It might be useful to have planning algorithms that improve more gradually. Such algorithms exist for certain kinds of path planning, for instance, in which some path is returned at the beginning, but the algorithm works to make the path shorter or more efficient. There is still a difficult decision to be made, however, about whether to take the first step on a plan that is known to be non-optimal or to plan for a while longer.

4.3 Goal Reduction Rules

Given a set of goal-reduction rules, an action map is specified by a top-level goal for the agent. The rules can be "executed" by using them, on each call to the action function, to reduce the top-level goal to a primitive action that is suitable for the currently-perceived state of the world. If the reduction rules are not re-

cursive, execution time has a bound linear in the number and size of the reduction rules. If they are recursive, the goal-reduction process may not terminate, so bounded reaction time cannot be guaranteed.

5 Compilation

We have seen that the high-level languages in which it is convenient for human programmers to specify action maps are often intractable for an agent to execute. Conversely, languages that can be efficiently executed tend to be tedious for human programmers to use. It is possible to bridge this gap, to some degree, by adding a compilation stage in which the language used by the programmer to specify the action map is translated into another language for execution by the agent. This section discusses the compilation of action maps specified as operator descriptions and as goal reduction rules, then considers when it is desirable to perform this compilation.[1]

5.1 Compiling operator descriptions

Operator descriptions can be compiled into a directly executable language. If the initial state description and the goal state description are known at compile time *and* the world is completely deterministic and the operator descriptions absolutely correct, then compilation can simply be planning. The result would be a list of actions to be taken by the agent. This is a very limited approach, because it assumes that the agent has some fixed goal of achievement (unless the planner is a very sophisticated one, capable of synthesizing plans with loops and conditionals, a goal of maintenance would require an infinite list of actions).

When a description of the goal state is known, but the initial state is not known or when the operator descriptions are not completely reliable, descriptions of the goal and the operators can be compiled into an action map specified by a set of condition-action rules. The rules map every possible situation into an action that, according to the operator descriptions, is a useful step toward the goal.

Schoppers' algorithm for synthesizing universal plans [Schoppers, 1989] performs compilation of this sort, although the reaction time of the compiled code may not have a constant bound on execution time.

A disadvantage of compiling operator descriptions into condition-action rules is that very large program structures can result, and although they are as robust as possible, the majority of the program will never be consulted. Additionally, the top-level goal is frozen into the compiled structure.

5.2 Compiling goal-reduction rules

The Gapps language includes an algorithm for compiling a set of goal-reduction rules and a top-level goal into a set of condition-action rules. Whereas the goal-reduction rules could not, in general, be executed by the agent in bounded time, the condition action rules are efficiently executable. To enable this compilation, the top-level goal must be fixed at compile time. It is still possible for the compiled program to respond to externally specified run-time goals, but the goal-reduction mechanism cannot be used at run time [Kaelbling, 1988].

The Gapps compilation procedure described above can be used, in conjunction with standard operators described in terms of a regression function, to compile a set of operator descriptions and a top-level goal into a set of condition-action rules. This compilation method requires that the operator descriptions be used to define a function (**regress** *p alpha*), which returns the weakest condition in the world such that, if operator *alpha* is executed, *p* will be true.

```
(defgoalr (ach p)
    (if (regress p alpha)
        (do alpha)
        (ach (regress p alpha))))
```

This goal-reduction rule says that the goal of achieving a condition p can be reduced to performing some action alpha if that action will cause p to be true in one step (the condition denoted by (**regress p alpha**)); otherwise, it can be reduced to the condition of being one step away from p. If the Gapps compiler is modified to have a depth bound, so that only plans of finite length may be considered, it will generate a partial action map that has an action for every situation from which the goal can be achieved in a number of steps that is less than the depth bound.

5.3 Online versus Offline Compilation

There is a middle ground between compiling all of the declarative structure and flexibility away and performing large search computations on each action step. When the agent can encounter a wide range of goals or initial states at run time, it may be more efficient to retain a compact declarative description of the agent's abilities in terms of operator descriptions or goal-reduction rules and use them to derive actions at run time. This approach is an instance of a space versus time trade-off. It would always be possible to do the complete compilation in advance, but storing the result could take a huge amount of space. In addition, we have already seen that complete direct execution of the declarative specification is intractable. Thus, for each world, agent, and task specification there is an appropriate degree of compilation.

An agent's behavior usually stems from the requirements of a number of constraints. They may be ever-present constraints, such as not running out of power or avoiding running into walls; they may be reflex constraints, such as pulling away from touching a hot object; or they may be dynamic goal constraints, such as going to the store to get a sandwich because someone asked you to. An intuitively reasonable place to divide compilation from run-time interpretation is according to when the constraints are known. Thus, all of the agent's background and reflex constraints might

[1] In the terms used by Russell to discuss knowledge compilation [Russell, 1989], the methods described in this section perform heterogeneous compilation, mapping knowledge of types A, B, and F into knowledge of type D.

be compiled into a program that is intersected with a program obtained by run-time interpretation associated with a particular dynamic goal that has been received by the agent. Another intuitive dividing line would be to compile those parts of the agent's behavior that are appropriate for the situations in which it is most likely to find itself. If the agent finds itself without a compiled response to a particular situation, it can fall back on dynamic interpretation of high-level structures.

Blythe and Mitchell have explored incremental compilation methods in a mobile robot [Blythe and Mitchell, 1989]. The robot uses a traditional planner to solve problems initially, but it caches the results of the planning as situation-action rules. Whenever the robot re-encounters a situation, it can act reactively. This is a relatively simple but reliable way to ensure that the agent can react quickly to common occurrences.

6 Conclusion

The selection of a formalism for specifying the action map of an embedded agent is much the same as the selection of a programming language for a conventional programming project. Two important considerations are the ease of use of the formalism for the agent's designer and the efficiency of execution of the formalism by the agent. These two considerations are often in direct conflict, but that conflict can be mediated very successfully by a variety of compilation methods. It is an important research direction to develop new compilation methods, exploring the trade-offs between efficiency of the compiler and efficiency of the compiled code and between online and offline compilation.

Acknowledgments

Many of these ideas are a result of joint work with Stan Rosenschein. This paper was also influenced by helpful discussions with David Chapman on action formalisms as programming languages.

References

[Allen *et al.*, 1990] James Allen, James Hendler, and Austin Tate, editors. *Readings in Planning.* Morgan Kaufmann, San Mateo, California, 1990.

[Blythe and Mitchell, 1989] Jim Blythe and Tom M. Mitchell. On becoming reactive. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 255–257, Ithaca, New York, 1989. Morgan Kaufmann.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–378, 1987.

[Dean and Boddy, 1988] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Minneapolis-St. Paul, Minnesota, 1988.

[Drummond and Bresina, 1990] Mark Drummond and John Bresina. Anytime synthetic projection. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 138–144, Boston, Massachusetts, 1990. Morgan Kaufmann.

[Hayes, 1990] Patrick J. Hayes. The frame problem and related problems in artificial intelligence. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning.* Morgan Kaufmann, San Mateo, California, 1990.

[Kaelbling, 1987a] Leslie Pack Kaelbling. An architecture for intelligent reactive systems. In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning About Actions and Plans*, pages 395–410. Morgan Kaufmann, 1987.

[Kaelbling, 1987b] Leslie Pack Kaelbling. Rex: A symbolic language for the design and parallel implementation of embedded systems. In *Proceedings of the AIAA Conference on Computers in Aerospace*, Wakefield, Massachusetts, 1987.

[Kaelbling, 1988] Leslie Pack Kaelbling. Goals as parallel program specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Minneapolis-St. Paul, Minnesota, 1988.

[Nilsson, 1985] Nils J. Nilsson. Triangle tables: A proposal for a robot programming language. Technical Report 347, Artificial Intelligence Center, SRI International, Menlo Park, California, 1985.

[Rosenschein and Kaelbling, 1986] Stanley J. Rosenschein and Leslie Pack Kaelbling. The synthesis of digital machines with provable epistemic properties. In Joseph Halpern, editor, *Proceedings of the Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 83–98. Morgan Kaufmann, 1986. An updated version appears as Technical Note 412, Artificial Intelligence Center, SRI International, Menlo Park, California.

[Rosenschein, 1985] Stanley J. Rosenschein. Formal theories of knowledge in AI and robotics. *New Generation Computing*, 3(4):345–357, 1985.

[Russell, 1989] Stuart J. Russell. Execution architectures and compilation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 15–20, Detroit, Michigan, 1989. Morgan Kaufmann.

[Schoppers, 1989] Marcel J. Schoppers. *Representation and Automatic Synthesis of Reaction Plans.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1989.

[Winograd, 1985] Terry Winograd. Frame representations and the declarative/procedural controversy. In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in Knowledge Representation.* Morgan Kaufmann, 1985.

Plan Evaluation under Uncertainty

John D. Lowrance and **David E. Wilkins**
Artificial Intelligence Center, SRI International
Menlo Park, California 94025
Prepared for: DARPA Planning Workshop, November 1990

Introduction

SRI International (SRI) has a long history of work in automated planning and uncertain reasoning. In a recent effort[1], we began to explore the problem of planning in uncertain environments. Here we present our results on how to evaluate the likelihood that plans will accomplish their intended goals given both an uncertain description of the initial state of the world and the use of probabilistically reliable operators. We begin by reviewing our approach to developing a new planning architecture, for the DARPA/RADC Knowledge-Based Planning Initiative, that will incorporate these results[2].

Planning in Uncertain and Dynamic Environments

SRI intends to develop a planning system capable of coping with the inherent complexity and requirements of many real-world domains. The requirements include uncertain information, competing goals, real-time response, intelligent application of standard operating procedures, integration of multiple plans, dynamic plan modification, and interaction with a human planner. Our approach includes the development of a hybrid methodology that is able to use classical methods such as optimization techniques when appropriate, and will use AI methods to organize and evaluate the evolving plan.

The basis for the proposed flexible, integrated planning system will be several high-performance AI technologies. These include classical, search-based planning; structured, procedural reasoning; and evidential reasoning. Various SRI research programs and results have contributed to the state-of-the-art in each of these areas: SIPE-2 [10] provides a "classical" framework for hierarchically elaborating plans and subplans, tracking resources, and monitoring plan execution; PRS [4] provides a means for bringing expert planning and domain knowledge to bear on the planning problem; Gister's implementation of evidential reasoning [7] provides a natural and effective representation for reasoning from limited uncertain information to assess the present and future states of the world during plan execution. SIPE-2, PRS, and Gister are implemented and tested systems[3], not theoretical exercises. SIPE-2 has been applied to construction tasks and the scheduling of process lines in a real manufacturing environment [11], PRS has been used for monitoring and controlling the Reaction Control System of the NASA Space Shuttle [4] and for battle management aboard a Grumman E-2C [5], and Gister has been applied to naval-intelligence decision problems [7] and helicopter route planning [3], among others.

To provide the capabilities that are required by real-world problems but are not provided by currently available systems, we propose to:

- Introduce evidential-reasoning methods into SIPE-2. Evidential reasoning constructs will be used to represent nondeterministic operators and uncertain and incomplete situational knowledge.
- Implement procedural templates as PRS "Knowledge Areas" for representation and use of expert planning

[1]Contract No. N00039-88-C-0248, Space and Naval Warfare Systems Command/Defense Advanced Research Project Agency.

[2]Contract No. F30602-90-C-0086, Rome Air Development Center/Defense Advanced Research Project Agency.

[3]SIPE-2, PRS, and Gister are trademarks of SRI International.

knowledge. These templates will be used to outline plans for stereotypical situations (where standard operating procedures would be used); they will be applied by PRS in real-time; they will be used by SIPE-2 to provide methods for improving planner operation when resources (including time) are constrained. This capability is a necessity for controlling the uncertainty-induced growth of the solution space.

- Integrate generic and problem-specific optimization methods with AI planning techniques.
- Develop techniques to control planning and plan execution that will enable the system to react quickly when necessary, and to consider more alternatives when time permits.
- Design mechanisms for combining plans that have been produced in a distributed manner.
- Validate the technical developments through a proof-of-concept demonstration based upon selected logistics problems.

Existing Technologies

In the remainder of this paper, we describe preliminary work that demonstrates some of the ways that our planning technology can be beneficially combined with evidential reasoning. The ideas described are supported by an implementation that incorporates both SIPE-2 and Gister, and demonstrates that the theory works in practice. Brief introductions to these two systems will provide the necessary terminology for the remainder of this paper.

SIPE-2

Faced with the overwhelming complexity of planning, SIPE-2 has attempted to balance epistemological and heuristic adequacy. It retains enough expressive power to be useful, yet makes enough restricting assumptions to produce a viable, efficient implementation. Unlike most AI planning research, the design of SIPE-2 has taken heuristic adequacy as one of its primary goals.

SIPE-2 provides a domain-independent formalism for describing *operators* (the planner's representation of actions), and utilizes the knowledge encoded in these operators, together with heuristics for handling the combinatorics of the problem, to plan means to achieve given goals in diverse problem domains. The plans include a *plan rationale* so that the system can modify these plans in response to unanticipated events during plan execution. Automatically, or under interactive control, the system generates possibly nonlinear plans containing conditionals that will achieve the given goals when executed in the given initial situation. It can intermingle planning and execution, and can accept arbitrary descriptions, in the language used to describe the domain, of unexpected occurrences during execution and modify its plan to take these into account.

To achieve heuristic adequacy, SIPE-2 incorporates special techniques for solving a number of problems; these techniques are described elsewhere [10]. The technique most important for the current work is the *truth criterion.* A planner's *truth criterion* is its algorithm for determining whether a formula is true in a particular world state. As Chapman has shown [1], nonlinearity makes the truth criterion NP-complete, given a reasonably powerful representation. SIPE-2 incorporates several heuristics for making its truth criterion efficient [10]. For present purposes, these are unimportant except to note that each node in a SIPE-2 plan network implicitly and economically encodes a world state by specifying the changes that have occurred since the previous world state. Thus a plan node can be passed back to Gister as a representation of a world state. Of course, only by using this node in the context of the whole plan network and using SIPE-2's truth criterion to process this network can the implicit representation be properly interpreted.

Gister

Gister incorporates a body of techniques for automated reasoning from evidence that we call *evidential reasoning.* The techniques are based upon the mathematics of belief functions developed by Dempster and Shafer [2, 8, 9] and have been applied to a variety of problems including multisensor integration, situation analysis, route planning, diagnosis, plan execution monitoring, and process control.

We have developed both a formal basis and a framework for implementing automated reasoning systems based upon these techniques [6]. Both the formal and practical approach can be divided into four parts: (1) specifying a set of distinct propositional spaces (i.e., *frames of discernment*), each of which delimits a set of possible world situations; (2) specifying the interrelationships among these propositional spaces (i.e., *compatibility relations*); (3) representing bodies of evidence as belief distributions over these propositional spaces

(i.e., *mass distributions*); and (4) establishing paths (i.e., *analyses*) for the bodies of evidence to move through these propositional spaces by means of evidential operations, eventually converging on spaces where the target questions can be answered. These steps specify a means for arguing from multiple bodies of evidence toward a particular (probabilistic) conclusion.

Evaluating Plans in Uncertain Worlds

The approach we are taking in our preliminary work is to combine our existing systems into a test bed suitable for exploring different technological solutions. Later in this effort, when our solutions have stabilized, we will determine the target architecture for the final implementation. By combining our existing systems, we are attempting to accumulate the benefits of each. This is facilitated by the fact that each system can support a common representation, namely first-order logic. To facilitate this, we posed the problem of evaluation of a given plan in an uncertain environment. By *uncertain environment*, we mean a world where the initial state is not known with certainty and where the effects of actions are not known with certainty.

Since Gister supports reasoning about uncertainty, but SIPE-2 and PRS do not, our approach is to have Gister evaluate a given plan in an uncertain environment. The plan will not incorporate uncertain information, rather it will be a plan produced by SIPE-2, or a standard operating procedure that has been selected by PRS, or a plan created by the user. By *evaluate a plan*, we mean that Gister will be able to predict the probabilistic results of executing a plan given that neither the initial state of the world nor the effects of applying operators (in known states) are known with certainty.

Frame Logic

The first step in applying Gister to a selected domain of application is to define the *frame logic*. Suppose that the answer to some question $\mathcal{A}$ is contained in a finite set Θ_A. That is, each element a_i of Θ_A corresponds to a distinct possible answer to the question $\mathcal{A}$, no two of which can be simultaneously true. For example, $\mathcal{A}$ might be a question concerning the configuration of a set of blocks. In this case, Θ_A would consist of all the possible configurations under consideration. Θ_A is called a *frame of discernment*. If there are exactly three blocks, labeled "A", "B", and "C", and each can rest on top of one other block or on a table, then Θ_A might be defined as follows:

$$\begin{aligned}\Theta_A \;=\; \{&\mathsf{ABC}, \mathsf{ACB}, \mathsf{AB\text{-}C}, \mathsf{AC\text{-}B}, \mathsf{BAC},\\ &\mathsf{BCA}, \mathsf{BC\text{-}A}, \mathsf{BA\text{-}C}, \mathsf{CAB},\\ &\mathsf{CBA}, \mathsf{CA\text{-}B}, \mathsf{CB\text{-}A}, \mathsf{A\text{-}B\text{-}C}\}\ ,\end{aligned}$$

where AB-C corresponds to block A resting on top of block B, and blocks B and C resting on the table.

Propositions Once a frame of discernment has been established for a given question, it formalizes a *variable* where each possible value for the variable is an element of the frame. A statement pertaining to the value of this variable is discerned by the frame, just in case the impact of the statement is to focus on some subset of the possible values in the frame as containing the true value. In other words, a propositional statement A_i about the answer to question $\mathcal{A}$ corresponds to a subset of Θ_A. For example, if the statement is "block A is on block B," then it corresponds to the set of block configuration in Θ_A where block A rests on block B.

$$\mathsf{A\text{-}ON\text{-}B} = \{\mathsf{ABC}, \mathsf{AB\text{-}C}, \mathsf{CAB}\} \subseteq \Theta_A\ .$$

Other propositions related to this question can be similarly represented as subsets of Θ_A (i.e., as elements of the power set of Θ_A, denoted 2^{Θ_A}); the subset A_j might correspond to all those configurations in Θ_A that have no block on top of block C. Once this has been accomplished, logical questions involving multiple statements can be posed and resolved in terms of the frame. Given two propositions, A_i and A_j, and their corresponding sets, A_i and A_j, the following logical operations and relation can be resolved through the associated set operations and relation:

$$\begin{aligned}\neg\mathsf{A}_i &\iff \Theta_A - A_i\\ \mathsf{A}_i \wedge \mathsf{A}_j &\iff A_i \cap A_j\\ \mathsf{A}_i \vee \mathsf{A}_j &\iff A_i \cup A_j\\ \mathsf{A}_i \Rightarrow \mathsf{A}_j &\iff A_i \subseteq A_j\ .\end{aligned}$$

Thus, when two statements pertaining to the same question are available, and they are each represented as subsets of the same frame, their joint impact is calculated by intersecting those two subsets. Given "A is on B" (A-ON-B) and "the top of C is clear" (CLEAR-C), their joint impact is

$$\begin{aligned} \text{CLEAR-C} &= \{\text{AB-C}, \text{BA-C}, \text{CAB}, \text{CBA}, \\ &\quad \text{CA-B}, \text{CB-A}, \text{A-B-C}\} \\ \text{A-ON-B} \cap \text{CLEAR-C} &= \{\text{AB-C}, \text{CAB}\} \end{aligned}$$

All other statements that correspond to supersets of this result in Θ_A, are implicitly true (e.g., "a block is on B"); all of those statements whose corresponding sets are disjoint from this result are implicitly false (e.g., "A is on C"); and all others statements' truthfulness are undetermined (e.g., "B is on the table"). As additional information becomes available, it can be combined with the current result in the same way. Since intersection is commutative and associative, the order that information enters is of no consequence.

Translating Propositions Suppose that another question of interest B has been separately framed. For example, if A corresponds to the state of the blocks at time 1, then B might correspond to the state of the blocks at time 2. Its frame of discernment, Θ_B, is defined as the set of possible block configurations at time 2 (for this example, Θ_A and Θ_B are equivalent).

$$\begin{aligned} \Theta_B &= \{b_1, b_2, \ldots, b_m\} \\ B_j &\subseteq \Theta_B \quad . \end{aligned}$$

If something is known about the state of the blocks at time 1, we would like to take advantage of this information to narrow the possibilities at time 2. To do this, one must first define a *compatibility relation* between the two frames. A compatibility relation simply describes which elements from the two frames can be true simultaneously i.e., which elements are compatible. For this example, if at most one block can be moved in a single unit of time, then state AB-C from Θ_A is compatible with AB-C, CAB, and A-B-C from Θ_B, since these are the only states that could immediately follow AB-C. Thus, a compatibility relation between frames Θ_A and Θ_B is a subset of the cross product of the two frames. A pair (a_i, b_j) is included if and only if they are compatible. Typically, there is at least one pair (a_i, b_j) included for each a_i in Θ_A (the analogue is true for each b_j):

$$\Pi_{(A,B)} \subseteq \Theta_A \times \Theta_B \quad .$$

Using the compatibility relation $\Pi_{(A,B)}$ we can define a *compatibility mapping* $\Gamma_{A \mapsto B}$ for translating propositional statements expressed relative to Θ_A to statements relative to Θ_B. If a statement A_k is true, then the statement $\Gamma_{A \mapsto B}(A_k)$ is also true:

$$\begin{aligned} \Gamma_{A \mapsto B} : 2^{\Theta_A} &\mapsto 2^{\Theta_B} \\ \Gamma_{A \mapsto B}(A_k) &= \{\, b_j \mid (a_i, b_j) \in \Pi_{(A,B)},\ a_i \in A_k \,\} \quad . \end{aligned}$$

In our example, the compatibility relation $\Pi_{(A,B)}$ delimits all possible state changes between time 1 and 2. However, when evaluating a plan, additional information is available, namely, the specific action or operation that is to be performed. One means of incorporating this information is to define a distinct compatibility relation corresponding to each operation. For example, the compatibility relation $\Pi_{\text{PUT-C-ON-A}}$ would have CAB as the only state in Θ_B compatible with AB-C in Θ_A; those states in Θ_A that already have block C on block A (e.g., CAB) or that have some block on C, thus preventing it from being moved, are compatible with the same state in Θ_B.

Given the propositions A-ON-B and CLEAR-C at time 1, we conclude that the initial state is either AB-C or CAB; if we do not know which, if any, operator is applied in this state, then we conclude that any of three states are possible at time 2, AB-C, CAB, or A-B-C; on the other hand, if we know that *PUT-C-ON-A* is applied, then we conclude that the state at time 2 must be CAB. Presuming that the possible states for any time i are the same as those for times 1 and 2, and that the possible operations and their effects are the same in moving from any time i to $i+1$, these frames and compatibility relations can be used to calculate the effects of any planned sequence of actions.

Framing Evidence

When information is inconclusive, partial beliefs replace certainty; probabilistic distributions over statements discerned by a frame replace Boolean valued propositions. These distributions are called *mass distributions*. Each body of evidence is represented as a mass distribution (e.g., m_A) that distributes a unit of belief over propositional statements discerned by a frame (e.g., Θ_A):

$$\begin{aligned} m_A : 2^{\Theta_A} &\mapsto [0,1] \\ \sum_{A_i \subseteq \Theta_A} m_A(A_i) &= 1 \\ m_A(\emptyset) &= 0 \quad . \end{aligned}$$

For example, if we are told that there is an 80% chance that block A is on B and a 20% chance that block A is not on B, then this is represented by a mass distribution $m_{\text{A-ON-B}}$ that attributes 0.8 to the set corresponding to A-ON-B, 0.2 to the complement of A-ON-B with respect to Θ_A, and 0.0 to all other subsets of Θ_A.

Interpreting Evidence To *interpret* a body of evidence relative to the statement A_j, we calculate its *support* and *plausibility* to derive its *evidential interval* as follows:

$$\begin{aligned} Spt(A_j) &= \sum_{A_i \subseteq A_j} m_A(A_i) \\ Pls(A_j) &= 1 - Spt(\Theta_A - A_j) \\ [Spt(A_j), Pls(A_j)] &\subseteq [0,1] \ . \end{aligned}$$

Given the body of evidence represented by $m_{\text{A-ON-B}}$, the evidential interval for A-ON-B is $[0.8, 0.8]$, for CLEAR-C is $[0.0, 1.0]$, for {ABC} is $[0.0, 0.8]$, and for CLEAR-B is $[0.0, 0.2]$.

Propositional statements that are attributed nonzero mass are called the *focal elements* of the distribution. When a mass distribution's focal elements are all single element sets, the distribution corresponds to a classical *additive* probability distribution and the evidential interval, for any proposition discerned by the frame, collapses to a point i.e., support is equivalent to plausibility. For any other choice of focal elements, some propositional statement discerned by the frame will have an evidential interval with support strictly less than plausibility. This reflects the fact that mass attributed to a set consisting of more than one element represents an incomplete assessment; if additional information were available, the mass attributed to this set of elements would be distributed over its single element subsets. Thus, an evidential interval with support strictly less than plausibility is indicative of incomplete information relative to the frame.

For example, consider another point of evidence. A computer vision system reports that block C is clear. Based upon our previous experience with this system, we know that it always correctly determines if a block is clear or not, but 10% of the time it misidentifies the block. In other words, although we do not doubt that some block is clear, we are uncertain whether the block observed was C. Assuming that there is a 90% chance that the observed block was C and a 10% chance that it was not, then this evidence is represented by a mass distribution $m_{\text{CLEAR-C}}$ that attributes 0.9 to CLEAR-C and 0.1 to the set of all possible configurations, since every configuration has at least one clear block. Based upon this distribution, the evidential interval corresponding to the proposition that block C is clear is $[0.9, 1.0]$, that it is not clear $[0.0, 0.1]$, and that it is any particular configuration where C is clear is $[0.0, 1.0]$.

Fusing Evidence When two mass distributions m_A^1 and m_A^2 representing independent opinions are expressed relative to the same frame of discernment, they can be *fused* (i.e., combined) using *Dempster's Rule of Combination*. Dempster's rule pools mass distributions to produce a new mass distribution m_A^3 that represents the consensus of the original disparate opinions. That is, Dempster's rule produces a new mass distribution that leans towards points of agreement between the original opinions and away from points of disagreement. Dempster's rule is defined as follows:

$$\begin{aligned} m_A^3(A_k) &= m_A^1 \oplus m_A^2(A_k) \\ &= \frac{1}{1-\kappa} \sum_{A_i \cap A_j = A_k} m_A^1(A_i) m_A^2(A_j) \\ \kappa &= \sum_{A_i \cap A_j = \emptyset} m_A^1(A_i) m_A^2(A_j) \\ &< 1 \ . \end{aligned}$$

Combining the two bodies of evidence $m_{\text{A-ON-B}}$ and $m_{\text{CLEAR-C}}$ by Dempster's rule results in a mass distribution that attributes 0.72 to C being clear and A being on B (i.e., {CAB, AB-C}), 0.18 to C being clear and A not on B (i.e., {A-B-C, CB-A, CA-B, CBA, BA-C}), 0.08 to A on B (i.e., A-ON-B), and 0.02 to A not on B (i.e., the complement of A-ON-B). This induces the following evidential intervals: $[0.9, 1.0]$ for CLEAR-C, $[0.72, 1.0]$ for C-ON-A, $[0.8, 0.8]$ for A-ON-B, $[0.0, 0.8]$ for {CAB} and {AB-C}, $[0.0, 0.2]$ for {CBA}, and $[0.0, 1.0]$ for CLEAR-A.

Since Dempster's rule is both commutative and associative, multiple (independent) bodies of evidence can be combined in any order without affecting the result. If the initial bodies of evidence are independent, then the derivative bodies of evidence are independent as long as they share no common ancestors.

The *conflict* (i.e., κ) generated during the application of Dempster's rule quantifies the degree to which the mass distributions being combined are incompatible, that is, the degree to which the two distributions

are directly contradictory. When $\kappa = 1$, the distributions are in direct and complete contradiction to one another and no consensus exists (i.e., Dempster's rule is undefined); when $\kappa = 0$, there is no contradiction and the evidential intervals based upon the consensus distribution will be contained within the bounds of the evidential intervals based upon the component distributions i.e., the combination is *monotonic*; otherwise, the component distribution are partially contradictory. In this case, Dempster's rule focuses the consensus on the compatible portions of the component distributions by eliminating the contradictory portions and normalizing what remains; some evidential intervals based upon the consensus distributions will not fall within the bounds of intervals based upon the component distributions i.e., the combination is *nonmonotonic*.

Translating Evidence If a body of evidence is to be interpreted relative to a question expressed over a different frame from the one over which the evidence is expressed, a path of compatibility relations connecting the two frames is required. The mass distribution expressing the body of evidence is then repeatedly *translated* from frame to frame, via compatibility mappings, until it reaches the ultimate frame of the question. In our planning example, interpreting the effects of a body of evidence about time 1 on propositions at time 5 requires that the evidence be translated from frame to frame, for each planned action between time 1 and time 5.

In translating m_A from frame Θ_A to frame Θ_B via compatibility mapping $\Gamma_{A \mapsto B}$, the following computation is applied to derive the translated mass distribution m_B:

$$\begin{aligned} m_B(B_j) &= \frac{1}{1-\kappa} \sum_{\Gamma_{A \mapsto B}(A_i)=B_j} m_A(A_i) \\ \kappa &= \sum_{\Gamma_{A \mapsto B}(A_i)=\emptyset} m_A(A_i) \\ &< 1 \ . \end{aligned}$$

Intuitively, if we (partially) believe A_i, and A_i implies B_j, then we should (partially) believe B_j; if some focal element A_i is incompatible with every element in Θ_B, then there is *conflict* (i.e., κ) between the evidence and the logic of the frames and compatibility relation. This is equivalent to the conflict in Dempster's rule.

In our example, to evaluate the effect of applying the *PUT-C-ON-A* operator, given the two independent bodies of evidence about the initial state, $m_{\text{A-ON-B}}$ and $m_{\text{CLEAR-C}}$, we first combine these mass distributions using Dempster's rule and then translate the result via compatibility mapping $\Gamma_{\text{PUT-C-ON-A}}$ to frame Θ_B. The result is a mass distribution that attributes 0.72 to {CAB}, 0.18 to {CA-B, CBA, BA-C}, 0.08 to {CAB, ABC}, and 0.02 to the complement of {A-B-C}; this induces the following evidential intervals: $[0.9, 1.0]$ for CLEAR-C, $[0.72, 1.0]$ for C-ON-A, $[0.8, 0.8]$ for A-ON-B, $[0.72, 0.8]$ for {CAB}, $[0.0, 0.2]$ for {CBA}, $[0.0, 0.1]$ for CLEAR-A, and $[0.0, 0.0]$ for {AB-C}. Given a sequence of operators to be applied after executing *PUT-C-ON-A*, we simply perform successive translations until the sequence is exhausted.

When multiple bodies of evidence are available over different frames, they must be translated to a common frame before they can be combined using Dempster's rule. They can all be translated to a single frame and combined, or subsets of the available evidence can be translated and combined at intermediate frames, and these intermediate results then translated and combined until the final destination frame is reached. If during plan execution, intermediate observations are made, resulting in additional bodies of evidence about the state of world at time i, these bodies of evidence can be combined with the body of evidence representing the presumed state of the world at time i, to refine the predicted outcome of the plan. So, in our example, if we had additional information about the state of the world at time 2, it could be combined with our result for that time before additional translations are performed.

Implementing Evidential Reasoning

In the preceding discussion, we have defined the frame logic in terms of set theoretic concepts. This is the way that it is most often presented since the audience is usually more familiar with multivariate decision theory and statistics than with propositional logic. However, all of the evidential reasoning operations can be recast using propositional logic. These modified definitions follow.

Interpretation:

$$\begin{aligned} Spt(\mathsf{A}_j) &= \sum_{\mathsf{A}_i \Rightarrow \mathsf{A}_j} m_{\mathsf{A}}(\mathsf{A}_i) \\ Pls(\mathsf{A}_j) &= 1 - Spt(\neg \mathsf{A}_j) \\ [Spt(\mathsf{A}_j), Pls(\mathsf{A}_j)] &\subseteq [0,1] \ . \end{aligned}$$

Fusion:

$$\begin{aligned} m^3_{\mathsf{A}}(\mathsf{A}_i \wedge \mathsf{A}_j) &= m^1_{\mathsf{A}} \oplus m^2_{\mathsf{A}}(\mathsf{A}_i \wedge \mathsf{A}_j) \\ &= \frac{1}{1-\kappa} \sum_{\mathsf{A}_i \wedge \mathsf{A}_j} m^1_{\mathsf{A}}(\mathsf{A}_i) m^2_{\mathsf{A}}(\mathsf{A}_j) \\ \kappa &= \sum_{\neg(\mathsf{A}_i \wedge \mathsf{A}_j)} m^1_{\mathsf{A}}(\mathsf{A}_i) m^2_{\mathsf{A}}(\mathsf{A}_j) \\ &< 1 \ . \end{aligned}$$

Translation:

$$\begin{aligned} m_{\mathsf{B}}(\mathsf{B}_j) &= \frac{1}{1-\kappa} \sum_{\Gamma_{\mathsf{A}\rightarrow\mathsf{B}}(\mathsf{A}_i)=\mathsf{B}_j} m_{\mathsf{A}}(\mathsf{A}_i) \\ \kappa &= \sum_{\Gamma_{\mathsf{A}\rightarrow\mathsf{B}}(\mathsf{A}_i)=\mathsf{FALSE}} m_{\mathsf{A}}(\mathsf{A}_i) \\ &< 1 \ . \end{aligned}$$

Implementing evidential-reasoning systems can be divided into two independent subproblems: How to represent mass distributions and perform numeric calculations on them? How to represent propositions and perform logical inferences? Accordingly, Gister's implementation of evidential reasoning consists of two distinct components: one that manipulates and interprets mass distributions and another that performs logical reasoning. As mass distributions are manipulated by the first component, logical questions are posed to the second component. The implementation of each of these components is independent of the other. The best suited implementations depends upon the characteristics of the domain of application and upon the characteristics of the host computational environment. Most importantly, the numeric component places no constraints on the representation of propositions or the implementation of the logical operations, just so long as the logical questions posed by the numeric component are answered by the logical component.

Since different logical representations are better suited to different applications, Gister allows a frame logic implementation to essentially be given as a parameter. A frame logic implementation is represented as a distinct object (using object-oriented programming techniques) capable of answering all of the logical questions required to support the numeric module's evidential operations. One such implementation is based on set theory, mirroring the set-theoretic presentation of the frame logic in this paper.

However, it should be clear from the simple blocks world example in this paper that this representation is not suitable for real-world planning. The representation is too cumbersome since each possible world state must be enumerated, and the compatibility relations must specify all compatible states for every possible world state. Furthermore, continually translating from one frame to another during evaluation of a plan will be inefficient, and partially ordered plans will cause problems. Instead, we propose to develop a frame logic based upon SIPE-2's representation of plans and techniques for reasoning about them.

A SIPE-2 Logic

The central idea behind the current combination of SIPE-2 and Gister is that the former can provide the logic used by the latter when the domain is planning, with several advantages. Briefly, these advantages are compactness of representation (one does not enumerate every possible world state nor elements of compatibility relations), SIPE-2's efficiency when determining the truth of a proposition in a world state, the use of nonlinear plans under conditions imposed by SIPE-2's heuristics, and the ability of the planner to generate plans automatically when Gister eventually asks that goals be satisfied rather than that operators be applied.

We have implemented a SIPE-2 frame logic for Gister as described below, and have tested it by evaluating and interactively constructing plans in a blocks world with uncertain states. When Gister evaluates plans, the SIPE-2 logic provides the algorithms and representations for determining whether a proposition is true in a world state, for determining whether two world states are equivalent, and for performing translations using a compact SIPE-2 operator.

World States and Propositions

The different possible initial worlds states are represented in Gister as SIPE-2 plan nodes of type *planhead*. Planhead nodes explicitly list all predicates that are true at that node. All other worlds states, i.e., those generated by planned actions, are represented in Gister by SIPE-2 plan nodes of type *process*. The planner creates these plan nodes in response to requests from Gister to perform translations (see next section). Process nodes implicitly represent the world state since they list only predicates that have changed since the previous node. Gister represents an incompletely specified world state as a set of plan nodes.

For example, given the propositions A-ON-B and CLEAR-C at time 1, we represent the resulting incompletely specified world state as {AB-C, CAB}, where AB-C and CAB are names for SIPE-2 planhead nodes. Suppose an operator is applied in this state. As described in the next section, this will cause SIPE-2 to produce plan networks for each element of this set, and the uncertain world state at time 2 would be represented as {P13, P19}, where P13 and P19 are names for SIPE-2 process nodes in plan networks.

Gister needs to query the truth of propositions in specific world states. In our implementation, Gister accepts propositions specified in SIPE-2 input syntax, which are then passed on to the planner together with the plan node representing the desired world state. (Checking a proposition in an incompletely specified state may result in several such calls to the planner). SIPE-2 parses the propositions into appropriate data structures and simply applies its truth criterion to the proposition at the plan node. The plan node is part of a whole plan network that the truth criterion uses to compute its result. This computation has been shown in practice to be efficient, even in fairly realistic domains [11].

In the blocks world example, the SIPE-2 predicates, plans, and operators are exactly the same as they are in published examples [10]. The only limitation of this technique is that certain restrictions are placed by SIPE-2 on the form of the input propositions [10]. We do not expect this to be problematic, and some restrictions could be easily relaxed. This implementation provides both the compactness of plan nodes as a representation and the efficiency of computing on them with the truth criterion.

Translations

As discussed earlier, Gister could use a set to represent a compatibility relation that captures the effects of an action. However, this representation is much too cumbersome in practice since an element must be included from every possible pair of successive world states.

The SIPE-2 logic does translations for Gister by using its operators to generate plan networks. Gister still has a name for each possible compatibility mapping, but need not represent these mappings in any more detail. Gister will call SIPE-2 to translate from one world state to another using the named compatibility mapping. The planner translates Gister's name into a goal or process node in a plan network. For example, a translation request for the compatibility mapping PUT-A-ON-B will cause the planner to add a process/goal node to a plan network. The goal node would specify (ON A B) as the goal, while the process node would specify that the standard PUTON operator be applied to the arguments A and B. The current implementation creates process nodes, the use of goal nodes is described in the next section.

The node is added to the plan at the point that represents the state from which we are translating. The planner then expands this plan in more detail to obtain the final representation of the new world state. This process makes use of the SIPE-2's causal theory for deducing the effects of actions. The plan node returned to Gister will be the last node in the expansion of the added node. Since Gister may make the same request several times, SIPE-2 uses its context mechanism to keep track of all expansions and returns an already constructed plan node whenever appropriate.

Suppose we apply PUT-A-ON-C in the incompletely specified state {AB-C, CAB} at time 1. SIPE-2 will create process nodes for applying PUTON to A and C after each of the two planhead nodes AB-C and CAB. The first one will be expanded by the planner and a process node, say P13, will be returned. As described later, an equivalence test in the SIPE-2 frame logic will allow Gister to merge P13 with AC-B. The second process node cannot be expanded because the precondition of the operator is not satisfied. Currently, SIPE-2 will return the previous world state, CAB in this case, on the assumption that the executing agent recognizes the unexecutable action and ignores it. Thus the resulting uncertain state at time 2 would be {P13, CAB}. Domain-specific knowledge about the effects of attempting unexecutable actions could easily be incorporated. For example, if the agent would knock C off A while attempting to put A on C in CAB, an operator could be written to encode this knowledge and the precondition of this operator would allow it to expand the node for putting A on C.

In this system, the compatibility mappings are represented by SIPE-2 operators. Since the Gister names may encode a list of arguments, one operator with variables can represent any number of Gister compatibility mappings. For example, the single standard PUTON operator is used to represent all 9 blocks world compatibility mappings. Thus a significant economy is achieved, and the economic representation can be computed with efficiently.

Another advantage of using plan networks is that they might contain nonlinear plans, yet the nonlinearity would be invisible to Gister. The process node returned to Gister would be after the unordered actions in the nonlinear plan, so only SIPE-2's truth criterion needs to address the question of nonlinearity. Thus, the restriction to linear sequences can be partially alleviated.

Generating Plans for Translations One extension of this scheme is to allow the translation to be described as a goal node to be achieved. The planner could then build an arbitrary plan for achieving this goal and use it to represent the compatibility mapping. One complication is that this means the "compatibility mapping" might vary depending on the situation (since different plans might be generated). However, achieving a goal in an uncertain world state will require that the same plan be used for each world state in the mass distribution. While this complication does not appear to pose theoretical difficulties, it has not yet been implemented. This capability of translating via goals would be useful for letting the planner fill in the details of a more abstract plan that has been provided.

Equivalent States

It is important to notice when two world states are equivalent in Gister, since this can significantly collapse the size of the sets that the system must reason about, which in turn significantly reduces the combinatorics. This is particularly useful in the blocks world because the simple states mean that all sorts of plan networks might result in the same world state. In more complex, realistic domains it may be rare for different sequences of actions to result in exactly the same state. However, even in these domains it will eventually be necessary to recognize states as equivalent in all relevant aspects so that the combinatorics can be reduced.

For this reason, we have not written code to determine the equality of two states in SIPE-2 (a possibly expensive computation). Instead we allow the user to specify the relevant aspects for dividing states into equivalence classes. While this puts more of a burden on the user, we view it as necessary for obtaining heuristic adequacy in complex domains. This is accomplished by defining an "equivalence" operator that is designated for this checking. This is a standard SIPE-2 operator with a list of arguments and a precondition, but nothing else. Matching the precondition in a particular world state will return a list of instantiations for variables in the operator that effectively specify its equivalence class. Thus, when Gister asks whether two world states are equivalent, SIPE-2 simply calls its truth criterion on the precondition of the equivalence operator at each of the two states. If the result is failure in both cases, or success with the same variable instantiations in both cases, then the two states are equivalent. Again, the efficiency of the truth criterion is used to significantly improve on an algorithm for determining the equality of any two states.

For example, our equivalence operator in the blocks world has a precondition of (ON A OBJECT1) $\wedge$ (ON B OBJECT2) $\wedge$ (ON C OBJECT3), where the OBJECTn are variables to be instantiated. In a world where A, B, and C are the only blocks, this condition distinguishes every state, effectively implemented a test for equality with the efficiency obtained from using the equivalence-operator mechanism. In our previous example, P13 and AC-B were equivalent. This is easily determined by matching the equivalence condition at each of these two nodes, and getting C, TABLE, and TABLE as the instantiations for the OBJECTn in both cases.

Probabilistic Operators

The discussion to this point has focused on plan evaluation when the initial (and therefore subsequent) state of the world is uncertain. Another source of uncertainty that needs to be taken into account is the nondeterministic nature of many real-world operators. Within the blocks world, one can imagine that if a robot is attempting to move the blocks as specified in a plan, that each operation will only probabilistically achieve the intended goal. For example, if the operation is to put block C on block A, the initial grasp for block C might fall short leaving block C in its original position or the placement of block C on top of block A might fail, causing block C to fall to the table. These probabilistically accurate operators can be incorporated into an evidential model as probabilistic translations.

As previously discussed, given two frames, Θ_A and Θ_B, and a compatibility relation, $\Pi_{(A,B)}$, propositional statements can be translated between these two frames. Alternatively, instead of translating propositional statements between these two frames via $\Gamma_{A \mapsto B}$ and $\Gamma_{B \mapsto A}$, we might choose to translate these statements to a common frame that captures all of the information and then on to the target frame. This common frame, $\Theta_{(A,B)}$, is identical to the compatibility relation $\Pi_{(A,B)}$. Frames

Θ_A and Θ_B are trivially related to frame $\Theta_{(A,B)}$ via the following compatibility relations and compatibility mappings:

$$\begin{aligned}
\Theta_{(A,B)} &= \Pi_{(A,B)} \subseteq \Theta_A \times \Theta_B \\
\Pi_{(A,(A,B))} &= \{\, (a_i, (a_i, b_j)) \mid (a_i, b_j) \in \Pi_{(A,B)} \,\} \\
\Pi_{((A,B),B)} &= \{\, ((a_i, b_j), b_j) \mid (a_i, b_j) \in \Pi_{(A,B)} \,\} \\
\Gamma_{A \mapsto (A,B)}(A_k) &= \{\, (a_i, b_j) \mid (a_i, b_j) \in \Pi_{(A,B)},\ a_i \in A_k \,\} \\
\Gamma_{(A,B) \mapsto B}(X_k) &= \{\, b_j \mid (a_i, b_j) \in \Pi_{(A,B)},\ (a_i, b_j) \in X_k \,\}
\end{aligned}$$

Given these three frames, Θ_A, $\Theta_{(A,B)}$, and Θ_B, and two compatibility mappings, $\Gamma_{A \mapsto (A,B)}$ and $\Gamma_{(A,B) \mapsto B}$, a mass distribution over Θ_A can be translated to $\Theta_{(A,B)}$ and then on to Θ_B; the result will be identical to that produced through a single translation from Θ_A to Θ_B via $\Gamma_{A \mapsto B}$.

Once this intermediate frame has been introduced, probabilistic information about the relationship between Θ_A and Θ_B can be taken into account. This information, expressed as a mass distribution, $m_{(A,B)}$, over $\Theta_{(A,B)}$, provides a means of "weighting" translations to favor some elements of $\Theta_{(A,B)}$ over others. The probabilistic translation is accomplished by translating the mass distribution over Θ_A to $\Theta_{(A,B)}$, fusing the result with $m_{(A,B)}$, and translating the fused result to Θ_B.

In our blocks world example, if $\Pi_{(A,B)}$ (and consequently $\Theta_{(A,B)}$) delimits all possible state changes between time i and $i+1$, then each nonprobabilistic operator can be represented by a mass distribution that assigns all of its mass to a single set, the set consisting of paired states from Θ_A and Θ_B where the state from Θ_A is transformed into the state from Θ_B by applying that operator. For the operator *PUT-C-ON-A* this set assigned unit mass is designated PUT-C-ON-A. Given similarly constructed sets, PUT-C-ON-TABLE and DO-NOTHING, corresponding to the operators for putting C on the table and doing nothing (i.e., no changes in state), we can represent a probabilistically accurate operator for putting C on A by a mass distribution $m_{\text{PUT-C-ON-A}}$. This mass distribution might attribute 0.9 to PUT-C-ON-A and 0.1 to the union of PUT-C-ON-TABLE and DO-NOTHING, representing the knowledge that 90% of the time this operator acts as intended, but 10% of the time it functions as if the intended action were to put C on the table or to do nothing.

Using this probabilistic version of PUT-C-ON-A in combination with the evidence about the initial state, $m_{\text{A-ON-B}}$ and $m_{\text{CLEAR-C}}$, we conclude [0.9, 1.0] for CLEAR-C, [0.65, 1.0] for C-ON-A, [0.8, 0.8] for A-ON-B, [0.65, 0.8] for {CAB}, [0.0, 0.2] for {CBA}, [0.0, 0.19] for CLEAR-A, and [0.0, 0.8] for {AB-C}. Comparing these results with previous ones obtained using a nonprobabilistic version of PUT-C-ON-A, we find that the support for C-ON-A and {CAB} has decreased, the plausibility for CLEAR-A and {AB-C} has increased, while the evidential intervals for the others have remained unchanged. This reflects the fact that C is less likely to be on top of A and more likely to be elsewhere.

Importantly, this approach to probabilistic operators requires no changes to the SIPE-2 frame logic previously described.

Conclusion

We have implemented a SIPE-2 logic within Gister, and have tested it by evaluating and interactively constructing plans in a blocks world with uncertain world states. This work demonstrates some of the ways that our planning technology can be beneficially combined with evidential reasoning. Several advantages are obtained by this combination: compactness of representation, the efficiency of SIPE-2 operators to determine the effects of actions, SIPE-2's efficiency when determining the truth of a proposition in a world state, the use of nonlinear plans, and the ability of the planner to generate plans automatically while Gister manages the uncertain aspects of the situation.

References

[1] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

[2] Arthur P. Dempster. A generalization of Bayesian inference. *Journal of the Royal Statistical Society*, 30:205–247, 1968.

[3] Thomas D. Garvey. Evidential reasoning for geographic evaluation for helicopter route planning. Technical Report 405, SRI International Artificial Intelligence Center, 333 Ravenswood Avenue, Menlo Park, California, December 1986.

[4] Michael P. Georgeff and François Félix Ingrand. Research on procedural reasoning systems. Final Report Phase 1, SRI International Artificial Intelligence Center, 333 Ravenswood Avenue, Menlo Park, California, October 1988.

[5] François Félix Ingrand, Jack Goldberg, and Janet D. Lee. SRI/GRUMMAN Crew Members' Associate Program: Development of an authority manager. Final Report SRI Project 7025, SRI International Artificial Intelligence Center, 333 Ravenswood Avenue, Menlo Park, California, March 1989.

[6] John D. Lowrance. Automated argument construction. *Journal of Statistical Planning and Inference*, 20:369–387, 1988.

[7] John D. Lowrance, Thomas M. Strat, and Thomas D. Garvey. Application of artificial intelligence techniques to naval intelligence analysis. Final Report SRI Contract 6486, SRI International Artificial Intelligence Center, 333 Ravenswood Avenue, Menlo Park, California, June 1986.

[8] Glenn Shafer. *A Mathematical Theory of Evidence.* Princeton University Press, Princeton, New Jersey, 1976.

[9] Glenn Shafer. Belief functions and possibility measures. *The Analysis of Fuzzy Information*, 1, 1986.

[10] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers Inc., San Mateo, California, 1988.

[11] David E. Wilkins. Can AI planners solve practical problems? Technical Report 468R, SRI International Artificial Intelligence Center, 333 Ravenswood Avenue, Menlo Park, California, November 1989.

Planning Reactive Behavior: A Progress Report

Drew McDermott*
Yale Computer Science Department
P.O. Box 2158 Yale Station
New Haven, Connecticut 06520
mcdermott@cs.yale.edu

Abstract

Recently AI planning theory has concerned itself with the behavior of realistic agents, which involves sensing and reacting. The plans that control this behavior have to be more complicated than traditional action sequences, which makes generating and modifying them much more difficult. I describe a novel planning architecture that is intended to surmount these problems, by providing transformational planning capabilities on top of a reactive plan interpreter for a robot delivery truck. Planning is implemented by way of a set of critics and schedulers that anticipate problems with the plan by projecting it ahead of time, and seek to transform the plan to alleviate these problems. For the plans to be transformable, they must be simple and modular. We make simplicity more likely by providing high-level control structures, and we encourage modularity by providing a uniform way of referring to the tasks generated by executing the plan.

1 Introduction

The focus of our research project is on planning for agents in a dynamic, not fully controllable world. It has been suggested that in such an environment agents cannot and should not have plans. [2, 1, 5] Having a plan seems to imply planning ahead, and the benefits of planning ahead are often limited. We agree with this view up to a point, but it can be overstated. Planning ahead is indeed of little value — except when it is of great value. An agent with several trips to undertake can gain a lot by coordinating them, rather than by doing them in no particular order or all at once. For the agent to be able to take advantage of such planning opportunities, it must have the ability to think about the future, in particular about what it intends to do and what else will happen. We take the agent's *plan* simply to be that part of its future intentions that it is in a position to reason about. Being able to plan means being able to model and improve some piece of one's program before it is executed.

Classical planners focused entirely on program manipulation. Consequently they could be based on the assumption that the programs were quite simple, typically sequences of actions involving the manipulation of objects with standard names. Now that we are taking agenthood more seriously, we realize that plans have to contain explicit steps for sensing the world and reacting to it. Such plans are better able to guide agent behavior over long stretches of time. But they are harder to reason about. One response to this difficulty is to give up and just tune all plans by hand. But a less pessimistic response is to try to make reactive plans transparent enough that a planner has a chance of reasoning about them and improving them. The payoffs of even a small planning capacity seem large enough that it is worth trying to develop one.

Several people are working on ways of combining planning with reaction. [20, 33, 26, 14] It is too early to tell which approaches are best. Hence the views I outline here are basically in the form of a manifesto rather than an argument.

We assume that an agent always has a plan. New assignments are given to it in the form of abstract plans, with steps like "Make such-and-such a state true." But even abstract plans have default methods for carrying them out, so that the agent's plan is always in some sense executable, even if the default methods have little chance of succeeding. Actually, we expect that the default method will under normal circumstances be quite capable. We do not insist that the planner be able to do something useful within a bounded period of time. [4, 18, 19] In any case, the present paper is concerned with general architectural issues, not the adequacy of particular plans.

Under this view, planning is the operation of improving the agent's existing plan. It goes on in the background, at whatever time scale its natural evolution requires. Whenever the planner thinks it has an improved plan, it swaps it for the current plan. The interpreter must be built so this can happen smoothly. If things are happening too fast, then the planner may never catch up with the interpreter, in which case the interpreter's existing plan had better be good enough.

One of the most elegant ideas in planning theory is

*The work reported herein was supported by the Defense Advanced Research Projects Agency, contract number DAAA15-87-K-0001, administered by the Ballistic Research Laboratory.

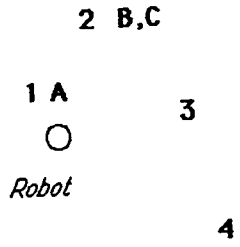


Figure 1: A Simple Delivery Problem

that a planner can operate by refining abstract plans. [28, 23] Unfortunately, our architecture makes that idea inapplicable. We have to find ways of editing an already fleshed-out plan. [13, 29, 30] We can distinguish between two sorts of editors: schedulers and debuggers.[1] A *scheduler* is a transformation that reorders tasks in order to optimize time or other resource usage. [9, 25, 3]. A *debugger* is a transformation that attempts to eliminate a bug. [32] A *bug* is simply a detectable problem, discovered by a process of *projecting* the plan — simulating its operation — to see how well it will work. [35, 15, 16] A scheduler cannot run in response to bug detection, because such a transformation is worth doing whenever it would substantially improve the plan, and usually the only way to verify that it will is to try it. Hence schedulers should run whenever they can. One advantage of doing things this way is that schedulers tend to increase the amount of order in a plan, and an ordered plan is easier to reason about. [24]

Figure 1 shows a simple example of the kind of problem we want to solve. The system is given three jobs, to take object A from location 1 to location 3, and take objects B and C from location 2 to location 4. Its orders arrive as a plan to do these three things in no particular order — interleaving steps if necessary. This specification is already executable, but the interpreter by default would do the errands in some random order, unlikely to be optimal. As the interpreter starts to work (we assume these trips take a substantial amount of time, on the order of minutes or hours), the planner starts to think. The first thing it does is call a heuristic scheduler. In this domain, a good way to proceed is to extract a traveling-salesman algorithm from the plan, solve it approximately (in polynomial time [27]), and impose the constraints derived on the plan. The result, as shown in the figure, is a reasonable schedule, in which all the loads happen before any of the unloads.

To make the problem interesting, let us suppose that the scheduler, black box that it is, is incapable of taking any constraints into account except for those depending on the space-time location of tasks. In particular, it has no idea that the robot has only two hands and cannot carry three objects at once. Detecting and solving this bug is the job of another module, the *overload bug detector*. In the present case, its only remedy for the bug is to remove all constraints imposed by the scheduler, add the constraint that an unloading (say, of A) occur before the overload, and call the scheduler again. This time it decides to deliver A before loading B or C.

[1] I used the term "debug" differently in [23].

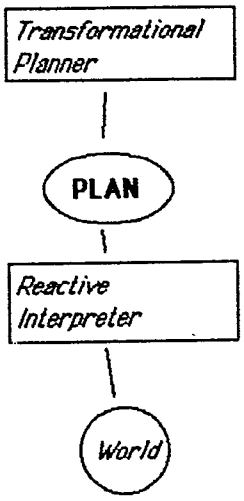


Figure 2: Block Diagram of XFRM Planner/Interpreter

Our goal is to devise and implement a planner that can carry this plan out, with or without optimizing it. The simultaneous requirements that the plan must

- be complex enough to be able to survive interference;
- but be transparent enough to be manipulated by a planner

impose strong constraints on the design of the interpreter, the planner, and the plan itself. In the rest of this paper, I will explain the design decisions that are the result of these constraints.

2 Agent Architecture

Figure 2 shows a top-level view of the agent architecture. There is a central plan that is manipulated by a transformational planner and a reactive executor. We call the planner XFRM.

One of the major issues in deciding which transformation to do next is deciding how to coordinate debuggers and schedulers. The debuggers ought to take precedence, because a buggy plan is probably too awful to try to optimize. But we can't arrange to run the schedulers after the debuggers, simply because the schedulers may themselves introduce bugs. Hence when a debugger runs, it may have to undo some work done by a scheduler.

That means that (a) the orderings imposed by the scheduler must be clearly marked as undoable; (b) schedulers must be rerun when their work might have been undone. Achieving (a) is just a matter of bookkeeping. Achieving (b) is trickier. We don't want to run every scheduler periodically, and we don't want to try to guess which need to run. It seems best to have any program

that undoes a schedule to be responsible for rerunning the appropriate schedulers. That will have the additional advantage that after every plan revision that plan will be as scheduled as possible. The more ordered a plan is, the easier it is to work with.

We use the classic phrase *plan critic* [32] to refer to a module that looks for a particular class of problem and proposes corrections. A critic is a procedure that takes a plan and a set of timelines resulting from projecting it, and returns a list of bugs. A *bug* is a data structure with the following slots:

1. *Penalty:* How much it costs a plan to have this bug.
2. *Signature:* A symbolic description of the bug.
3. *Comparer:* A procedure for comparing the signature with that of another bug. Two bugs can be the same except for severity. (E.g., the same deadline can be missed by a minute or by an hour.)
4. *Transformation:* A procedure that will try to fix the bug by making a change in the plan. It will return zero or more new plans that purport to be improvements.

XFRM operates as follows. It keeps a queue of buggy plans. Each plan has several bugs, but the worst bug is special in that it is the one whose transformation will be run if the plan is selected for future search. In the steady state, the planner repeatedly executes this loop:

1. Select the most promising plan on the queue, and run its transformation, thus generating some new plans.
2. For each new plan, run the projector to generate timelines; run the critics on the plan and timelines to find bugs; score the resulting plan.
3. Sort and merge the new plans into the queue.

Formally this system is a best-first seacher. However, in practice it had better focus on a very narrow "beam" in the search space, or else the whole idea will collapse. Here we are inspired by past transformational systems ([13, 29, 30]) that avoided keeping track of more than one plan at all. (Hammond's planner picked one transformation and tried it. Simmons's system tried several transformations, but picked one of the resulting plans based on a heuristic score, and discarded all the others.)

The reason for comparing bug signatures is to address a hard question, how to measure progress in eradicating bugs. It could easily happen that a transformation could fail to eliminate a bug, or even reintroduce one previously eliminated. By encouraging critics to provide simple symbolic descriptions of a bug, we make it likely that a persistent bug will be recognized. The plan evaluation function adds heavy penalties for a bug that has reappeared.

Our plans are general robot plans, which raises special problems for the projection and transformation algorithms. The projector basically mimics the plan interpreter and the world's response, but it must do more than that. Suppose the plan contains a *monitor* [8], a command to wait for a condition and then do something. Assuming the condition is not under the planner's control, there are essentially an infinite number of points when it can become true. The best we can do is investigate a random sample.

Once a bug is found, it must be accompanied by a transformation that claims to know how to fix it. The transformation must be able to rearrange portions of a plan, where the plan is a complex program. For example, suppose that the agent is supposed to plan a set of deliveries, and a critic detects that the cargo capacity will be overloaded given the current schedule. The repair transformation could rerun the scheduler after constraining the next possible unloading step to come before the step that caused the overload. (This tactic may not eliminate the problem, but the planner will presumably be able to detect that it is getting better.) To allow such edits, our plan notation must be as transparent as possible.

3 The Interpreter

We call our notation RPL, for Reactive Plan Language. It can be considered to be the next generation of Firby's RAPS system [8], but it is also related to MACNET [5], PRS [12], COAL [6], and robot programming languages like OWL [7]. There are two major differences between RPL and RAPS:

1. The syntax is more "recursive," more in the style of Lisp
2. More high-level concepts (interrupts, monitors) have been made into explicit constructs.

The language is still evolving.

A RPL plan looks like a program. Indeed, I will use the terms "plan" and "program" interchangeably. RPL looks so much like Lisp that many Lisp programs are valid RPL plans. However, that is not the intended use for the language. Mechanisms are provided to allow and encourage the plan writer to let sensory input guide the behavior of the system, rather than complex data and control structures.

One of these mechanisms is the *fluent*, or time-varying quantity. Of course, all program variables are time-varying quantities, but in plans we want behavior to be governed by the temporal changes. For example, in RPL we can say `(FILTER c e)` to mean, "Execute e while the fluent c remains true." If c becomes false, the execution of e is "evaporated" [22], that is, rendered unnecessary. Fluents can be defined in terms of other fluents. For example,

```
(FILTER (AND C1 (> I 5)) (CARRY-OUT A))
```

does `(CARRY-OUT A)` only while `C1` remains true and `I` remains greater than 5. Here `C1` is a Boolean-valued fluent, and `I` is a numerical-valued one.

Fluents can be set by sensors, thus allowing immediate sensory control of actions. The combination of Boolean combination and program control by fluents puts much of the functionality of MACNET [5] into RPL (while dispensing with the idea of combinational-logic compilation).

RPL contains a `LET*` construct for binding local variables, which behaves a lot like Lisp's. One use for this facility is in perceiving objects in the world. E.g., the plan

```
(LET* ((X (Find a block)))
   (PICKUP X))
```

picks up a block. I will say more about this sort of thing in Section 3.3. A more mundane use of local variables is for counting or keeping track of lists, just as in regular programming.

As a RPL plan is executed, a task network is constructed. The task network corresponds to the stack in a standard programming language. However, an important difference is that the "stack frames" can come into being before the interpreter reaches them, and the planner and interpreter can refer to them in advance. A task is an occurrence of an action that the planner has carried out or might try to carry out. A task has two kinds of subtask [22]: syntactic subtasks and reducing subtasks. A *syntactic subtask* of a task T is one that is generated from a piece of the text of the action of T. For example, if the action of T is `(LOOP (A) (B))`, then one of its syntactic subtasks might be "the second step in the third iteration of T," i.e., the third execution of `(B)`. We denote this task with the expression `(SUB STEP 2 (SUB ITER 3 T))`. This example shows that in principle a task can have an infinite number of syntactic subtasks, but that only a finite number can actually be executed (or even be committed to).

Tasks are accessible using plan variables. The top task is the value of global variable `TOP-TASK*`, and one can use `SUB` expressions to work one's way down to a particular subtask. However, it is often more convenient to `TAG` a subtask. In this plan:

```
(SEQ (A) (TAG STEP2 (B)) (C))
```

the second step can be referred to using the variable `STEP2`. This is most useful in the `PLAN` construct, which expects explicit constraints among steps:

```
(PLAN ((TAG STEP1 (A))
       (TAG STEP2 (B))
       (TAG STEP3 (C)))
      (ORDER STEP1 STEP3)
      (ORDER STEP2 STEP3))
```

`TAG` actually binds local variables to tasks.

A new task is created for every "step" of a plan. To avoid generating a huge pile of tasks, we distinguish between *steps* and *expressions*. The latter are pieces of the plan that are *evaluated* rather than being *executed*. For example, in `(IF e a b)`, the expression e is evaluated, resulting in the usual choice of a or b. A task with an `IF` action has two subtasks, one for the true arm and one for the false arm. There is no subtask for the test, e.

I also mentioned *reducing* subtasks above. The planner can step in and declare that a certain set of tasks is to be carried out using a specified RPL plan. This hangs a *reduction* off the tasks in question, pointing to a new reducing subtask. When the interpreter is to execute these tasks, it executes the reducing subtask instead. This facility is not yet well developed.

A *policy* is an action defined as a constraint on other actions. [21] An example is the action "Avoid making any noise." For the RPL interpreter, a policy is an action that can fail but whose successful termination is not an end in itself. The RPL construct `(WITH-POLICY p a)` carries out action a after starting action p. If a or p fails, the whole thing fails, but a must succeed for the `WITH-POLICY` to succeed. That is, p is behaving as a policy during the execution of a. When a finishes, the task for p evaporates. One useful construct to serve as such a p is `(WHENEVER c i)`, which executes i whenever fluent c becomes true. This action can never succeed, although it can fail.

An important policy class are protections. [32] A *protection of state P* is the policy of keeping P true. As Firby [8] observes, there are many different policies that might fall under this description. We distinguish three: A "soft" protection is one that is routinely expected to lapse and be restored. A "hard" protection is one that should not lapse, although the plan has resources for restoring it when it does. A "rigid" protection is one that must not lapse; if the planner foresees the violation of a rigid protection, it must take steps to correct it or expect its plan to fail. Historically, most planners have assumed protections to be rigid, and have worked hard to make protection violations impossible.

RPL provides the following construct:

```
(PROTECTION [:RIGID|:HARD|:SOFT]
            state
            fluent
            repair)
```

The *state* is a predicate-calculus pattern summarizing what is protected. This is of use to the projector and planner. The *fluent* is the actual run-time entity whose truth the interpreter cares about. If it should become false, the *repair* is run. If the *repair* fails to make *fluent* true, then the `PROTECTION` fails (and so, presumably, does the plan it occurs in).

3.1 Example

Space does not permit inclusion of a RPL manual, so I will give an annotated example to convey the flavor of the language. The plan library consists of subroutines, defined using `DEF-INTERP-PROC`. The plan in Figure 3 achieves the goal of transporting `OBJ` from location `X1,Y1` to location `X2,Y2`:

3.2 Implementation

The interpreter is run by a "cpu" that keeps track of threads of control. It looks something like a real-time operating system [31], except that we focus on issues of flexibility rather than speed of response. The interpreter takes plan constructions and turns them into control threads. Unlike a traditional operating system, scheduling is done "depth-first" instead of "round-robin." That is, the interpreter stays focused on a particular thread and its successors until interrupted, rather than trying to run all enabled threads in some fair way. It would not be hard to make the scheduling policy be more flexible.

A cpu thread consists of three things: a priority, an aliveness checker, and a continuation. Calling the continuation is supposed to return zero or more threads. The aliveness checker is a function that returns NIL when the thread has died for some reason. One use of this is in `(TRY-ALL` $a_1 \ldots a_N$`)`, where each action is alive only if

```
(DEF-INTERP-PROC TRANSPORT (OBJ X1 Y1 X2 Y2)
    (LET* ((HAND (FREE-HAND))
           (DROPPED-IT '#F)
           (DROP-X 0) (DROP-Y 0))
       ;; First, pick the object up.  FREE-HAND finds a hand that is
       ;; not in use if it can, or picks one randomly.
       (AT-LOCATION X1 Y1
          (ACHIEVE-IN-HAND OBJ))
       ;; Note that the hand is in use,
       (CONCLUDE (IN-USE HAND))
       ;; Request the ''resource'' WHEELS, using variable
       ;; I-HAVE-THE WHEELS to keep track of whether it has been
       ;; seized by some other plan:
       (USING-RESOURCE WHEELS -1
                       I-HAVE-THE-WHEELS
          ;; Go to the new location,
          (PLAN ((TAG DOIT (AT-LOCATION X2 Y2
                                ;; and release the object.
                                (TAG LET-GO (UNHAND HAND))
                                (CONCLUDE (NOT (IN-USE HAND))))))
              ;; However, keep track of whether something disturbs
              ;; the hand en route.
              (POLICY (TASK-BEGIN DOIT)
                      (TASK-BEGIN (TAGGED LET-GO DOIT))
                      (WHENEVER (NOT DROPPED-IT)
                          (SEQ (WAIT-FOR (EMPTY HAND))
                               (CONCLUDE DROPPED-IT)
                               (!= < DROP-X DROP-Y >
                                   (COORDS-HERE)))))
              ;; If something does, pick the object up again.
              (PROTECTION :HARD
                          (TASK-BEGIN DOIT)
                          (TASK-BEGIN (TAGGED LET-GO DOIT))
                          '(TAKING ,OBJ)
                  ;; Don't worry about the protection violation
                  ;; until this plan has the wheels under its control.
                  (NOT (AND I-HAVE-THE-WHEELS
                            DROPPED-IT))
                  (SEQ (!= HAND (FREE-HAND))
                       ;; (A different hand may be used this time.)
                       (AT-LOCATION DROP-X DROP-Y
                           (ACHIEVE-IN-HAND OBJ))
                       (CONCLUDE (IN-USE HAND))
                       (CONCLUDE (NOT DROPPED-IT))))))))
```

Figure 3: A RPL Procedure

the (TRY-ALL ...) hasn't succeeded yet. Another is in FILTER, where the value of the filtering fluent is checked.

The priority of a thread may be controlled by the construct (PRIORITY n -*body*-), which executes *body* with priority n. Lower numbers represent more urgent priorities.

3.3 Perception

Classical planning often seemed to assume that, to enable an object to be manipulated, it needed only to be bound to a predicate-calculus constant. A better way of putting it is that classical planning took no position at all on the question how A was to be found in order to perform (GRASP A). As we move toward more realistic domains, this gap has become more glaring. Revisionists like Agre and Chapman [1, 5] have argued that the classical model is bankrupt, and have diagnosed the underlying illness as reliance on "objective" instead of "deictic" semantics.

Actually, a careful analysis shows there is nothing really wrong with the classical account of the way names work in AI programs. There is no problem executing (GRASP A) if the plan for carrying out a GRASP can look up the coordinate of its argument; for instance, if there's an assertion (COORDS A <X,Y,Z>) stored in the database.

If there isn't, then the agent has to do some work, of the following sort: It scans the place where it expects A to be. Anything in that vicinity that resembles A it takes to be A (assuming there is just one such object). In other words, the vision system must be given a description of A, and must return the scene parsed into "things that look like that" and, at a lower resolution, the background. Each such thing has a new name (what Firby [8] calls a sensor name), complete with all the information the vision system can extract about it, including, let us suppose, its coordinates. So now the agent has an assertion (COORDS OB991 <X,Y,Z>) in its database. The crucial step is to assume A=OB991, so that now it knows (or thinks it knows) the coordinates of A. It can proceed to grasp A, and so forth. (While it is grasping A, the equation "A=object-grasped" can be assumed, and so forth.)

There is a further layer of indirection to be dealt with, however. Consider the following robot plan:

Repeat
    Look for a widget coming down the chute
    Pick it up
    Put it in the bin

Now, the question is, how do we analyze "it" in the "pick it up" step? Classical planning has had remarkably little to say about this question, and has tended to focus on the case where all the objects are "known" beforehand.

If you inquire of roboticists how they handle this situation, you find that they write programs in which there is no reference to widgets at all. Instead, the "look for" step becomes code to scan images for things that look like projections of widgets. The image fragments found are translated into 3-d coordinates or the like, and this information is what gets passed to the "pick it up" step. It's hard to argue with this approach,

The way to modulate this to the classical-representation view is to assume that "it" is a variable. That is, what we have is

X := widget-like thing in chute
pickup X

where X is bound to a thing with properties such as 3-d coordinates, etc. This way of thinking of the situation allows us to tie the plan to robotics while still notating it in the usual compact way. The question is what sort of a "thing" this is. Presumably there is no magic way to guarantee that every time the same object is seen it will be assigned an EQ entity in memory. Firby's research dealt with how to drop that assumption. But once we have dropped it, we're left with the classical theory, pretty much intact. We see that there was no harm after all in saying (GRASP A), just so long as we realized that A might have been freshly consed a millisecond ago by the sensory system.[2]

In our simulated world, we model perception by having a list of objects at each location. There is a primitive operation to scan that list for objects having certain perceptual properties. The objects found are returned, not as pointers, but as descriptions, called *desigs*, including "coordinates," which are simply given as the position of the object in the list. (This is contrived, but picture the world as inhabiting a separate address space, so that it would be impossible to return a pointer.) If new objects arrive at this location, or if the robot moves, the desig can become wrong, but there is no way for the robot to test for that, without comparing the actual object at the coordinate with the properties it expects, which are stored in the desig. Hence if an object is to be manipulated over some stretch of time, then its desig must be "reacquired" when necessary, by the process, described at the beginning of this section, of searching for an object like it and equating the old desig to the new desig for the found object.

4 Transformational Planning

We now return to the topic of planning — how plans get improved by XFRM before being executed. To refresh your memory, the process consists of *projecting* the plan to allow *critics* to foresee problems with it, where the critics propose *transformations* to fix those problems. Many transformations call a *scheduler* to optimize the order of tasks.

4.1 The Projector

The output of the projector is a set of timelines (or "time maps"), each a story about how execution might go. Coupled with each timeline is an elaborated task network that explains which tasks succeeded and which failed. The hope is that in case of projected failure the

[2] Close analysis of the often confusing literature on "deictic" alternatives to the classical view show that they do not differ much in practice from what I am proposing; only they prefer the word "register" to the word "variable."

data gathered during projection can suggest patches to the plan.

Our most elegant and powerful projector was built by Steve Hanks. [15] It generates a *scenario tree* that describes all but the most unlikely ways that a plan could work. A planner can inspect the tree to find hidden disasters as well as most probable outcomes. We are currently working on a simpler cousin of Hanks's projector as a module for the XFRM planner. The original version tends to generate big trees for big plans, because of its commitment to finding every way a plan could be executed, even when the ways don't differ much. An alternative idea is to have the projector generate a random sample of projections. That is, we project the plan, making random choices, then project it again a few times from the beginning, until we have a collection of timelines. The main advantage of this approach is that the planner can quickly predict whether a plan is basically good or bad, because the the first few samples are probably among the most probable. The danger is that improbable catastrophes will be overlooked until their probability has risen.

Plan projection is a good place to apply probability theory. Typically all we need to estimate is the probability that a given state will result from a certain event. It might be thought that keeping track of the interdependencies among these assessments might be impossible, but we can arrange to avoid that work. Whenever the projector estimates the probability of a state at a point in time, it flips a coin based on that probability and actually adds an assumption that the state is true or false from that point on (for some lifetime). Future assessments that are dependent on this state will be affected by the recorded assumption. Hanks's original projector would keep track of both outcomes, splitting the projection into two different scenarios. In our "Monte Carlo" version, we retain just one of them; the other might be generated next time.

For example, if the success of a plan step depends on whether it is raining, and the chance of rain is 20%, then 20% of the time we install the assumption that it is raining; 80%, that it is dry. The subsequent assessment of any other probability that depends upon the chance of rain must then take this assumption into account.

Projection is relatively easy for straight-line sequences of plan steps. But in Section 3 we expanded the scope of plan notations considerably, and the projector must be able to cope with all of these. Our basic approach is to treat the projector as just another interpreter, indeed, just another incarnation of the interpreter, running in *projection mode*. In this mode, instead of dealing with the real world, the planner acts by adding events to the timeline, and senses by querying the timeline. Real time is replaced by "projection time," as recorded in the last event stored in the time map. As new events are added to the growing timeline, projection time marches on. In the dullest case, an event is simply added with a new date, thus simulating the passage of time by the corresponding amount. However, if the world contains autonomous processes and agents, then they might cause events to occur during that time period. In the current system, we simply provide a hook, a procedure `WORLD-PROJECTOR*` that is called whenever an interval passes. It can roll some dice to decide if it rains, notice that sunset has occurred, or do whatever else is necessary to simulate the world. It adds the resulting new events to the timeline before allowing time to proceed.

Because the projector is just the interpreter, it must be able to handle all the variable-binding and setting constructs. The tricky case is a Lisp global variable, e.g., a fluent tickled by a sensor. It would be inappropriate for the projector either to read, set, or destructively alter the *current* value of such a variable midway through a projection. Instead, when the interpreter discovers that a variable has a global binding, it must copy its value and use the copy from then on. Any data type that the projector might encounter must respond to a `COPY` operation,[3] or an error will be generated. Fluents are an example of an easily copied data type. The copy can be named "Copy of fluent so-and-so" so the two can be related when necessary.

4.2 Critics and Transformations

After projecting the current plan revision, critics are run on the resulting scenarios in order to recognize bugs and suggest fixes. The fix suggestions, or *transformations*, are responsible for keeping the plan scheduled.

Let's look at a detailed example, that arises in the course of solving the problem in Figure 1, using (among other plans) the `TRANSPORT` plan of Section 3.1. After the first version of the plan has been scheduled, projection shows that an overload will occur after object C is loaded. In particular, when `FREE-HAND` fails to find a hand that is not in use, it returns a hand that is in use, which gets emptied, triggering a protection violation. (Let me remind you that none of this is really happening, but is just being predicted during projection.) The projection continues, and the plan completes successfully, but the protection was marked `:HARD`, so XFRM notes the violation, and the "overload critic" will try to get rid of it, in order to avoid wasting time going back to get object A after C is unloaded. This critic is called whenever the following constellation of events occurs:

- A protection violation occurs in a protection set up by `TRANSPORT` during the transport of object B_1 to destination D_1
- The violation occurred due to a call to `FREE-HAND` in during the transportation of object B_2 to destination D_2
- The violation occurred at location V

If the distance from V to D_2 is longer than the distance from V to D_1, then the critic recommends that a new ordering be introduced: from the `UNHAND` step for B_1 to the step that picks up B_2. This criterion is fairly arbitrary, but will be supplanted by a more refined estimate when the transformed plan is rescheduled and reprojected. In

[3] This would be a good place to use CLOS, the Common Lisp Object System, but in fact the current implementation does not.

the present case, the resulting plan is a significant improvement.

For such transformations to be expressible, the planner needs a flexible way of talking about plan steps, and the syntactic subtask idea from Section 3 provides one. Every step has a name that reflects its place in the plan, and these names can be used to add order constraints to the top-level `PLAN`.

Another transformation we are working on can be expressed informally as, "If you have lots of deliveries to make in the same area, then consider getting a box to put all the things in." It might be thought that applying this transformation would require editing every pickup action, transforming it to an action to put an object in the box. However, we can avoid that work by rewriting the `TRANSPORT` procedure so that it will use a box if one is at hand. Then transforming the plan will require simply adding steps to acquire a box.

It is not at all clear as yet how many transformations our planner will need. In our contrived delivery domain, we won't need many, but as domains get more complicated the problem of coordinating transformations could become severe.

5 Status and Prospects

The system described here is partially implemented. The interpreter is running, as are several versions of the projector, scheduler, and world simulator. We hope to get a working prototype that includes everything in the next few weeks.

The RPL interpreter has been ported to GE's Corporate Research and Development Laboratory for use in an emergency-advice application. RPL is used to write scripts for advising personnel how to react to an emergency. The transparency of the notation makes it possible to display an informative checklist of actions to be taken.

There are lots of problems left to be solved in the development of XFRM. One is the provision of a formal semantics for RPL. Another is the exploration of ways of projecting and transforming loops. A plan with loops can go on for a long time, generating a long boring timeline. To circumvent this problem, the projector needs to engage in a little *aggregation*, in the phrase of Weld ([34], cf. [10, 11]). That is, having run the loop a couple of times, it should try to summarize what's happening in such a way that it can estimate the number of iterations, and predict in general terms what the world will be like when the loop is done.

One question that needs to be addressed are the criteria for judging this work. Our focus is on notation and architecture rather than on particular schedulers or transformations. The payoff we expect is in the ability to write reactive plans that are easy to understand and modify, that make realistic assumptions about execution platforms, and that support the development of a battery of plan transformations. Hence the work will be successful to the extent that it supports the evolution of a new generation of planners.

Acknowledgements: This work is the result of a collaboration with many students, including Jim Firby, Steve Hanks, Joshy Joseph, and Yuval Shahar. Bruce Pomeroy and Bill Cheetham helped adapt RPL to the emergency-advice domain. My thoughts on perception were honed by electronic correspondence with Phil Agre, who no doubt disagrees with all of them.

References

[1] Philip E. Agre 1989 The dynamic structure of everyday life. MIT AI Lab Report 1085

[2] Philip E. Agre and David Chapman 1987 Pengi: an implementation of a theory of activity. *Proc. AAAI* 6, pp. 268–272

[3] Colin E. Bell and Kwangho Park 1989 Solving resource-constrained project scheduling by A* search. University of Iowa Working paper.

[4] Mark Boddy and Thomas Dean 1989 Solving time-dependent planning problems. *Proc. Ijcai* 11

[5] David Chapman 1990 Vision, instruction, and action. MIT AI Lab Tech Report 1204

[6] Ernest Davis 1984 A high-level real-time programming language. NYU Dept. of CS Report 145 (Robotics Report 36

[7] Marc Donner 1987 *Real-time control of walking* Boston: Birkhäuser

[8] R.J. Firby 1989 *Adaptive Execution in Complex Dynamic Worlds.* Yale University CS Dept. TR 672

[9] Mark S. Fox and Stephen F. Smith 1984 ISIS: a knowledge-based system for factory scheduling. *Expert Systems* 1, no. 1, pp.25–49

[10] Andrew Gelsey 1990 Automated reasoning about machines. Yale Computer Science Department Report 785

[11] Andrew Gelsey and Drew McDermott 1988 Spatial reasoning about mechanisms. Yale CS Department Report YALEU/DCS/RR-641. To appear in *Advances in Spatial Reasoning* 1, Su-Shing Chen, ed. Ablex Publishing.

[12] Michael Georgeff and Amy Lansky 1987 Reactive reasoning and planning. *Proc. AAAI* 7, pp. 677–682

[13] Kristian Hammond 1988 *Case-based Planning: An Integrated Theory of Planning, Learning, and Memory.* New York: Academic Press.

[14] Kristian Hammond, Timothy Converse, and Charles Martin 1990 Integrating planning and acting in a case-based framework. *Proc. AAAI* 8, pp. 292–297

[15] Steven Hanks 1990 *Projecting Plans for Uncertain Worlds.* Yale Yale Computer Science Department Technical Report 756.

[16] Steven Hanks 1990 Practical temporal projection. *Proc. AAAI* 8

[17] Jerry Hobbs and Robert C. Moore (eds.) 1985 *Formal Theories of the Commonsense World*, Ablex Publishing Corporation

[18] Eric Horvitz 1988 Reasoning under varying and uncertain resource constraints. *Proc. AAAI* **7**, pp. 111–116

[19] Eric Horvitz, G.F. Cooper, and D.E. Heckerman 1989 Reflection and action under scarce resources: theoretical principles and empirical study. *Proc. Ijcai* **11**, pp. 1121–1127

[20] Leslie Pack Kaelbling and Stanley J. Rosenschein 1990 Action and planning in embedded agents. In Patti Maes (ed.) *New Architectures for Autonomous Agents: Task-level Decomposition and Emergent Functionality*, Cambridge: MIT Press

[21] Drew McDermott 1978 Planning and acting, *Cognitive Science* **2**, no. 2, pp. 71–109

[22] Drew McDermott 1985 Reasoning about plans. In [17], pp. 269–317

[23] Drew McDermott 1989 Regression planning. Yale Computer Science Report 752. To appear (with revisions) in *Int. J. of Intelligent Sys.* 1990

[24] David Miller 1985 *Planning by Search through Simulations.* Yale Computer Science Department Tech Report 423.

[25] David Miller 1988 A task and resource scheduling system for automated planning. *Annals of Operations Res.* **12**, pp. 169–198

[26] Tom M. Mitchell 1990 Becoming increasingly reactive. *Proc. AAAI* **8**, pp. 1051–1058

[27] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis II 1977 An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.* **6**, no. 3, pp. 563–581

[28] Earl Sacerdoti 1977 *A Structure for Plans and Behavior.* American Elsevier Publishing Company, Inc.

[29] Reid Gordon Simmons 1988 *Combining Associational and Causal Reasoning to Solve Interpretation and Planning Problems.* MIT AI Laboratory TR 1048.

[30] Reid Gordon Simmons 1988 A theory of debugging plans and interpretations. *Proc. AAAI* **7**, pp. 94–99.

[31] David B. Stewart, Donald E. Schmitz, and Pradeep K. Khosla 1990 Implementing real-time robotic systems using CHIMERA II. *Proc. IEEE Int. Conf. on Robotics and Automation*, Cincinatti

[32] Gerald J. Sussman 1975 *A Computer Model of Skill Acquisition.* American Elsevier Publishing Company

[33] Jennifer Turney and Alberto Segre 1989 A framework for learning in planning domains with uncertainty. Cornell Department of Computer Science Report 89-1009

[34] Daniel Weld 1986 The use of aggregation in causal simulation. *Artificial Intelligence* **30**, no. 1, pp. 1–34

[35] Robert Wilensky 1983 *Planning and Understanding.* Reading, Mass.: Addison-Wesley

Becoming Increasingly Reactive

Tom M. Mitchell

School of Computer Science[1]
Carnegie Mellon University
Pittsburgh, PA 15213
Tom.Mitchell@cs.cmu.edu

Abstract

We describe a robot control architecture which combines a stimulus-response subsystem for rapid reaction, with a search-based planner for handling unanticipated situations. The robot agent continually chooses which action it is to perform, using the stimulus-response subsystem when possible, and falling back on the planning subsystem when necessary. Whenever it is forced to plan, it applies an explanation-based learning mechanism to formulate a new stimulus-response rule to cover this new situation and others similar to it. With experience, the agent becomes increasingly reactive as its learning component acquires new stimulus-response rules that eliminate the need for planning in similar subsequent situations. This Theo-Agent architecture is described, and results are presented demonstrating its ability to reduce routine reaction time for a simple mobile robot from minutes to under a second.

1. Introduction and Motivation

Much attention has focused recently on *reactive* architectures for robotic agents that continually sense their environment and compute appropriate reactions to their sense stimuli within bounded time (e.g., [Brooks 86, Agre and Chapman 87, Rosenschein 85]). Such architectures offer advantages over more traditional open-loop search-based planning systems because they can react more quickly to changes to their environment, and because they can operate more robustly in worlds that are difficult to model in advance. *Search-based* planning architectures, on the other hand, offer the advantage of more general-purpose (if slower) problem solving mechanisms which provide the flexibility to deal with a more diverse set of unanticipated goals and situations.

This paper considers the question of how to combine the benefits of reactive and search-based architectures for controlling autonomous agents. We describe the Theo-Agent architecture, which incorporates both a reactive component and a search-based planning component. The fundamental design principle of the Theo-Agent is that it reacts when it can, plans when it must, and learns by augmenting its reactive component whenever it is forced to plan. When used to control a laboratory mobile robot, the Theo-Agent in simple cases learns to reduce its reaction time for new tasks from several minutes to less than a second.

The research reported here is part of our larger effort toward developing a general-purpose learning robot architecture, and builds on earlier work described in [Blythe and Mitchell 89]. We believe that in order to become increasingly successful, a learning robot will have to incorporate several types of learning:

- It must become *increasingly correct* at predicting the effects of its actions in the world.
- It must become *increasingly reactive*, by reducing the time required for it to make rational choices; that is, the time required to choose actions consistent with the above predictions and its goals.
- It must become *increasingly perceptive* at distinguishing those features of its world that impact its success.

This paper focuses on the second of these types of learning. We describe how the Theo-Agent increases the scope of situations for which it can quickly make rational decisions, by adding new stimulus-response rules whenever it is forced to plan for a situation outside the current scope of its reactive component. Its explanation-based learning mechanism produces rules that recommend precisely the same action as recommended by the slower planner, in exactly those situations in which the same plan rationale would apply. However, the learned rules infer the desired action immediately from the input sense data in a single inference step--without considering explicitly the robot's goals, available actions, or their predicted consequences.

1.1. Related Work

There has been a great deal of recent work on architectures for robot control which continually sense the environment and operate in bounded time (e.g., [Brooks 86, Schoppers 87, Agre and Chapman 87]), though this

[1]This is a reprint of a paper which appeared in the *Proceedings of the 1990 AAAI Conference*, August 1990, Boston.

type of work has not directly addressed issues of learning. Segre's ARMS system [Segre 88] applies explanation-based learning to acquire planning tactics for a simulated hand-eye system, and Laird's RoboSoar [Laird and Rosenbloom 90] has been applied to simple problems in a real hand-eye robot system. While these researchers share our goal of developing systems that are increasingly reactive, the underlying architectures vary significantly in the form of the knowledge being learned, underlying representations, and real response time. Sutton has proposed an inductive approach to acquiring robot control strategies, in his DYNA system [Sutton 90], and Pommerleau has developed a connectionist system which learns to control an outdoor road-following vehicle [Pommerleau 89]. In addition to work on learning such robot control strategies, there has been much recent interest in robot learning more generally, including work on learning increasingly correct models of actions [Christiansen, et al. 90, Zrimic and Mowforth 88], and work on becoming increasingly perceptive [Tan 90].

The work reported here is also somewhat related to recent ideas for compiling low-level reactive systems from high-level specifications (e.g., [Rosenschein 85]). In particular, such compilation transforms input descriptions of actions and goals into effective control strategies, using transformations similar to those achieved by explanation-based learning in the Theo-Agent. The main difference between such design-time compilation and the explanation-based learning used in the Theo-Agent, is that for the Theo-Agent learning occurs incrementally and spread across the lifetime of the agent, so that the compilation transformation is incrementally focused by the worlds actually encountered by the agent, and may be interleaved with other learning mechanisms which improve the agent's models of its actions.

The next section of this paper describes the Theo-Agent architecture in greater detail. The subsequent section presents an example of its use in controlling a simple mobile robot, the learning mechanism for acquiring new stimulus-response rules, and timing data showing the effect of caching and rule learning on system reaction time. The final section summarizes some of the lessons of this work, including features and bugs in the current design of the architecture.

2. The Theo-Agent Architecture

The design of the Theo-Agent architecture is primarily driven by the goal of combining the complementary advantages of reactive and search-based systems. Reactive systems offer the advantage of quick response. Search-based planners offer the advantage of broad scope for handling a more diverse range of unanticipated worlds. The Theo-Agent architecture employs both, and uses explanation-based learning to incrementally augment its reactive component whenever forced to plan. In addition, the architecture makes widespread use of caching and dependency maintenance in order to avoid needless recomputation of repeatedly accessed beliefs. The primary characteristics of the Theo-Agent are:

- It continually reassesses what action it should perform. The agent runs in a tight loop in which it repeatedly updates its sensor inputs, chooses a control action, begins executing it, then repeats this loop.
- It reacts when it can, and plans when it must. Whenever it must choose an action, the system consults a set of stimulus-response rules which constitute its reactive component. If one of these rules applies to the current sensed inputs, then the corresponding action is taken. If no rules apply, then the planner is invoked to determine an appropriate action.
- Whenever forced to plan, it acquires a new stimulus-response rule. The new rule recommends the action which the planner has recommended, in the same situations (i.e., those world states for which the same plan justification would apply), but can be invoked much more efficiently. Learning is accomplished by an explanation-based learning algorithm (EBG [Mitchell, et al 86]), and provides a demand-driven incremental compilation of the planner's knowledge into an equivalent reactive strategy, guided by the agent's experiences.
- Every belief that depends on sensory input is maintained as long as its explanation remains valid. Many beliefs in the Theo-Agent, including its belief of which action to perform next, depend directly or indirectly on observed sense data. The architecture maintains a network of explanations for every belief of the agent, and deletes beliefs only when their support ceases. This caching of beliefs significantly improves the response time of the agent by eliminating recomputation of beliefs in the face of unchanging or irrelevant sensor inputs.
- It determines which goal to attend to, based on the perceived world state, a predefined set of goal activation and satisfaction conditions, and given priorities among goals.

Internal structure of agent: A Theo-Agent is defined by a frame structure whose slots, subslots, subsubslots, etc. define the agent's beliefs, or internal state[2]. The two most significant slots of the agent are Chosen.Action, which describes the action the agent presently chooses to perform; and Observed.World, which describes the agent's current perception of its world. As indicated in Figure 2-1

[2]The Theo-Agent is implemented on top of a generic frame-based problem solving and learning system called Theo [Mitchell, et al. 90], which provides the inference, representation, dependency maintenance, and learning mechanisms.

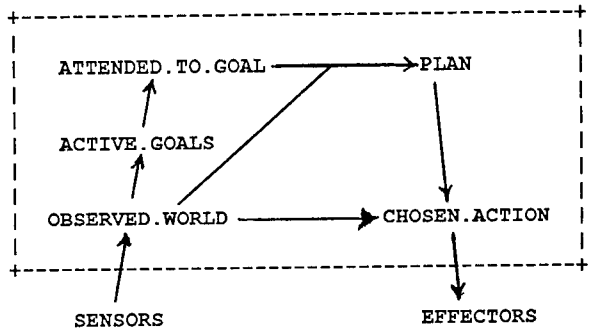


Figure 2-1: Data Flow in a Theo-Agent

the agent may infer its Chosen.Action either directly from its Observed.World, or alternatively from its current Plan. Its Plan is in turn derived from its Observed.World and Attended.To.Goal. The Attended.To.Goal defines the goal the agent is currently attempting to achieve, and is computed as the highest priority of its Active.Goals, which are themselves inferred from the Observed.World.

Agent goals: Goals are specified to the agent by defining conditions under which they are active, satisfied, and attended to. For example, an agent may be given a goal Recharge.Battery which is defined to become active when it perceives its battery level to be less than 75%, becomes satisfied when the battery charge is 100%, and which is attended to whenever it is active and the (higher priority) goal Avoid.Oncoming.Obstacle is inactive.

Caching policy: The basic operation of the Theo-Agent is to repeatedly infer a value for its Chosen.Action slot. Each slot of the agent typically has one or more attached procedures for obtaining a value upon demand. These procedures typically access other slots, backchaining eventually to queries to slots of the Observed.World. Whenever some slot value is successfully inferred, this value is cached (stored) in the corresponding slot, along with an explanation justifying its value in terms of other slot values, which are in turn justified in terms of others, leading eventually to values of individual features in the Observed.World, which are themselves inferred by directly accessing the robot sensors. Values remain cached for as long as their explanations remain valid. Thus, the agent's Active.Goals and Chosen.Action may remain cached for many cycles, despite irrelevant changes in sensor inputs. This policy of always caching values, deleting them immediately when explanations become invalid, and lazily recomputing upon demand, assures that the agent's beliefs adapt quickly to changes in its input senses, without needless recomputation.

Control policy: The Theo-Agent is controlled by executing the following loop:

Do Forever:

1. **Sense** and update readings for all *eagerly sensed* features of Observed.World, and delete any cached values for *lazily sensed* features.
2. **Decide** upon the current Chosen.Action
3. **Execute** the Chosen.Action

When the Chosen.Action slot is accessed (during the decision portion of the above cycle), the following steps are attempted in sequence until one succeeds:

1. Retrieve the cached value of this slot (if available)
2. Infer a value based on the available stimulus-response rules
3. Select the first step of the agent's Plan (inferring a plan if necessary)
4. Select the default action (e.g., WAIT)

Sensing policy: Each primitive sensed input (e.g., an array of input sonar data) is stored in some slot of the agent's Observed.World. Higher level features such as edges in the sonar array, regions, region width, etc., are represented by values of other slots of the Observed.World, and are inferred upon demand from the lower-level features. The decision-making portions of the agent draw upon the entire range of low to high level sensory features as needed. In order to deal with a variety of sensing procedures of varying cost, the Theo-Agent distinguishes between two types of primitive sensed features: those which it *eagerly* senses, and those which it *lazily* senses. Eagerly sensed features are refreshed automatically during each cycle through the agent's main loop, so that dependent cached beliefs of the agent are retained when possible. In contrast, lazily sensed features are simply deleted during each cycle. They are recomputed only if the agent queries the corresponding slot during some subsequent cycle. This division between eagerly and lazily refreshed features provides a simple focus of attention which allows keeping the overhead of collecting new sense data during each cycle to a minimum.

Learning policy: Whenever the agent is forced to plan in order to obtain a value for its Chosen.Action, it invokes its explanation-based generalization routine to acquire a new stimulus-response rule to cover this situation. The details of this routine are described in greater detail in the next section. The effect of this learning policy is to incrementally extend the scope of the set of stimulus-response rules to fit the types of problem instances encountered by the system in its world.

3. Example and Results

This section describes the use of the Theo-Agent architecture to develop a simple program to control a Hero 2000 mobile robot to search the laboratory to locate garbage cans[3]. In particular, we illustrate how goals and actions are provided to the robot with no initial stimulus-

[3]A detailed description of the modified Hero 2000 robot used here is available in [Lin, et al. 89].

response rules, how it initially selects actions by constructing plans, and how it incrementally accumulates stimulus-response rules that cover its routine actions.

The robot sensors used in this example include an ultrasonic sonar mounted on its hand, a rotating sonar on its head, and a battery voltage sensor. By rotating its hand and head sonars it is able to obtain arrays of sonar readings that measure echo distance versus rotation angle. These raw sonar readings are interpreted (on demand) to locate edges in the sonar array, as well as regions, and properties of regions such as region width, distance, direction, and identity. The primitive sensing operations used in the present example include Battery, which indicates the battery voltage level, Sonarw, which measures sonar range with the wrist sonar pointed directly forward, and Sweep.Wrist.Roll, which obtains an array of sonar readings by rotating the wrist from left to right. Of these sensed features, Sonarw is eagerly sensed, and the others are lazily sensed.

The robot actions here include Forward.10 (move forward 10 inches), Backward.10 (move backward 10 inches), Face.The.Object (turn toward the closest sonar region in front of the robot), and Measure.The.Object (obtain several additional sonar sweeps to determine whether the closest sonar region in front of the robot is a garbage can). The.Object refers to the closest sonar region in front of the robot, as detected by the sense procedure Sweep.Wrist.Roll.

This Theo-Agent has been tested by giving it different sets of initial goals, leading it to compile out different sets of stimulus-response rules exhibiting different behaviors. In the simple example presented here, the agent is given three goals:

- Goal.Closer: approach distant objects. This goal is activated when the Sonarw sense reading is between 25 and 100 inches, indicating an object at that distance. It is satisfied when Sonarw is less that 25 inches, and attended to whenever it is active.
- Goal.Further: back off from close objects. This is activated when Sonarw is between 3 and 15 inches, satisfied when Sonarw is greater than 15 inches, and attended to whenever it is active.
- Goal.Identify.The.Object: determine whether the nearest sonar region corresponds to a garbage can. This is activated when there is an object in front of the robot whose identity is unknown, satisfied when the object identity is known, and attended to whenever it is active and Goal.Closer and Goal.Further are inactive.

In order to illustrate the operation of the Theo-Agent, consider the sequence of events that results from setting the robot loose in the lab with the above goals, actions, and sensing routines: During the first iteration through its sense-decide-execute loop, it (eagerly) senses a reading of 41.5 from Sonarw, reflecting an object at 41.5 inches. In the decide phase of this cycle it then queries its Chosen.Action slot, which has no cached value, and no associated stimulus-response rules. Thus, it is forced to plan in order to determine a value for Chosen.Action. When queried, the planner determines which goal the agent is attending to, then searches for a sequence of actions which it projects will satisfy this goal. Thus, the planner queries the Attending.To.Goal slot, which queries the Active.Goals slots, which query the Observed.World, leading eventually to determining that the Attending.To.Goal is Goal.Closer. The planner, after some search, then derives a two-step plan to execute Forward.10 two times in a row (this plan leads to a projected sonar reading of 21.5 inches, which would satisfy Goal.Closer). The inferred value for the Chosen.Action slot is thus Forward.10 (the first step of the inferred plan).

The agent caches the result of each of the above slot queries, along with an explanation that justifies each slot value in terms of the values from which it was derived. This network of explanations relates each belief (slot value) of the agent eventually to sensed features of its Observed.World.

In the above scenario the agent had to construct a plan in order to infer its Chosen.Action. Therefore, it formulates a new stimulus-response rule which will recommend this chosen action in future situations, without planning. The agent then executes the action and begins a new cycle by eagerly refreshing the Sonarw feature and deleting any other sensed features (in this case the observed Battery level, which was queried by the planner as it checked the preconditions for various actions). During this second cycle, the stimulus-response rule learned during the first cycle applies, and the agent quickly decides that the appropriate Chosen.Action in the new situation is to execute Forward.10. As it gains experience, the agent acquires additional rules and an increasing proportion of its decisions are made by invoking these stimulus-response rules rather than planning.

3.1. Rule Learning

The rule acquisition procedure used by the Theo-Agent is an explanation-based learning algorithm called EBG [Mitchell, et al 86]. This procedure explains why the Chosen.Action of the Theo-Agent is justified, finds the weakest conditions under which this explanation holds, and then produces a rule that recommends the Chosen.Action under just these conditions. More precisely, given some Chosen.Action, ?Action, the Theo-Agent explains why ?Action satisfies the following property:

Justified.Action(?Agent, ?Action) ←
- (1) The Attending.To.Goal of the ?Agent is ?G
- (2) ?G is Satisfied by result of ?Agent's plan
- (3) The tail of ?Agent's plan will not succeed without first executing ?Action
- (4) ?Action is the first step of the ?Agent's plan

```
(hero justified.action) = face.the.object
 <--prolog--
 (hero attending.to.goals) = goal.identify.object
  <--prolog--
  (hero monitored.goals) = goal.identify.object
  (hero goal.identify.object attending.to?) = t
   <--prolog--
   (hero goal.identify.object active?) = t
    <--prolog--
    (hero observed.world) = w0
    (w0 the.object identity known?) = nil
  (hero goal.closer active?) = nil
   <--prolog--
   (hero observed.world) = w0
   (w0 sonarw) = 22.5
  (hero goal.further active?) = nil
   <--prolog--
   (hero observed.world) = w0
   (w0 sonarw) = 22.5
 (world376 goal.identify.object wsatisfied?) = t
  <--prolog--
  (world376 the.object identity known?) = t
   <--expected.value--
   (world376 previous.state) = world159
   (world159 measure.the.object prec.sat?) = t
    <--prolog--
    (world159 battery) = 100
     <--expected.value--
     (world159 previous.state) = w0
     (w0 battery) = 100
      <--observed.value--
      (w0 battery observed.value) = 100
    (world159 the.object distance) = 22
     <--expected.value--
     (world159 previous.state) = w0
     (w0 face.the.object prec.sat?) = t
      <--prolog--
      (w0 battery) = 100
       <--observed.value--
       (w0 battery observed.value) = 100
      (w0 the.object direction known?) = t
     (w0 the.object distance) = 22
      <--observed.value--
      (w0 the.object distance
                    observed.value) = 22
    (world159 the.object direction) = 0
     <--expected.value--
     (world159 previous.state) = w0
     (w0 face.the.object prec.sat?) = t
      <--prolog--
      (w0 battery) = 100
       <--observed.value--
       (w0 battery observed.value) = 100
      (w0 the.object direction known?) = t
 (w0 measure.the.object prec.sat?) = nil
  <--prolog--
  (w0 the.object direction) = 10
   <--observed.value--
   (w0 the.object direction observed.value) = 10
```

Figure 3-1: Explanation for (Hero Justified.Action) = Face.The.Object

EBG constructs an explanation of why the Chosen.Action is a Justified.Action as defined above, then determines the weakest conditions on the Observed.World under which this explanation will hold[4]. Consider, for example, a scenario in which the Hero agent is attending to the goal Goal.Identify.The.Object, and has constructed a two-step plan: Face.The.Object, followed by Measure.The.Object. Figure 3-1 shows the explanation generated by the system for why Face.The.Object is its Justified.Action. In this figure, each line corresponds to some belief of the agent, and level of indentation reflects dependency. Each belief is written in the form (frame slot subslot subsubslot ...)=value, and arrows such as "<-- observed.value--" indicate how the belief above and left of the arrow was inferred from the beliefs below and to its right. For example, the leftmost belief that the Hero's Justified.Action is Face.The.Object, is supported by the three next leftmost beliefs that (1) the (Hero Attending.To.Goals)=Goal.Identify.Object, (2) the (World376 Goal.Identify.Object Satisfied?)=t, and (3) (W0 Measure.The.Object Prec.Sat?)=nil. W0 is the current Observed.World, World376 is the world state which is predicted to result from the agent's plan, and Prec.Sat? is the predicate indicating whether the preconditions of an action are satisfied in a given world state. These three supporting beliefs correspond to the first three clauses in the above definition of Justified.Action[5]. Notice the third clause indicates that in this case the tail of the agent's plan cannot succeed since the preconditions of the second step of the plan are not satisfied in the initial observed world.

```
IF
 (1) Identity of The.Object in Observed.World
       is not Known
 (1) Sonarw in Observed.World = ?s
 (1) Not [3 < ?s < 15]
 (1) Not [25 < ?s < 100]
 (2) Battery in Observed.World > 70
 (2) Distance to The.Object in Observed.World
       = ?dist
 (2) 15 <= ?dist <= 25
 (2,3) Direction to The.Object in Observed.World
       = ?dir
 (3) Not [-5 <= ?dir <= 5]

THEN
     Chosen.Action of Hero = Face.The.Object
```

Figure 3-2: Rule for Explanation from Figure 3-1

[4]Notice that the third clause in the definition of Justified.Action requires that the first step of the plan be essential to the plan's success. Without this requirement, the definition is too weak, and can produce rules that recommend non-essential actions such as WAIT, provided they can be followed by other actions that eventually achieve the goal.

[5]The fourth clause is not even made explicit, since this is satisfied by defining the rule postcondition to recommend the current action.

Figure 3-2 shows the english description of the rule produced by the Theo-Agent from the explanation of Figure 3-1. The number to the left of each rule precondition indicates the corresponding clause of Justified.Action which is supported by this precondition. For example, the first four lines in the rule assure that the robot is in a world state for which it should attend to the goal Goal.Identify.Object (i.e., they assure that this goal will be active, and that all higher priority goals will be inactive). Of course this rule need not explicitly mention this goal or any other, since it instead mentions the observed sense features which imply the activation of the relevant goals. Similarly, the rule need not mention the plan, since it instead mentions those conditions, labeled (2) and (3), which assure that the first step of the plan will lead eventually to achieving the desired goal.

In all, the agent typically learns from five to fifteen stimulus-response rules for this set of goals and actions, depending on its specific experiences and the sequence in which they are encountered. By adding and removing other goals and actions, other agents can be specified that will "compile out" into sets of stimulus-response rules that produce different behaviors.

3.2. Impact of Experience on Agent Reaction Time

With experience, the typical reaction time of the Theo-Agent in the above scenario drops from a few minutes to under a second, due to its acquisition of stimulus-response rules and its caching of beliefs. Let us define *reaction time* as the time required for a single iteration of the sense-decide-execute loop of the agent. Similarly, define *sensing time*, *decision time*, and *execution time* as the time required for the corresponding portions of this cycle. Decision time is reduced by two factors:

- Acquisition of stimulus-response rules. Matching a stimulus-response rule requires on the order of ten milliseconds, whereas planning typically requires several minutes.
- Caching of beliefs about future world states. The time required by planning is reduced as a result of caching all agent beliefs. In particular, the descriptions of future world states considered by the planner (e.g., "the wrist sonar reading in the world that will result from applying the action Forward.10 to the current Observed.World") are cached, and remain as beliefs of the agent even after its sensed world is updated. Some cached features of this imagined future world may become uncached each cycle as old sensed values are replaced by newer ones, but others tend to remain.

The improvement in agent reaction time is summarized in the timing data from a typical scenario, shown in table 3-1. The first line shows decision time and total reaction time for a sense-decide-execute cycle in which a plan must be created. Notice that here decision time constitutes the bulk of reaction time. The second line of this table shows

	Decision Time	Reaction Time
1. Construct simple plan:	34.3 sec	36.8 sec
2. Construct similar plan:	5.5 sec	6.4 sec
3. Apply learned rules:	0.2 sec	0.9 sec

Table 3-1: Effect of Learning on Agent Response Time

(Timings are in CommonLisp on a Sun3 workstation)

the cost of producing a very similar plan on the next cycle. The speedup over the first line is due to the use of slot values which were cached during the first planning episode, and whose explanations remain valid through the second cycle. The third line shows the timing for a third cycle in which the agent applied a set of learned stimulus-response rules to determine the same action. Here, decision time (200 msec.) is comparable to sensing time (500 msec) and the time to initiate execution of the robot action (200 msec.), so that decision time no longer constitutes the bulk of overall reaction time. The decision time is found empirically to require 80 + 14r msec. to test a set of r stimulus-response rules[6].

Of course the specific timing figures above are dependent on the particular agent goals, sensors, training experience, actions, etc. Scaling to more complex agents that require hundreds or thousands of stimulus-response rules, rather than ten, is likely to require more sophisticated methods for encoding and indexing the learned stimulus-response pairings. Approaches such as Rete matching, or encoding stimulus-response pairings in some type of network [Rosenschein 85, Brooks 86] may be important for scaling to larger systems. At present, the significant result reported here is simply the existence proof that the learning mechanisms employed in the Theo-Agent are sufficient to reduce decision time by two orders of magnitude for a real robot with fairly simple goals, so that decision time ceases to dominate overall reaction time of the agent.

4. Summary, Limitations and Future Work

The key design features of the Theo-Agent are:

- A stimulus-response system combined with a planning component of broader scope but slower response time. This combination allows quick response for routine situations, plus flexibility to plan when novel situations are encountered.
- Explanation-based learning mechanism for

[6]Rules are simply tested in sequence with no sophisticated indexing or parallel match algorithms.

incrementally augmenting the stimulus-response component of the system. When forced to plan, the agent formulates new stimulus-response rules that produce precisely the same decision as the current plan, in precisely the same situations.

- The agent chooses its own goals based on the sensed world state, goal activation conditions and relative goal priorities. Goals are explicitly considered by the agent *only* when it must construct plans. As the number of learned stimulus-response rules grows, the frequency with which the agent explicitly considers its goals decreases.
- Caching and dependency maintenance for all beliefs of the agent. Every belief of the agent is cached along with an explanation that indicates those beliefs on which it depends. Whenever the agent sense inputs change, dependent beliefs which are affected are deleted, to be recomputed if and when they are subsequently queried.
- Distinction between eagerly and lazily refreshed sense features. In order to minimize the lower bound on reaction time, selected sense features are eagerly updated during each agent cycle. Other features are lazily updated by deleting them and recomputing them if and when they are required. This provides a simple focus of attention mechanism that helps minimize response time. In the future, we hope to allow the agent to dynamically control the assignment of eagerly and lazily sensed features.

There are several reasonable criticisms of the current TheoAgent architecture, which indicate its current limitations. Among these are:

- The kind of planning the TheoAgent performs may be unrealistically difficult in many situations, due to lack of knowledge about the world, the likely effects of the agent's actions, or other changes in the world. One possible response to this limitation is to add new decision-making mechanisms beyond the current planner and stimulus-response system. For example, one could imagine a decision-maker with an evaluation function over world states, which evaluates actions based on one-step lookahead (similar to that proposed in Sutton's DYNA [Sutton 90].). As suggested in [Kaelbling 86], a spectrum of multiple-decision makers could trade off response speed for correctness. However, learning mechanisms such as those used here might still compile stimulus-response rules from the decisions produced by this spectrum of decision-makers.
- Although the TheoAgent learns to become increasingly reactive, its decisions do not become increasingly correct. The acquired stimulus-response rules are only as good as the planner and action models from which they are compiled. We are interested in extending the system to allow it to inductively learn better models of the effects of its actions, as a result of its experience. Preliminary results with this kind of learning using a hand-eye robot are described in [Christiansen, et al. 90, Zrimic and Mowforth 88].
- The current planner considers the correctness of its plans, but not the cost of sensing or effector commands. Therefore, the plans and the stimulus-response rules derived from them may refer to sense features which are quite expensive to obtain, and which contribute in only minor ways to successful behavior. For instance, in order to guarantee correctness of a plan to pick up a cup, it might be necessary to verify that the cup is not glued to the floor. The current system would include such a test in the stimulus-response rule that recommends the grasp operation, provided this feature was considered by the planner. We must find a way to allow the agent to choose which tests are necessary and which can be ignored in order to construct plausible plans that it can then attempt, and recover from as needed.
- Scaling issues. As noted in the previous section, the current robot system requires only a small set of stimulus-response rules to govern its behavior. We must consider how the approach can be scaled to more complex situations. Some avenues are to (1) explore other strategies for indexing learned knowledge (e.g., index rules by goal, so that many subsets of rules are stored rather than a single set), (2) develop a more selective strategy for invoking learning only when the benefits outweigh the costs, and (3) consider representations of the control function that sacrifice expressive precision for fixed computational cost (e.g., fixed topology neural networks with constant response time).

We believe the notion of incrementally compiling reactive systems from more general but slower search-based systems is an important approach toward extending the flexibility of robotic systems while still achieving respectable (asymptotic) response times. The specific design of the Theo-Agent illustrates one way to organize such a system. Our intent is to extend the current architecture by adding new learning mechanisms that will allow it to improve the correctness of its action models and its abilities to usefully perceive its world. These additional learning capabilities are intended to complement the type of learning presented here.

Acknowledgements. This work is based on extensions to earlier joint work with Jim Blythe, reported in [Blythe and Mitchell 89]. I am most grateful for Jim's significant contributions to the design of the Theo-Agent. Thanks

also to the entire Theo group, which produced the Theo system on which Theo-Agent is built. Theo provides the underlying inference, representation, and learning mechanisms used by the Theo-Agent. Finally, thanks to Long-Ji Lin who developed a number of the routines for interfacing from workstations to the robot. This research is supported by DARPA under research contract N00014-85-K-0116 and by NASA under research contract NAGW-1175.

References

[Agre and Chapman 87]
Agre, P. and Chapman, D.
Pengi: An Implementation of a Theory of Activity.
In *Proceedings of the National Conference on Artificial Intelligence*, pages 268-272. Morgan Kaufmann, July, 1987.

[Blythe and Mitchell 89]
Blythe, J., and Mitchell, T.
On Becoming Reactive.
In *Proceedings of the Sixth International Machine Learning Workshop*, pages 255-259. Morgan Kaufmann, June, 1989.

[Brooks 86] Brooks, R.A.
A Robust Layered Control System for a Mobile Robot.
IEEE Journal of Robotics and Automation 2(1), March, 1986.

[Christiansen, et al. 90]
Christiansen, A., Mason, M., and Mitchell, T.
Learning Reliable Manipulation Strategies without Initial Physical Models.
In *Proceedings of the IEEE International Conference on Robotics and Automation*. IEEE Press, May, 1990.

[Kaelbling 86] Kaelbling, L.P.
An Architecture for Intelligent Reactive Systems.
In M.P. Georgeff and A.L. Lansky (editor), *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*. Morgan Kaufmann , 1986.

[Laird and Rosenbloom 90]
Laird, J.E. and Rosenbloom, P.S.
Integrating Planning, Execution, and Learning in Soar for External Environments.
In *Proceedings of AAAI '90*. AAAI, 1990.

[Lin, et al. 89] Lin, L., Philips, A., Mitchell, T., and Simmons, R.
A Case Study in Robot Exploration.
Robotics Institute Technical Report CMU-RI-89-001, Carnegie Mellon University, Robotics Institute, January, 1989.

[Mitchell, et al 86]Mitchell, T.M., Keller, R.K., and Kedar-Cabelli, S.
Explanation-Based Generalization: A Unifying View.
Machine Learning 1(1), 1986.

[Mitchell, et al. 90]
Mitchell, T. M., J. Allen, P. Chalasani, J. Cheng, O. Etzioni, M. Ringuette, and J. Schlimmer.
Theo: A Framework for Self-improving Systems.
In VanLehn, K. (editor), *Architectures for Intelligence*. Erlbaum, 1990.

[Pommerleau 89] Pommerleau, D.A.
ALVINN: An Autonomous Land Vehicle In a Neural Network.
In Touretzky, D. (editor), *Advances in Nerual Information Processing Systems, Vol. 1*. Morgan Kaufmann, 1989.

[Rosenschein 85] Rosenschein, S.
Formal Theories of Knowledge in AI and Robotics.
New Generation Computing 3:345-357, 1985.

[Schoppers 87] Schoppers, M.J.
Universal Plans for Reactive Robots in Unpredictable Environments.
In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039-1046. AAAI, August, 1987.

[Segre 88] Segre, A.M.
Machine Learning of Robot Assembly Plans.
Kluwer Academic Press, 1988.

[Sutton 90] Sutton, R.
First Results with DYNA, an Integrated Architecture for Learning, Planning, and Reacting.
In *Proceedings of AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments*, pages 136-140. AAAI, March, 1990.

[Tan 90] Tan, M.
CSL: A Cost-Sensitive Learning System for Sensing and Grasping Objects.
In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*. IEEE, May, 1990.

[Zrimic and Mowforth 88]
Zrimic, T., and Mowforth, P.
An Experiment in Generating Deep Knowledge for Robots.
In *Proceedings of the Conference on Representation and Reasoning in an Autonomous Agent*. 1988.

A Preliminary Analysis of the Soar Architecture as a Basis for General Intelligence* ,† ,‡

Paul S. Rosenbloom§
Information Sciences Institute
University of Southern California

John E. Laird
Department of Electrical Engineering and Computer Science
The University of Michigan

Allen Newell
School of Computer Science
Carnegie Mellon University

Robert McCarl
Department of Electrical Engineering and Computer Science
The University of Michigan

Abstract

In this article we take a step towards providing an analysis of the Soar architecture as a basis for general intelligence. Included are discussions of the basic assumptions underlying the development of Soar, a description of Soar cast in terms of the theoretical idea of multiple levels of description, an example of Soar performing multi-column subtraction, and three analyses of Soar: its natural tasks, the sources of its power, and its scope and limits.

The central scientific problem of artificial intelligence (AI) is to understand what constitutes intelligent action and what processing organizations are capable of such action. Human intelligence — which stands before us like a holy grail — shows to first observation what can only be termed *general intelligence*. A single human exhibits a bewildering diversity of intelligent behavior. The types of goals that humans can set for themselves or accept from the environment seem boundless. Further observation, of course, shows limits to this capacity in any individual — problems range from easy to hard, and problems can always be found that are too hard to be solved. But the general point is still compelling.

Work in AI has already contributed substantially to our knowledge of what functions are required to produce general intelligence. There is substantial, though certainly not unanimous, agreement about some functions that need to be supported: symbols and goal structures, for example. Less agreement exists about what mechanisms are appropriate to support these functions, in large part because such matters depend strongly on the rest of the system and on cost-benefit tradeoffs. Much of this work has been done under the rubric of AI tools and languages, rather than AI systems themselves. However, it takes only a slight shift of viewpoint to change from what is an aid for the programmer to what is structure for the intelligent system itself. Not all features survive this transformation, but enough do to make the development of AI languages as

*This paper will appear in *Foundations of Artificial Intelligence*, Editted by Kirsh and Hewitt, MIT Press, Cambridge MA, and *Artificial Intelligence* in 1991. It is reprinted here by permission.

†This research was sponsored by the Proceedings of the Defense Advanced Research Projects Agency (DOD) under contract N00039-86-C-0133 and by the Sloan Foundation. Computer facilities were partially provided by NIH grant RR-00785 to Sumex-Aim. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the US Government, the Sloan Foundation, or the National Institutes of Health.

‡We would like to thank Beth Adelson, David Kirsh, and David McAllester for their helpful comments on an earlier draft of this article.

§Much of this work was done while the first author was affiliated with the Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

much substantive research as tool building. These proposals provide substantial ground on which to build.

The Soar project has been building on this foundation in an attempt to understand the functionality required to support general intelligence. Our current understanding is embodied in the Soar architecture [Laird, 1986; Laird *et al.*, 1987]. This article represents an attempt at describing and analyzing the structure of the Soar system. We will take a particular point of view — the description of Soar as a hierarchy of levels — in an attempt to bring coherence to this discussion.

The idea of analyzing systems in terms of multiple levels of description is a familiar one in computer science. In one version, computer systems are described as a sequence of levels that starts at the bottom with the device level and works up through the circuit level, the logic level, and then one or more program levels. Each level provides a description of the system at some level of abstraction. The sequence is built up by defining each higher level in terms of the structure provided at the lower levels. This idea has also recently been used to analyze human cognition in terms of levels of description [Newell, 1990]. Each level corresponds to a particular time scale, such as ~100 msec. and ~1 sec., with a new level occurring for each new order of magnitude. The four levels between ~10 msec. and ~10 sec. comprise the cognitive band (Figure 1). The lowest cognitive level — at ~10 msec. — is the symbol-accessing level, where the knowledge referred to by symbols is retrievable. The second cognitive level — at ~100 msec. — is the level at which elementary deliberate operations occur; that is, the level at which encoded knowledge is brought to bear, and the most elementary choices are made. The third and fourth cognitive levels — at ~1 sec. and ~10 sec. — are the simple-operator-composition and goal-attainment levels. At these levels, sequences of deliberations can be composed to achieve goals. Above the cognitive band is the rational band, at which the system can be described as being goal oriented, knowledge-based, and strongly adaptive. Below the cognitive band is the neural band.

In the architecture section we describe Soar as a sequence of three cognitive levels: the memory level, at which symbol accessing occurs; the decision level, at which elementary deliberate operations occur; and the goal level, at which goals are set and achieved via sequences of decisions. The goal level is an amalgamation of the top two cognitive levels from the analysis of human cognition.

In this description we will often have call to describe mechanisms that are built into the architecture of Soar. The architecture consists of all of the fixed structure of the Soar system. According to the levels analysis, the correct view to be taken of this fixed structure is that it comprises the set of mechanisms provided by the levels underneath the cognitive band. For human cognition this is the neural band. For artificial cognition, this may be a connectionist band, though it need not be. This view notwithstanding, it should be remembered that it is the Soar architecture which is primary in our research. The use of the levels viewpoint is simply an attempt at imposing a particular, hopefully illuminating, theoretical structure on top of the existing architecture.

In the remainder of this paper we describe the methodological assumptions underlying Soar, the structure of Soar, an illustrative example of Soar's performance on the task of multi-column subtraction, a set of preliminary analyses of Soar as an architecture for general intelligence.

Methodological Assumptions

The development of Soar is driven by four methodological assumptions. It is not expected that these assumptions will be shared by all researchers in the field. However, the assumptions do help explain why the Soar system and project have the shapes that they do.

The first assumption is the utility of focusing on the cognitive band, as opposed to the neural or rational bands. This is a view that has traditionally been shared by a large segment of the cognitive science community; it is not, however, shared by the connectionist community, which focuses on the neural band (plus the lower levels of the cognitive band), or by the logicist and expert-systems communities, which focus on the rational band. This assumption is not meant to be exclusionary, as a complete understanding of general intelligence requires the understanding of all of these descriptive bands.[1] Instead the assumption is that there is important work to be done by focusing on the cognitive band. One reason is that, as just mentioned, a complete model of general intelligence will require a model of the cognitive band. A second reason is that an understanding of the cognitive band can constrain models of the neural and rational bands. A third, more applied reason, is that a model of the cognitive band is required in order to be able to build practical intelligent systems. Neural-band models need the higher levels of organization that are provided by the cognitive band in order to reach complex task performance. Rational-band models need the heuristic adequacy provided by the cognitive band in order to be computationally feasible. A fourth reason is that there is a wealth of both psychological and AI data about the cognitive band that can be used as the basis for elucidating the structure of its levels. This data can help us understand what type of symbolic architecture is required to support general intelligence.

The second assumption is that general intelligence can most usefully be studied by not making a dis-

[1] Investigations of the relationship of Soar to the neural and rational bands can be found in [Newell, 1990; Rosenbloom, 1989; Rosenbloom *et al.*, 1990].

Rational Band	...	
Cognitive Band	~10 sec.	Goal attainment
	~1 sec.	Simple operator composition
	~100 msec.	Elementary deliberate operations
	~10 msec.	Symbol accessing
Neural Band	...	

Figure 1: Partial hierarchy of time scales in human cognition.

tinction between human and artificial intelligence. The advantage of this assumption is that it allows wider ranges of research methodologies and data to be brought to bear to mutually constrain the structure of the system. Our research methodology includes a mixture of experimental data, theoretical justifications, and comparative studies in both artificial intelligence and cognitive psychology. Human experiments provide data about performance universals and limitations that may reflect the structure of the architecture. For example, the ubiquitous power law of practice — the time to perform a task is a power-law function of the number of times the task has been performed — was used to generate a model of human practice [Newell & Rosenbloom, 1981; Rosenbloom & Newell, 1986], which was later converted into a proposal for a general artificial learning mechanism [Laird *et al.*, 1984; Laird *et al.*, 1986a; Steier *et al.*, 1987]. Artificial experiments — the application of implemented systems to a variety of tasks requiring intelligence — provide sufficiency feedback about the mechanisms embodied in the architecture and their interactions [Hsu *et al.*, 1988; Rosenbloom *et al.*, 1985; Steier, 1987; Steier & Newell, 1988; Washington & Rosenbloom, 1988]. Theoretical justifications attempt to provide an abstract analysis of the requirements of intelligence, and of how various architectural mechanisms fulfill those requirements [Newell, 1990; Newell *et al.*, 1989; Rosenbloom, 1989; Rosenbloom *et al.*, 1988b; Rosenbloom *et al.*, 1990]. Comparative studies, pitting one system against another, provide an evaluation of how well the respective systems perform, as well as insight about how the capabilities of one of the systems can be incorporated in the other [Etzioni & Mitchell, 1989; Rosenbloom & Laird, 1986].

The third assumption is that the architecture should consist of a small set of orthogonal mechanisms. All intelligent behaviors should involve all, or nearly all, of these basic mechanisms. This assumption biases the development of Soar strongly in the direction of uniformity and simplicity, and away from modularity [Fodor, 1983] and toolkit approaches. When attempting to achieve a new functionality in Soar, the first step is to determine in what ways the existing mechanisms can already provide the functionality. This can force the development of new solutions to old problems, and reveal new connections — through the common underlying mechanisms — among previously distinct capabilities [Rosenbloom *et al.*, 1988a]. Only if there is no appropriate way to achieve the new functionality are new mechanisms considered.

The fourth assumption is that architectures should be pushed to the extreme to evaluate how much of general intelligence they can cover. A serious attempt at evaluating the coverage of an architecture involves a long-term commitment by an extensive research group. Much of the research involves the apparently mundane activity of replicating classical results within the architecture. Sometimes these demonstrations will by necessity be strict replications, but often the architecture will reveal novel approaches, provide a deeper understanding of the result and its relationship to other results, or provide the means of going beyond what was done in the classical work. As these results accumulate over time, along with other more novel results, the system gradually approaches the ultimate goal of general intelligence.

Structure of Soar

In this section we build up much of Soar's structure in levels, starting at the bottom with memory and proceeding up to decisions and goals. We then describe how learning and perceptual-motor behavior fit into this picture, and wrap up with a discussion of the default knowledge that has been incorporated into the system.

Level 1: Memory

A general intelligence requires a memory with a large capacity for the storage of knowledge. A variety of types of knowledge must be stored, including declarative knowledge (facts about the world, including facts about actions that can be performed), procedural knowledge (facts about how to perform actions, and control knowledge about which actions to perform when), and episodic knowledge (which actions were

done when). Any particular task will require some subset of the knowledge stored in the memory. Memory access is the process by which this subset is retrieved for use in task performance.

The lowest level of the Soar architecture is the level at which these memory phenomena occur. All of Soar's long-term knowledge is stored in a single production memory. Whether a piece of knowledge represents procedural, declarative, or episodic knowledge, it is stored in one or more productions. Each production is a condition-action structure that performs its actions when its conditions are met. Memory access consists of the execution of these productions. During the execution of a production, variables in its actions are instantiated with values. Action variables that existed in the conditions are instantiated with the values bound in the conditions. Action variables that did not exist in the conditions act as generators of new symbols.

The result of memory access is the retrieval of information into a global working memory. The working memory is a temporary memory that contains all of Soar's short-term processing context. Working memory consists of an interrelated set of objects with attribute-value pairs. For example, an object representing a green cat named Fred might look like `(object o025 ^name fred ^type cat ^color green)`. The symbol `o025` is the identifier of the object, a short-term symbol for the object that exists only as long as the object is in working memory. Objects are related by using the identifiers of some objects as attributes and values of other objects.

There is one special type of working memory structure, the preference. Preferences encode control knowledge about the acceptability and desirability of actions, according to a fixed semantics of preference types. Acceptability preferences determine which actions should be considered as candidates. Desirability preferences define a partial ordering on the candidate actions. For example, a better (or alternatively, worse) preference can be used to represent the knowledge that one action is more (or less) desirable than another action, and a best (or worst) preference can be used to represent the knowledge that an action is at least as good (or as bad) as every other action.

In a traditional production-system architecture, each production is a problem-solving operator (see, for example, [Nilsson, 1980]). The right-hand side of the production represents some action to be performed, and the left-hand side represents the preconditions for correct application of the action (plus possibly some desirability conditions). One consequence of this view of productions is that the productions must also be the locus of behavioral control. If productions are going to act, it must be possible to control which one executes at each moment; a process known as conflict resolution. In a logic architecture, each production is a logical implication. The meaning of such a production is that if the left-hand side (the antecedent) is true, then so is the right-hand side (the consequent).[2] Soar's productions are neither operators nor implications. Instead, Soar's productions perform (parallel) memory retrieval. Each production is a retrieval structure for an item in long-term memory. The right-hand side of the rule represents a long-term datum, and the left-hand side represents the situations in which it is appropriate to retrieve that datum into working memory. The traditional production-system and logic notions of action, control, and truth are not directly applicable to Soar's productions. All control in Soar is performed at the decision level. Thus, there is no conflict resolution process in the Soar production system, and all productions execute in parallel. This all flows directly from the production system being a long-term memory. Soar separates the retrieval of long-term information from the control of which act to perform next.

Of course it is possible to encode knowledge of operators and logical implications in the production memory. For example, the knowledge about how to implement a typical operator can be stored procedurally as a set of productions which retrieve the state resulting from the operator's application. The productions' conditions determine when the state is to be retrieved — for example, when the operator is being applied and its preconditions are met. An alternative way to store operator implementation knowledge is declaratively as a set of structures that are completely contained in the actions of one or more productions. The structures describe not only the results of the operator, but also its preconditions. The productions' conditions determine when to retrieve this declarative operator description into working memory. A retrieved operator description must be interpreted by other productions to actually have an affect.

In general, there are these two distinct ways to encode knowledge in the production memory: procedurally and declaratively. If the knowledge is procedurally encoded, then the execution of the production reflects the knowledge, but does not actually retrieve it into working memory — it only retrieves the structures encoded in the actions. On the other hand, if a piece of knowledge is encoded declaratively in the actions of a production, then it is retrievable in its entirety. This distinction between procedural and declarative *encodings* of knowledge is distinct from whether the knowledge is declarative (represents facts about the world) or procedural (represents facts about procedures). Moreover, each production can be viewed in either way, either as a procedure which implicitly represents conditional information, or as the indexed storage of declarative structures.

[2]The directionality of the implication is reversed in logic programming languages such as Prolog, but the point still holds.

Level 2: Decisions

In addition to a memory, a general intelligence requires the ability to generate and/or select a course of action that is responsive to the current situation. The second level of the Soar architecture, the decision level, is the level at which this processing is performed. The decision level is based on the memory level plus an architecturally provided, fixed, decision procedure. The decision level proceeds in a two phase elaborate-decide cycle. During elaboration, the memory is accessed repeatedly, in parallel, until quiescence is reached; that is, until no more productions can execute. This results in the retrieval into working memory of all of the accessible knowledge that is relevant to the current decision. This may include a variety of types of information, but of most direct relevance here is knowledge about actions that can be performed and preference knowledge about what actions are acceptable and desirable. After quiescence has occurred, the decision procedure selects one of the retrieved actions based on the preferences that were retrieved into working memory and their fixed semantics.

The decision level is open both with respect to the consideration of arbitrary actions, and with respect to the utilization of arbitrary knowledge in making a selection. This openness allows Soar to behave in both plan-following and reactive fashions. Soar is following a plan when a decision is primarily based on previously generated knowledge about what to do. Soar is being reactive when a decision is based primarily on knowledge about the current situation (as reflected in the working memory).

Level 3: Goals

In addition to being able to make decisions, a general intelligence must also be able to direct this behavior towards some end; that is, it must be able to set and work towards goals. The third level of the Soar architecture, the goal level, is the level at which goals are processed. This level is based on the decision level. Goals are set whenever a decision cannot be made; that is, when the decision procedure reaches an impasse. Impasses occur when there are no alternatives that can be selected (*no-change* and *rejection* impasses) or when there are multiple alternatives that can be selected, but insufficient discriminating preferences exist to allow a choice to be made among them (*tie* and *conflict* impasses). Whenever an impasse occurs, the architecture generates the goal of resolving the impasse. Along with this goal, a new *performance context* is created. The creation of a new context allows decisions to continue to be made in the service of achieving the goal of resolving the impasse — nothing can be done in the original context because it is at an impasse. If an impasse now occurs in this subgoal, another new subgoal and performance context are created. This leads to a goal (and context) stack in which the top-level goal is to perform some task, and lower-level goals are to resolve impasses in problem solving. A subgoal is terminated when either its impasse is resolved, or some higher impasse in the stack is resolved (making the subgoal superfluous).

In Soar, all symbolic goal-oriented tasks are formulated in problem spaces. A problem space consists of a set of states and a set of operators. The states represent situations, and the operators represent actions which when applied to states yield other states. Each performance context consists of a goal, plus roles for a problem space, a state, and an operator. Problem solving is driven by decisions that result in the selection of problem spaces, states, and operators for their respective context roles. Given a goal, a problem space should be selected in which goal achievement can be pursued. Then an initial state should be selected that represents the initial situation. Then an operator should be selected for application to the initial state. Then another state should be selected (most likely the result of applying the operator to the previous state). This process continues until a sequence of operators has been discovered that transforms the initial state into a state in which the goal has been achieved. One subtle consequence of the use of problem spaces is that each one implicitly defines a set of constraints on how the task is to be performed. For example, if the Eight Puzzle is attempted in a problem space containing only a `slide-tile` operator, all solution paths maintain the constraint that the tiles are never picked up off of the board. Thus, such conditions need not be tested for explicitly in desired states.

Each problem solving decision — the selection of a problem space, a state, or an operator — is based on the knowledge accessible in the production memory. If the knowledge is both correct and sufficient, Soar exhibits highly controlled behavior; at each decision point the right alternative is selected. Such behavior is accurately described as being algorithmic or knowledge-intensive. However, for a general intelligence faced with a broad array of unpredictable tasks, situations will arise — inevitably and indeed frequently — in which the accessible knowledge is either incorrect or insufficient. It is possible that correct decisions will fortuitously be made, but it is more likely that either incorrect decisions will be made or that impasses will occur. Under such circumstances search is the likely outcome. If an incorrect decision is made, the system must eventually recover and get itself back on a path to the goal, for example, by backtracking. If instead an impasse occurs, the system must execute a sequence of problem space operators in the resulting subgoal to find (or generate) the information that will allow a decision to be made. This processing may itself be highly algorithmic, if enough control knowledge is available to uniquely determine what to do, or it may involve a large amount of further search.

As described earlier, operator implementation knowledge can be represented procedurally in the production memory, enabling operator implementation to

be performed directly by memory retrieval. When the operator is selected, a set of productions execute that collectively build up the representation of the result state by combining data from long-term memory and the previous state. This type of implementation is comparable to the conventional implementation of an operator as a fixed piece of code. However, if operator implementation knowledge is stored declaratively, or if no operator implementation knowledge is stored, then a subgoal occurs, and the operator must be implemented by the execution of a sequence of problem space operators in the subgoal. If a declarative description of the to-be-implemented operator is available, then these lower operators may implement the operator by interpreting its declarative description (as was demonstrated in work on task acquisition in Soar [Steier *et al.*, 1987]). Otherwise the operator can be implemented by decomposing it into a set of simpler operators for which operator implementation knowledge is available, or which can in turn be decomposed further.

When an operator is implemented in a subgoal, the combination of the operator and the subgoal correspond to the type of deliberately created subgoal common in AI problem solvers. The operator specifies a task to be performed, while the subgoal indicates that accomplishing the task should be treated as a goal for further problem solving. In complex problems, like computer configuration, it is common for there to be complex high-level operators, such as `Configure-computer` which are implemented by selecting problems spaces in which they can be decomposed into simpler tasks. Many of the traditional goal management issues — such as conjunction, conflict, and selection — show up as operator management issues in Soar. For example, a set of conjunctive subgoals can be ordered by ordering operators that later lead to impasses (and subgoals).

As described in [Rosenbloom *et al.*, 1988b], a subgoal not only represents a subtask to be performed, but it also represents an introspective act that allows unlimited amounts of meta-level problem-space processing to be performed. The entire working memory — the goal stack and all information linked to it — is available for examination and augmentation in a subgoal. At any time a production can examine and augment any part of the goal stack. Likewise, a decision can be made at any time for any of the goals in the hierarchy. This allows subgoal problem solving to analyze the situation that led to the impasse, and even to change the subgoal should it be appropriate. One not uncommon occurrence is for information to be generated within a subgoal that, instead of satisfying the subgoal, causes the subgoal to become irrelevant and consequently to disappear. Processing tends to focus on the bottom-most goal because all of the others have reached impasses. However, the processing is completely opportunistic, so that when appropriate information becomes available at a higher level, processing at that level continues immediately and all lower subgoals are terminated.

Learning

All learning occurs by the acquisition of chunks — productions that summarize the problem solving that occurs in subgoals [Laird *et al.*, 1986a]. The actions of a chunk represent the knowledge generated during the subgoal; that is, the results of the subgoal. The conditions of the chunk represent an access path to this knowledge, consisting of those elements of the parent goals upon which the results depended. The results of the subgoal are determined by finding the elements generated in the subgoal that are available for use in supergoals — an element is a result of a subgoal precisely because it is available to processes outside of the subgoal. The access path is computed by analyzing the traces of the productions that fired in the subgoal — each production trace effectively states that its actions depended on its conditions. This dependency analysis yields a set of conditions that have been implicitly generalized to ignore irrelevant aspects of the situation. The resulting generality allows chunks to transfer to situations other than the one in which it was learned. The primary system-wide effect of chunking is to move Soar along the space-time trade-off by allowing relevantly similar future decisions to be based on direct retrieval of information from memory rather than on problem solving within a subgoal. If the chunk is used, an impasse will not occur, because the required information is already available.

Care must be taken to not confuse the power of chunking as a learning mechanism with the power of Soar as a learning system. Chunking is a simple goal-based, dependency-tracing, caching scheme, analogous to explanation-based learning [DeJong & Mooney, 1986; Mitchell *et al.*, 1986; Rosenbloom & Laird, 1986] and a variety of other schemes [Rosenbloom & Newell, 1986]. What allows Soar to exhibit a wide variety of learning behaviors are the variations in the types of subgoals that are chunked; the types of problem solving, in conjunction with the types and sources of knowledge, used in the subgoals; and the ways the chunks are used in later problem solving. The role that a chunk will play is determined by the type of subgoal for which it was learned. State-no-change, operator-tie, and operator-no-change subgoals lead respectively to state augmentation, operator selection, and operator implementation productions. The content of a chunk is determined by the types of problem solving and knowledge used in the subgoal. A chunk can lead to skill acquisition if it is used as a more efficient means of generating an already generatable result. A chunk can lead to knowledge acquisition (or knowledge level learning [Dietterich, 1986]) if it is used to make old/new judgments; that is, to distinguish what has been learned from what has not been learned [Rosenbloom *et al.*, 1987; Rosenbloom *et al.*, 1988a;

Rosenbloom *et al.*, 1990].

Perception and Motor Control

One of the most recent functional additions to the Soar architecture is a perceptual-motor interface [Wiesmeyer, 1988b; Wiesmeyer, 1989]. All perceptual and motor behavior is mediated through working memory; specifically, through the state in the top problem solving context. Each distinct perceptual field has a designated attribute of this state to which it adds its information. Likewise, each distinct motor field has a designated attribute of the state from which it takes it commands. The perceptual and motor systems are autonomous with respect to each other and the cognitive system.

Encoding and decoding productions can be used to convert between the high-level structures used by the cognitive system, and the low-level structures used by the perceptual and motor systems. These productions are like ordinary productions, except that they examine only the perceptual and motor fields, and not any of the rest of the context stack. This autonomy from the context stack is critical, because it allows the decision procedure to proceed without waiting for quiescence among the encoding and decoding productions, which may never happen in a rapidly changing environment.

Default Knowledge

Soar has a set of productions (55 in all) that provide default responses to each of the possible impasses that can arise, and thus prevent the system from dropping into a bottomless pit in which it generates an unbounded number of content-free performance contexts. Figure 2 shows the default production that allows the system to continue if it has no idea how to resolve a conflict impasse among a set of operators. When the production executes, it rejects all of the conflicting operators. This allows another candidate operator to be selected, if there is one, or for a different impasse to arise if there are no additional candidates. This default response, as with all of them, can be overridden by additional knowledge if it is available.

```
If there is an impasse because of an operator
   conflict and there are no candidate
   problem spaces available
then reject the conflicting operators.
```

Figure 2: A default production.

One large part of the default knowledge (10 productions) is responsible for setting up operator subgoaling as the default response to no-change impasses on operators. That is, it attempts to find some other state in the problem space to which the selected operator can be applied. This is accomplished by generating acceptable and worst preferences in the subgoal for the parent problem space. If another problem space is suggested, possibly for implementing the operator, it will be selected. Otherwise, the selection of the parent problem space in the subgoal enables operator subgoaling. A sequence of operators is then applied in the subgoal until a state is generated that satisfies the preconditions of an operator higher in the goal stack.

Another large part of the default knowledge (33 productions) is responsible for setting up lookahead search as the default response to tie impasses. This is accomplished by generating acceptable and worst preferences for the *selection* problem space. The selection problem space consists of operators that evaluate the tied alternatives. Based on the evaluations produced by these operators, default productions create preferences that break the tie and resolve the impasse. In order to apply the evaluation operators, domain knowledge must exist that can create an evaluation. If no such knowledge is available, a second impasse arises — a no-change on the evaluation operator. As mentioned earlier, the default response to an operator no-change impasse is to perform operator subgoaling. However, for a no-change impasse on an evaluation operator this is overridden and a lookahead search is performed instead. The results of the lookahead search are used to evaluate the tied alternatives.

As Soar is developed, it is expected that more and more knowledge will be included as part of the basic system about how to deal with a variety of situations. For example, one area on which we are currently working is the provision of Soar with a basic arithmetical capability, including problem spaces for addition, multiplication, subtraction, division, and comparison. One way of looking at the existing default knowledge is as the tip of this large iceberg of background knowledge. However, another way to look at the default knowledge is as part of the architecture itself. Some of the default knowledge — how much is still unclear — must be innate rather than learned. The rest of the system's knowledge, such as the arithmetic spaces, should then be learnable from there.

Example: Multi-column Subtraction

Multi-column subtraction is the task we will use to demonstrate Soar. This task has three advantages. First, it is a familiar and simple task. This allows the details of Soar not to be lost in the complexities of understanding the task. Second, previous work has been done on modeling human learning of subtraction in the Sierra architecture [VanLehn, 1983]. Our implementation is inspired by the Sierra framework. Third, this task appears to be quite different from many standard search-intensive tasks common in AI. On the surface, it appears difficult to cast subtraction within the problem-space framework of Soar — it is, after all, a procedure. One might also think that chunking could not learn such a procedure. However, in this example, we will demonstrate that multi-column subtraction can

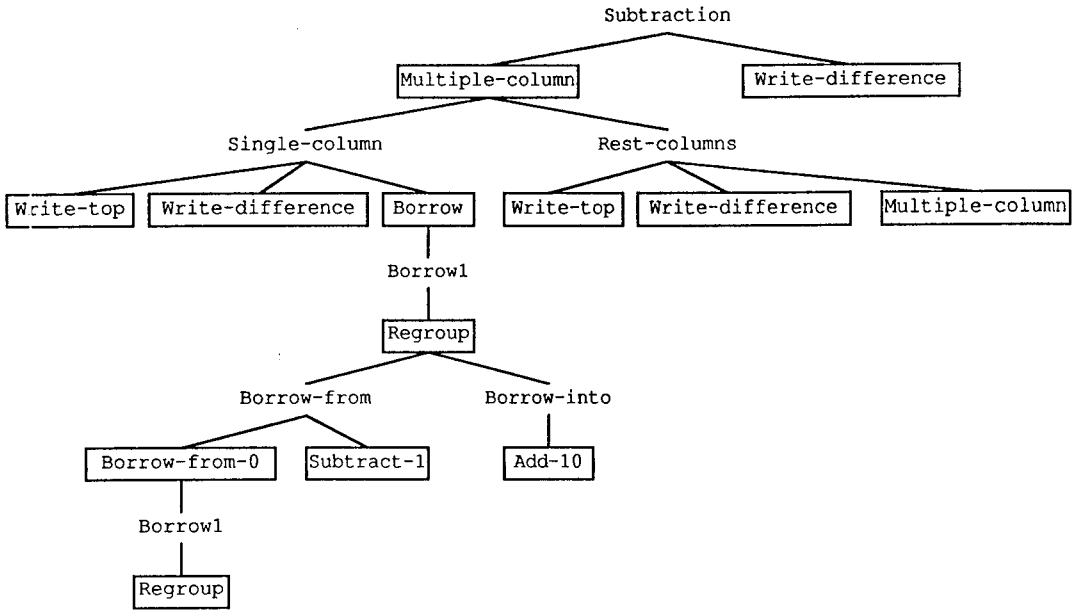


Figure 3: A goal hierarchy for multi-column subtraction.

be performed by Soar and that important parts of the procedure can be learned through chunking.

There exist many different procedures for performing multi-column subtraction. Different procedures result in different behaviors, both in the order in which scratch marks — such as borrowing notations — are made and in the type of mistakes that might be generated while learning [VanLehn & Ball, 1987]. For simplicity, we will demonstrate the implementation of just one of the many possible procedures. This procedure uses a borrowing technique that recursively borrows from a higher-order column into a lower-order column when the top number in the lower-order column is less than the bottom number.

A Hierarchical Subtraction Procedure

One way to implement this procedure is via the processing of a goal hierarchy that encodes what must be done. Figure 3 shows a subtraction goal hierarchy that is similar to the one learned by Sierra.[3] Under each goal are shown the subgoals that may be generated while trying to achieve it. This Sierra goal hierarchy is mapped onto a hierarchy of operators and problem spaces in Soar (as described in the architecture section). The boxed goals map onto operators and the unboxed goals map onto problem spaces. Each problem space consists of the operators linked to it from below in the figure. Operators that have problem spaces below them are implemented by problem solving in those problem spaces. The other operators are implemented directly at the memory level by productions (except for multiple-column and regroup, which are recursive). These are the primitive acts of subtraction, such as writing numbers or subtracting digits.

The states in these problem spaces contain symbolic representations of the subtraction problem and the scratch marks made on the page during problem solving. The representation is very simple and direct, being based on the spatial relationships among the digits as they would appear on a page. The state consists of a set of columns. Each column has pointers to its top and bottom digits. Additional pointers are generated when an answer for a column is produced, or when a scratch mark is made as the result of borrowing. The physical orientation of the columns on the page is represented by having "left" and "right" pointers from columns to their left and right neighbors. There is no inherent notion of multi-digit numbers except for these left and right relations between columns. This representation is consistent with the operators, which treat the problem symbolically and never manipulate multi-digit numbers as a whole.

Using this implementation of the subtraction procedure, Soar is able to solve all multi-column subtraction problems that result in positive answers. Unfortunately, there is little role for learning. Most of the control knowledge is already embedded in the productions that select problem spaces and operators. Within each problem space there are only a few operators from which to select. The preconditions of the few operators in each problem space are sufficient for perfect behavior. Therefore, goals arise only to implement opera-

[3] Sierra learned a slightly more elaborate, but computationally equivalent, procedure.

- *Operators:*

  Write-difference: If the difference between the top digit and the bottom digit of the current column is known, then write the difference as an answer to the current column.

  Write-top: If the lower digit of the current column is blank, then write the top digit as the answer to the current column.

  Borrow-into: If the result of adding 10 to the top digit of the current column is known, and the digit to the left of it has a scratch mark on it, then replace the top digit with the result.

  Borrow-from: If the result of subtracting 1 from the top digit in the current column is known, then replace that top digit with the result, augment it with a scratch mark and shift the current column to the right.

  Move-left: If the current column has an answer in it, shift the current column left.

  Move-borrow-left: If the current column does not have a scratch mark in it, shift the current column left.

  Subtract-two-digits: If the top digit is greater than or equal to the lower digit, then produce a result that is the difference.

  Subtract-1: If the top digit is not zero, then produce a result that is the top digit minus one.

  Add-10: Produce a result that is the top digit plus ten.

- *Goal Test:* If each column has an answer, then succeed.

Figure 4: Primitive subtraction problem space.

tors. Chunking these goals produces productions that are able to compute answers without the intermediate subgoals.[4]

A Single Space Approach

One way to loosen up the strict control provided by the detailed problem-space/operator hierarchy in Figure 3, and thus to enable the learning of the control knowledge underlying the subtraction procedure, is to have only a single subtraction problem space that contains all of the primitive acts (writing results, changing columns, and so on). Figure 4 contains a description of the problem space operators and the goal test used in this second implementation. The operators can be grouped into four classes: the basic acts of writing answers to a single column problem (write-difference, write-top); borrow actions on the upper digits (borrow-into, borrow-from); moving from one column to the next (move-left, move-borrow-left); and performing very simple arithmetic computations (subtract-two-digits, subtract-1, add-10). With this simple problem space, Soar must learn the subtraction procedure by acquiring control knowledge that correctly selects operators.

Every operator in the subtraction problem space is considered for every state in the space. This is accomplished by having a production for each operator that generates an acceptable preference for it. The conditions of the production only test that the appropriate problem space (subtraction) is selected. Similar productions existed in the original implementation, except that those productions also contained additional tests which ensured that the operators would only be considered when they were the appropriate ones to apply.

In addition to productions which generate acceptable preferences, each operator has one or more productions which implement it. Although every operator is made acceptable for every state, an operator will actually be applied only if all of the conditions in the productions that implement it are satisfied. For example, write-difference will only apply if the difference between the top and bottom numbers is known. If an operator is selected, but the conditions of the productions that implement it are not satisfied, an impasse arises. As described in the architecture section, the default response to this type of impasse is to perform operator subgoaling.

Figure 5 shows a trace of Soar's problem solving as it performs a simple two-column subtraction problem, after the learning of control knowledge has been completed. Because Soar's performance prior to learning on this problem is considerably more complicated, it is described after this simpler case. The top goal in this figure is to have the result of subtracting 3 from 22. Problem solving in the top goal proceeds from left to right, diving to a lower level whenever a subgoal is created in response to an impasse. Each state is a partially solved subtraction problem, consisting of the statement of the subtraction problem, a * designating the current column, and possibly column results and/or scratch marks for borrowing. Operator applications are represented by arrows going from left to right. The only impasses that occur in this trace are a result of the failure of operator preconditions — a form of operator no-change impasse. These impasses are designated by circles disrupting the operator-application arrows, and are labeled in the order they arise (A and B). For exam-

[4] This work on subtraction was done in an earlier version of Soar that did not have the perceptual-motor interface described in the architecture section. In that version, these chunks caused Soar to write out all of the column results and scratch marks in parallel — not very realistic motor behavior. To work around this problem chunking was disabled for goals in this task during which environmental interactions occurred.

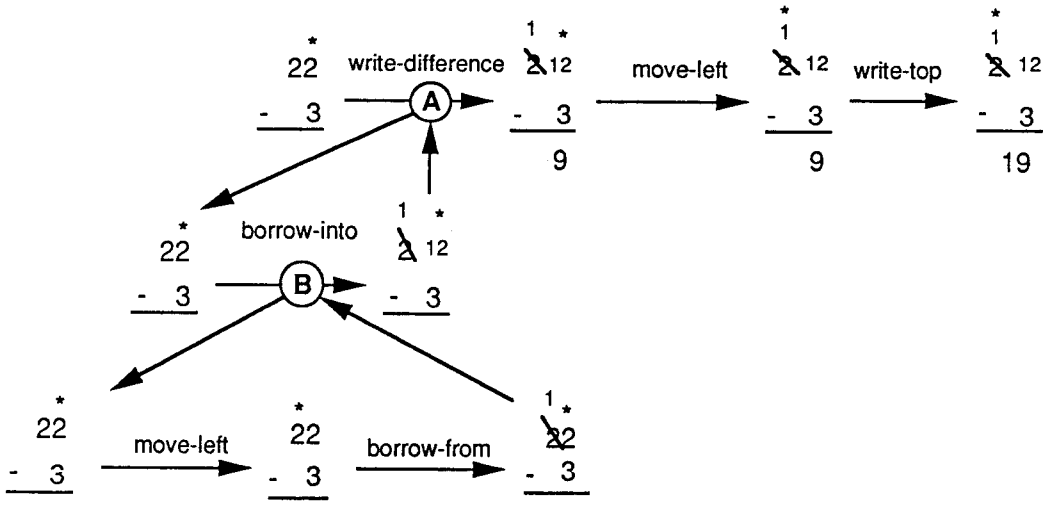


Figure 5: Trace of problem solving after learning for 22 – 3.

ple, impasse A arises because write-difference cannot apply unless the lower digit in the current column (3) is less than the top digit (2).

For impasse A, operator subgoaling occurs when the subtraction problem space is selected in the subgoal. The preconditions of the write-difference operator are met when a state has been generated whose top digit has been changed from 2 to 12 (by borrowing). Once this occurs, the subgoal terminates and the operator applies, in this case writing the difference between 12 and 3. In this implementation of subtraction, operator subgoaling dynamically creates a goal hierarchy that is similar to the one programmed into the original implementation.

Performance Prior to Learning

Prior to learning, Soar's problem solving on this task is considerably more complicated. This added complexity arises because of an initial lack of knowledge about the results of simple arithmetic computations and a lack of knowledge about which operators should be selected for which states. Figure 6 shows a partial trace of Soar's pre-learning problem solving. Although many of the subgoals are missing, this small snapshot of the problem solving is characteristic of the impasses and subgoals that arise at all levels.

As before, the problem solving starts at the upper left with the initial state. As soon as the initial state is selected, a tie impasse (A) arises because all of the operators are acceptable and there are no additional preferences that distinguish between them. Default productions cause the selection space to be selected for this impasse. Within this space, operators are created to evaluate the tied operators. This example assumes that evaluate-object(write-difference) is selected, possibly based on advice from a teacher. Then, because there is no knowledge available about how to evaluate the subtraction operators, a no-change impasse (B) occurs for the evaluation operator. More default productions lead to a lookahead search by suggesting the original problem space (subtraction) and state and then selecting the operator that is being evaluated. The operator then applies, if it can, creating a new state. In this example, an operator subgoal impasse (C) arises when the attempt is made to apply the write-difference operator — its preconditions are not satisfied. Problem solving continues in this subgoal, requiring many additional impasses, until the write-difference operator can finally be applied. The lookahead search then continues until an evaluation is generated for the write-difference operator. Here, this happens shortly after impasse C is resolved. The system was given the knowledge that a state containing an answer for the current column is a (partial) success — such states are on the path to the goal. This state evaluation is then converted by default productions into an evaluation of "success" for the operator, and from there into a best preference for the operator. The creation of this preference breaks the operator tie, terminating the subgoals, and leading to the selection of the preferred operator (write-difference). The overall behavior of the system during this lookahead search is that of depth-first search — where backtracking occurs by subgoal termination — intertwined with operator subgoaling. Once this search is completed, further impasses (N) arise to actually apply the selected operator, but eventually, a solution is found.

One way in which multi-column subtraction differs from the classic AI search tasks is that the goal test is underspecified. As shown in Figure 4, the goal test

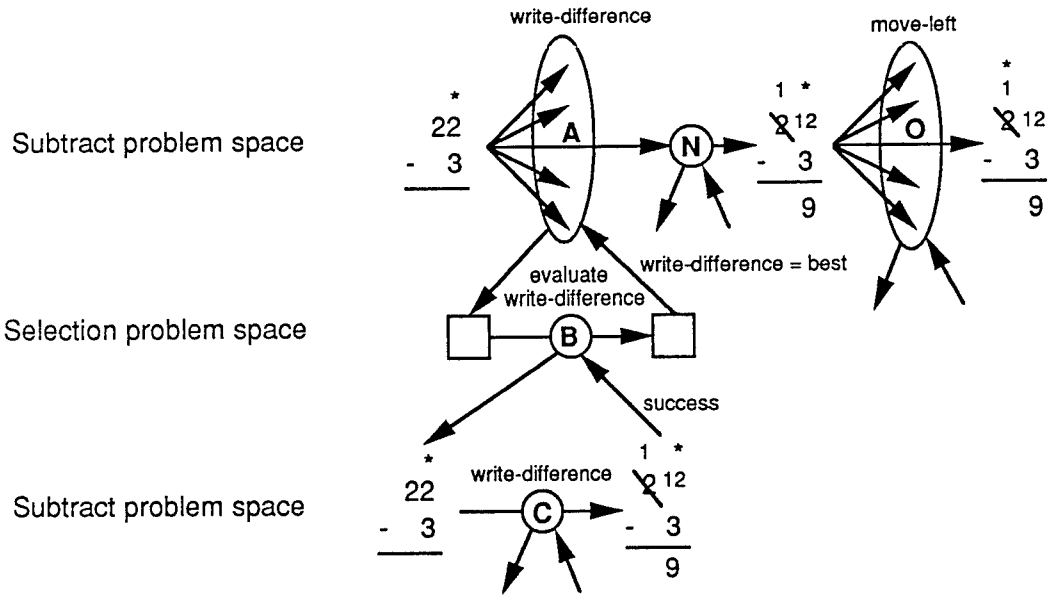


Figure 6: Trace of problem solving before learning for 22 – 3.

used here is that a result has been generated for each column of the problem. This determines whether some answer has been given for the problem, but is inadequate to determine whether the correct answer has been generated. The reason for this is that when solving a subtraction problem, the answer is in general not already available. It is theoretically (and practically) possible to use an addition procedure to test whether the subtraction procedure has generated the correct result. However, that corresponds to a deliberate strategy of "checking your work", rather than to the normal procedural goal test of determining whether the sequence of steps has been completed.

One consequence of having an underspecified goal test is that the combination of the problem space and goal test are not sufficient to ensure correct performance. Additional knowledge — the control knowledge which underlies the subtraction procedure — must also be provided in some form. VanLehn provided Sierra with worked out examples which included the order in which the primitive external actions were to be performed [VanLehn, 1983]. The approach that we have taken is to provide advice to Soar [Golding *et al.*, 1987] about which task operators it should evaluate first in the selection problem space. This ensures that the first answer generated during the lookahead search is the correct one.

Learning in Subtraction

When chunking is used during subtraction problem solving, productions are created which reproduce the results of the subgoals in similar future situations. For the subgoals created because of tie impasses, the chunks create best preferences for the operators that led to the solution. These chunks essentially cache the results of the lookahead searches. A set of such chunks corresponds to a plan (or procedure) — they determine at every step what should be done — thus chunking converts Soar's behavior from search into plan (or procedure) following. When Soar is rerun on the same problem, the tie impasses do not arise and the solution is found directly, as in Figure 5.

One important issue concerning the chunked productions is their generality. Does Soar only learn chunks that can apply to the exact same problem, or are the chunks general enough so that advice is no longer needed after a few subtraction problems have been completed? The answer is that the learned control chunks are quite general — so general that only one or two are required per operator. Once these chunks are acquired, Soar is able to solve perfectly all multi-column subtraction problems that have a positive answer. One sample control chunk for the borrow-into operator is shown in Figure 7. Similar chunks are learned for each of the other major operators.

```
If the super-operator is write-difference,
   and the bottom digit is greater than
       the top digit,
then make a best preference for borrow-into.
```

Figure 7: A control chunk for borrow-into.

One reason for this generality is that operator subgoaling leads to a fine-grained goal hierarchy. There

are a large number of relatively simple goals having to do with satisfying the preconditions of an operator. Because the problem solving for these goals is relatively minimal, the resulting chunks are quite general. A second reason for the generality of the learning is that the control chunks do not test for the specific digits used in the problems — if such tests were included, the chunks would transfer to many fewer problems.[5]

Though the control chunks that are learned are quite general, many specialized implementation chunks are also learned for the simple arithmetic operators. For example, the set of chunks that are eventually learned for the subtract-two-digits operator comprise a partial subtraction table for one and two-digit numbers. Conceivably, these chunks could have been learned before multi-column subtraction is ever attempted — one can imagine that most of these simple digit manipulations are learned during earlier lessons on addition and single-column subtraction. Alternatively, these chunks can continue to be acquired as more multi-column subtraction problems are solved. The control chunks would all be acquired after a few trials, but learning of arithmetic knowledge would continue through later problems.

Analysis of Soar

There are a variety of analyses that could be performed for Soar. In this section we take our cue from the issues provided by the organizers of the 1987 Workshop on the Foundations of Artificial Intelligence [Hewitt & Kirsh, 1987]. We examine the set of tasks that are natural for Soar, the sources of its power, and its scope and limits.

Natural Tasks

What does it mean for a task to be natural for an architecture? To answer this question we first must understand what a task is, and then what it means for such a task to be natural. By "task" we will mean any identifiable function, whether externally specified, or completely internal to the system. Computer configuration and maneuvering through an obstacle course are both tasks, and so are inheritance and skill acquisition. One way to define the idea of naturalness for a combination of a task and architecture is to say that a task is natural for an architecture if the task can be performed within the architecture without adding an extra level of interpretation within the software. This definition is appealing because it allows a distinction to be made between the tasks that the architecture can perform directly and those that can be done, but for which the architecture does not provide direct support. However, applying this definition is not without its problems. One problem is that, for any particular task, it is possible to replace the combination of an interpreter and its interpreted structures with a procedure that has the same effect. Some forms of learning — chunking, for example — do exactly this, by compiling interpreted structures into the structure of the interpreter. This has the effect of converting an unnatural task implementation into a natural one. Such a capability causes problems for the definition of naturalness — naturalness cannot be a fixed property of the combination of a task and an architecture — but it is actually a point in favor of architectures that can do such learning.

A second problem is that in a system that is itself built up in levels, as is Soar, different tasks will be performed at different levels. In Soar, tasks can be performed directly by the architecture, by memory retrieval, by a decision, or by goal-based problem solving. A task is implemented at a particular level if that level and all lower levels are involved, but the higher levels are not. For example, consider the task of inheritance. Inheritance is not directly implemented by the Soar architecture, but it can be implemented at the memory level by the firing of productions. This implementation involves the memory level plus the architecture (which implements the memory level), but not the decision or goal levels. Alternatively, inheritance could be implemented at the decision level, or even higher up at goal level. As the level of implementation increases, performance becomes more interpretive, but the model of computation explicitly includes all of these levels as natural for the system.

One way out of this problem is to have pretheoretic notions about the level at which a particular task ought to be performable. The system is then natural for the task if it can be performed at that level, and unnatural if it must be implemented at a higher level. If, for example, the way inheritance works should be a function of the knowledge in the system, then the natural level for this capability is at the memory level (or higher).

In the remainder of this section we describe the major types of tasks that appear to us to be natural in Soar. Lacking any fundamental ways of partitioning the set of all tasks into principled categories, we will use a categorization based on four of the major functional capabilities of Soar: search-based tasks, knowledge-based tasks, learning tasks, and robotic tasks. The naturalness judgments for these task types are always based on assumptions about the natural level of imple-

[5] Chunking would include tests for the digits if their specific values were examined during the lookahead searches. However, the actual manipulation of the numbers is performed by the simple arithmetic operators: add-10, subtract-1 and subtract-two-digits. Before an operator such as write-difference is applied, an operator subgoal is created in which subtract-two-digits is selected and applied. The chunk for this subgoal reproduces the result whenever the same two digits are to be subtracted, eliminating the need for subtract-two-digits in such situations in the future. In the following lookahead searches, only pointers to the digits rather than the actual digits are ever tested, thereby leading to control chunks that are independent of the actual digits.

mentation for a variety of subtasks within each type of task. We will try to be as clear as possible about the levels at which the subtasks are being performed, so that others may also be able to make these judgments for themselves.

Search-based tasks Soar performs search in two qualitatively different ways: within context and across context. Within-context search occurs when Soar "knows" what to do at every step, and thus selects a sequence of operators and states without going into a subgoal. If it needs to backtrack in within-context search, and the states in the problem space are internal (rather than states of the outside world), it can do so by reselecting a previously visited state. Within-context search corresponds to doing the task, without lookahead, and recovering if anything goes wrong. Across-context search occurs when the system doesn't know what to do, and impasses arise. Successive states in the search show up in successively lower contexts. Backtracking occurs by terminating subgoals. Across-context search corresponds to lookahead search, hypothetical scenario generation, or simulation.

Various versions of Soar have been demonstrated to be able to perform over 30 different search methods [Laird, 1983; Laird & Newell, 1983; Laird *et al.*, 1987]. Soar can also exhibit hybrid methods — such as a combination of hill-climbing and depth-first search or of operator subgoaling and depth-first search — and use different search methods for different problem spaces within the same problem.

Search methods are represented in Soar as method increments — productions that contain a small chunk of knowledge about some aspect of a task and its action consequences. For example, a method increment might include knowledge about how to compute an evaluation function for a task, along with the knowledge that states with better evaluations should be preferred. Such an increment leads to a form of hill climbing. Other increments lead to other search methods. Combinations of increments lead to mixed methods.

The basic search abilities of making choices and generating subgoals are provided by the architecture. Individual method increments are at the memory level, but control occurs at the decision level, where the results of all of the method increments can be integrated into a single choice. Some search knowledge, such as the selection problem space, exists at the goal level.

Knowledge-based tasks Knowledge-based tasks are represented in Soar as a collection of interacting problem spaces (as are all symbolic goal-oriented tasks). Each problem space is responsible for a part of the task. Problem spaces interact according to the different goal-subgoal relationships that can exist in Soar. Within each problem space, the knowledge is further decomposed into a set of problem space components, such as goal testing, state initialization, and operator proposal [Yost & Newell, 1989]. These components, along with additional communication constructs, can then be encoded directly as productions, or can be described in a high-level problem space language called TAQL [Yost & Newell, 1989], which is then compiled down into productions. Within this overall problem space organization, other forms of organization — such as object hierarchies with inheritance — are implementable at the memory level by multiple memory accesses. Task performance is represented at the goal level as search in problem spaces.

Several knowledge-based tasks have been implemented in Soar, including the R1-Soar computer configuration system [Rosenbloom *et al.*, 1985], the Cypress-Soar and Designer-Soar algorithm design systems [Steier, 1987; Steier & Newell, 1988], the Neomycin-Soar medical diagnosis system [Washington & Rosenbloom, 1988], and the Merl-Soar job-shop scheduling system [Hsu *et al.*, 1988].

These five knowledge-based systems cover a variety of forms of both construction and classification tasks. Construction tasks involve assembling an object from pieces. R1-Soar — in which the task is to construct a computer configuration — is a good example of a construction task. Classification tasks involve selecting from among a set of objects. Neomycin-Soar — in which the task is to diagnose an illness — is a good example of a classification task.[6] In their simplest forms, both construction and classification occur at the decision level. In fact, they both occur to some extent within every decision in Soar — alternatives must be assembled in working-memory and then selected. These capabilities can require trivial amounts of processing, as when an object is constructed by instantiating and retrieving it from memory. They can also involve arbitrary amounts of problem solving and knowledge, as when the process of operator-implementation (or, equivalently, state-construction) is performed via problem solving in a subgoal.

Learning tasks The architecture directly supports a form of experiential learning in which chunking compiles goal-level problem solving into memory-level productions. Execution of the productions should have the same effect as the problem solving would have had, just more quickly. The varieties of subgoals for which chunks are learned lead to varieties in types of productions learned: problem space creation and selection; state creation and selection; and operator creation, selection, and execution. An alternative classification for this same set of behaviors is that it covers procedural, episodic and declarative knowledge [Rosenbloom *et al.*, 1990]. The variations in goal outcomes lead to both learning from success and learning from failure. The

[6] In a related development, as part of an effort to map the Generic Task approach to expert system construction onto Soar, the Generic Task for classification by establish-refine has been implemented in Soar as a general problem space [Johnson *et al.*, 1989].

ability to learn about all subgoal results leads to learning about important intermediate results, in addition to learning about goal success and failure. The implicit generalization of chunks leads to transfer of learned knowledge to other subtasks within the same problem (within-trial transfer), other instances of the same problem (across-trial transfer), and other problems (across-task transfer). Variations in the types of problems performed in Soar lead to chunking in knowledge-based tasks, search-based, and robotic tasks. Variations in sources of knowledge lead to learning from both internal and external knowledge sources. A summary of many of the types of learning that have so far been demonstrated in Soar can be found in [Steier *et al.*, 1987].

The apparent naturalness of these various forms of learning depends primarily on the appropriateness of the required problem solving. Towards the natural end of the spectrum is the acquisition of operator selection productions, in which the problem solving consists simply of a search with the set of operators for which selection knowledge is to be learned. Towards the unnatural end of the spectrum is the acquisition of new declarative knowledge from the outside environment. Many systems employ a simple store command for such learning, effectively placing the capability at the memory level. In Soar, the capability is situated two levels further up, at the goal level. This occurs because the knowledge must be stored by chunking, which can only happen if the knowledge is used in subgoal-based problem solving. The naturalness of this learning in Soar thus depends on whether this extra level of interpretation is appropriate or not. It turns out that the problem solving that enables declarative learning in Soar takes the form of an understanding process that relates the new knowledge to what is already known. The chunking of this understanding process yields the chunks that encode the new knowledge. If it is assumed that new knowledge should always be understood to be learned, then Soar's approach starts to look more natural, and verbatim storage starts to look more inappropriate.

Robotic tasks Robotic tasks are performed in Soar via its perceptual-motor interface. Sensors autonomously generate working memory structures representing what is being sensed, and motor systems autonomously take commands from working memory and execute them. The work on robotics in Soar is still very much in its infancy; however, in Robo-Soar [Laird *et al.*, 1989], Soar has been successfully hooked up to the combination of a camera and a Puma arm, and then applied to several simple blocks-world tasks.[7] Low-level software converts the camera signal into information about the positions, orientations and identifying characteristics of the blocks. This perceptual information is then input to working memory, and further interpreted by encoding productions. Decoding productions convert the high-level robot commands generated by the cognitive system to the low-level commands that are directly understood by the controller for the robot arm. These low-level commands are then executed through Soar's motor interface.

Given a set of operators which generate motor commands, and knowledge about how to simulate the operators and about the expected positions of blocks following the actions, Robo-Soar is able to successfully solve simple blocks world problems and to learn from its own behavior and from externally provided advice. It also can make use of a general scheme for recovering from incorrect knowledge [Laird, 1988] to recover when the unexpected occurs — such as when the system fails in its attempt to pick up a triangular prism — and to learn to avoid the failure in the future. Robo-Soar thus mixes planning (lookahead search with chunking), plan execution and monitoring, reactivity, and error recovery (with replanning). This performance depends on all of the major components of the architecture, plus general background knowledge — such as how to do lookahead search and how to recover from errors — and specific problem spaces for the task.

Where the Power Resides

Soar's power and flexibility arise from at least four identifiable sources. The first source of power is the universality of the architecture. While it may seem that this should go without saying, it is in fact a crucial factor, and thus important to mention explicitly. Universality provides the primitive capability to perform any computable task, but does not by itself explain why Soar is more appropriate than any other universal architecture for knowledge-based, search-based, learning, and robotic tasks.

The second source of power is the uniformity of the architecture. Having only one type of long-term memory structure allows a single, relatively simple, learning mechanism to behave as a general learning mechanism. Having only one type of task representation (problem spaces) allows Soar to move continuously from one extreme of brute-force search to the other extreme of knowledge-intensive (or procedural) behavior without having to make any representational decisions. Having only one type of decision procedure allows a single, relatively simple, subgoal mechanism to generate all of the types of subgoals needed by the system.

The traditional downside of uniformity is weakness and inefficiency. If instead the system were built up as a set of specialized modules or agents, as proposed in [Fodor, 1983; Minsky, 1986], then each of the modules could be optimized for its own narrow task. Our approach to this issue in Soar has been to go strongly

[7] The work on Robo-Soar has been done in the newest major release of Soar (version 5) [Laird *et al.*, 1990], which differs in a number of interesting ways from the earlier versions upon which the rest of the results in this article are based.

with uniformity — for all of the benefits listed above — but to achieve efficiency (power) through the addition of knowledge. This knowledge can either be added by hand (programming) or by chunking.

The third source of power is the specific mechanisms incorporated into the architecture. The production memory provides pattern-directed access to large amounts of knowledge; provides the ability to use strong problem solving methods; and provides a memory structure with a small-grained modularity. The working memory allows global access to processing state. The decision procedure provides an open control loop that can react immediately to new situations and knowledge; contributes to the modularity of the memory by allowing memory access to proceed in an uncontrolled fashion (conflict resolution was a major source of nonmodularity in earlier production systems); provides a flexible control language (preferences); and provides a notion of impasse that is used as the basis for the generation of subgoals. Subgoals focus the system's resources on situations where the accessible knowledge is inadequate; and allow flexible meta-level processing. Problem spaces separate control from action, allowing them (control and action) to be reasoned about independently; provide a constrained context within which the search for a desired state can occur; provide the ability to use weak problem solving methods; and provide for straightforward responses to uncertainty and error (search and backtracking). Chunking acquires long-term knowledge from experience; compiles interpreted procedures into non-interpreted ones; and provides generalization and transfer. The perceptual-motor system provides the ability to observe and affect the external world in parallel with the cognitive activity.

The fourth source of power is the interaction effects that result from the integration of all of the capabilities within a single system. The most compelling results generated so far come about from these interactions. One example comes from the mixture of weak methods, strong methods, and learning that is found in systems like R1-Soar. Strong methods are based on having knowledge about what to do at each step. Because strong methods tend to be efficient and to produce high-quality solutions, they should be used whenever possible. Weak methods are based on searching to make up for a lack of knowledge about what should be done. Such methods contribute robustness and scope by providing the system with a fall-back approach for situations in which the available strong methods do not work. Learning results in the addition of knowledge, turning weak methods into strong ones. For example, in R1-Soar it was demonstrated how computer configuration could be cast as a search problem, how strong methods (knowledge) could be used to reduce search, how weak methods (subgoals and search) could be used to make up for a lack of knowledge, and how learning could add knowledge as the result of search.

Another interesting interaction effect comes from work on abstraction planning, in which a difficult problem is solved by first learning a plan for an abstract version of the problem, and then using the abstract plan to aid in finding a plan for the full problem [Newell & Simon, 1972; Sacerdoti, 1974; Unruh *et al.*, 1987; Unruh & Rosenbloom, 1989]. Chunking helps the abstraction planning process by recording the abstract plan as a set of operator-selection productions, and by acquiring other productions that reduce the amount of search required in generating a plan. Abstraction helps the learning process by allowing chunks to be learned more quickly — abstract searches tend to be shorter than normal ones. Abstraction also helps learning by enabling chunks to be more general than they would otherwise be — the chunks ignore the details that were abstracted away — thus allowing more transfer and potentially decreasing the cost of matching the chunks (because there are now fewer conditions).

Scope and Limits

The original work on Soar demonstrated its capabilities as a general problem solver that could use any of the weak methods when appropriate, across a wide range of tasks. Later, we came to understand how to use Soar as the basis for knowledge-based systems, and how to incorporate appropriate learning and perceptual-motor capabilities into the architecture. These developments increased Soar's scope considerably beyond its origins as a weak-method problem solver. Our ultimate goal has always been to develop the system to the point where its scope includes everything required of a general intelligence. In this section we examine how far Soar has come from its relatively limited initial demonstrations towards its relatively unlimited goal. This discussion is divided up according to the major components of the Soar architecture, as presented in the architecture section: memory, decisions, goals, learning, and perception and motor control.

Level 1: Memory The scope of Soar's memory level can be evaluated in terms of the amount of knowledge that can be stored, the types of knowledge that can be represented, and the organization of the knowledge.

Amount of knowledge. Using current technology, Soar's production memory can support the storage of thousands of independent chunks of knowledge. The size is primarily limited by the cost of processing larger numbers of productions. Faster machines, improved match algorithms and parallel implementations [Gupta & Tambe, 1988; Tambe *et al.*, 1989; Tambe *et al.*, 1988] may raise this effective limit by several orders of magnitude over the next few years.

Types of knowledge. The representation of procedural and propositional declarative knowledge is well developed in Soar. However, we don't have well worked-out approaches to many other knowledge representation problems, such as the representation of quanti-

fied, uncertain, temporal, and episodic knowledge. The critical question is whether architectural support is required to adequately represent these types of knowledge, or whether such knowledge can be adequately treated as additional objects and/or attributes. Preliminary work on quantified [Polk & Newell, 1988] and episodic [Rosenbloom *et al.*, 1990] knowledge is looking promising.

Memory organization. An issue which often gets raised with respect to the organization of Soar's memory, and with respect to the organization of production memories in general, is the apparent lack of a higher-order memory organization. There are no scripts [Schank & Ableson, 1977], frames [Minsky, 1975], or schemas [Bartlett, 1932] to tie fragments of related memory together. Nor are there are any obvious hierarchical structures which limit what sets of knowledge will be retrieved at any point in time. However, Soar's memory does have an organization, which is derived from the structure of productions, objects, and working memory (especially the context hierarchy).

What corresponds to a schema in Soar is an object, or a structured collection of objects. Such a structure can be stored entirely in the actions of a single production, or it can be stored in a piecemeal fashion across multiple productions. If multiple productions are used, the schema as a unit only comes into existence when the pieces are all retrieved contemporaneously into working memory. The advantage of this approach is that it allows novel schemas to be created from fragments of separately learned ones. The disadvantage is that it may not be possible to determine whether a set of fragments all originated from a single schema.

What corresponds to a hierarchy of retrieval contexts in Soar are the production conditions. Each combination of conditions implicitly defines a retrieval context, with a hierarchical structure induced by the subset relationship among the combinations. The contents of working memory determines which retrieval contexts are currently in force. For example, problem spaces are used extensively as retrieval contexts. Whenever there is a problem solving context that has a particular problem space selected within it, productions that test for other problem space names are not eligible to fire in that context. This approach has worked quite well for procedural knowledge, where it is clear when the knowledge is needed. We have just begun to work on appropriate organizational schemes for episodic and declarative knowledge, where it is much less clear when the knowledge should be retrieved. Our initial approach has been based on the incremental construction, via chunking, of multi-production discrimination networks [Rosenbloom *et al.*, 1988a; Rosenbloom *et al.*, 1990]. Though this work is too premature for a thorough evaluation in the context of Soar, the effectiveness of discrimination networks in systems like Epam [Feigenbaum & Simon, 1984] and Cyrus [Kolodner, 1983] bodes well.

Level 2: Decisions The scope of Soar's decision level can be evaluated in terms of its speed, the knowledge brought to bear, and the language of control.

Speed. Soar currently runs at approximately 10 decisions/second on current workstations such as a Sun4/280. This is adequate for most of the types of tasks we currently implement, but is too slow for tasks requiring large amounts of search or very large knowledge bases (the number of decisions per second would get even smaller that it is now). The principal bottleneck is the speed of memory access, which is a function of two factors: the cost of processing individually expensive productions (the *expensive chunks* problem) [Tambe & Newell, 1988], and the cost of processing a large number of productions (the *average growth effect* problem) [Tambe, 1988]. We now have a solution to the problem of expensive chunks which can guarantee that all productions will be cheap — the match cost of a production is at worst linear in the number of conditions [Tambe & Rosenbloom, 1989] — and are working on other potential solutions. Parallelism looks to be an effective solution to the average growth effect problem [Tambe, 1988].

Bringing knowledge to bear. Iterated, parallel, indexed access to the contents of long-term memory has proven to be an effective means of bringing knowledge to bear on the decision process. The limited power provided by this process is offset by the ability to use subgoals when the accessible knowledge is inadequate. The issue of devising good access paths for episodic and declarative knowledge is also relevant here.

Control language. Preferences have proven to be a flexible means of specifying a partial order among contending objects. However, we cannot yet state with certainty that the set of preference types embodied in Soar is complete with respect to all the types of information which ultimately may need to be communicated to the decision procedure.

Level 3: Goals The scope of Soar's goal level can be evaluated in terms of the types of goals that can be generated and the types of problem solving that can be performed in goals. Soar's subgoaling mechanism has been demonstrated to be able to create subgoals for all of the types of difficulties that can arise in problem solving in problem spaces [Laird, 1983]. This leaves three areas open. The first area is how top-level goals are generated; that is, how the top-level task is picked. Currently this is done by the programmer, but a general intelligence must clearly have grounds — that is, motivations — for selecting tasks on its own. The second area is how goal interactions are handled. Goal interactions show up in Soar as operator interactions, and are normally dealt with by adding explicit knowledge to avoid them, or by backtracking (with learning) when they happen. It is not yet clear the extent to which Soar could easily make use of more

sophisticated approaches, such as non-linear planning [Chapman, 1987]. The third area is the sufficiency of impasse-driven subgoaling as a means for determining when meta-level processing is needed. Two of the activities that might fall under this area are goal tests and monitoring. Both of these activities can be performed at the memory or decision level, but when they are complicated activities it may be necessary to perform them by problem solving at the goal level. Either activity can be called for explicitly by selecting a "monitor" or "goal-test" operator, which can then lead to the generation of a subgoal. However, goals for these tasks do not arise automatically, without deliberation. Should they? It is not completely clear.

The scope of the problem solving that can be performed in goals can itself be evaluated in terms of whether problem spaces cover all of the types of performance required, the limits on the ability of subgoal-based problem solving to access and modify aspects of the system, and whether parallelism is possible. These points are addressed in the next three paragraphs.

Problem space scope. Problem spaces are a very general performance model. They have been hypothesized to underlie all human, symbolic, goal-oriented behavior [Newell, 1980]. The breadth of tasks that have so far been represented in problem spaces over the whole the field of AI attests to this generality. One way of pushing this evaluation further is to ask how well problem spaces account for the types of problem solving performed by two of the principal competing paradigms: planning [Chapman, 1987] and case-based reasoning [Kolodner, 1988].[8] Both of these paradigms involve the creation (or retrieval) and use of a data structure that represents a sequence of actions. In planning, the data structure represents the sequence of actions that the system expects to use for the current problem. In case-based reasoning, the data structure represents the sequence of actions used on some previous, presumably related, problem. In both, the data structure is used to decide what sequence of actions to perform in the current problem. Soar straightforwardly performs procedural analogues of these two processes. When it performs a lookahead search to determine what operator to apply to a particular state, it acquires (by chunking) a set of search control productions which collectively tell it which operator should be applied to each subsequent state. This set of chunks forms a procedural plan for the current problem. When a search control chunk transfers between tasks, a form of procedural case-based reasoning is occurring.

Simple forms of declarative planning and case-based reasoning have also been demonstrated in Soar in the context of an expert system that designs floor systems [Reich, 1988]. When this system discovers, via lookahead search, a sequence of operators that achieves a goal, it creates a declarative structure representing the sequence and returns it as a subgoal result (plan creation). This plan can then be used interpretively to guide performance on the immediate problem (plan following). The plan can also be retrieved during later problems and used to guide the selection of operators (case-based reasoning). This research does not demonstrate the variety of operations one could conceivably use to modify a partial or complete plan, but it does demonstrate the basics.

Meta-level access. Subgoal-based problem solving has access to all of the information in working memory — including the goal stack, problem spaces, states, operators, preferences, and other facts that have been retrieved or generated — plus any of the other knowledge in long-term memory that it can access. It does not have direct access to the productions, or to any of the data structures internal to the architecture. Nonetheless, it should be able to indirectly examine the contents of any productions that were acquired by chunking, which in the long run should be just about all of them. The idea is to reconstruct the contents of the production by going down into a subgoal and retracing the problem solving that was done when the chunk was learned. In this way it should be possible to determine what knowledge the production cached. This idea has not yet been explicitly demonstrated in Soar, but research on the recovery from incorrect knowledge has used a closely related approach [Laird, 1988].

The effects of problem solving are limited to the addition of information to working memory. Deletion of working memory elements is accomplished by a garbage collector provided by the architecture. Productions are added by chunking, rather than by problem solving, and are never deleted by the system. The limitation on production creation — that it only occurs via chunking — is dealt with by varying the nature of the problem solving over which chunking occurs [Rosenbloom *et al.*, 1990]. The limitation on production deletion is dealt with by learning new productions which overcome the effects of old ones [Laird, 1988].

Parallelism. Two principal sources of parallelism in Soar are at the memory level: production match and execution. On each cycle of elaboration, all productions are matched in parallel to the working memory, and then all of the successful instantiations are executed in parallel. This lets tasks that can be performed at the memory level proceed in parallel, but not so for decision-level and goal-level tasks.

Another principal source of parallelism is provided by the motor systems. All motor systems behave in parallel with respect to each other, and with respect to the cognitive system. This enables one form of task-level parallelism in which non-interfering external tasks can be performed in parallel. To enable further

[8] The work on Robo-Soar also reveals Soar's potential to exhibit reactive planning [Georgeff & Lansky, 1987]. The current version of Soar still has problems with raw speed and with the unbounded nature of the production match (the expensive chunks problem), but it is expected that these problems will be solved in the near future.

research on task-level parallelism we have added the experimental ability to simultaneously select multiple problem space operators within a single problem solving context. Each of these operators can then proceed to execute in parallel, yielding parallel subgoals, and ultimately an entire tree of problem solving contexts in which all of the branches are being processed in parallel. We do not yet have enough experience with this capability to evaluate its scope and limits.

Despite all of these forms of parallelism embodied in Soar, most implementations of the architecture have been on serial machines, with the parallelism being simulated. However, there is an active research effort to implement Soar on parallel computers. A parallelized version of the production match has been successfully implemented on an Encore Multimax, which has a small number (2-20) of large-grained processors [Tambe *et al.*, 1988], and unsuccessfully implemented on a Connection Machine [Hillis, 1985], which has a large number (16K-64K) of small-grained processors [Flynn, 1988]. The Connection Machine implementation failed primarily because a complete parallelization of the current match algorithm can lead to exponential space requirements. Research on restricted match algorithms may fix this problem in the future. Work is also in progress towards implementing Soar on message-passing computers [Tambe *et al.*, 1989].

Learning In [Steier *et al.*, 1987] we broke down the problem of evaluating the scope of Soar's learning capabilities into four parts: when can the architecture learn; from what can the architecture learn; what can the architecture learn; and when can the architecture apply learned knowledge. These points are discussed earlier, and need not be elaborated further here.

One important additional issue is whether Soar acquires knowledge that is at the appropriate level of generalization or specialization. Chunking provides a level of generality that is determined by a combination of the representation used and the problem solving performed. Under varying circumstances, this can lead to both overgeneralization [Laird *et al.*, 1986b] and overspecialization. The acquisition of overgeneral knowledge implies that the system must be able to recover from any errors caused by its use. One solution to this problem that has been implemented in Soar involves detecting that a performance error has occurred, determining what should have been done instead, and acquiring a new chunk which leads to correct performance in the future [Laird, 1988]. This is accomplished without examining or modifying the overgeneral production; instead it goes back down into the subgoals for which the overgeneral productions were learned.

One way to deal with overspecialization is to patch the resulting knowledge gaps with additional knowledge. This is what Soar does constantly — if a production is overspecialized, it doesn't fire in circumstances when it should, causing an impasse to occur, and providing the opportunity to learn an additional chunk that covers the missing case (plus possibly other cases). Another way to deal with overspecialized knowledge is to work towards acquiring more general productions. A standard approach is to induce general rules from a sequence of positive and negative examples [Mitchell, 1982; Quinlan, 1986]. This form of generalization must occur in Soar by search in problem spaces, and though there has been some initial work on doing this [Rosenbloom, 1988; Saul, 1984], we have not yet provided Soar with a set of problem spaces that will allow it to generate appropriate generalizations from a variety of sets of examples. So, Soar cannot yet be described as a system of choice for doing induction from multiple examples. On the other hand, Soar does generalize quite naturally and effectively when abstraction occurs [Unruh & Rosenbloom, 1989]. The learned rules reflect whatever abstraction was made during problem solving.

Learning behaviors that have not yet been attempted in Soar include the construction of a model of the environment from experimentation in it [Rajamoney *et al.*, 1985], scientific discovery and theory formation [Langley *et al.*, 1987], and conceptual clustering [Fisher & Langley, 1985].

Perception and motor control The scope of Soar's perception and motor control can be evaluated in terms of both its low-level I/O mechanisms and its high-level language capabilities. Both of these capabilities are quite new, so the evaluation must be even more tentative than for the preceding components.

At the low-level, Soar can be hooked up to multiple perceptual modalities (and multiple fields within each modality) and can control multiple effectors. The critical low-level aspects of perception and motor control are currently done in a standard procedural language outside of the cognitive system. The resulting system appears to be an effective testbed for research on high-level aspects of perception and motor-control. It also appears to be an effective testbed for research on the interactions of perception and motor control with other cognitive capabilities, such as memory, problem solving, and learning. However, it does finesse many of the hard issues in perception and motor control, such as selective attention, shape determination, object identification, and temporal coordination. Work is actively in progress on selective attention [Wiesmeyer, 1988a].

At the high end of I/O capabilities is the processing of natural language. An early attempt to implement a semantic grammar parser in Soar was only a limited success [Powell, 1984]. It worked, but did not appear to be the right long-term solution to language understanding in Soar. More recent work on NL-Soar has focused on the incremental construction of a model of the situation by applying comprehension operators to each incoming word [Lewis *et al.*, 1989]. Comprehension operators iteratively augment and refine the situation model, setting up expectations for the part of the utterance still to be seen, and satisfying ear-

lier expectations. As a side effect of constructing the situation model, an utterance model is constructed to represent the linguistic structure of the sentence. This approach to language understanding has been successfully applied to acquiring task specific problem spaces for three immediate reasoning tasks: relational reasoning [Johnson-Laird, 1988], categorical syllogisms, and sentence verification [Clark & Chase, 1972]. It has also been used to process the input for these tasks as they are performed. Though NL-Soar is still far from providing a general linguistic capability, the approach has proven promising.

Conclusion

In this article we have taken a step towards providing an analysis of the Soar architecture as a basis for general intelligence. In order to increase understanding of the structure of the architecture we have provided a theoretical framework within which the architecture can be described, a discussion of methodological assumptions underlying the project and the system, and an illustrative example of its performance on a multi-column subtraction task. In order to facilitate comparisons between the capabilities of the current version of Soar and the capabilities required to achieve its ultimate goal as an architecture for general intelligence, we have described the natural tasks for the architecture, the sources of its power, and its scope and limits. If this article has succeeded, it should be clear that progress has been made, but that more work is still required. This applies equally to the tasks of developing Soar and analyzing it.

References

[Bartlett, 1932] F. C. Bartlett. *Remembering: A Study in Experimental and Social Psychology.* Cambridge University Press, Cambridge, Eng., 1932.

[Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[Clark & Chase, 1972] H. H. Clark & W. G. Chase. On the process of comparing sentences against pictures. *Cognitive Psychology*, 3:472–517, 1972.

[DeJong & Mooney, 1986] G. DeJong & R. J. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1:145–176, 1986.

[Dietterich, 1986] T. G. Dietterich. Learning at the knowledge level. *Machine Learning*, 1:287–315, 1986.

[Etzioni & Mitchell, 1989] O. Etzioni & T. M. Mitchell. A comparative analysis of chunking and decision analytic control. In *Proceedings of the AAAI Spring Symposium on Limited Rationality and AI*, Stanford, CA, 1989.

[Feigenbaum & Simon, 1984] E. A. Feigenbaum & H. A. Simon. Epam-like models of recognition and learning. *Cognitive Science*, 8:305–336, 1984.

[Fisher & Langley, 1985] D. H. Fisher & P. Langley. Approaches to conceptual clustering. In *Proceedings of IJCAI-85*, pages 691–697, Los Angeles, CA, 1985.

[Flynn, 1988] R. Flynn. Placing Soar on the connection machine. Prepared for and distributed at the AAAI Mini-Symposium "How Can Slow Components Think So Fast", 1988.

[Fodor, 1983] J. A. Fodor. *The Modularity of Mind.* Bradford Books, MIT Press, Cambridge, MA, 1983.

[Georgeff & Lansky, 1987] M. P. Georgeff & A. L. Lansky. Reactive reasoning and planning. In *Proceedings of AAAI-87*, pages 677–682, Seattle, WA, 1987.

[Golding *et al.*, 1987] A. Golding, P. S. Rosenbloom, & J. E. Laird. Learning general search control from outside guidance. In *Proceedings of IJCAI-87*, Milan, 1987.

[Gupta & Tambe, 1988] A. Gupta & M. Tambe. Suitability of message passing computers for implementing production systems. In *Proceedings of AAAI-88*, pages 687–692, St. Paul, 1988.

[Hewitt & Kirsh, 1987] C. Hewitt & D. Kirsh. Personal communication. 1987.

[Hillis, 1985] W. D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[Hsu *et al.*, 1988] W. Hsu, M. Prietula, & D. Steier. Merl-Soar: Applying Soar to scheduling. In *Proceedings of the Workshop on Artificial Intelligence Simulation, The National Conference on Artificial Intelligence*, pages 81–84, 1988.

[Johnson *et al.*, 1989] T. R. Johnson, J. W. Jr. Smith, & B. Chandrasekaran. Generic Tasks and Soar. In *Working Notes of the AAAI Spring Symposium on Knowledge System Development Tools and Languages*, pages 25–28, Stanford, CA, 1989.

[Johnson-Laird, 1988] P. N. Johnson-Laird. Reasoning by rule or model? In *Proceedings of the 10th Annual Conference of the Cognitive Science Society*, pages 765–771, Montreal, 1988.

[Kolodner, 1983] J. L. Kolodner. Maintaining order in a dynamic long-term memory. *Cognitive Science*, 7:243–280, 1983.

[Kolodner, 1988] J. L. Kolodner, editor. *Proceedings of the DARPA Workshop on Case-Based Reasoning.* Clearwater Beach, FL, 1988.

[Laird & Newell, 1983] J. E. Laird & A. Newell. A universal weak method. Technical Report 83-141, Department of Computer Science, Carnegie-Mellon University, June 1983.

[Laird *et al.*, 1984] J. E. Laird, P. S. Rosenbloom, & A. Newell. Towards chunking as a general learning mechanism. In *Proceedings of AAAI-84*, pages 188–192, Austin, 1984.

[Laird *et al.*, 1986a] J. E. Laird, P. S. Rosenbloom, & A. Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[Laird *et al.*, 1986b] J. E. Laird, P. S. Rosenbloom, & A. Newell. Overgeneralization during knowledge compilation in Soar. In T. G. Dietterich, editor, *Proceedings of the Workshop on Knowledge Compilation*, Otter Crest, 1986. AAAI/Oregon State U.

[Laird *et al.*, 1987] J. E. Laird, A. Newell, & P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Laird *et al.*, 1989] J. E. Laird, E. S. Yager, C. M. Tuck, & M. Hucka. Learning in tele-autonomous systems using Soar. In *Proceedings of the NASA Conference on Space Telerobotics*, Pasadena, CA, 1989.

[Laird *et al.*, 1990] J. E. Laird, K. Swedlow, E. Altmann, & C. B. Congdon. *Soar 5 User's Manual.* The University of Michigan, 1990. In preparation.

[Laird, 1983] J. E. Laird. *Universal Subgoaling.* PhD thesis, Carnegie-Mellon University, 1983. (Available in Laird, J. E., Rosenbloom, P. S., & Newell, A. *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies*, Hingham, MA: Kluwer, 1986).

[Laird, 1986] J. E. Laird. Soar user's manual (version 4). Technical Report ISL-15, Xerox Palo Alto Research Center, 1986.

[Laird, 1988] J. E. Laird. Recovery from incorrect knowledge in Soar. In *Proceedings of AAAI-88*, pages 618–623, St. Paul, 1988.

[Langley *et al.*, 1987] P. Langley, H. A. Simon, G. L. Bradshaw, & J. M. Zytkow. *Scientific Discovery: Computational Explorations of the Creative Processes.* MIT Press, Cambridge, MA, 1987.

[Lewis *et al.*, 1989] R. L. Lewis, A. Newell, & T. A. Polk. Toward a Soar theory of taking instructions for immediate reasoning tasks. In *Proceedings of the 11th Annual Conference of the Cognitive Science Society*, Ann Arbor, MI, 1989.

[Minsky, 1975] M. Minsky. A framework for the representation of knowledge. In P. Winston, editor, *The Psychlogy of Computer Vision.* McGraw-Hill, New York, 1975.

[Minsky, 1986] M. Minsky. *The Society of Mind.* Simon and Schuster, New York, 1986.

[Mitchell *et al.*, 1986] T. M. Mitchell, R. M. Keller, & S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986.

[Mitchell, 1982] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[Newell & Rosenbloom, 1981] A. Newell & P. S. Rosenbloom. Mechanisms of skill acquisition and the law of practice. In J. R. Anderson, editor, *Cognitive Skills and their Acquisition*, pages 1–55. Erlbaum, Hillsdale, NJ, 1981.

[Newell & Simon, 1972] A. Newell & H. A. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, 1972.

[Newell *et al.*, 1989] A. Newell, P. S. Rosenbloom, & J. E. Laird. Symbolic architectures for cognition. In M. I. Posner, editor, *Foundations of Cognitive Science.* Bradford Books/MIT Press, Cambridge, MA, 1989.

[Newell, 1980] A. Newell. Reasoning, problem solving and decision processes: The problem space as a fundamental category. In R. Nickerson, editor, *Attention and Performance VIII.* Erlbaum, Hillsdale, N.J., 1980.

[Newell, 1990] A. Newell. *Unified Theories of Cognition.* Harvard University Press, Cambridge, MA, 1990.

[Nilsson, 1980] N. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, 1980.

[Polk & Newell, 1988] T. A. Polk & A. Newell. Modeling human syllogistic reasoning in Soar. In *Proceedings of the 10th Annual Conference of the Cognitive Science Society*, pages 181–187, Montreal, 1988.

[Powell, 1984] L. Powell. Parsing the picnic problem with a Soar3 implementation of Dypar-1. Department of Computer Science, Carnegie-Mellon University. Unpublished, 1984.

[Quinlan, 1986] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Rajamoney *et al.*, 1985] S. Rajamoney, G. F. DeJong, & B. Faltings. Towards a model of conceptual knowledge acquisition through directed experimentation. In *Proceedings of IJCAI-85*, pages 688–690, Los Angeles, CA, 1985.

[Reich, 1988] Y. Reich. Learning plans as a weak method for design. Department of Civil Engineering, Carnegie Mellon University. Unpublished, 1988.

[Rosenbloom & Laird, 1986] P. S. Rosenbloom & J. E. Laird. Mapping explanation-based generalization onto Soar. In *Proceedings of AAAI-86*, pages 561–567, Philadelphia, 1986.

[Rosenbloom & Newell, 1986] P. S. Rosenbloom & A. Newell. The chunking of goal hierarchies: A generalized model of practice. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach, Volume II*, pages 247–288. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[Rosenbloom *et al.*, 1985] P. S. Rosenbloom, J. E. Laird, J. McDermott, A. Newell, & E. Orciuch. R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7:561–569, 1985.

[Rosenbloom *et al.*, 1987] P. S. Rosenbloom, J. E. Laird, & A. Newell. Knowledge level learning in Soar. In *Proceedings of AAAI-87*, pages 499–504, Seattle, 1987.

[Rosenbloom *et al.*, 1988a] P. S. Rosenbloom, J. E. Laird, & A. Newell. The chunking of skill and knowledge. In B. A. G. Elsendoorn & H. Bouma, editors, *Working Models of Human Perception*, pages 391–410. Academic Press, London, 1988.

[Rosenbloom *et al.*, 1988b] P. S. Rosenbloom, J. E. Laird, & A. Newell. Meta-levels in Soar. In P. Maes & D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 227–240. North Holland, Amsterdam, 1988.

[Rosenbloom *et al.*, 1990] P. S. Rosenbloom, A. Newell, & J. E. Laird. Towards the knowledge level in Soar: The role of the architecture in the use of knowledge. In K. VanLehn, editor, *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, 1990. In preparation.

[Rosenbloom, 1988] P. S. Rosenbloom. Beyond generalization as search: Towards a unified framework for the acquisition of new knowledge. In G. F. DeJong, editor, *Proceedings of the AAAI Symposium on Explanation-Based Learning*, pages 17–21, Stanford, CA, 1988. AAAI.

[Rosenbloom, 1989] P. S. Rosenbloom. A symbolic goal-oriented perspective on connectionism and Soar. In R. Pfeifer, Z. Schreter, F. Fogelman-Soulie, & L. Steels, editors, *Connectionism in Perspective*. Elsevier, Amsterdam, 1989.

[Sacerdoti, 1974] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Saul, 1984] R. H. Saul. A Soar2 implementation of version-space inductive learning. Department of Computer Science, Carnegie-Mellon University. Unpublished, 1984.

[Schank & Ableson, 1977] R. Schank & R. Ableson. *Scripts, Plans, Goals and Understanding*. Lawrence Erlbaum, Hillsdale, NJ, 1977.

[Steier & Newell, 1988] D. M. Steier & A. Newell. Integrating multiple sources of knowledge into Designer-Soar an automatic algorithm designer. In *Proceedings of AAAI-88*, pages 8–13, St. Paul, MN, 1988.

[Steier *et al.*, 1987] D. M. Steier, J. E. Laird, A. Newell, P. S. Rosenbloom, R. Flynn, A. Golding, T. A. Polk, O. G. Shivers, A. Unruh, & G. R. Yost. Varieties of learning in Soar: 1987. In P. Langley, editor, *Proceedings of the Fourth International Workshop on Machine Learning*, pages 300–311, Los Altos, CA, 1987. Morgan Kaufmann Publishers, Inc.

[Steier, 1987] D. Steier. Cypress-Soar: A case study in search and learning in algorithm design. In *Proceedings of IJCAI-87*, pages 327–330, Milan, 1987.

[Tambe & Newell, 1988] M. Tambe & A. Newell. Some chunks are expensive. In J. Laird, editor, *Proceedings of the Fifth International Conference on Machine Learning*, pages 451–458, Ann Arbor, MI, 1988.

[Tambe & Rosenbloom, 1989] M. Tambe & P. S. Rosenbloom. Eliminating expensive chunks by restricting expressiveness. In *Proceedings of IJCAI-89*, Detroit, 1989.

[Tambe *et al.*, 1988] M. Tambe, Kalp D., A. Gupta, C. L. Forgy, B. Milnes, & A. Newell. Soar/PSM-E: Investigating match parallelism in a learning production system. In *Proceedings of ACM/SIGPLAN symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 146–161, 1988.

[Tambe *et al.*, 1989] M. Tambe, A. Acharya, & A. Gupta. Implementation of production systems on message passing computers: Simulation results and analysis. Technical Report CMU-CS-89-129, School of Computer Science, Carnegie Mellon University, April 1989.

[Tambe, 1988] M. Tambe. Speculations on the computational effects of chunking. Department of Computer Science, Carnegie Mellon University. Unpublished, 1988.

[Unruh & Rosenbloom, 1989] A. Unruh & P. S. Rosenbloom. Abstraction in problem solving and learning. In *Proceedings of IJCAI-89*, Detroit, 1989.

[Unruh *et al.*, 1987] A. Unruh, P. S. Rosenbloom, & J. E. Laird. Dynamic abstraction problem solving in Soar. In *Proceedings of the Third Annual Aerospace Applications of Artificial Intelligence Conference*, pages 245–256, Dayton, OH, 1987.

[VanLehn & Ball, 1987] K. VanLehn & W. Ball. Flexible execution of cognitive procedures. Technical Report PCG-5, Department of Psychology, Carnegie-Mellon University, June 1987.

[VanLehn, 1983] K. VanLehn. Felicity conditions for human skill acquisition: Validating an AI-based theory. Technical Report CIS-21, Xerox Palo Alto Reserch Center, November 1983.

[Washington & Rosenbloom, 1988] R. Washington & P. S. Rosenbloom. Applying problem solving and learning to diagnosis. Department of Computer Science, Stanford University. Unpublished, 1988.

[Wiesmeyer, 1988a] M. Wiesmeyer. Personal communication. 1988.

[Wiesmeyer, 1988b] M. Wiesmeyer. *Soar I/O Reference Manual, Version 2.* Department of EECS, University of Michigan, 1988.

[Wiesmeyer, 1989] M. Wiesmeyer. *New and Improved Soar IO.* Department of EECS, University of Michigan, 1989.

[Yost & Newell, 1989] G. R. Yost & A. Newell. A problem space approach to expert system specification. In *Proceedings of IJCAI-89*, Detroit, MI, 1989.

An Implementation of Indexical/Functional Reference for Embedded Execution of Symbolic Plans

Marcel Schoppers and **Richard Shu**
Advanced Decision Systems
1500 Plymouth Street
Mountain View, CA 94043

Abstract

We describe how we modified the Universal Plans execution engine to provide indexical/functional reference capabilities, thus allowing Universal Plans to interact with several identical, physical objects at once. This advance makes it easier for automatically constructed, symbolic plans, to reactively control physical robots. Our implementation of indexical/functional reference complements that of Agre and Chapman, in that our implementation:

- is designed for use in executing symbolic plans;
- does not require the planner to reason indexically;
- is capable of interacting with any number of objects at once;
- supports recursive plans for dismantling block towers of arbitrary size;
- finds objects to satisfy indefinite descriptions; and
- dynamically constructs the indefinite descriptions to satisfy.

We also explain why the use of any implementation of indexical/functional reference will complicate the detection of surprise events (serendipities).

1 Objectives

The work reported herein was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the U.S. Army Missile Command under contract DAAH01-90-C-0080, in part by IR&D funding from Advanced Decision Systems, and in part by the authors' own resources.

Deictic representation was introduced to the planning community in the PENGI paper [AGRE and CHAPMAN, 1987] and was devised to address the problem of how an embedded agent could manipulate physical objects. This problem is not addressed by logic-based representations (such as those used by automated planning programs) that do not indicate how a symbol inside an agent can be made to refer to any particular object in the real world. The association between a symbol and the real object it represents – if there is one – usually exists only in the head of some human being. If an artificial agent is to interact with real objects it must be able to dynamically create, destroy, and manipulate references to those objects: associations between structures inside the agent and objects outside the agent must be maintained by the agent itself. Agre and Chapman showed how to do that by devising an agent capable of establishing indexical/functional references to external objects, and as part of their solution, advocated interactionist, deictic representation over mentalist, logic-based representation.

Our involvement in both planning and situated activity gave us cause to examine the indexical/functional reference capabilities accruing from the use of deictic representation, and to integrate those capabilities with the use of a more conventional plan representation. Since Universal Plans were known to be amenable to symbolic representation, automatic synthesis, and reactive execution, we undertook to incorporate the capabilities of indexical/functional reference into our Universal Plans execution engine, without modifying the plan representation itself. Further, by taking careful note of the changes we had to make in the plan execution software along the way, we would not only come to a clearer understanding of the capabilities of indexical/functional reference, but would be in a position to describe it as constructed from a set of primitive capabilities. If, on the other hand, our attempt to achieve indexical/functional reference failed because of our self-imposed constraints, we would have isolated the precise point of conflict between it and assumptions built into symbolic planning technology.

Note that we were not trying to reconstruct Agre and Chapman's *implementation* of indexical/functional reference. Indeed, by requiring that our own implementation must be compatible with the use of symbolic plan representation and construction, we ran quite contrary to the motive of machine parsimony that drove the original implementation [AGRE, 1988]. We also wished to avoid such an implementation as would require the *planner* to reason indexically, and thus ran contrary to the direction of [SUBRAMANIAN and WOODFILL, 1989]. Our goal was to provide indexical/functional execution capabilities for plans produced by ordinary planners.

We summarize our results here: indexical/functional reference can indeed be achieved by proper construction of the execution engine for objective symbolic plan

representations. As a result, symbolic plans can now cause embedded agents to interact with physical objects, even when those objects are objectively indistinguishable. More interestingly, the symbolic implementation confers some advantages, such as making the number of relevant objects dynamically expandable, and allowing dynamic determination of the type of object to be referenced.

2 Experimental Setup

For this experiment we have constructed a modified Blocks World, in which blocks have names, labels, colors, and shapes. Each block's name is unique, and serves to identify the block. Block labels are letters, supposedly written on the block (e.g. "a"), and there may be any number of blocks sporting the same letter. Hence a plan can name a specific block, as usual, or can describe a desired block as one bearing a given letter. Similarly, a plan can describe a desired block as one having a given color, and there may be many blocks with the same color. Again, blocks can be spherical, pyramidal, or box-shaped.

If a plan referred to a desired color or shape, the plan would be describing, not naming, its objects. Similarly, if a plan referred to the labels printed on the blocks, the plan would be describing, not identifying, blocks. Only the blocks' names serve as designators in the logico-objective sense (as the atom **tweety** designates the only bird of that name).

The Universal Plan for building block towers consists of all the usual domain constraints of the Blocks World (e.g. a block can't be supported by two others simultaneously), plus a description of the effects of the available primitive actions, plus some additional information discovered by the planner. The planner considers the domain constraints, the effect descriptions, and the goals, and adds some new rules that function as advice to the plan interpreter concerning the order in which goals should be achieved. How confinement rules are discovered is detailed in [SCHOPPERS, 1989].

Universal Plan interpretation involves the backward chaining of domain constraints, effect descriptions, and confinement rules, subject to the truth or falsity (in the environment) of the plan's goals and the rules' preconditions – clearly, a goal that is already true in the environment does not have to be achieved. In the process of this backward chaining, the interpreter constructs a stack that holds the goals and supergoals of the current action. In other words, the interpreter traverses part of a goal tree (Figure 1). Although the interpreter makes a bee-line from the root node of the goal tree down a single path, it is much more enlightening to see the goal tree as a whole. An equivalent decision tree is shown in Figure 2. Neither the goal tree nor the decision tree ever exist in their entirety; they are merely pedagogical tools. The Universal Plans interpreter uses its knowledge of the domain to behave *as if* it were executing a decision tree.

Notice that the goal tree and the equivalent decision tree are tree *schemas*, containing unbound variables (in the Prolog convention logical variables begin with an upper case letter). This is important in allowing us

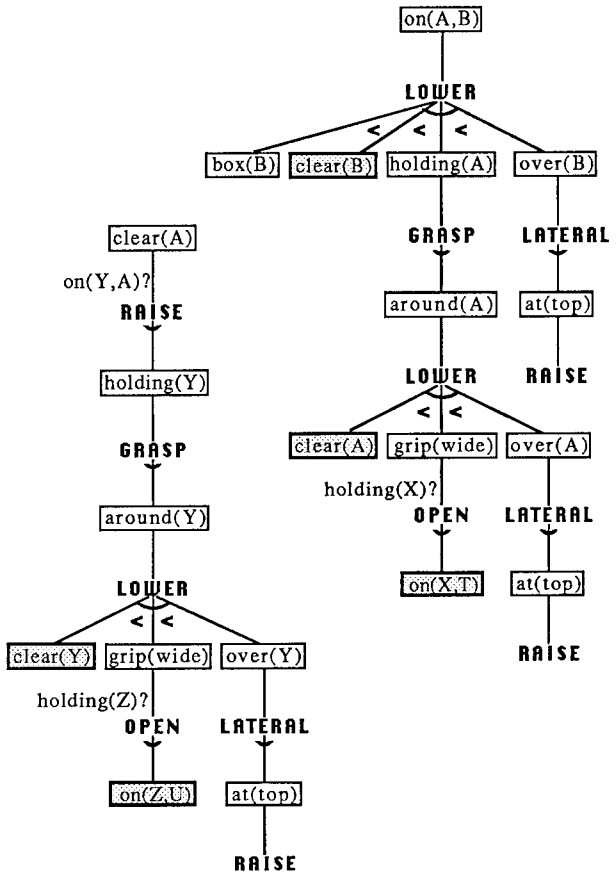


Figure 1: Part of the Universal Plan for STACK(a,b).

```
on(A,B) ?
  T) NO-OP
  F) box(B) ?
       T) clear(B) ?
       .    T) holding(A) ?
       .    .    T) over(B) ?
       .    .    .    T) LOWER
       .    .    .    F) at(top) ?
       .    .    .         T) LATERAL
       .    .    .         F) RAISE
       .    .    F) [subplan to GRASP A]
       .    F) [subplan to CLEAROFF B]
       F) FAIL
```

Figure 2: Decision Tree Equivalent of the STACK(A,B) Plan.

to invoke the plan (tree) with whatever parameters we want. Instead of invoking the plan to achieve **on(a,b)** – thus supposedly identifying particular blocks by means of unique atomic designators **a** and **b** – we may want to invoke the plan to achieve **on**("a","b"), with the intention that the plan should stack any one of the blocks *labelled* "a" onto any one of the blocks labelled "b". More explicitly, we might invoke the plan to achieve **on**(*p1,p2*) where *p1* and *p2* were descriptors, with *p1* describing (for example) "a thing that is a cubical block and is red and is labelled 'a'". That Universal Plans are plan schemas turned out to be very useful for our experiment.

3 Fundamentals of Indexical/Functional Reference

Now suppose we pose the goal **on**(*p1,p2*), where *p1* and *p2* are both (indefinite) descriptions of "any red cubical block labelled "a'". This goal might appear problematic, but need not be: if the world has several red cubical blocks labelled "a", the goal may be interpreted as meaning that we want one such block stacked atop another. If, furthermore, all red cubical blocks labelled "a" are identical, the goal can *still* mean the same thing. Why, then, have all planners to date resorted to creating such artificial distinctions as unique names? The problem is caused in part by the god's-eye view, the ability to identify every object, assumed by the prevailing logical formalisms, and in part by their "mentalism" [AGRE, 1988] – their inability to establish causal relationships with objects outside the representation itself. By posing the goal with descriptions rather than with names of blocks, we have already avoided the god's-eye view, and have thus opened a door to the possibility that a conventional plan representation might be capable of manipulating several identical blocks at once. Consider that in a constraint posting system, two variables may be identically constrained, yet may be bound differently. Similarly, two descriptions of blocks may be identical, and yet the two descriptions might apply to different blocks.

Clearly there is more to the issue than the binding of variables; the other half of the problem is the ability to establish causal relationships with objects outside the representation. Classically assembled plans merely name or describe objects, leaving the relationship between the object's representation and the object itself to the imagination of some human. Agre and Chapman built PENGI to *interact* with objects, and although it did so only in simulation, it seems plausible that PENGI could interact with *real* objects. If that capability is assumed, then PENGI could also interact with several *identical* real objects, and could do so without either making artificial distinctions or getting the objects confused.

We have already noted that two identical descriptions can (in principle) refer to different plan objects. Since Agre and Chapman push past plan objects to real objects, the problem for us becomes, how identical descriptions in a plan might be made to refer to different physical objects when the plan is executed, even if the objects themselves are perceptually identical. We solved that problem as follows.

Throughout the experiment we dealt with two software systems that had to communicate with each other. One system was a simulated robot arm in a simulated Blocks World. The arm could perform RAISE, LOWER, GRASP, OPEN and LATERAL actions, and the simulator would see to it that blocks moved with the arm as appropriate. The other system was a simulated agent, equipped with a database in which the agent could store any beliefs about the state of the simulated world, and controlled by a Universal Plan. The Universal Plan was the one of Figure 2. The two systems are shown in Figure 3; arrows represent flow of information.

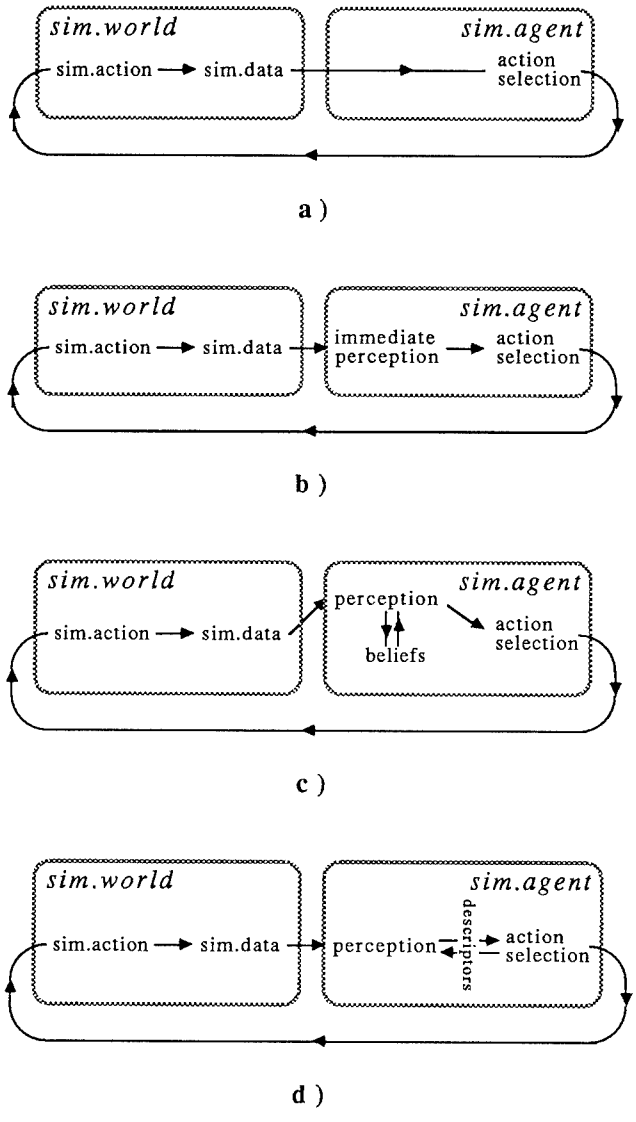


Figure 3: Steps Toward Indexical/Functional Reference.

In order to get the systems working together we began by cheating: we allowed the agent to examine the simulator's world model directly. The resulting flow of information is shown in Figure 3a, is exceedingly common in systems that make use of simulations, and is entirely pre-

posterous. To determine whether the condition **on(a,b)** is true or false, the agent must know what **a** and **b** refer to in the (simulated) world. Unfortunately, **a** is only a symbol, and this becomes obvious as soon as one tries to write code to evaluate **on(a,b)**. That code must traverse the *simulator's data structures* looking for a model of a block that has the label **a**. But the real world has no list of all known blocks! An alternative approach is to somehow map the symbol **a** into the machine-memory address of a data structure in the simulator. But again, no physical object in the real world is accessible via a machine-memory address!

To achieve a more plausible information flow from the (simulated) world to the agent, we constructed a perceptual interface and insisted that the agent could know nothing of the state of the (simulated) world except by using that perceptual interface (Figure 3b). We wrote code to implement some sensors, such as a camera, contact sensors on the agent's hand, and position sensors in the agent's arm. Naturally, those sensors had to have access to the simulator's data structures, so one might argue that the sensors did nothing to answer the criticisms of the previous paragraph. Nevertheless, there was a crucial difference. On the assumption that no-one will quarrel with any agent's ability to read contact and position sensors, let us consider our agent's use of its camera. We defined the camera as an effector that had to be controlled by the agent. The code that evaluated the agent's plan's conditions was thereby completely unable to make any use of the symbol **a** – what does **a** mean to a camera platform? Instead, the agent was forced to point its camera in some specified direction such that the camera's field of view included the location of the block being referred to as **a**. With the camera so positioned, the agent was then allowed to examine the *image* to determine whether the block being viewed looked as expected. The test of **on(a,b)** thus became a comparison of the spatial coordinates of two blocks. With all three types of sensors just mentioned – contact, position and camera – we were able to implement all of the tests needed for the Universal Plan.

There immediately arose a problem of how to map the plan symbol **a** to the location of some block. At the start of the agent's activities it knew nothing at all about the state of the (simulated) world. To solve this problem we implemented a camera movement procedure that systematically scanned the table until the camera viewed a block having the desired label. This whole scanning procedure was controlled by means of camera positioning coordinates.

Now when we executed the Universal Plan, we saw the camera scanning the table every time the plan needed to know anything about any block – numerous times *per block per action*. Worse, when we executed the Universal Plan using the non-unique block labels, the agent regularly got confused about *which* block labelled "a" it was working with.

This sad state of affairs was what we had been expecting. Clearly, when a plan representation refers to one of many identical blocks by means of a non-specific description, *something* must be added to that description to discriminate a particular one of the candidate blocks. The traditional solution had been to create unique names, but once one takes seriously the idea that the thing being named is out in the physical world and is *not* displaying its unique name in any way, that solution is seen to be bankrupt.

We repaired the agent's behavior by allowing perception to store beliefs about the positions of blocks, and to make use of existing beliefs to re-find blocks (Figure 3c). This use of beliefs solved two problems at once, a performance problem and a competence problem:

- It short-circuited the scan for blocks having a desired label, because the presence of beliefs about locations of previously found labels allowed the camera to find those labels again (if they were still there).
- It facilitated the tracking of specific blocks through time, even when several available blocks had the same desired label, because the location of the particular block being manipulated was enough to distinguish that block from other identical blocks.

Consequently, when perception was allowed to utilize beliefs, the agent could stack blocks, despite the presence of duplicates.

Although the perception component remembered where it last saw the blocks of interest, it made no assumption that blocks would actually be found where they were last seen. If a belief turned out to be significantly wrong, the agent would resort to scanning to find another block having the desired label, would determine the next action based on the location of that newly found block, and would continue with plan execution from there. Conversely, even if a belief about a block's location turned out to be approximately right, the believed location would still be updated. Beliefs used for perception purposes might more appropriately be called "expectations", a labelling that underscores both their dependence on the past, and their defeasibility.

Our agent could now interact with one specific member of a set of identical blocks, but could not achieve goals such as **on("a","a")** which require the agent to interact with *two* identical blocks. The problem was that although the agent's beliefs could properly distinguish (by their positions) two blocks both labelled "a", the plan itself made no such distinction, leaving the perception component confused about which block was meant when the plan referred to the label "a".

Our solution was to turn the plan's parameters – the descriptions of the relevant objects – into record structures that contained a slot for the location of the described block, and to have the perception component update the location slots. In this way the block descriptions being used by the plan were unambiguously associated with the perception component's knowledge about the actual blocks. We had thus completed the agent's ability to unambiguously refer to (and manipulate) physical blocks, even when two or more of the blocks being manipulated were perceptually identical.

Since the descriptor records were a natural place to put all descriptive information, we endowed them with slots for block label, color and shape, and made them

fully general. This meant that the agent's perception component could track blocks not by using descriptions to access beliefs, but by accessing and updating the descriptors themselves (Figure 3d). This eliminated the need for beliefs as literals in a database.

Although our use of descriptor records (or perhaps, distinct pointers to initially identical descriptions) might be regarded as "creating an artificial distinction", it was not nearly the same thing as forcing domain theories to give every block a different name. In our case the "artificial distinction" existed only between the blocks the plan was manipulating at the time, not between all blocks that could ever be in view. Our distinction might be regarded as a dynamically made one, whereas the logico-objective name distinction is a statically made one.

It remained only to assure ourselves that the set of capabilities provided by Agre's marker control operators [AGRE, 1988, p.220ff] could be emulated within our framework. These capabilities fell into five groups (according to Agre): marker comparison, marker inspection, marker assignment, indexing, and object comparison. Marker comparison and inspection operators provided such abilities as thresholding the distance between two tracked objects, testing whether two objects were approaching each other, and testing whether two markers referred to the same object. These tests could clearly be emulated by examining the location slots of our descriptors. The indexing and assignment operators caused markers to pick out objects of a specified type, or to pick out objects having a specified position relative to another object already being tracked. Again, we could emulate these either by manipulating the camera directly, or by manipulating the location slots of our descriptors. The object comparison operators checked whether objects were adjacent, whether objects were separated by empty space, and whether they were in given directions from each other. Consequently we are confident that we have captured the complete range of indexical/functional reference capabilities.

4 Further Developments

4.1 Treating Descriptions as Goals

Despite its being tidy and convenient, our generalization of descriptors to include slots for label, shape and color was soon felt to be a mistake. Remember that the preconditions of stacking one block on another include the requirement that the lower block must be a box, and that that precondition appears in the plan as a goal. Yet, when we asked the plan to stack one box-block on another, that very same condition would be part of the plan parameter, i.e. of the descriptor record. In both cases the condition would have to be checked perceptually. We considered it undesirable to be imposing the same condition in two different ways. And so we came to regard the conditions established by satisfying an object description as being "of one cloth" with the conditions established by goal achievement, and came to regard descriptions as goals. Then, when we wanted the plan to put any red sphere on any blue box, we would pose the goal

color(X,red) ∧ shape(X,sphere) ∧ color(Y,blue) ∧ shape(Y,box) ∧ on(X,Y)

and regarded it as part of goal achievement to find suitable objects for X and Y to refer to.

This change in viewpoint is not as strange as it may at first seem. If we were to augment the agent's capabilities by introducing a painting action, the above goal might induce the agent to paint things, an appropriate behavior that could not arise if object color was left purely as a slot in a description record. At the same time, one way of satisfying a color goal is to search out an object that already has the desired color; indeed, that is the only way to achieve a color goal when there is no painting action. Thus we came to regard painting and scanning as alternative ways of achieving color goals (like the build-or-buy choice in the constraint-based planner MOLGEN [STEFIK, 1981]), and came to regard scanning and verification as first-class actions that changed the agent's mental state:

```
SCANNING(X) ==  true +> known located(X)

COLORCHECK(X,C) ==
     located(X) +> known-whether color(X,C)
```

(The plan executor achieves **known-whether** P whenever it wants to know the truth value of any predicate P. See [SCHOPPERS, Sep 1990] for details.)

As a result of our removing label, color and shape information from the plan's description-record parameters, those records came to contain nothing but spatial location information. In the simulated Blocks World, the location of a block was a Cartesian coordinate triple (we had a three-dimensional world). In our application of indexical/functional reference to the NASA EVA Retriever robot, the location of an external object is specified by azimuth and elevation measured relative to the robot's body axes [SCHOPPERS, Sep 1990] we may yet decide to add distance information).

4.2 Conjunctive Descriptions and Perceptual Searching

When object descriptions are conjunctive, e.g. **color(X,red) ∧ shape(X,sphere)**, it is likely that an object found to satisfy one constraint will not satisfy the other. Thus the search for a red object will have succeeded and stopped, before the object's shape is examined. In our implementation, such failures lead to behavior that is at once backtracking and a resumption of visual scanning. The location of the last object to fail the conjoined goals becomes the starting point of the resumed visual scan.

If there is ultimately no object that satisfies a description, the plan executor resorts to normal backtracking, trying to achieve the goal by means of other actions. Our plan executor always performs a perceptual search first. In general there is a decision to be made for each constraint in a description: how long to try finding a suitable object readymade, or how much effort to spend on coercing an unsuitable object until it suits.

4.3 Creating and Constraining Variables Dynamically

Imagine a situation where the goal is to put a red box onto a blue box, but there is a green pyramid on the red box. The pyramid must be removed and put down elsewhere. Notice that according to the block stacking plan of Figure 1, the block to be moved must be clear, which may require removal of the block on top of it. The unwanted block is an object variable (Y) in the plan and does not appear in the goal, so is dynamically created by the plan executor. The newly created variable is then made to indexically/functionally refer to the green pyramid when the plan executor tries to achieve **known-whether on(Y,A)** with the action

```
LOOKON(A,Y) ==
      located(A) +> known-whether on(Y,A) .
```

Iff there is an object on A, the LOOKON action will cause Y to refer to the space above A and will return **true**. (Note, incidentally, that LOOKON need not scan for Y but can look directly at the space above A, whose location must be known.)

Now imagine a situation where the goal is once again to put one block on another, but the robot hand is already holding a green pyramid. The pyramid must be put down on something. That "something" is another object variable in the plan, and also does not appear in the goal. It is however constrained by the LOWERING action, which insists that only box-shaped blocks can function as supports for other blocks. Since the new object variable does not initially specify the location of a particular box, the plan executor performs a visual scan for any box-shaped object, and when one is found, deposits the green pyramid thereon. Thus the plan executor has dynamically created an object variable (and/or marker), has dynamically constrained the desired object to be any box, and has used the constrained variable to establish an indexical/functional reference in the usual way.

4.4 A Recursive Plan for Block Tower Dismantling

The mechanism described in the preceding subsection also allows a Universal Plan of fixed size to dismantle block towers containing an arbitrary number of blocks. Suppose the plan executor were given a goal to stack a red box onto a blue box, and it happened that the only red box was on the bottom of a tall tower of blocks. The plan executor would dynamically create an object variable for the block on top of the red box, then would try to remove it, and would realize that it too needed to be cleared first, thus calling the block-clearing plan recursively, and ultimately creating new object variables on-the-fly for each of the blocks in the tower. This behavior is similar to the recursive plan derived by [MANNA and WALDINGER, 1987], augmented with indexical/functional references to physical rather than represented objects.

The use of variables and recursion means that the size of the block tower affects not the size of the plan itself, but only the planning time, the plan execution time, and the size of the plan execution stack. As in BLOCKHEAD [CHAPMAN, 1989], our use of variables conclusively answers the plan-size objections raised by [GINSBERG, 1989]. The essential ingredient is an ability to dynamically create and bind variables to objects; the binding need not involve indexical/functional reference. (A Universal Plan of fixed size can also construct block towers of arbitrary size, but we cannot explain that here due to space limitations.)

4.5 Consequences for Serendipity Detection

We were surprised by indexical/functional reference in only one respect, namely that it seemed to completely thwart the ability of Universal Plans to detect serendipitous events. If we instructed the plan to stack any red box on any blue box, and we then put our own red box onto the blue box picked out by the plan, the plan would *remove* our red box so as to put its own red box on the blue box! While not exactly wrong, this behavior is inefficient and would be warranted only if we pointed out a particular red box and told the plan to put *that* red box on a blue box. We conclude that indexical/functional reference represents an extreme of firmness of purpose, and intend to explore mechanisms for automatically restoring the ability to detect serendipitous circumstances.

5 Conclusions

All the system designs of Figure 3 require the agent's plan predicates to take parameters that somehow indicate the objects to be examined. The arrangement of Figure 3b allowed the agent to manipulate objects external to it, but only if the objects were non-identical. The step from Figure 3b to Figure 3c gave the agent a tracking capability, which allowed the agent to manipulate a particular one of several identical objects. The step from Figure 3c to Figure 3d allowed the agent to manipulate several identical objects simultaneously. These steps show that indexical/functional reference is a confluence of: **1)** the use of indefinite descriptions of target objects, e.g. "any red sphere"; **2)** the ability to perceptually pick out and spatially locate candidate objects, e.g. "that thing"; **3)** tracking (when perception is visual), e.g. "that thing" (when it is one of several, or happens to be moving); **4)** an unambiguous association between plan objects and perceived object locations; and **5)** the ability to perceptually examine tracked objects, e.g. "is <tracked-object< red and a sphere". Indexical/functional reference falls naturally out of this combination, e.g. "that red sphere".

It may be of some interest that our block stacking plan acquired the ability to manipulate multiple identical blocks without being modified in any way that would have required indexical reasoning on the planner's part. For example, the constraints on objects were indefinite descriptions, or in logical terms, existentially quantified formulae. This means that for the planner, object variables could have logico-objective semantics, while for the plan executor they could have indexical/functional semantics. Is this possible difference of semantics between planner and plan executor a deficiency or an advantage?

Finally, our implementation of indexical/functional reference within a symbolic framework extended the former by providing the ability to dynamically create, constrain and assign references.

References

[Agre and Chapman, 1987] P. AGRE and D. CHAPMAN. Pengi: an implementation of a theory of activity. In *Proc AAAI*, pages 268–272, 1987.

[Agre, 1988] P. AGRE. *The Dynamic Structure of Everyday Life.* Tech rept, AI Lab, MIT, 1988.

[Chapman, 1989] D. CHAPMAN. Penguins can make cake. *AI Magazine*, 10:4:45–50, 1989.

[Ginsberg, 1989] M. GINSBERG. Universal planning: an (almost) universally bad idea. *AI Magazine*, 10:4:40–44, 1989.

[Manna and Waldinger, 1987] Z. MANNA and R. WALDINGER. How to clear a block: a theory of plans. *Journal of Automated Reasoning*, 3:4:343–377, 1987.

[Schoppers, 1989] M. SCHOPPERS. *Representation and Automatic Synthesis of Reaction Plans.* Report, Dept of Computer Science, University of Illinois at Urbana-Champaign, 1989.

[Schoppers, Sep 1990] M. SCHOPPERS. Automatic synthesis of perception driven discrete event control laws. In *Proc 5th IEEE Internat'l Symp on Intelligent Control*, page XX, Sep 1990.

[Stefik, 1981] M. STEFIK. Molgen part 1: planning with constraints. *Artificial Intelligence*, 16:141ff, 1981.

[Subramanian and Woodfill, 1989] D. SUBRAMANIAN and J. WOODFILL. Making situation calculus indexical. In *Proc 1st Internat'l Conf on Principles of Knowledge Rep'n and Reasoning*, pages 467–474. Morgan Kaufman, Los Altos, 1989.

OPIS: An Integrated Framework for Generating and Revising Factory Schedules[1]

Stephen F. Smith, Peng Si Ow[2], Nicola Muscettola, Jean-Yves Potvin[3] and Dirk C. Matthys

Center for Integrated Manufacturing Decision Systems
The Robotics Institute
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Practical solutions to the production scheduling problem must provide two broad capabilities: (1) an ability to efficiently generate schedules that reflect the actual constraints and objectives of the manufacturing environment, and (2) an ability to incrementally revise these schedules over time in response to unexpected executional circumstances. In this paper, we advocate a common view of predictive and reactive scheduling as an incremental problem solving process that is opportunistically focused by characteristics of the current solution constraints. We describe the architecture of OPIS (OPportunistic Intelligent Scheduler), which defines a general framework for configuring scheduling systems according to this view. We then examine the scheduling knowledge (e.g. analysis and scheduling methods, schedule generation/revision strategies) that is exploited within this architecture by the current OPIS scheduler. Experimental studies with the OPIS scheduler have demonstrated the potential of this constraint-directed scheduling methodology in both predictive and reactive scheduling contexts. [4]

1. Introduction

The broad goal of production scheduling is to produce a factory behavior where parts are produced in a timely and cost-effective manner. In most manufacturing environments, achievement of this goal is confounded by two factors: the *complexity* of operational level decision-making and the *unpredictability* of factory operations. Problem complexity derives from the need to determine assignments of shared resources (machines, operators, transport devices) to the manufacturing activities of many competing production processes over time which are both feasible from the standpoint of temporal process restrictions and resource capacity limitations, and satisfactory in the sense that they result in good overall factory performance. This latter requirement typically involves compromise among a diverse and conflicting set of production objectives (e.g. meeting deadlines, minimizing work-in-process, etc.). Problem complexity argues strongly for the advance development of production schedules. This provides a basis for anticipating constraint interactions (in particular, resource contention) and minimizing their harmful effects on factory performance. At the same time, the factory floor is typically a dynamically changing environment. Machines break down, raw materials fail to arrive on time, partially manufactured parts fail to meet quality control standards and require rework, operators call in sick, etc. Thus, even an ability to produce advance schedules that accurately reflect the constraints and objectives of the production environment is likely to be of limited practical utility without a companion ability to reactively manage these schedules in response to changing circumstances. The reactive scheduling problem raises additional requirements. In addition to maintaining the quality of the schedule from the standpoint of production objectives, it is also important to maintain a degree of stability in planned operations (since of a schedule sets a large number of interdependent processes in motion) and to produce results within acceptable response time constraints (to keep the manufacturing system operating).

---

[1]This paper previously appeared in *Journal of the Operational Research Society*, Vol. 41, No. 6, pp. 539-552 (1990). It is reprinted here by permission.

[2]Current Address: IBM Entry Systems Division, Austin, Texas.

[3]Current Address: Centre de Recherche sur les Transports, Universite de Montreal, Canada.

[4]This research was supported in part by the Air Force Office of Scientific Research under contract F49620-82-K-0017, International Business Machines Inc. under contract number 71223046, and by the CMU Robotics Institute. Additional support was provided to Jean-Yves Potvin by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Historically, the production scheduling problem has been treated strictly from either a global optimization perspective or a local control perspective. One body of work (e.g. [Graves 81]) has focused on the development of optimal solutions to various classes of scheduling problems. Unfortunately, such results have been obtainable only under very restrictive problem assumptions, which bear little relationship to actual manufacturing environments. Other research has focused on the development of local dispatch priority heuristics [Panwalker&Iskander 77]. These approaches do provide a robust basis for operational decision-making in the face of an unpredictable environment. However, the ability of such local decision-making to effectively optimize overall performance depends on the sensitivity of the decision rule to the dynamics of the manufacturing system, and again the simplifying assumptions made in most of this work are not reflective of most actual manufacturing environments. Scheduling techniques used in practice typically provide only rough guidance for operational decision-making, and there is no support for reactively revising this guidance as unexpected events occur. Often, predictive plans are based on artificial constraints (e.g. standard lead times) that accommodate factory floor inefficiencies, in essence advocating predictable factory results as a substitute for good factory results.

More recent work in knowledge-based scheduling [Fox 83, Fox&Smith 84, Smith&Ow 85, Smith et. al. 86] has attempted to provide more effective solutions to the production scheduling problem, emphasizing the use of heuristic scheduling techniques that are directed by knowledge of the active constraints and objectives in the target production environment. This work has led to an integrative view of predictive and reactive scheduling as an opportunistic problem solving process [Ow&Smith 88, Ow et. al. 88], which forms the basis of the OPIS (OPportunistic Intelligent Scheduler) factory scheduling system. The term opportunistic reasoning has been used to characterize a problem solving process whereby activity is consistently directed toward those actions that appear most promising in terms of furthering the current problem solving state. In the case of OPIS, it refers to an incremental scheduling methodology where characteristics of current solution constraints (e.g. likely areas of resource contention, schedule conflicts resulting from unanticipated external events) are used to dynamically focus attention on the most critical decisions that remain to be made/revised.

OPIS implements this approach to scheduling via a "blackboard style" system organization [Erman et. al. 80], wherein a set of distinct methods, referred to as knowledge sources (KSs), are selectively employed to generate, revise or analyze specific components of the overall schedule. Scheduling methods vary in the types of subproblems they can solve (i.e. the problem decomposition assumptions they are based on), and in the types of constraints and objectives that are emphasized. A control cycle that combines constraint propagation and consistency maintenance techniques with heuristics for subproblem formulation is used to coordinate overall scheduling activity. Experimental results with the OPIS scheduler in the context of realistic production scheduling problems have demonstrated the potential of this approach in both predictive and reactive scheduling contexts.

In this paper, we describe the structure and operation of the OPIS scheduler. First, we consider the basic principles that motivate our approach. Next we describe the principal components of the OPIS scheduling architecture and the framework it provides for opportunistic constraint-directed scheduling. We then examine the scheduling knowledge (i.e. analysis and scheduling methods, schedule generation and revision strategies) that is exploited within this architecture by the current OPIS scheduler, and summarize experimental results that have been obtained. We conclude with a discussion of the directions of our current research.

2. Opportunistic, Constraint-Directed Scheduling

The generation of an assignment of resources, start times and end times to operations that satisfies temporal process and resource capacity constraints and effectively balances a set of conflicting objectives is a combinatorial search problem. To manage the complexity of this search, it is necessary to make heuristic assumptions about how the problem is to be decomposed and where search effort is to be concentrated. Such assumptions, however, do affect the quality of the result. In dispatch-based scheduling approaches, for example, the problem is decomposed principally in an event-based fashion (i.e., scheduling decisions are made in chronological order), and secondarily into a set of local resource scheduling problems. Thus, search is confined to the operations that are eligible to acquire a free resource at a given point in time. On the other hand, the ability to optimize overall performance objectives is limited by the lack of a view of the future consequences of the scheduling decisions that are made. A decision made at a given point in time may unnecessarily restrict alternatives for critical future decisions which eventually leads to otherwise avoidable problems (e.g. unnecessary downstream congestion).

In simplest terms, OPIS advocates an approach to incremental scheduling wherein the order and manner in which decisions are made (or revised) is not fixed a priori but are instead determined dynamically according to the structure of the constraints implied by the current solution state. The approach is motivated by the desire to circumvent the inherent limitations of any fixed

decomposition strategy with respect to optimization of various scheduling objectives while retaining an efficient search process. Two basic types of problem decomposition strategies (or local scheduling perspectives) are considered within OPIS as a basis for heuristically structuring the scheduling process: resource-based and order-based. Each offers distinct advantages and disadvantages from the standpoint of addressing various scheduling objectives. A resource-based approach yields subproblems that localize attention to the schedule of a specific resource, and promotes optimal resolution of conflicts involving the set of operations that are competing for that resource (e.g., scheduling decisions that minimize setups and overall order tardiness). At the same time, interactions among operations belonging to the same order due to precedence constraints cut across several subproblems, and cannot be effectively addressed. An order-based approach provides an orthogonal viewpoint. Here, each subproblem relates to the schedule of a particular order, and optimal resolution of conflicts involving the operations that must be performed to produce a given order (e.g. scheduling decisions that minimize work-in-process time) is promoted. But interactions among operations competing for the same resource are now fall outside of any one subproblem.

Opportunistic use of multiple local scheduling perspectives raises a number of difficult issues with respect to overall control of the scheduling process: What is the most appropriate way to approach the problem at any point? What is the most important subproblem to solve? In what manner is a given problem most effectively solved? How does one resolve inconsistencies that arise, due either to interactions between different solved subproblems or to unexpected results that occur on the factory floor? As suggested above, OPIS relies on repeated analysis of the characteristics of current solution constraints to guide this decision-making. For example, if analysis of initial problem constraints indicates resources that are highly contended for, then schedule generation will proceed by first constructing candidate schedules for these bottleneck resources. These decisions can be seen as most critical to the quality of the overall solution, and also represent the subproblems with the fewest scheduling alternatives (since there is no reason to consider insertion of idle time in bottleneck resource schedules). These candidate bottleneck schedules then anchor the search by constraining alternatives for the decisions that remain to be made.

Similarly, analysis of the constraints that comprise inconsistent solution states provides information relating to the criticality of revising various scheduling decisions and the opportunities (flexibilities) that exist for efficient reaction. In this case, analysis is centered around the actual conflicts that exist in the current schedule, and subproblem formulation heuristics are directed at local resolution of these conflicts. Depending on the nature of the formulated subproblem, it is quite possible that subproblem solution will lead to the introduction of additional conflicts that must be subsequently responded to. For example, indication of an unexpected machine failure might necessitate considerable rescheduling of other substitutable machines, which, in turn, is likely to have disruptive effects on the downstream activities of rescheduled orders. Given the tightly coupled nature of scheduling decisions, it is extremely difficult (if not impossible) to predict the effects (i.e. the ripple effect) of a given local reaction. In the absence of an understanding of the behavior of the manufacturing environment, there is little alternative to such an opportunistic approach.

3. The OPIS Scheduling Architecture

The OPIS scheduling system can be seen at two levels. At one level, it defines a specific (albeit complex) heuristic procedure for generating and revising factory schedules opportunistically. At another level, it defines a general framework, or scheduling architecture, for organizing and applying scheduling heuristics in an opportunistic, constraint-directed fashion. The OPIS scheduling architecture thus provides a structure for developing other opportunistic scheduling procedures [Potvin&Smith 89]. In this section, we describe OPIS from an architectural standpoint. In later sections, we turn attention to the currently implemented OPIS scheduler.

As indicated at the outset, The OPIS scheduling architecture incorporates principles of standard blackboard style architectures and similarly assumes an organization comprised of a number of knowledge sources (KSs) that extend, revise and analyze a globally accessible solution (in this case the factory schedule). Within OPIS, two types of KSs are distinguished: *Analysis* KSs, which examine specific components of the global schedule and build abstract characterizations of the current solution state, and *Scheduling* KSs, which constitute alternative methods for manipulating the global schedule. Scheduling KSs implement the different local scheduling perspectives that might be adopted.

In support of opportunistic schedule generation and revision, the OPIS scheduling architecture combines two principal components: a schedule maintenance subsystem, for incrementally maintaining a representation of current solution constraints, and an event-driven control cycle, for coordinating the use of scheduling and analysis KSs. The former provides both a basis for analyzing aspects of the current scheduling state and a means for communicating scheduling constraints among different formulated subproblems. The latter provides a structure for specifying and a mechanism for applying the control knowledge necessary to effectively employ various KSs.

3.1. Schedule Maintenance

At the core of the system is an incrementally maintained representation of the current schedule. This representation is defined and maintained relative a hierarchical model of the constraints of the production environment. In this model, production plans for various part types are represented as hierarchies of operations, with aggregate operations designating either more detailed sub-processes or sets of exclusive alternatives. Operation precedence and duration constraints are embedded in these hierarchies, as well as specifications of required resources. Individual resources are grouped into successively larger work areas to provide resource descriptions at each level of abstraction in the production plans, and constraints on resource allocation (e.g. capacity, hours of operation) are specified at each level. A utility-based preference representation is used to encode various scheduling objectives and operational preferences (e.g. machine preferences).

This factory model provides a structure for representing and maintaining current solution constraints at multiple levels of abstraction. This representation includes a specification of the current time bounds on the execution of each manufacturing operation that has been or may be scheduled (since some operations designate alternatives which will become undefined as choices are made), and a specification of the current available capacity of each resource over time. As additional scheduling decisions are made or the constraints implied by particular factory status updates are introduced, the schedule maintenance system combines these new constraints with the constraints specified in the underlying factory model (e.g. order release/due dates, operation durations, operation precedence relations, resource requirements, resource capacity limitations) to update the time bounds and available capacity representations respectively of related operations and resources at all defined levels of abstraction. Thus, an unscheduled operation's time bounds at any point reflect the set of allocation decisions compatible with both the specified constraints on factory operations and any scheduling decisions that have been made.

Constraint propagation in response to schedule changes can lead to the detection of two types of conflicts:

- time conflicts: situations where either the time bounds or scheduled execution times of two operations belonging to the same order violate a defined precedence constraint.
- capacity conflicts: situations where the resource requirements of a set of currently scheduled operations exceed the available capacity of a specific resource over some interval of time.

The recognition of such conflicts signals the need for schedule revision and provides a basis for focusing this activity.

Details of this approach to schedule maintenance can be found in [LePape&Smith 87]. Representation of the underlying factory model is described in [Smith 89].

3.2. Coordinating the Scheduling Process

The use of various analysis and scheduling KSs in the solution of specific scheduling problems is coordinated by a designated KS called the *top-level manager*. This KS implements the system's basic control cycle, which, in turn, defines a framework for specification and organization of the system's (heuristic) control knowledge.

Generally speaking, coordination of the scheduling effort by the top level manager proceeds as an event-driven process. Changes in the state of the schedule, introduced either by internal problem-solving activity (e.g. generating a schedule for a given order) or by external factory status updates (e.g. notification of a machine breakdown), are detected by the schedule maintenance system and posted as control events to the top-level manager at the beginning of each problem solving cycle. Three basic types of control events can be posted: (1) an *incomplete hypothesis* event, which indicates that unscheduled operations remain, (2) an *elementary conflict* event, which indicates the presence of an inconsistency in the schedule (i.e. a time or capacity constraint violation), and an *opportunity* event, which indicates the possibility for schedule improvement due to an unexpected "loosening" of time or capacity constraints[5]. On any given cycle, the set of posted events leads to the execution of a particular scheduling action, and problem solving continues until the set of posted control events becomes empty. In such a case, a complete and consistent schedule has been obtained.

Figure 3-1 depicts the top-level control cycle in more detail, and identifies its 4 main steps. Each is briefly summarized below.

3.2.1. Event Selection

The first step of the top-level control cycle is event selection, which is concerned with identifying, from the set of currently posted events, the most appropriate problem to focus on. This is accomplished in two steps:

1. Event aggregation. It is often the case that individual events are related in some manner and would be better addressed simultaneously. During event aggregation, knowledge of such relationships (provided to the system as a set of aggregation heuristics) is applied to the set of posted events. In cases where specific relationships are detected, *aggregate* events are created and added to the list of posted events. The aggregation heuristics employed in the current OPIS scheduler consider only one

[5] opportunity events are not exploited within the current OPIS scheduler

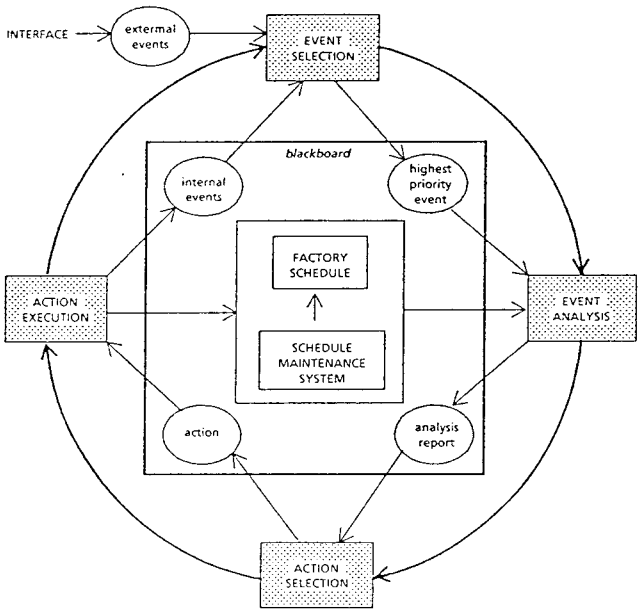


Figure 3-1: The top-level control cycle

type of relationship: commonality in the resources involved in posted elementary conflicts (e.g. capacity conflicts involving same resource). However, other aggregation heuristics are certainly possible and potentially useful (e.g. commonality of the orders involved in the conflicts, conflicts involving high priority orders or tardy orders).

2. Event prioritization. After this preprocessing of posted control events, prioritization heuristics are applied to select the specific event to be responded to in the current cycle. Within the current scheduler, event priority is a function first of event type (aggregate conflicts are more important than elementary conflicts, which in turn are more important than incomplete-hypothesis events), and second of the urgency of the event (with the highest priority event being the one that is closest in time to the current time). All events other than the highest priority event are left pending until the next cycle.

3.2.2. Event Analysis

Having identified a focal point for problem solving (as represented by the highest priority control event), an analysis KS is selected to examine the portion of the current schedule "surrounding" this focal point in more detail. The goal of this event analysis step is to summarize essential aspects of current time and capacity constraints (i.e. their relative looseness or tightness), providing a basis for determining how to best respond to the event (see action selection below). This characterization is referred to as an *analysis report*. Within the current implementation, the selection of analysis KS depends on the type of event under consideration. In the case of an incomplete-hypothesis event, where the problem solving focus is one of extending the current schedule, a "capacity analysis" is performed. In the case of a conflict event, where the focus is schedule revision, a "conflict analysis" is performed (an overview of these KSs is provided later).

3.2.3. Action Selection

The goal of action selection is to formulate the most appropriate scheduling task to execute in response to the control event under consideration. This requires determination of a particular component of the overall schedule to extend or revise, selection of a particular scheduling KS to carry out the task, and, depending on the KS selected, parameterization of the solution procedure. Action selection is accomplished through application of a set of subproblem formulation heuristics to the analysis report produced during event analysis. These heuristics combine knowledge of the implications of various characteristics of current time and capacity constraints with knowledge of the strengths and weaknesses of different scheduling KSs, and constitute the core of the system's theory of constraint-directed scheduling. The subproblem formulation heuristics currently employed in the OPIS scheduler are also described later in this paper.

3.2.4. Action Execution

The final step of the top-level control cycle is execution of the formulated scheduling task. This yields changes to the current schedule, and the consequences of these changes are inferred by the schedule maintenance system. Any detected constraint conflicts and/or unscheduled operations are posted as control events to the top-level manager and the control cycle repeats.

4. The OPIS Scheduler

The OPIS scheduling architecture provides a general framework for opportunistic, constraint-directed scheduling. We now turn attention to the specific scheduling and analysis methods that are exploited in the current OPIS scheduler, and the subproblem formulation heuristics that govern their use.

4.1. Generating and Revising Scheduling Decisions

As indicated previously, scheduling KSs constitute the actual methods available to the system for extending and revising the current schedule. Scheduling KSs vary in the types of scheduling subproblems they can solve (i.e. the local scheduling perspective that is assumed) and in the types of scheduling constraints and objectives that they emphasize. Accordingly, each has particular strengths and weaknesses with regard to generating and maintaining the

overall schedule. Table 4-1 summarizes the behavioral characteristics of each method along different dimensions. The entries for a given scheduling action are assigned qualitative values in the range from 0 to 1, with 0 being the lowest possible rating and 1 the highest. The first two rows indicate the relative strengths of each method from the standpoint of optimizing various scheduling objectives. The remaining three rows characterize the disruptive behavior of each method when used in schedule revision contexts. Each of these scheduling KSs is summarized below.

Order Scheduler (OSC)

The Order Scheduler provides a method for generating or revising scheduling decisions relative to some contiguous portion of a specific order's production plan. It implements the constraint-directed heuristic search technique originally developed in the ISIS scheduling system [Fox&Smith 84]. This method is characterized by the use of a beam search to explore alternative sets of resource assignments and execution intervals, evaluating various alternatives with respect to how well the decisions satisfy relevant preference constraints (e.g. meeting due dates, work-in-process time objectives, machine preferences, etc.). Heuristic knowledge relating to the relative importance of different preferences and the relative utility of the various alternatives over which each preference is defined provides the basis for this evaluation. In invoking OSC, resource availability constraints can be made more or less "visible". It can either be constrained to consider only execution intervals for which resource capacity currently exists, which we designate as the complete visibility (CV) OSC, or allowed to consider capacity allocated to lower priority orders as available, which we designate as the prioritized visibility (PV) OSC. Since the latter case admits the possibility of introducing additional capacity conflicts into the schedule (leading to "bumping" of lower priority orders), a decision to invoke the PV-OSC trades off potential additional disruption for some ability to perform resource-based optimization (hence the value 0.5 for this characteristic in Table 4-1). OSC can also be parameterized to conduct its search either forward or backward through an order's production plan from a given start or end time anchor. In situations where resource capacity is fairly plentiful, the search tends to lead to minimization of work-in-process time in the direction of the search anchor.

Resource Scheduler (RSC)

The Resource Scheduler provides a method for generating or revising the schedule of a designated resource or collection of substitutable resources (i.e. an aggregate resource). The method is predicated on the assumption that contention for the resource in question is high and, thus, emphasizes efficient resource utilization (e.g. there is no need to consider slack time between operations). It generates scheduling decisions using an iterative dispatch-based approach, adding one or more operations to the schedule of the resource under consideration at each dispatch cycle. A collection of dispatch heuristics are selectively employed to provide sensitivity to different preference constraints, the principal being Ow's Idle Time rule [Ow 85]. Full details of this approach can be found in [Ow&Smith 88]. When invoked in reactive contexts, an attempt is made to retract only as many scheduling decisions as necessary to resolve the problem at hand. This is accomplished by assuming that a new schedule will be generated forward in time from the point of the current problem, but remembering the old schedule. After each dispatch scheduling cycle, a check is performed to see if the new schedule can be consistently merged with the fragment of the old schedule consisting of the operations that have yet to be placed in the new schedule.

Right Shifter (RSH)

The RSH implements a considerably less sophisticated reactive method which simply "pushes" the scheduled execution times of designated operations forward in time by some designated amount. Such initial shifts can introduce both time conflicts and capacity conflicts. However, these conflicts are internally resolved by propagating the shifts through resource and order schedules to the extent necessary. Thus, the RSH will not introduce any new conflicts into the overall schedule.

Demand Swapper (DSW)

Demand Swapping is a specialized reactive method applicable in situations where an operation has become unexpectedly and significantly delayed (e.g. as a result of required rework if a part fails to meet quality standards). It exchanges the remaining portion of the affected order's schedule with the correspondent portion of the schedule of another order of the same type so as to minimize their combined tardiness. Note that the DSW is not necessarily a conflict resolution strategy. It is more appropriately viewed as a scheduling action that improves the character of the conflict.

4.2. Analysis of Current Solution Constraints

Analysis of current scheduling constraints provides the basis for differentiating between potential scheduling actions at each point in the scheduling process. As previously indicated, different analysis KSs are employed within OPIS in generative and reactive scheduling contexts. Each produces an analysis report summarizing constraint characteristics relevant to action selection in the context it supports.

Capacity Analysis

The Capacity Analyzer (CAN) is invoked when the

	RSC	CV-OSC	PV-OSC	RSH	DSW
resource-based optimization	1	0	0.5	0	0.5
order-based optimization	0	1	1	0	1
time conflict avoidance	0	1	1	1	0.5
cap. conflict avoidance	1	1	0	1	1
sequence stability	0	0	0	1	0

Table 4-1: Characteristics of Scheduling KSs

scheduler's current focus is incomplete-hypothesis event (i.e. when unscheduled manufacturing operations have been detected). The control decision at hand in this case is how to best extend the current schedule. To support this decision-making, the Capacity Analyzer computes estimates of the expected level of contention for resources required by operations that remain to be scheduled. Operating at an abstract level in the hierarchical model, capacity analysis proceeds by first constructing a rough, infinite capacity schedule that satisfies the time constraints of all unscheduled operations (employing a general line balancing heuristic where choices in resource assignments exist), and then superimposing resource availability constraints to compute resource "demand/supply" ratios over time. These estimates are used to identify likely bottleneck areas in the evolving schedule. An alternative, probabilistic approach to providing this global view of resource contention is described in [Muscettola&Smith 87].

Conflict Analysis

The Conflict Analyzer (CONAN), alternatively, is invoked when the system's current focus is a conflict event (indicating that one or more inconsistencies have been introduced into the schedule). In this case, the control decision to be made concerns how to best revise the current schedule to restore feasibility. In contrast to the capacity analysis, conflict analysis is concerned with characterizing a localized set of current solution constraints. The Conflict Analyzer computes measures that characterize the magnitude of conflict itself (duration, number of orders involved) as well as measures of the current flexibility (or lack of) in the capacity and time constraints of the resource and orders involved in the conflict (level of contention for the resource involved, projected lateness of the orders involved, upstream and downstream slack in these orders' schedules, and variance in the projected lateness of all orders). The value of each these measures has specific implications with respect to the continuing validity of previous scheduling decisions, the amount of disruption to be expected by various scheduling actions, and/or the relative emphasis that should be placed on various scheduling objectives in resolving the conflict. This knowledge is encoded in the subproblem formulation heuristics used to direct schedule revision (see below). Details of the measures computed during conflict analysis may be found in [Ow et. al. 88].

4.3. Subproblem formulation heuristics

In this section, we describe the subproblem formulation heuristics used to opportunistically focus the scheduling process. We consider, in turn, the heuristics employed for generating and revising scheduling decisions.

In generative scheduling contexts, subproblem formulation decisions are driven first and foremost by the expectations regarding resource contention that are provided by capacity analysis. If bottleneck resources are identified, then RSC is applied to schedule the bottleneck that is estimated to be the severe one. These decisions are seen as most critical to the overall solution. The RSC is repeatedly applied to any further bottlenecks that are identified on subsequent control cycles. When the results of capacity analysis indicate that no bottleneck resources remain, a shift in perspective to order-based scheduling is made.

Both the scope of order-based subproblems and the manner in which they are prioritized and solved depend on the state of the current partial solution. Consider the case of a single scheduled bottleneck resource. The heuristic used here aims at constructing order schedules outward from these fixed points in an manner that facilitates minimization of work-in-process time (the principal strength of OSC). To this end, order scheduling subproblems relating to the upstream portions of the orders' production plans are prioritized in reserve order of the their scheduled start times at the bottleneck resource, and CV-OSC is applied in a backward scheduling fashion through the upstream portion of a given order's production plan. Conversely, order scheduling problems relating to the downstream portions of orders' production plans are prioritized in order of their scheduled end times on the bottleneck resource, and CV-OSC is applied in a forward fashion. Order scheduling subproblems relating to the portions of production plans falling between two bottlenecks are prioritized and solved in the same manner as downstream subproblems, however in this case minimizing work-in-process time is not really a concern (i.e. there are two temporal anchors). Complete order scheduling subproblems (i.e. in situations where there are no bottlenecks) are prioritized according to closeness to due date.

With respect to overall coordination of order scheduling

before, between, and after scheduled bottlenecks, the strategy here is to move forward through time, solving all subproblems upstream of all bottlenecks first, and so. The motivation here is to recognize and respond to any conflicts that might arise in the developing schedule as early as possible. Given the fact that bottleneck scheduling proceeds with only a local view of the overall problem, incompatibilities may develop as order scheduling proceeds (note that such problems will not occur downstream of all bottlenecks since due dates can be relaxed). In such cases, the heuristics described below for schedule revision become relevant and the bottleneck schedule is revised. The underlying propagation of time bounds, coupled with the fact that important capacity related problems are addressed, acts against the occurrence of such problems, and typically the conflicts that are introduced are relatively minor in nature.

In schedule revision contexts, characteristics of the current conflict state provide the basis determining appropriate scheduling actions. Figure 4-1 contains a decision tree reflecting the set of heuristics that are currently utilized. Interpreting this tree, we see that if the size of the conflict is small, then RSH is advocated. This heuristic appeals to sensitivity analysis [Bean&Birge 85], assuming that the sequencing decisions in the current schedule are still valid in such situations, and the knowledge that this scheduling KS resolves conflicts efficiently in a way that preserves existing sequencing decisions. On the other hand, if either the conflict size, the conflict duration or both is large, then the implication is that some amount of resequencing is likely to be necessary. If it is additionally the case that capacity constraints are very tight on the resource involved in the conflict (designated as "low fragmentation" in the figure), a resource-based perspective is needed to ensure that the action taken optimizes utilization of this resource. Conversely, a highly fragmented resource schedule implies an opportunity for order-based optimization.

If a resource-based perspective is to be taken, then a number of different scheduling actions are possible. If the average projected lateness of the conflicting orders is positive, and the variance in the projected lateness of all orders is high, then there may be an opportunity to either resolve or reduce the conflict using DSW. This action is applied whenever possible, and then removed from consideration for the next control cycle. If, alternatively, the above conditions are not met, then the likelihood that DSW will yield productive results is low since either there is no need for pair-wise minimization of tardiness (conflicting orders are early) or there is little opportunity (all orders are equally early or late). In this case, either RSC or PV-OSC (with its scope limited to just the conflicting operations) are possible actions. If either the number of conflicting operations is high or there is upstream slack that can be exploited for resequencing purposes, then RSC is the most efficient and effective reactive action that can be taken. This follows from its strength in optimizing the utilization of a particular resource. However, if only one or two conflicting operations are present and there is little upstream slack, then PV-OSC may be sufficient. In this case the resequencing problem is constrained to one of simply repositioning the conflicting operation(s) within the focal point resource's schedule.

If we presume instead that an order-based perspective is appropriate (i.e., fragmentation is high), the projected average lateness of conflicting orders provides a basis for differentiating among possible actions. If there is considerable slack in meeting the due dates of orders involved in the conflict, the CV-OSC is preferable as it minimizes disruption to the existing schedule without threat of the order being tardy. If, however, order tardiness is a concern, then a more aggressive approach to order scheduling is needed. When there is a high variance in the projected lateness of jobs, an opportunity may exist to swap demands with DSW. If not, PV-OSC provides a consistent, more aggressive approach to order scheduling.

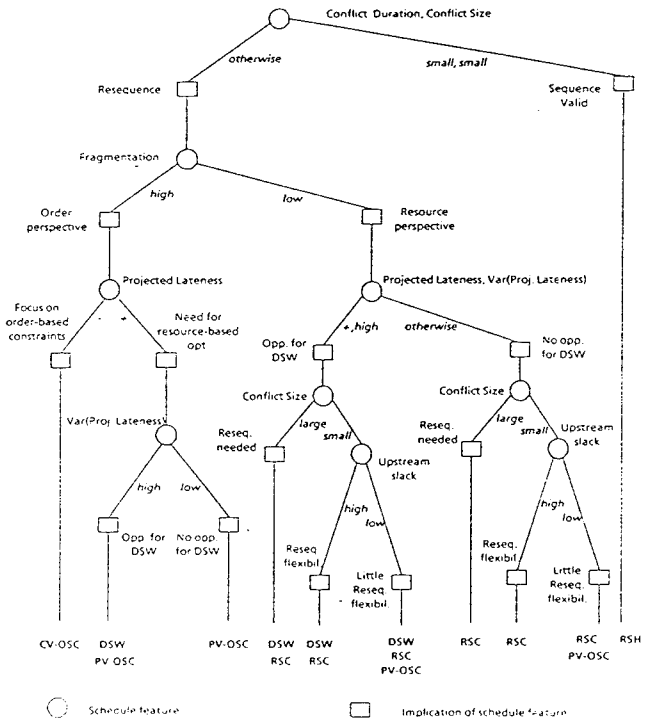


Figure 4-1: Revising Schedules

One aspect of the reactive scheduling heuristics which is not reflected in figure 4-1 is the scope of the formulated subproblem. In the case of resource-based reactions, the

scope is naturally the resource involved in the conflict. However, in the case of order-based reactions, the scope depends on the extent of resource contention further downstream. Specifically, if downstream slack is high (indicating the presence of downstream bottleneck resources), then the scope of an order-based scheduling action is limited to the portion of the order's production plan that precedes the first downstream bottleneck operation. This provides the opportunity to take full advantage of the strengths of resource-based scheduling actions for downstream bottlenecks.

5. Experimental Results

An experimental study previously reported [Ow&Smith 88] has demonstrated the utility of the "bottlenecks first" generative scheduling strategy described in the previous section in the case of a single fixed bottleneck problem. This study conducted an comparative analysis of an earlier version of OPIS (employing OSC and RSC) with both the predecessor ISIS scheduling system (a order-based scheduler), and a dispatch-based simulation approach. The context of this study was a scaled down model of an actual job shop consisting of 30 machines organized into various work areas (with 7 machines in the bottleneck area). Six product types were included in the model, each with linear production plans ranging from 4 to 6 operations. The schedules generated by each system were evaluated with respect to tardiness costs, work-in-process time, and number of machine setups.

The analysis involved solution, by each system, of 22 test problems, 20 requiring 120 orders to be scheduled and 2 requiring on 75. Test problems were defined by manipulating 4 parameters: the pattern of order releases, the number of orders released simultaneously, the product mix, and the setting of due dates. The problem set was grouped into 18 categories, representing different shop conditions and load factors ranging from 70% to 120% of the capacity of the bottleneck area.

With respect to all three performance metrics, OPIS outperformed the other two systems across all experiments. As expected, ISIS performed well with respect to minimizing WIP time (given its order-based perspective), but its performance with respect to tardiness costs suffered because of its inability to effectively manage resource contention. ISIS schedules contained close to twice the number of setups as did the OPIS schedules. Relative to the dispatch-based approach, OPIS schedules exhibited > 25% improvement in tardiness costs in 70% of the experiments (>10% improvement over all experiments) and > 50% improvement in average WIP time over all experiments. A complete account of this study may be found in [Ow&Smith 88].

More recently, a preliminary experimental analysis was performed to assess the performance of the reactive decision-making model described above (hereafter referred to simply as the tree model). This study was carried out in the context of a specific computer board assembly and test line. At the level of detail modeled, the line consisted of 11 "sectors", each possessing the capacity to simultaneously process a number of boards. Board process routings varied in length from 12 to 30 operations and included both planned and unplanned looping through various sectors. In the following, we summarize the experimental design and the overall results obtained.

The performance of the tree model was contrasted with that of the set of simpler reactive strategies that are defined by assuming that a particular OPIS KS is unconditionally applied as the first reaction to any conflict that arises. In cases where the designated first action did not define a complete reactive strategy (e.g. the resource scheduler may introduce additional conflicts into the schedule which must be subsequently resolved), the tree model was used as a basis for selection of subsequent actions. A "random" decision model was also defined to provide a final point of comparison. Within this model, the choice of specific actions was biased by the relative frequencies with which actions were used following the tree strategy in solving the test problems. The point of the random model was to verify that the knowledge encoded in the decision tree was in fact significant. Each alternative reactive strategy was applied to a series of 26 test problems. Each reflected the occurrence of machine breakdowns and/or quality control failures (implying order rework) in the midst of executing a pre-generated schedule. Conflict points in the current schedule were chosen so as to include circumstances that covered a large number of branches in the tree model.

The results of these experiments were analyzed with respect to the following six performance criteria, combining schedule quality and scheduling disruption objectives. With respect to schedule quality, change in total tardiness time (Tardy/min) , change in the number of tardy orders (#OrdTardy), and change in total work-in-process time (ChgWIP) were measured. With respect to schedule disruption, number of resources with changed schedules (#ResChg), number of orders with changed schedules (#OrdChg), and average time change per rescheduled operation (Chg/res) were measured. To provide a basis for comparison, the set of performance values obtained for each strategy i in test problem k,were normalized according to the following formula:

$$v_n(i,j,k) = \frac{v(i,j,k) - \overline{V}(j)}{\sigma(j)}$$

where $\overline{V}(j)$ is the grand mean value for that criterion obtained for all strategies over all experiments, and $\sigma(j)$ is

the grand standard deviation. A score for each strategy in each experiment was then computed as follows:

$$S(i,k) = \sum_{j \in Perf.Criteria} w(j)\, v_n(i,j,k)$$

where $w(j)$ is the weight associated with performance criterion j.

Strategies were evaluated with the following distribution of weights: [tardy/min = 0.6, chgWIP = 0.05, #OrdTardy = 0.1, #OrdChg = 0.1, #ResChg = 0.05, Chg/res = 0.1] which reflect the assumptions on which the tree model is based. The following results were obtained. First, the action prescribed by the tree model produced the best $S(i,k)$ in 60% of the test problems. Furthermore, the action prescribed by the tree produced a $S(i,k)$ that was either best or second best in 88% of experiments. The overall behavior of each strategy is characterized in Table 5-1.

	TREE	RSC	PV-OSC	CV-OSC	RSH	RDM
Ave.	-0.327	-0.205	-0.123	-0.021	0.170	0.178
Std. Dev.	0.447	0.397	0.753	0.799	0.799	1.089

Table 5-1: Overall behavior of alternative reactive strategies

While it is not possible to draw any general conclusions from these preliminary experimental results, there are a couple of observations that can be made:

- With respect to average overall performance, the tree model did perform better than all competing approaches. The "resource scheduler first" strategy (RSC) was the second best performer. This is in fact the most similar model to the tree model, since it nearly always incorporates decisions from the tree model to complete the schedule revision.
- The right shifter first strategy (which is a complete strategy) was found to perform as poorly as the random model. This is interesting given the fact that right shifting is essentially equivalent to not reacting and just delaying production. This result suggests the value of more sophisticated reactive decision-making.
- Finally, we found the relative overall performance of the tree model to be insensitive to 5% changes in the weights of individual criteria.

6. Conclusions

In this paper, we have presented an approach to coordinating production activities that advocates a common view of predictive and reactive scheduling as an opportunistic constraint-directed process. Accordingly, the OPIS scheduling architecture is intended to provide a framework that integrates periodic expansion/refinement of predictive schedules with incremental revision in response to conflicts that are introduced as a result of factory operation.

Our current research builds on this work and is concerned with the following issues:

- Understanding the multiplicative effects of incremental revisions to the schedule over time - Our current model for conflict analysis and reaction selection is based on the assumption that a new solution in the "neighborhood" of the old solution is desirable (or somewhat equivalently that the starting schedule is a good one). We currently have little understanding as to whether (and at what rate) the quality of the schedule can be expected to degrade over time. Our goal here is development of a methodology for recognizing when more global rescheduling is warranted.
- Reacting to opportunities - A second point regarding schedule revision is the incompleteness of our current reactive model. If factory floor decision-making is to be driven by predictive guidance (a schedule) then schedule revision must be driven by opportunities (e.g. unanticipated resource capacity) as well as conflicts. We are currently extending the reactive model in this regard.
- What level of predictive guidance is appropriate - This issue concerns the nature of the constraints imposed by the schedule (e.g. level of detail, temporal precision, etc.), and the extent to which execution-time decision-making is assumed. Knowledge of the sources and degrees of unpredictability (and regularity for that matter) in any particular manufacturing environment should dictate the level of detail of different aspects of the maintained schedule, both in terms of imposing constraints on factory floor decision making and in terms of establishing system expectations against which the need for reactive scheduling action can be gauged. This implies some amount of execution-time decision making in most manufacturing environments. We are investigating methods for representing and exploiting knowledge of the sources of unpredictability within the scheduler, and the development of control policies that interpret the schedule at execution time with knowledge of the scheduler's uncertainty assumptions. More generally, we are interested in understanding how the characteristics and constraints of a given manufacturing environment (e.g. extent of process unpredictability, demand patterns) should influence

decisions as to level of predictive guidance.

- Decentralization - Finally, we recognize that effective coordination of production schedules requires planning and reaction at different levels. Decisions made at higher levels (e.g. decisions regarding manpower requirements and shifts of operation in different areas of the factory) provide constraints on the more detailed decisions that must be made at lower levels (e.g. decisions regarding the short term schedule for a particular area in the factory). Similarly, the results of factory operation necessitate reactive actions that may involve decision-making at several different levels. Given both the complexity of this overall process and the concurrency of manufacturing activities, we view decentralization of coordination responsibility as a central component of any practical framework for production management.

References

[Bean&Birge 85] Bean, J.C., and J.R. Birge.
Match-Up Real-Time Scheduling.
Technical Report 85-22, Univ. of Michigan, Dept. of Industrial and Operations Engineering, June, 1985.

[Erman et. al. 80] Erman, L.D., F. Hayes-Roth, V.R. Lesser and D.R. Reddy.
The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty.
*Computing Surveys*12:213-253, 1980.

[Fox 83] Fox, M.S.
Constraint-Directed Search: A Case Study of Job Shop Scheduling.
PhD thesis, Computer Science Department, CMU, 1983.

[Fox&Smith 84] Fox, M.S., and S.F. Smith.
ISIS: A Knowledge-Based System for Factory Scheduling.
*Expert Systems*1(1):25-49, July, 1984.

[Graves 81] Graves, S.C.
A Review of Production Scheduling.
*Operations Research*29(4):646-675, July-August, 1981.

[LePape&Smith 87] LePape, C. and S.F. Smith.
Management of Temporal Constraints for Factory Scheduling.
Technical Report TR-CMU-RI-87-13, The Robotics Institute, CMU, June, 1987.

[Muscettola&Smith 87] Muscettola, N. and S.F. Smith.
A Probabilistic Framework for Resource-Constrained Multi-Agent Planning.
In *Proceedings IJCAI-87*, Milano, Italy, August, 1987.

[Ow 85] Ow, P.S.
Focused Scheduling in Proportionate Flowshops.
*Management Science*31(7), 1985.

[Ow et. al. 88] Ow, P.S., S.F. Smith, and A. Thiriez.
Reactive Plan Revision.
In *Proceedings AAAI-88*, St. Paul, Minn., August, 1988.

[Ow&Smith 88] Ow, P.S. and S.F. Smith.
Viewing Scheduling as an Opportunistic Problem Solving Process.
*Annals of Operations Research*12:85-108, 1988.

[Panwalker&Iskander 77] Panwalker, S.S. and W. Iskander.
A Survey of Scheduling Rules.
*Operations Research*25:45-61, 1977.

[Potvin&Smith 89] Potvin, J.Y. and S.F. Smith.
Flexible Systems for the Design of Heuristic Algorithms in Complex OR Domains.
In *Publications in Operations Research.* Volume 9: *Impacts of Recent Computer Advances on Operations Research*, pages 332-344. North-Holland, 1989.

[Smith 89] Smith, S.F.
The OPIS Framework for Modeling Manufacturing Systems.
Technical Report TR-CMU-RI-89-30, The Robotics Institute, CMU, December, 1989.

[Smith et. al. 86] Smith, S.F., M.S. Fox, and P.S. Ow,
Constructing and Maintaining Detailed Production Plans: Investigations into the Development of Knowledge-Based Factory Scheduling Systems.
*AI Magazine*7, 1986.

[Smith&Ow 85] Smith, S.F. and P.S. Ow.
The Use of Multiple Problem Decompositions in Time-Constrained Planning Tasks.
In *Proc. IJCAI-85*, Pages 1013-1015. Los Angeles, CA, August, 1985.