REPORT DOCUMENTATION PAGE

FINAL/01 AUG 91 TO 31 JUL 94

LEARNING IN AN INTENTIONAL SYSTEM

2304/GS
AFOSR-91-0341

LAWERENCE BIRNBAUM AND GREG COLLINS

NORTHWESTERN UNIVERSITY
THE INSTITUTE FOR THE LEARNING SCIENCES
1890 MAPLE AVE
EVANSTON, ILLINOIS 60201

AFOSR-TR- 95 0007

AFOSR/NM
110 DUNCAN AVE, SUTE B115
BOLLING AFB DC 20332-0001

AFOSR-91-0341

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

The goal of the project, as outlined in the original proposal, was to carry out research aimed at the construction of adaptive planning systems that can learn in response to planning failures, i.e., modelling learning to plan as a process of debugging. Previous approaches to failure-driven learning, notably Sussman (1975), have been based on a traditional models of planning. in which the planner generates a monolithic, self-contained plan to be passed on to another module for execution. Such approaches have thus focused on the process of debugging a plan that has proven to be faulty during execution. However, such traditional models of planning have increasingly come under attack, as it has come to be recognized that planners operating in the real world must be reactive, changing plans on the fly to cope with unexpected circumstances. In such a model, there may be no single, monolithic plan that governs the systems behavior during execution.

SECURITY CLASSIFICATION OF REPORT	SECURITY CLASSIFICATION OF THIS PAGE	SECURITY CLASSIFICATION OF ABSTRACT	
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR(SAME AS REPORT)

Learning in an Intentional System: Final Report

submitted

December 1994

to

The Air Force Office of Scientific Research

for

Grant no. AFOSR-91-0341-DEF

by

Lawrence Birnbaum and Gregg Collins

Northwestern University
The Institute for the Learning Sciences
1890 Maple Avenue
Evanston, Illinois 60201

(708) 491-3500
fax (708) 491-5258
{birnbaum, collins}@ils.nwu.edu

1 Introduction

This document outlines the final results of a three year project entitled "Learning in an Intentional System" carried out at The Institute for the Learning Sciences, Northwestern University, with support from the Air Force Office of Scientific Research (grant number AFOSR-91-0341-DEF). Further details may be found in the references.

1.1 Project Goal and Basic Approach

The goal of the project, as outlined in the original proposal, was to carry out research aimed at the construction of adaptive planning system that can learn in response to planning failures, i.e., modelling learning to plan as a process of debugging. Previous approaches to failure-driven learning, notably Sussman (1975), have been based on a traditional models of planning, in which the planner generates a monolithic, self-contained plan to be passed on to another module for execution. Such approaches have thus focused on the process of debugging a plan that has proven to be faulty during execution. However, such traditional models of planning have increasingly come under attack, as it has come to be recognized that planners operating in the real world must be reactive, changing plans on the fly to cope with unexpected circumstances. In such a model, there may be no single, monolithic plan that governs the systems behavior during execution.

In response to this, we have focused on the issue of debugging the *planner*, rather than debugging the plan. To use a simple analogy, we can regard the agent's behavior as the product of a complex set of interacting processes, like the output of a factory. If the products turned out by a factory are unsatisfactory, then it is the factory itself, and not simply the individual products, that should be fixed. Similarly, if a planning agent's behavior is judged unsatisfactory, it the planning system itself that must be debugged.

In order to support this debugging process, we have chosen to apply the paradigm of model-based reasoning (see, e.g., de Kleer, et al., 1977; Sussman and Stallman, 19077; Davis, 1984). In this approach, a model of a complex system is used as a basis for generating expectations for that systems behavior, pinpointing faults when these expectations are violated, and generating repairs that will correct the problem in the future. In order to apply model-based reasoning to the task of debugging a planning system, it is thus necessary to create detailed, explicit models of how that planning system functions. The construction of such models has thus been the central focus of out research effort.

1.2 Project Structure

Our work is being carried out in the context of three related research projects:

- Model-based debugging: building planning systems that improve their performance over time, e.g., *Castle.*
- Extending traditional planning models: building systems that handle uncertainty and real-time planning and execution, e.g., *Cassandra, Pareto.*
- Domain-independent design: Using models of user tasks as a basis for design processes that create artifacts to support those tasks.

The first two of these project are well underway, and have produced significant published results, as well as a number of prototype systems. The third project is in its early phases. The remainder of this report will describe the contributions of these projects in detail.

2 Adaptive Planning

Machine learning research has concentrated almost exclusively on how to learn—that is, the learning system is focused on a particular, well-defined learning problem by the researcher. The question of *what* to learn has thus been largely ignored. The Castle system decides what to learn by analyzing observed failures of its own plans, and implementing repairs. Determining what to change in response to a failure depends on determining which component of the system's cognitive architecture is responsible for that failure. For example, when a chess player fails to make a defensive move that would have avoided a costly capture, analysis of the failure should distinguish, e.g., a failure to notice that the move was possible ("Why didn't I think of that?") from a failure to realize that the move was worthwhile ("I thought of that, why didn't I do it?").

Castle implements a failure driven, model-based approach to learning from planning failures in the domain of chess. As part of its decision-making cycle, Castle posts expectations about the outcomes of its plans. When an expectation fails, the diagnosis process is invoked. The outcome of this process is an identification of the underlying fault that caused the expectation failure. This is used as an input to the repair module, which suggests an appropriate modification to the decision-making apparatus to prevent the recurrence of such a failure. Castle's decision-making architecture is primarily concerned with the detection of threats and opportunities, and the anticipation of possible opponent plans.

2.1 Model-Based Diagnosis and Explanation-Based Repair

As described in the previous section, Castle's learning process is triggered by the observed failure of an expectation. When such a failure is observed, the system must determine the underlying cause of this failure, in order to generate an appropriate repair. In effect, the failed expectation is a symptom, and Castle must diagnose the underlying disease. The knowledge necessary to support this diagnosis process is captured in the form of *justification structures,* which record the dependence of system's expectations on its assumptions—including in particular assumptions about the operation of its own components stemming from its self-model. When an expectation is violated, the system traces back through these justification structures to find the faulty assumption(s) underlying the failed expectation. The basic technology for the maintenance and analysis of such justification structures was originally developed in support of model-based reasoning for devices (see, e.g., de Kleer et al., 1977; Stallman and Sussman, 1977; Doyle, 1979; Davis, 1984; Hamscher and Davis, 1984; Smith et al., 1985; Simmons, 1988).

The primitive assumptions that underlie Castle's justification structures consist of functional specifications of the components of its planning architecture—e.g., the assumption that a threat-detection component will identify every instance of a particular class of threats. A completed diagnosis will thus involve the identification of one or more components that failed to fulfill their functional specifications.

Once Castle has identified such a failed assumption, it must then determine what can be done to correct it. This involves, first, generating an explanation of why the component's specification should have applied to this case, and, second, generalizing this explanation using EBL (see DeJong and Mooney, 1986; Mitchell et al., 1986). The result of this process is a new rule that, when added to the faulty component, prevents similar problems in the future.

2.2 Abstraction in Task Models

Castle's diagnosis process depends upon it's model of its own decision-making architecture—that is, upon its cognitive model of its task. Maximizing the benefit of the learning process requires that the system's cognitive task models be abstract enough to supply leverage across a wide variety of domains. In other words, the cognitive model must be specified in terms of components that carry out tasks that are not idiosyncratic to any particular domain, but rather play a role in planning in general.

For example, consider the task of *threat detection:* The goal of threat detection is to notice instances of threats against the system's goals early enough to be able to counterplan against them, that is, to do something to prevent them from coming to fruition. Thus the task entails, among other things, *determining "perceptual"*

features associated with instances of such threats, *monitoring* the environment for the presence of such features, and *notifying* some decision-making authority when such features are detected. At the next level of detail, determining an appropriate set of features to be monitored entails discovering or generating features that meet certain specifications: The features must *arise early* enough in the course of a threat instance to enable the system to block the threat; they must be reasonably *diagnostic* of threats of a given type, e.g., they must not present the decision-making authority with too many *false alarms;* they must be reasonably *easy to detect;* and so on.

The key thing to notice about the above description of the issues that arise in threat detection is that it is entirely general in nature (Collins, Birnbaum, Krulwich, and Freed, 1991). The concepts involved have nothing to do with detecting threats in any particular domain: Rather they pertain to the task of threat detection in general. This suggests that threat detection, and counterplanning generally, is an aspect of planning behavior that is highly dependent upon general planning and design knowledge. In particular, the ability to modify threat detection behavior in order to successfully meet the particular circumstances in which the planner finds itself depends crucially upon such knowledge.

Consider the following example: A night watchman completes his rounds in a certain period of time, say one hour. At some point during the night, he discovers that while he was elsewhere on his rounds, one of the areas that he was guarding has been successfully broken into by burglars, and valuable property has been stolen. What should he do to prevent this from happening in the future?

One obvious strategy he might follow is to speed up his rounds. The question of interest here is, what makes this obvious? The answer lies in our general knowledge about the monitoring sub-task in threat detection. Monitoring a feature means focusing the planner's perceptual mechanisms on an area in which a threat might arise and computing whether that feature is in fact present at that time. Since we cannot in general afford to monitor all features continuously, we must instead monitor them intermittently. The rate at which we monitor a given feature must be a function of, among other things, the rapidity with which the threat it predicts can be carried out: A threat that takes a long time to come to fruition need only be checked for occasionally. On the other hand, a threat that can be executed quickly must be monitored at short intervals, in order to ensure that enough time remains after its detection to prevent it from being carried out. Given this understanding of the situation, it can be seen that the speed with which the night watchman makes his rounds is what controls the rate at which threats are being monitored. It is part of our general planning knowledge that if threats are not being detected early enough, one possible repair strategy is to increase the rate at which they are being monitored, which in this case means speeding up the watchman's rounds.

2.3 CASTLE Case Study

Consider the *discovered attack*, a classic chess maneuver in which the movement of one piece opens a line of attack for another piece. Novices often fall prey to such attacks, and the key point is that this is *not* because they fail to understand the mechanism of the threat, i.e., the way in which the piece can move to make the capture. Instead, the problem appears to be one of attention: Novices simply fail to consider new threats arising from pieces other than the one just moved. In other words, the problem lies not in their ability to detect the given threat in principle, but rather in their decisions about where to look for threats in practice. In fact, without such a distinction—e.g., if the method for detecting threats entailed scanning the entire board after each move—the problem of discovered attacks, and the need to distinguish them from other sorts of attacks, would not even arise. The model of threat detection described above reflects this distinction by introducing a notion of focus rules that limit the application of threat-detection rules. The former embody knowledge of where to look; the latter, of what to look for. Our system is capable of learning both sorts of rules, depending upon the cause of the failure in a given situation.

The following example describes a scenario in which the system falls prey to a discovered attack, and thereby learns to improve the attention-focusing portion of its threat detection component. The system's opponent starts by advancing one of its pawns one square. This opens a discovered attack by the opponent's bishop on the system's rook. However, because of overly restrictive attention-focusing, the system fails to notice this threat, despite the fact that it has a threat detection rule which, if applied to the appropriate portions of the board, would have in fact detected the threat. The system chooses instead to capture the opponent's pawn with its knight, leaving the threat on the rook unaddressed. The opponent carries out the threat and captures the rook.

Let's examine the system's decision-making in some detail. Any decision to make a particular move must be based upon some assessment of how well that move compares to the available alternatives. Such a comparison, in turn, depends upon the ability to project the implications of each alternative considered. Once an alternative is selected, relevant aspects of the projection made in service of that choice become, in effect, predictions about the future course of events upon which the rationality of the system's chosen action depends. These predictions, in turn, depend upon assumptions stemming from the system's self-model, for example, the assumption that its threat detection component is capable of detecting all outstanding threats. In response to the failure of one of these predictions, the system will attempt to determine which aspect of its decision-making process was at fault, as described above.

In this particular case, the failed prediction is that capturing the opponent's pawn would be of greater value than blocking any extant threat. When the

opponent takes the system's rook, this prediction is seen to have failed. The justification structure underlying the failed prediction is roughly the following: The system believes that no such threat exists because it has not detected such a threat and it assumes that it can detect all threats. The latter assumption, in turn, is justified by the assumption that there is a threat rule capable of detecting any given threat, and the assumption that the threat rules have been evaluated using appropriate bindings. Finally, the latter is justified by the assumption that the system's focus rules have generated the appropriate bindings. This justification structure, and the assumptions it contains, are derived from the model of threat detection described above.

There are two components that might be at fault in this case: the threat rules, and the focus rules. Determining which component is at fault means exonerating all of the alternatives. In this case, the reasoning involved turns on the model's assumption that the use of attention focusing is not degrading the system's ability to detect threats. Recall that in our model, attention focusing is implemented by using focus rules to restrict the range of bindings over which threat rules will be evaluated. Thus, the assumption that the focus rules are not degrading performance amounts to the assumption that this limitation on the bindings of the threat rules will not prevent the detection of any existing threat, i.e., that any threat that can be detected by the threat detection rules *without* limiting their bindings in this way can also be detected when the limitations imposed by the focus rules are enforced.

Applied to our current example, the instantiated form of this assumption states that if the threat to the rook could have been detected at all, it could be detected with the focus rules in force. Attempting to fault this assumption means attempting to show its negation, namely that the threat to the rook could have been detected by the threat detection rules in principle, but could not be detected in practice because the threat lay outside of the range delimited by the focus rules. Since both of these are in fact true in this case, it can be concluded that the assumption in question is faulty, and that the problem therefore lies in the focus rules.

The repair for this fault is generated by retrieving the specification of the focusing component (which in effect described what a focusing rule is meant to do), and using this specification as a goal concept for EBL. The outcome of this generalization process is a new focusing rule for discovered attacks, which is added to the existing set of focus rules.

2.4 Castle Results

The Castle system can currently handle a set of examples that include a number of the classic chess tactics that novice players must learn, including:

- Interposition
- Running away

- Counterattacking
- Discovered attacks
- En passant
- Fork
- Pin
- Boxing in

We have analyzed several additional examples, but have not yet encoded the knowledge necessary to allow Castle to handle them. These are:

- Buying time
- Pawn line defense
- Sacrifice
- Midboard domination

Our experience with Castle demonstrates that abstract task models can be successfully applied in adaptive planning systems. An important lesson learned in this work is the importance of representing design rationale in task models. While Castle can represent the way in which components fit together to carry out a task, global considerations (e.g., efficiency) are not currently taken into account. In addition, although the competitive games domain has been useful in building our technological base, further progress depends crucially upon moving to a more realistic task domain. In order to address this, we have recently begun a new project to study model-based approaches to design, including in particular adaptive interfaces. This work is described in section 4, below.

3 Extending Traditional Models of Planning

Models of planning and plan execution play a pivotal role in our research program. Unfortunately, traditional models of planning have been extremely limiting in their assumptions about the planning domain, largely ignoring, among other issues, the problems of coping with uncertainty and planning in real time. In parallel with our work on Castle, we have pursued a project to extend the traditional models of planning. This project has resulted in the development and implementation of two planning systems, Cassandra and Pareto.

3.1 Contingency Planning: Cassandra

Cassandra is a planning system that is capable of building contingency plans in the face of uncertainty. Cassandra is a modified classical planner that can constructs such plans automatically. Cassandra's algorithm is based on the current generation of "classical" planners (e.g., Tweak, SNLP, UCPOP), which

combine elegant algorithms with nice formal properties: completeness, correctness, and systematicity. Unlike standard classical planners, however, Cassandra allows the representation of uncertainty in both the domain and its operators, and the generation of contingency plans to cope with uncertainty.

Our work on Cassandra has produced a number of important research contributions, in particular:

- Cassandra was the first partial-order planner to incorporate Pednault-style secondary preconditions.
- We have developed and implemented a simple and natural representation for uncertain effect in a classical planning framework.
- We have developed and implemented a representation for decisions steps and an algorithm for automatically adding such steps to contingency plans.
- We have developed and implemented a label-propagation scheme to carry out the bookkeeping necessary to keep track of plans with arbitrary numbers of contingency branches.

3.1.1 Overview

Many plans that we use in our everyday lives specify ways of coping with various problems that might arise during their execution. In other words, they incorporate *contingency plans*. The contingencies involved in a plan are often made explicit when the plan is communicated to another agent, e.g., "try taking Western Avenue, but if it's blocked use Ashland," or "crank the lawnmower once or twice, and if it still doesn't start jiggle the spark plug." So-called *classical planners* cannot construct plans of this sort, due primarily to their reliance on two *perfect knowledge* assumptions:

1. The planner will have full knowledge of the initial conditions in which the plan will be executed, e.g., whether Western Avenue will be blocked.
2. All actions have fully predictable outcomes, e.g., cranking the lawnmower will definitely either work or not work.

By effectively ruling out uncertainty, these assumptions make it impossible even to represent the notion of a contingency plan within a strictly classical framework. While these assumptions simplify the planning process, it is dangerous to make such assumption in general, since they may lead the planner to forgo options that would have been available had potential problems been anticipated in advance. For example, on the assumption that the weather will be sunny, as forecast, one may neglect to take along an umbrella; if the forecast later

turns out to be erroneous, the umbrella option will no longer be available. When the cost of recovering from failure is high, failing to prepare for possible problems in advance can be an expensive mistake. In order to avoid mistakes of this sort, an autonomous agent in a complex domain must have the ability to make and execute contingency plans.

Our work is embodied in the Cassandra program. Cassandra is a contingency planner that has a number of advantages over previous systems, including the following:

- A single mechanism handles both uncertainty due to incomplete knowledge and that due to unpredictable outcomes.
- The circumstances in which it is *possible* to perform an action are distinguished from the circumstances in which it is *necessary* to perform it.
- The plans constructed by Cassandra include specific steps to decide what contingencies are applicable.
- Information gathering steps are distinct from decision steps.
- Information gathering steps may have preconditions.
- Cassandra constructs a single plan that allows for all contingencies, rather than constructing separate plans that must later be merged.

For an example of the sort of plan that Cassandra constructs, consider a plan in which a delivery robot must pick up a package, in a situation in which the location of the package is not known at planning time. The resulting plan will involve a decision step, with some set of actions taken to achieve the knowledge preconditions of that decision step, i.e., to determine in which location the package is to be found. The decision discriminates between two possible subplans, each of which is correct given one possible package location.

Below is Cassandra's output for the package pickup example. Step 1 in Cassandra's plan determines the location of the box, while step two decides which subplan to execute based on the result of executing step 1. Cassandra encodes subplans using labels, e.g., [LOC0S: B]. Once a decision has been made, only steps that bear the label associated with that decision are executed.

```
Initial:        (AVAILABLE CAR-1)
                When [LOC0S: B]        (PACKAGE-AT LOCATION-2)
                When [LOC0S: A]        (PACKAGE-AT LOCATION-1)

Step 1 (2):     (ASK-ABOUT-PACKAGE)

Step 2 (1):     (DECIDE LOC0S)
                (and (PACKAGE-AT LOCATION-2)
                     T          ) => [LOC0S: B]
                (and (PACKAGE-AT LOCATION-1)
                     T          ) => [LOC0S: A]

Step 3 (4):     (DRIVE CAR-1 LOCATION-1)                YES:  [LOC0S: A]

Step 4 (3):     (DRIVE CAR-1 LOCATION-2)                YES:  [LOC0S: B]

Goal:   (AND (AT ?LOC) (PACKAGE-AT ?LOC))

        GOAL                                            YES:  [LOC0S: B]
          4 -> (AT LOCATION-2)
          0 -> (PACKAGE-AT LOCATION-2)                  NO :  [LOC0S: A]

        GOAL                                            YES:  [LOC0S: A]
          3 -> (AT LOCATION-1)
          0 -> (PACKAGE-AT LOCATION-1)                  NO :  [LOC0S: B]

Complete!
```

Cassandra's solution to the package problem

3.1.2 Cassandra's representation scheme and algorithm

Representation scheme: definitions

A plan **step** *S* represents an action. It may have enabling preconditions *EP*. It has at least one effect *E*. It is the instantiation of an operator *O*.

A plan step may be a **decision step** *D*. A decision step has enabling preconditions *EP*. Each enabling precondition is of the form (know-if *C*) for a condition *C*. *D* also has **decision rules** *DR*.

An **effect** *E* represents some results of an action. It is attached to a step *S*, representing that action. It may have secondary preconditions *SP*. It has at least one postcondition *C*, a condition that becomes true as the result of executing *S* when *SP* hold.

A **link** *L* represents a causal dependency in the plan, specifying how a condition *C* is established by an effect *E*, which has *C* as a postcondition.

E has secondary preconditions *SP* and is a result of step *S*. The condition *C* is one of:

- An enabling precondition of a step *ES*;
- A secondary precondition of an effect *EE* that's a result of step *ES*;
- The negation of a secondary precondition *ESP* of an effect *EE* that's a result of step *ES*, thus preserving link *EL*.

A link *L* is **unsafe** in a contingency *CT* in which it is required if there is a clobbering effect *CE* with postcondition *CC* resulting from step *CS* such that:

- Either *CC* can unify with *C*;
  Or *C* is of the form (know-if *KC*), and *CC* can unify with *KC*;
- Step *CS* can occur between steps *S* and *ES*;
- Effect CE can occur in contingency *CT*.

An **open condition** *OC* (an unachieved subgoal) is represented in Cassandra as an incomplete link, missing the information about the effect *E* that establishes it. The condition *C* is required to be established because it is one of:

- An enabling precondition of a step *ES*.
- A secondary precondition of an effect *EE* that's a result of step *ES*.
- The negation of a secondary precondition *ESP* of an effect *EE* that's a result of step *ES*, thus preserving link *EL*.

Plan **bindings** (codesignation constraints) specify the relationships between variables and constants. The following relationships are possible:

- Two variables may codesignate;
- A variable may designate a constant;
- A variable may be constrained not to designate a constant;
- Two variables may constrained not to codesignate.

An **ordering** constrains the order of two steps with respect to each other, so that step *S1* must precede step *S2* (*S1* < *S2*).

Every step, effect and open condition in a partial plan has two sets of **contingency labels** attached to it. In the interests of brevity, we also refer to the labels of a link *L*; in this case, we mean the labels of the step *ES* or effect *EE* that *L* establishes.

Each contingency label has two parts: a symbol representing the source of uncertainty, and a symbols representing a possible outcome of that source of uncertainty. **Positive contingency labels** denote the circumstances in which a plan element must or will necessarily occur; **negative contingency labels** denote the circumstances in which a plan element cannot or must not occur.

Contingency labels must be propagated through the plan. In general, positive contingency labels are propagated from goals to the effects that establish them, while negative contingency labels are propagated from step to their results. The details are as follows:

- A step *S* inherits the positive labels of the effects that result from it;
- A step *S* inherits the negative labels of the effects that establish its enabling preconditions;
- An effect *E* inherits the positive labels of the steps whose enabling preconditions it establishes;
- An effect *E* inherits the positive labels of the effects whose secondary preconditions it establishes;
- An effect *E* inherits the negative labels of the step from which it results;
- An effect *E* inherits the negative labels of the effects that establish its secondary preconditions;
- An open condition *OC* inherits the positive labels of the step or effect that it is required to establish.

Algorithm

The planning process starts by constructing a partial plan consisting of two steps:

- An initial step *IS* with no preconditions and with the initial conditions as its effects;
- A goal step *GS* with no effects and with the goal conditions as its enabling preconditions.

This plan is added to the (initially empty) list of partial plans *PQ*. Planning then proceeds as follows:

Plan(P, Q)

1 Choose a partial plan *P* from *PQ*;
2 If *P* is complete, then finish;
3 If there is an unsafe link *UL*,
Do *resolve(P, UL)* and add the resulting plans to *PQ*;
Return to step 1;
4 If there is an open condition *OC*,
Do *establish(P, OC)* and add the resulting plans to *PQ*;
Return to step 1.

It now remains to describe how threats to unsafe links are resolved and how open conditions are established.

Resolving threats to unsafe links

We denote by *CB* the extra bindings (if any) that are required in order for *C* and *CC* to unify, and by *CCT* the contingencies (if any) in which the link L is unsafe. Threats are resolves as follows

Resolve(P, L)

1 Initialize a list *PPL*;
2 If *CB* is non-empty
Make each possible modification to the bindings of *P* that ensures that *C* and *CC* cannot unify
Add each resulting partial plan to *PPL*;
3 If step *CS* can precede step *S*
Add an ordering to ensure that *CS* precedes *S*
Add the resulting partial plan to *PPL*;

4 If step *ES* can precede step *CS*
Add an ordering to ensure that *ES* precedes *CS*
Add the resulting partial plan to *PPL*;
5 Prevent effect *CE* occurring in each contingency *CT* in *CCT* by either
5a Add the negation of its secondary preconditions *CSP* as an open condition with positive contingency label *CT*
5b Add *CT* to the negative contingency labels of step *CS*
5c Add *CT* to the negative contingency labels of effect *EE* or step *ES*
If appropriate modify the relevant decision rule as discussed in section 4.2.5.
Add orderings to ensure that step *CS* occurs between steps *S* and *ES*
Propagate labels as appropriate.
Add each resulting partial plan to *PPL*;
6 Return *PPL*.

The methods shown in steps 2, 3, and 4 are standard methods found in SNLP and UCPOP; they are often termed **separation**, **demotion**, and **promotion** respectively. The method shown in step 5a is a modification of a standard method found in UCPOP and other planners that use secondary preconditions. Essentially, the idea to prevent an effect from occurring by ensuring that the context in which it occurs cannot hold.

We say that the methods in step 5 **disable** the threat; the methods in steps 5b and 5c are specific to a contingency planner. The methods in steps 5a and 5b ensure that the threatening effect does not occur in a given contingency; the method in step 5c notes that the established step or effect cannot occur in a given contingency. If any of these techniques result in inconsistent labeling of any plan element (so that, for example, it cannot occur in every contingency in which it is required) the resulting partial plan is abandoned, as it represents a dead end in the search space.

Establishing open conditions

The procedure is usedas follows:

Establish(P, OC)

1 Initialize lists *PPL* and *PPUL*;
2 If the open condition is not of type *unknown*
For each effect *E* resulting from a step *S* in *P*
If *E* can occur in every contingency in which *OC* must be established
and if *E* can precede *ES* or *EE*
and if there is a postcondition *C'* of *E* that can unify with condition *C*
Complete the link *OC* by using *E* as the establishing effect
Add the resulting partial plan to *PPL*;
3 If the open condition is not of type *unknown*
For each operator O with an effect E with a postcondition C' that can unify with C
Instantiate a new step S
Complete the link *OC* by using *E* as the establishing effect
Add the enabling preconditions EP of S as open conditions
Add the resulting partial plan to *PPL*;
4 For each plan in *PPL*
Add an ordering to ensure that *S* precedes *ES*
Add the bindings necessary to ensure that *C'* unifies with *C*
Add the secondary preconditions *SP* of *E* as open conditions
Propagate labels as appropriate;
5 If the open condition is of type *unknown* with source of uncertainty *U* and outcome *O*
Add a new decision step *S* for uncertainty *U*
If *U* is a new source of uncertainty in the plan, add new top-level goals as open conditions with the appropriate labels
Add the resulting partial plan to *PPUL*;
6 If the open condition is of type *unknown* with source of uncertainty *U* and outcome *O*
Find an existing decision step *S* for uncertainty *U*
Add the resulting partial plan to *PPUL*;
7 For each plan in *PPUL*
Modify the decision rule for *O* to include *C* as an antecedent
Add (know-if *C*) as an open condition required to establish *S*
Add orderings to ensure that *S* precedes *ES*
Propagate labels as appropriate
8 Return *PPL* and *PPUL*.

Steps 2 and 3, taken with step 4, show the methods of **adding** a new step and **reusing** an existing step; they are essentially the methods used in UCPOP extended to reflect the need to check and propagate contingency labels. Steps 5 and 6, together with step 7, show methods of adding a new **decision** or reusing an existing decision that are specific to Cassandra.

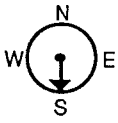
3.2 Opportunistic Planning: Pareto

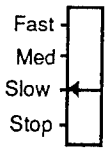
The Pareto project is aimed at the construction of planning systems that can take advantage of unforeseen opportunities discovered during plan execution. The Pareto system combines the features of deliberative planning and reactive execution—that is, it is able to react quickly to take advantage of opportunities that are presented in real time, but it is also able to reason in depth about the utility of pursuing a particular opportunity when this is warranted.

Pareto operates a delivery truck in a simulated environment created using Truckworld (Firby and Hanks). Its interface appears as follows:

TIRES

TIRE-13





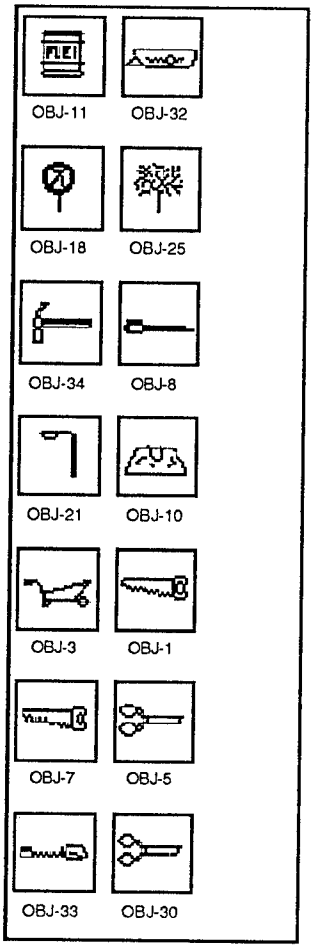
Time

190

Status

HAPPY

TOOL-DEPOT-2



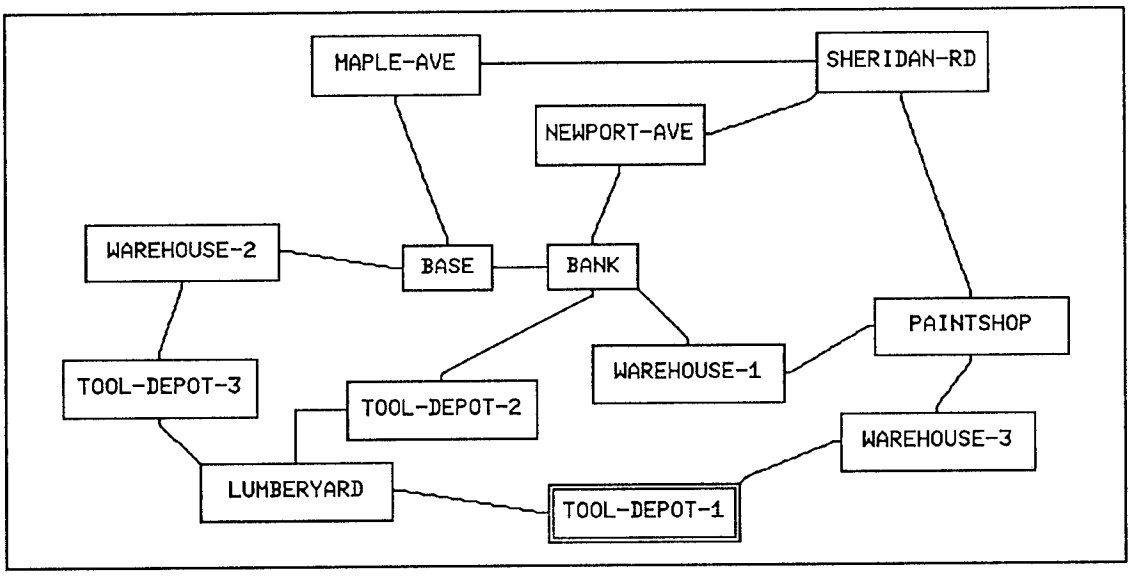
BAY1

BAY2

ARM1

ARM2

Pareto can carry objects in its bays, and can perceive objects in its immediate environment (in this case, Tool Depot 2). Pareto's task is to deliver required items to construction sites. A map of Pareto's world is as follows:



3.2.1 Overview

We can make the following observations about the problem of planning in time-pressured situations:

- Most of the effort in planning lies in the analysis of potential interaction between subplans.
- In well-structured environments, common plans can generally be assumed to be effectively independent.
- Detailed analysis to spot exception is costly; in time-critical situations, it can cause opportunities to be missed.

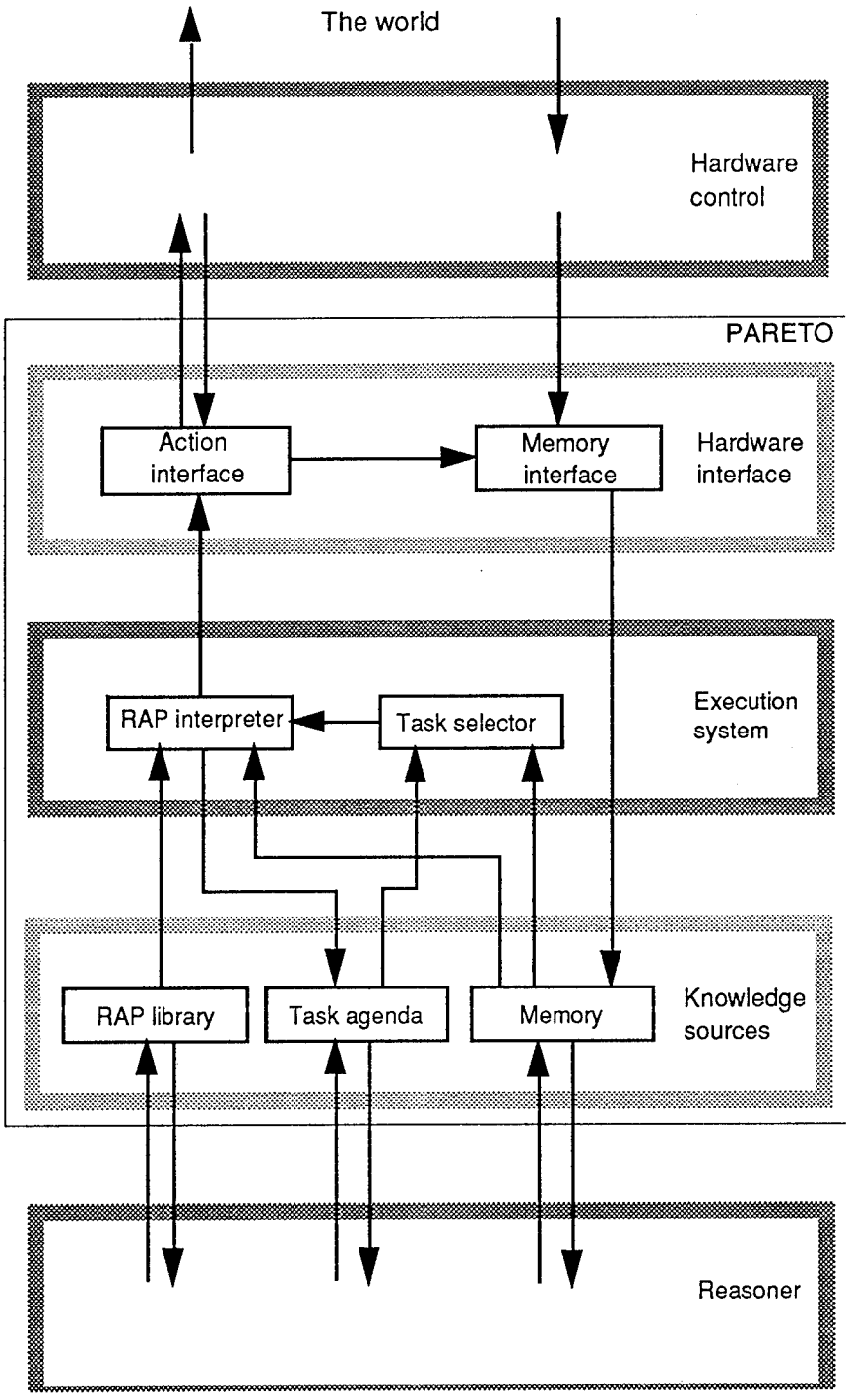
Traditional planning systems assume interactions between subplans, then reason exhaustively to either demonstrate that they will not arise, or to produce a modification of the plan that will avoid them. Pareto reverses this by assuming that subplans will *not* interact. Pareto performs a heuristic analysis process to spot situations in which problematic interactions are likely. In such situations, Pareto performs a more detailed analysis similar to traditional planners; otherwise, it is assumed that harmful interactions will not arise.

3.2.2 PARETO's Architecture

In order to perform an action an agent must specify the details of its execution. For example, in order to grasp an object the agent must know how wide its hand

must be opened and where the hand should be positioned so that closing the hand will make it grip the required object. Unfortunately, in an unpredictable world it is impossible to determine at the time that a plan is constructed exactly what the situation will be when the actions in it come to be executed. to return to a previous example, suppose you have a goal to make a bowl of cereal. Part of this plan will include actions to take a cereal bowl out of the cupboard, which involves grasping the bowl. At the time you decide to make a bowl of cereal you cannot possibly tell exactly where the bowl will be in the cupboard and exactly where you will be standing when you try to grasp the bowl. This information about the current state of the world can only be acquired as the plan is executed.

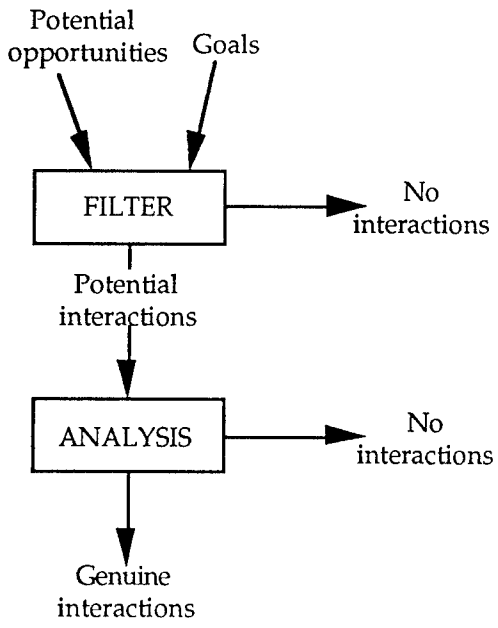
An agent in an unpredictable world must thus reconcile the impossibility of constructing plans that specify every detail of the actions to be performed with the need to specify the details of actions so that they may be actually executed. It is the job of the plan execution system to perform this reconciliation by turning incompletely specified plans into specific instructions that can be used to control the hardware that actually performs actions in the world. As described by Firby, the RAPs plan execution system is intended to serve as the middle level of the three level control architecture shown below.



Pareto's Architecture

3.2.3 Reference Features

At the heart of Pareto's ability to quickly spot potentially problematic plan interactions is its use of reference features. By associating these features with common elements in its planning domain, Pareto can use the reference features applicable in a given situation as indices to known plan interactions. This process, in turn, can be used as the basis of a filter that identifies those situations in which more analysis should be performed:



Reference features can be thought of as compiled knowledge about standard interactions. Reference features label important functional properties of object and other situational elements, e.g., sharp, fragile, heavy. These features summarize typical categories of interactions that particular types of objects participate in, e.g., cutting, breaking, etc. Reference features thus compile the results of previous observation and analysis; as such, they provide a basis for inexpensively analyzing situations for potential interactions. The table below shows some of the generally applicable reference features that we have identified.

Surface interactions:	smooth slippery sticky rough gritty greasy dull shiny glossy	Load-bearing interactions:	flimsy sturdy tough heavy light soft hard solid	Containment interactions:	bulky dense tiny large small big sheetlike stringlike hollow container
Cutting interactions:	sharp blunt soft hard	Breaking:	brittle fragile robust delicate	Deformation:	rigid flexible elastic bouncy
Liquid interactions:	imperm-eable perm-eable absorbent viscous	Temperature:	hot cold frozen	Vision and light:	transparent translucent solid
Use as tools:	round (cylindrical) prying pointed graspable	Tasks and plans:	urgent fiddly robust time-consuming	Stability:	stable unstable precarious balanced lopsided
Motion:	flat barrier	Miscellaneous:	valuable worthless		

Some reference features

4 Current and Future Work: Model-Based Design

The most important product of our research effort has been the development of a methodology for producing explicit models of complex planning tasks. Our purpose in developing this methodology has been the support of adaptive planning systems. However, such task models can also play a critical role in design support systems. We have thus begun to investigate the construction of model-based design systems.

4.1 ASK 'Bert

The most obvious way in which our work on adaptive planning transfers to design is in the creation of adaptive design systems, that model the design process as a series of tasks, and support failure-driven learning techniques that can support the adaptation of designs in response to observed failures of artifacts to meet their design goals. A step towards such a system is one that supports a human designer in the process of finding repairs for faulty designs. The ASK

Bert system, which is being built in Mosaic, and will shortly be available on the World Wide Web, uses vocabulary derived from an abstract model of the process of engineering design to index cases of known repairs for particular design flaws. ASK Bert thus combines our work on task modeling with case-based reasoning to produce a tool that can support design debugging across a broad spectrum of engineering domains. Currently, the application contains over one hundred engineering design cases, indexed under such problem categories as *excess capacity, insufficient capacity, complexity, by-products, failure,* and *inflexibility.*

4.2 Model-based Interface Design and Adaptive Interfaces

Despite the critical importance of human interfaces in the design of effective performance support systems, current interfaces are frequently arcane and difficult to use. At the same time, because the design of these interfaces depends upon ad hoc analyses of the tasks at hand, each interface must be hand-crafted to meet idiosyncratic requirements, a labor-intensive process that is all too often carried out by programmers who lack the necessary expertise. One promising approach to improving the interface design process is the construction of design aids that incorporate explicit, standardized models of the user tasks supported by the system.

To illustrate the rationale behind such an approach, consider the Sickle Cell Counselor (SCC), a system we have developed at ILS to teach visitors to the Museum of Science and Industry in Chicago about Sickle Cell anemia by allowing them to play the role of a genetics counselor to couples who may be at some risk of passing the disease on if they have children. A user of SCC is primarily engaged in an *investigation task,* and the system's interface has been carefully crafted to support the user in carrying out that task, e.g., by posing queries, running tests, and weighing evidence. The key observation underlying our approach is that many features of SCC's interface design would be equally appropriate in supporting other investigative tasks as well—*if* the commonalities could be identified and represented explicitly. We have begun to investigate the construction of interface design tools that, by incorporating explicit, standardized task models, would permit interface designers to access and incorporate such common task-relevant features into interfaces under construction simply by specifying the functionality of the underlying system.

It is clear that, to the extent that any software system is capable of carrying out essential interface functions at appropriate times, its design reflects some model of the user's task. In current interface design practice, however, this model is nowhere explicit in the operation of the interface itself, and is rarely even made explicit in the design process. Typically, a good interface design is the result of a skilled human designer relying on his or her intuitive knowledge of the task in order to make the interface do the right thing at the right time. Because of this, the current state of the art in interface design is such that every interface is custom designed and built, and its quality is totally dependent on the skill of the

interface designer. Interfaces are thus time-consuming and costly to construct. The result, all too often, is that insufficient resources are allocated to the interface design process, with the consequence that the overall system is difficult to understand and use.

As with any other engineering design process, improvement in the design methodology for interfaces depends, in large part, upon making explicit what has previously been implicit. To the extent that interface designs depend upon implicit models of the user's task, it is impossible to standardize the creation of such designs. This in turn makes it difficult to communicate and teach the design principles involved, or to automate aspects of the design process. The result is that interface design remains an expensive bottleneck.

Our approach aims to overcome the limitations of current technology through the use of *explicit, standardized* task models. We are developing a library of such models formulated in terms of a well-defined modeling language. These models capture common knowledge about basic cognitive tasks such as *information gathering, process monitoring, fault diagnosis, event tracking, scheduling, prioritizing.* We believe that it is possible to accurately describe most complex real world tasks as combinations of such basic cognitive tasks. In particular, we believe that a library consisting of a manageable number of models of these basic tasks will provide interface designers with a basis for modeling most real-world tasks that their systems are intended to support.

The task model libraries we are constructing will make it possible to associate standard interface objects and code templates with standard models of such tasks. In other words, our approach to interface design is one in which the interface designer's ad hoc concept of the task will be replaced by an explicit, well-defined specification, which will be used in the creation of interfaces that can support this task.

Bibliography

Agre, P., and Chapman, D. (1987). Pengi: An implementation of a theory of activity. *Proceedings of the 1987 AAAI Conference,* Seattle, WA, pp. 268-272.

Birnbaum, L. and Collins, G. (1988). The transfer of experience across planning domains through the acquisition of abstract strategies. In J. Kolodner, ed., *Proceedings of the 1988 Workshop on Case-Based Reasoning,* Morgan Kaufmann, San Mateo, CA, pp. 61-79.

Birnbaum, L. and Collins, G. (1989). Reminding and engineering design themes: A case study in indexing vocabulary. In K. Hammond, ed., *Proceedings of the 1989 Workshop on Case-Based Reasoning,* Morgan Kaufmann, San Mateo, CA, pp. 47-51.

Birnbaum, L., Collins, G., Freed, M., and Krulwich, B. (1990). Model-based diagnosis of planning failures. *Proceedings of the 1990 AAAI Conference,* Boston, MA, pp. 318-323.

Birnbaum, L., Collins, G., Brand, M., Freed, M., Krulwich, B., and Pryor, L. (1991). A model-based approach to the construction of adaptive case-based planning systems. *Proceedings of the 1991 Workshop on Case-Based Reasoning*, Washington, DC, pp. 215-224.

Birnbaum, L., Collins, G. (1993). Towards a general theory of planning and design. Technical report # 43, The Institute for the Learning Sciences, Northwestern University, August, 1993

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, vol. 2, no. 1.

Carbonell, J. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. Michalski, J. Carbonell, and T. Mitchell, eds., *Machine Learning: An Artificial Intelligence Approach, Volume II*, Morgan Kaufmann, Los Altos, CA, pp. 371-392.

Collins, G., Birnbaum, L., and Krulwich, B. (1989). An adaptive model of decision-making in planning. *Proceedings of the Eleventh IJCAI*, Detroit, MI, pp. 511-516.

Collins, G., Birnbaum, L., Krulwich, B., and Freed, M. (1991). Plan debugging in an intentional system. *Proceedings of the Twelfth IJCAI*, Sydney, Australia, pp. 353-358.

Collins, G. and L. Pryor. (1992a). "Achieving the functionality of filter conditions in a partial order planner." In *Proceedings of the Tenth National Conference on Artificial Intelligence, San Jose, CA*, AAAI.

Collins, G. and L. Pryor. (1992b). *Representation and performance in a partial order planner*. The Institute for the Learning Sciences. Technical Report #35.

Collins, G. and Pryor, L. (1994). Planning to perceive. In *Goal-Driven Learning*, A. Ram and D. Leake, eds. MIT Press.

Davis, R. (1984). Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, vol. 24, pp. 347-410.

DeJong, G., and Mooney, R. 1986. Explanation-based learning: An alternative view. *Machine Learning*, vol. 1, pp. 145-176.

deKleer, J., Doyle, J., Steele, G., and Sussman, G. 1977. AMORD: Explicit control of reasoning. *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, Rochester, NY, pp. 116-125.

deKleer, J., and Williams, B. (1987). Diagnosing multiple faults. *Artificial Intelligence*, vol. 32, pp. 97-130.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, vol. 12, pp. 231-272.

Fikes, R., and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, vol. 2, pp. 189-208.

Firby, R. (1989). Adaptive execution in complex dynamic worlds. Research report no. 672, Yale University, Dept. of Computer Science, New Haven, CT.

Freed, M. and Collins, G. (1994). Incrementally Improving Task Coordination. *Working Notes of the AAAI Spring Symposium on Goal-Directed Learning.*

Hammond, K. (1989a). *Case-Based Planning: Viewing Planning as a Memory Task.* Academic Press, San Diego.

Hammond, K. (1989b). Opportunistic memory. *Proceedings of the Eleventh IJCAI,* Detroit, MI, pp. 504-510.

Hayes-Roth, B, and Hayes-Roth, F. (1979). A cognitive model of planning. *Cognitive Science* , vol. 3, pp. 275-310.

Hayes-Roth, F. (1983). Using proofs and refutations to learn from experience. In R. Michalski, J. Carbonell, and T. Mitchell, eds., *Machine Learning: An Artificial Intelligence Approach, Vol. 1,* Tioga, Palo Alto, CA, pp. 221-240.

Kolodner, J. (1987). Capitalizing on failure through case-based inference. *Proceedings of the Ninth Cognitive Science Conference,* Seattle, WA, pp. 715-726.

Krulwich, B. (1991). Determining what to learn in a multi-component planning system. Proceedings of the Thirteenth Cognitive Science Conference, Chicago, IL, pp. 102-107.

Mitchell, T., Keller, R., and Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning,* vol. 1, pp. 47-80.

Newell, A., and Simon, H. (1963). GPS, a program that simulates human thought. In E. Feigenbaum and J. Feldman, eds., *Computers and Thought,* McGraw-Hill, New York, pp. 279-293.

Pryor, L. and Collins, G. (1991). "Information gathering as a planning task." In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society, Chicago, IL,* Lawrence Erlbaum Associates.

Pryor, L. and Collins, G. (1992). "Planning to perceive: A utilitarian approach." In *AAAI 1992 Spring Symposium on Control of Selective Perception,* .

Pryor, L. and G. Collins. (1993). *Cassandra: Planning with contingencies.* The Institute for the Learning Sciences, Northwestern University. Technical Report #41.

Collins, G. and Pryor, L. (1993). On the misuse of filter conditions: A critical analysis. *Proceedings of the Second European Workshop on Planning,*

Sacerdoti, E. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence,* vol. 5, pp. 115-132.

Sacerdoti, E. (1977). *A Structure for Plans and Behavior.* American Elsevier, New York.

Schank, R. (1982). *Dynamic Memory: A Theory of Reminding and Learning in Computers and People.* Cambridge University Press, Cambridge, England.

Schank, R., Collins, G., and Hunter, L. (1986). Transcending inductive category formation in learning. *The Behavioral and Brain Sciences,* vol. 9, pp. 639-686.

Simmons, R. (1988). A theory of debugging plans and interpretations. *Proceedings of the 1988 AAAI Conference,* St. Paul, MN, pp. 94-99.

Smith, R., Winston, H., Mitchell, T. and Buchanan, B. (1985) Representation and use of explicit justifications for knowledge base refinement. *Proceedings of the Ninth IJCAI,* Los Angeles, CA, pp. 673-680.

Stallman, R., and Sussman, G. (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence,* vol. 9, pp. 135-196.

Sussman, G. (1975). *A Computer Model of Skill Acquisition.* American Elsevier, New York.

Wilkins, D. (1984). *Domain independent planning: Representation and plan generation.* Artificial Intelligence, vol. 22, pp. 269-302.