(4)

RADC-TR-88-11, Vol VIII (of eight) Part A
Interim Technical Report
June 1988

NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT 1986 Time Oriented Problem Solving

Syracuse University

James F. Allen

This effort was funded partially by the Laboratory Director's fund.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss AFB, NY 13441-5700

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-11, Vol VIII (of eight), Part A
6a. NAME OF PERFORMING ORGANIZATION Northeast Artificial Intelligence Consortium (NAIC) — 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)
6c. ADDRESS (City, State, and ZIP Code) 409 Link Hall Syracuse University Syracuse NY 13244-1240	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center — 8b. OFFICE SYMBOL (If applicable) COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0008
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO: 62702F (over) — PROJECT NO: 5581 — TASK NO: 27 — WORK UNIT ACCESSION NO: 13
11. TITLE (Include Security Classification) NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT 1986 Time Oriented Problem Solving	
12. PERSONAL AUTHOR(S) James F. Allen	
13a. TYPE OF REPORT Interim — 13b. TIME COVERED FROM Jan 86 TO Dec 86	14. DATE OF REPORT (Year, Month, Day) June 1988 — 15. PAGE COUNT 134
16. SUPPLEMENTARY NOTATION This effort was performed as a subcontract by the University of Rochester to Syracuse University, Office of Sponsored Programs. (See reverse)	

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Artificial Intelligence, Temporal Reasoning, Planning, Plan Recognition

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose is to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress that has been made in the second year of the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photo interpretation, time oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system. The specific topic for this volume is a model theory and axiomatization of a logic for reasoning about planning in domains of concurrent actions.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Northrup Fowler III	22b. TELEPHONE (Include Area Code) (315) 330-7797 — 22c. OFFICE SYMBOL RADC (COES)

DD Form 1473, JUN 86 — Previous editions are obsolete.

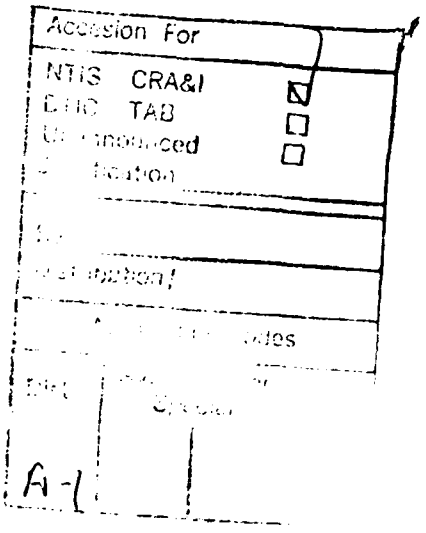
SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Block 10 (Cont'd)

Program Element Number	Project Number	Task Number	Work Unit Number
62702F	4594	18	E2
61101F	LDFP	15	C4
61102F	2304	J5	01
33126F	2155	02	10

Block 16 (Cont'd)

This effort was funded partially by the Laboratory Directors' Fund.



Task A: Time Oriented Problem Solving

James F. Allen
Computer Science Department
University of Rochester
Rochester, NY 14627

Table of Contents

A.1 Introduction

In the past year we have made significant progress, both on the development of our reasoning tools, and on basic research issues concerning planning in temporal world models. In particular, we completed the HORNE reasoning system and have made it available to other research labs and universities. This year we have distributed it to approximately 18 different sites and, combined with earlier versions distributed in previous years, have now distributed it to over 50 sites in North America. On the research side, we have finalized a model theory and axiomatization of a logic for reasoning about planning in domains where the planning agent may perform concurrent actions and may have to interact with events initiated by other agents and external forces. A description of why the standard state-based framework is inadequate in such domains and an outline of our approach to planning using our formalism is presented in [Pelavin & Allen, 1986]. A paper recently submitted to AAAI-87 goes into more detail describing the model theory, and Rich Pelavin's dissertation, to be completed this spring, provides a detailed presentation of the logic and issues related to planning with concurrent actions and external events.

The most recent work has involved the development of a simple planning algorithm that is based on our logic. We have rigorously proven that the algorithm corresponds to a valid proof in our logic. The algorithm differs from the standard state-based approach in that we do not use the STRIPS assumption to capture what actions affect and do not affect. This allows us to get around the many limitations that must be imposed when using the STRIPS assumption such as requiring a complete description of each actions effects and not allowing disjunctive effects. Our planning algorithm can also treat plans with concurrent actions, which cannot be treated in a state-based planner without providng additional mechanisms. Our underlying formalism proved a basis for treating action interactions, both sequential and concurrent, in a uniform manner.

Research in discourse analysis, story understanding, and user modeling for expert systems has shown great interest in plan recognition problems. In a plan recognition problem, one is given a fragmented description of actions performed by one or more agents, and expected to infer the overall plan or scenario which explains those actions. Henry Kautz's thesis, to be completed in Spring 1987, develops the first formal description of the plan recognition process. Beginning with a reified logic of events, the thesis presents a scheme for hierarchically structuring a library of event types. A semantic basis for non-deductive inference, based on minimal models, justifies the conclusions that one may draw from a set of observed actions. An equivalent proof theory forms a preliminary basis for mechanizing the theory. Finally, the thesis describes a number of recognition algorithms which correctly implement the theory. The analysis provides a firm theoretical foundation for much of what is loosely called "frame based inference," and directly accounts for problems of ambiguity, abstraction, and complex temporal interactions, which were ignored by previous work. In addition, the theory was applied to specific recognition problems in discourse and medical diagnosis. A Common Lisp implementation of one of the recognition algorithms was completed, and tested on plan recognition problems involving a micro-world about cooking, an operating system environment, and indirect speech acts in command understanding.

Future Plans

The successor to HORNE, named RHET, is now under development. This system is essentially a rewrite of HORNE with several important extensions: 1.) the ability to deal with contexts; 2.) increased code maintainability; 3.) the ability to handle negative assertions and proofs; 4.) improved user interface; and 5.) improved lispm oriented implementation. In addition, we are planning to add a reason maintenance facility. We have identified the useful ideas involved in the RMS (reason maintenance system) and we completed the design of a data structure for recording data-dependencies and an algorithm for monotonic retractions. We plan to handle non-monotonic updates of a database (addition causing retraction and retraction causing addition) by reducing it to a monotonic update by using a meta-predicate "unless" and explicitly asserting (unless P(a)) in the database.

A major focus of next year's research will be on the role of abstraction in planning formalisms. In particular, Josh Tenenberg has completed a preliminary study that focuses upon formalizing abstraction in problem solving and planning tasks. Many problems in Artificial Intelligence typically involve searching very large state-spaces, making exhaustive search intractable. As one approach to improving performance, we can map the representation that generates intractable spaces into successively simpler representations that have correspondingly simpler representations -- an *abstraction hierarchy*. In particular, we have defined a syntactic mapping, *predicate abstraction*, that allows one to map a theory encoded in a first-order predicate calculus (FOPC) axiomatization into a simpler theory having fewer predicates and axioms. In this way one might, for instance, map a theory about glasses and bottles into a simpler theory about containers. We plan to extend this work to a definition of abstraction in planning for STRIPS-type planners. This is another syntactic approach, similar, but less informal than the approach used by Sacerdoti in ABSTRIPS. We are considering performence metrics with which one can analytically demonstrate under what conditions a particular strategy (such as the strategy of using abstraction hierarchies) will result in actual performance improvements.

A.2 The HORNE & RHET systems

The first half of 1986 was spent on completing the HORNE system. This involved fixing all outstanding known bugs, correcting inconsistancies in the documentation, and generally cleaning things up for potential release to other sites. In addition, we looked at the feasibility of extending HORNE for contextual reasoning capabilities. For the most part, this served as foundation material for the decision to rewrite HORNE from scratch, which turned into the RHET project. We released HORNE and documentation at the end of August, 1986. Work began in earnest on RHET in September.

Rhet's goals were very much influenced by our collective HORNE experience. We had found ourselves spending more and more time maintaining old functionality in the HORNE system, which, due to its history was not very maintainable, or extensible along the lines we wanted to go. Our primary design goals were to:

- Support substantial additional functionality.

- Increase maintainablity via a coherent design vs. the HORNE design which had evolved through time, and via a coherent implementation vs. a collection of student projects.

- Increase (future) extensibility by improving code modularity and building in support for user provided subsystems.

- Increase (future) portability by increasing common-lisp compatibility.

- Improve performace by tailoring low-level functions to the target machine(s).

Most of the HORNE system's functionality is retained in the new implementation. In particular we retained both the backward and forward chaining reasoning modes, the Frame subsystem (i.e. types with roles, constructor functions), the use of universally quantified, potentially type restricted variables in an extended unification algorithm, full reasoning about equality between ground terms, and LISP compatibility (i.e. the system can be called from LISP, and predicates in the system can be defined to expand to lisp function calls). Any system using HORNE should be convertable to RHET with only minor changes.

To this, we plan to add a wide of new capabilities, including:

- An extended type calculus (handling set subtraction between types, e.g. ANIMAL-BIRD is the set of all animals that are not birds);

- Contextual reasoning, which includes access to the axioms and terms of a parent context, efficient creation and destruction of contexts, and making equality and type information also dependant on context;

- Allowing user-specifiable specialized reasoners, which will allow a user to inform the system that he will supply code to reason about objects of a particular type, which should allow the system to be used in specialized domains (TEMPOS, an extended version of our time relationship reasoner, will be the first example);

- A simplified programmatic interface, obtained by replacing HORNE's hashing setup functions and with automatically computed hashing as the need arises, and by using the facilities already on the lisp machines for editing axioms, etc.

By the end of 1986, we had implemented Rhet's basic functions for maintaining the knowledge base, including contexts and equality reasoning.

A.3 Representing Simultaneous Actions

We developed a model of action and time that represents concurrent actions and external events. Our starting point is Allen's interval logic [Allen 1984]. In this formalism, a global notion of time is developed that is independent from the agent's actions. Temporal intervals are introduced to refer to chunks of time in a global time line. An event is equated with the set of temporal intervals over which the change associated with the event takes place. Thus, there is a notion of what is happening

while an event is occurring. One can describe simultaneous events by starting that two events occur over intervals that overlap in time. Properties, which refer to static conditions, are treated similarly to events; they are equated with the temporal intervals over which they hold. Both relative and absolute specifications can be used to temporally relate events and properties.

Allen's logic can be used to describe what actually happens over time, but cannot be used to describe the different possibilities that the agent can bring about. It can be characterized as a linear time logic. Lacking from this logic is a construct like the result function in situation calculus that describes all the possible effects produced by executing different actions in different situations. To accommodate this deficiency, we extend Allen's logic with two modal operators. Both these extensions are introduced in chapter two after discussing Allen's logic in more detail. First, we add a modal operator that captures temporal possibility enabling us to describe the different possible futures at some specified time. This allows us to distinguish between conditions that are possibly true and conditions that are inevitably true at a specified time. This extended logic can be characterized as a branching time logic.

After extending Allen's logic with the inevitability operator, we show that this extension alone is not sufficient for our purposes. It does not distinguish whether a possibility is caused by the agent, caused by the external world, or caused by both factors. Making this distinction is necessary if we want to formalize what it means to say that a plan executed at a specified time solves the goal. For this purpose, we introduce a second modality *IFTRIED* which takes a *plan instance* and a sentence as arguments. Roughly, *plan instances* refer to both actions and plans to be executed at specified times in specified ways. Plan instances take the place of plans and actions in our theory. The *IFTRIED* modality represents statements that can be interpreted as saying "if plan instance *pi* were attempted then sentence S would be true." There are two ways to look at this modality, either as a subjunctive conditional or as a generalization of the result function from situation calculus.

A formal specification of the logic is then given. Our basic approach to the semantics can be characterized as *possible worlds semantics* which was developed by Hintikka [Hintikka 1962] and refined by Kripke [Kripke 1963]. In this framework, a set of objects called *possible worlds* is identified as part of a model. In our system, we refer to possible worlds as *world-histories* to emphasize that they correspond to worlds over time, not instantaneous snapshots. Each sentence in the language is given a truth value with respect to each world-history within a model. The truth value of sentences formed from the modal operators are given in terms of relations and functions defined on the set of world-histories. We pay particular attention to the functions that are used to interpret *IFTRIED*. This treatment derives from the semantic theories of conditionals developed by Stalnaker [Stalnaker 1985] and Lewis [Lewis 1973].

We then developed a proof theory that is sound with respect to the semantics. The axiomatization of most of the system is standard. The interval logic fragment is a first order theory which is formulated using standard first order axiomatization extended with axioms describing the properties of a small collection of predicates and function terms. The inevitability modal operator behaves like a S5 necessity operator for a fixed time argument. An axiomatization of these properties is taken from Hughes and Cresswell [Hughes & Cresswell 1968]. The properties that are unique to this operator capture the relations: conditions that hold earlier than or during time *i* are inevitable at *i*, and what is inevitable at some time is inevitable at later times. The axioms and rules capturing *IFTRIED* can be divided into three

categories: properties relating to a subjunctive conditional, the relation between *IFTRIED* and the inevitability operator, and properties describing the affect of attempting a plan instance composed out of two simpler ones.

Next, we analyzed the components of the planning problems using the logic that we have developed. These components include the specification of the goal, the specification of the planning environment, the conditions under which plan instances can be executed both alone in conjunction with other ones, and the effects produced by both simple and ocmposite plan instances. We pay particular attention to the interaction between plan instances, both sequential and concurrent, and to the *persistence* problem [McDermott 1982], which is the problem of determining how long a property remains true in a formalism that allows simultaneous events. By looking at these problems using our framework, we are able to explicate some of the problems that we mentioned earlier in conjunction with state-based systems. We illustrate how to represent some types of parallel interactions using our logic and describe how thsese interactions are used to determine the conditions under which two actions can be executed together.

Next year, we plan to develop a non-linear planning algorithm that exploits some of the properties investigated in above and prove that the algorithm is sound with respect to the semantics. Our algorithm will differ from previous planning systems in the method used to handle action interactions and the use of plan instances to maintain properties over temporal intervals. The interaction of two or more plan instances, concurrent or sequential, will be computed by only considering the interaction of two plan instances that overlap in time. The frame problem in our system will concern determining the conditions under which concurrent plan instances interfere witheach other. We will not need to use the STRIPS assumption or the persistence assumption to describe what actions do not affect, and thus this method should allow us to remove the restrictions that must be imposed when using the STRIPS assumption.

A.4 Abstraction and Planning

The focus of this year's work was concerned with the task of defining what it means for a theory or planning system to be an abstraction of another. This work was essentially divided into two parts. The first involved formalizing syntactic restrictions that allow one to construct an abstract theory from a detailed theory. The second involved defining a metric with which one can measure and compare the performance of different control strategies, thus providing a means to demonstrate under what conditions good performance will be exhibited by control strategies that exploit the presence of abstract theories.

Given a theory encoded as a first-order predicate calculus (FOPC) axiomatization, we can specify a syntactic mapping function that generates a new, abstract theory. The intuition behind the abstraction mapping is that there may be predicates that denote different objects or relationships at the primitive level that we wish not to distinguish between at the abstract level. For instance, in a primitive theory containing axioms about bottles and glasses, the primitive theory might state that all glasses always have a certain shape, and cannot be corked, while bottles have another shape, and can be corked. There will, however, be characteristics that are common to glasses and bottles - such as that both can hold liquids, and can be poured - and it is these characteristics that the abstract theory should capture. This can be achieved by defining a new predicate to take the place of both the predicate for

glass and the predicate for bottle, and keeping in the theory those statements that do not mention glasses or bottles, or are common to both glasses and bottles. The result (as described more fully in [Tenenberg 1987]) is that for every proof of theorem T' in the abstract theory, there exists a proof of theorem T in the original theory such that T' is the abstract mapping of T. In this way it can be demonstrated that the abstract theory is satisfiable if the original theory was.

In a similar vein, additional work was done on a set of syntactic restrictions for defining abstraction in planning systems. One of the few previous works along this line was the ABSTRIPS system of Sacerdoti at SRI [Sacerdoti 1974]. My research involved formalizing ABSTRIPS, as well as solving one its inherent deficits. The basic idea behind STRIPS-type planning systems [Fikes *et al.* 1972] is to consider world states as represented by sets of FOPC sentences, and actions as represented by additions and deletions to this set of sentences. In order to apply an action from a world state W, its preconditions (a set of sentences) must all hold in W. For instance, the precondition for the action of moving one block upon another is that both blocks must be free of any blocks on top, and the agent's hand must be free as well. Sacerdoti's idea with ABSTRIPS was to rank each of the preconditions according to some metric for the difficulty or importance of achieving that precondition, and then consider each action at different levels of abstraction by only looking at those preconditions with a ranking above a certain value. One searched for plans in a top-down fashion, by first searching through operators with the fewest constraints (that is, by considering only the preconditions with the highest rankings), and then using the resulting plan as a constraint on search at the next lower level of abstraction (that is, by considering preconditions with lower rankings). One of the unfortunate problems with Sacerdoti's formulation was that it was somewhat vague (as were most of the STRIPS systems) in terms of their precise syntax and semantics. In particular, there was no notion of what was required in order for a STRIPS system, or an ABSTRIPS system to remain consistent. My research then, provided these definitions. By expanding the definitions on the semantics of STRIPS recently provided by Lifschitz, I was able to demonstrate that an ABSTRIPS system can quite easily be brought into an inconsistent state, and that one must enforce syntactic restrictions on the ranking of preconditions if this is to be prevented. In addition, I provided a set of such restrictions, and demonstrated that they prevented an ABSTRIPS planning system from ever reaching an inconsistent state [Tenenberg 1987]. Work is proceeding in combining these two types of abstraction -- allowing for the mapping of predicates within a planning system.

Additional research has involved an attempt to define what is meant by the performance of a problem solving system. In particular, Leo Hartman and Josh Tenenberg focused upon a theorem proving system, and suggested a means for comparing the performance between two systems that use different search strategies to try to solve the same set of problems using the same axiomatizations. By performance is meant the amount of resource that an agent expends over its entire lifetime of use. In terms of computational complexity measures, the class of problems that theorem proving is a member of is the class of undecidable problems; by this is meant that there exists no uniform proof procedure which given some arbitrary FOPC axiomatization S and a sentence T expressable in the FOPC language of S will determine if T is a theorem of S. Unfortunately, such worst-case behavior does not provide much help to a system designer that is obliged to attempt the building of an autonomous (or semi-autonomous) agent. Our approach was to consider that the problem sample, those problems that an agent encounters over its lifetime, can be be viewed at a finer grain by dividing it into a set of problem classes such that there exist relatively quick solutions to all problems that fall into some of

these classes. In addition, the sample has certain measurable statistical properties, in that problems fall within these various classes with certain frequencies. Good performance, then, will result from exploiting the presence of high frequency problem classes -- if problems are solved by a particular strategy with a high frequency, then increasing the average amount of time that the strategy requires to solve the problems in its class will result in better overall resource utilization. This work will be presented at this year's IJCAI in Milano.

A.5 RAP: Planning and Classifying Objects

It seems that a major way that humans categorize objects is by their function, i.e. a "cup" is something one drinks from. Most work on the representation and formation of categories, however, has focused on intrinsic physical similarities which may or may not correspond to functional similarities. My work has been to explore the relationship between a model of planning and action and the category distinctions an agent finds useful, since a reasonable definition of functionality is an object's role in the performance of actions. This research program has involved work on automatic planning via the Rochester Abstract Planner (RAP) implementation, the use of goals in conceptual clustering, and the definition and use of abstraction in planning.

RAP is still under development; currently it is a forward-chaining interactive temporal planner, and does not yet employ abstraction as the name would suggest. RAP was implemented in Common Lisp on Symbolics and Explorer Lisp machines, using the (Prolog-like) HORNE Reasoning System for representing and querying about objects and actions, and the TIMELOGIC temporal interval reasoning system procedurally attached to special HORNE predicates. The basic action of the planner is this: given a goal, find a causation rule that specifies an event that has the goal as an effect. Then, if that event is basic (i.e. the agent can will the event to happen independent of context, as in turning on a motor) then pursue another goal. Otherwise, find a generation rule that specifies world properties and other events that when taken together imply the occurrence of the event (e.g. if a arm-raise action is performed while grasping a block, a lift-block action occurs). Recurse on those properties and events until done.

The definitions of actions in planners like RAP involve type restrictions on the objects involved, either explicitly in the header or implicitly as preconditions. An example of this is the action "Pickup(?x:block)", where the variable "?x" is constrained to be of type "block", which is defined somewhere else in the system in terms of physical features. This is a functional type, since its only use is to constrain the objects involved in actions like "Pickup", i.e. a block is something that can be picked up, or stacked, etc. When actions are composed, as is done while planning, new categories arise because the object constraints are conjoined. Thus object categories are not only implicit in the individual definitions of actions, but also in the ways that they are combined to satisfy context-dependent goals. This is the premise of my work on "Representing Goals for Goal-Oriented Classifications," which used the constraints gathered while planning within conceptual clustering problems (a la Michalski).

Categories also have hierarchical relationships. Not only do categories form a lattice by viewing them as sets, but it also seems natural and useful for an agent to perceive this organization. Any theory that relates the definitions of individual

categories to the definitions and structure of actions should also show how these hierarchical relationships are derived.

In short, the thesis of this work is that an agent's object classifications depend crucially on that agent's theory of action. How that theory is structured and why are then major questions behind a theory of categories and concepts. We feel that investigating how such knowledge representation structures can be built through learning will shed light on these questions.

A.6 References

Allen, J.F., "Towards a general theory of action and time," *Artificial Intelligence, 23*, no 2, 123-154, July 1984.

Fikes, R., P. Hart, and N. Nilsson, "Learning and executing generalized robot plans," *Artificial Intelligence 3*:251 - 288, 1972.

Hartman, L.B. and J. Tenenberg, "Performance in Practical Problem Solving," to appear, *Proceedings*, IJCAI-87.

Hintikka, J., *Knowledge and Belief: An Introduction to the Logic of the Two Notions.* Cornell University Press, 1962.

Hughes, G.E. and M.J. Cresswell, *An Introduction to Modal Lodic.* Methuen and Co. Ltd., 1968.

Kautz, H.A., "A formal theory of plan recognition," Ph.D. thesis, Computer Science Dept., Univ. Rochester, to appear, summer 1987.

Kripke, S., "Semantical analysis of modal logic," *Zeitschrift fur Mathematische Logik und Grundlagen der Mathemtik, 9*, 67-96, 1963.

Lifschitz, V., "On the semantics of STRIPS," *Proceedings of the Workshop on Planning and Reasoning about Action*, Timberline OR, 1986.

McDermott, D., "A temporal logic for reasoning about processes and plans," *Cognitive Science, 6*, no. 2, 101-155, April-June 1982.

Pelavin, R. and J.F. Allen, "A formal logic of plans in temporally rich domains," *IEEE Special Issue on Knowledge Representation, 74*, 10, October 1986.

Sacerdoti, E., "Planning in a hierarchy of abstraction spaces," *Artificial Intelligence 5*:115 - 135, 1974.

Stalnaker, R.C., "Chapter 3, Possible Worlds," *Inquiry*, MIT Press, Cambridge, MA, 1985.

Tenenberg, J., "Preserving consistency across abstraction mappings", TR 204, Computer Science Dept., Univ. Rochester, 1987.

Tenenberg, J. "Maintaining Logical Consistency in ABSTRIPS: A Formal Approach," TR 205, Computer Science Dept., Univ. Rochester, 1987.

Plans, Goals, and Natural Language*

James F. Allen
Computer Science Department
University of Rochester
Rochester, NY 14627

Diane J. Litman
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

1. Introduction

One of the more promising computational approaches to representing context in natural language systems has been based on work in general problem solving. In this approach, plans can be used both to represent the domain of discourse as well as the communication process itself. Using a uniform framework for both purposes allows a new set of techniques that allow natural language systems to handle sentence fragments, uses of indirect speech, helpful responses, and the tracking of the topic of conversations both with and without interruptions.

Examination of even simple dialogues illustrates the utility of extra-linguistic knowledge such as plans and goals. For example, imagine the demands that would be placed on a computer system capable of taking the role of the clerk in the following dialogue:

1) Passenger: The eight fifty to Milan?

2) Clerk: Eight fifty to Milan. Gate 7.

3) Passenger: Could you tell me where that is?

4) Clerk: Down there to the left. Second one on the left. No need to hurry though. The train is running late.

In order to process fragmental or incomplete utterances such as utterance (1), the system needs knowledge regarding some context of the utterance. Although many types of elliptical utterances can be understood using only the linguistic context provided by the previous dialogue, in the above example no dialogue precedes the problematic utterance. Thus, to find the missing phrases, the system will need to use extra-linguistic knowledge about the domain and likely goals of the speaker. For example, if the train clerk knows that persons seeking information typically are boarding a train, meeting a train, or looking for a room in the station, utterance (1) can be understood by recognizing that the speaker wants to board a train and that to do this the speaker needs to know what gate to go to. Plan analysis is also useful for understanding non-elliptical utterances. Since the system not only knows what was said but also why, recognition of how an utterance connects with a speaker's underlying goals provides a deeper level of understanding.

Knowledge of a speaker's domain plans and goals is also useful for understanding indirect speech and providing more information than requested. Consider indirect speech. Although utterance (3) is literally a yes-no question, the clerk responded as if the passenger had told the clerk to tell the passenger the location of gate 7. The clerk inferred that this was the intent behind the passenger's utterance, since the literal interpretation corresponded to achieving what was likely an already satisfied passenger goal (i.e., knowing if the clerk knew the location of gate 7). Furthermore, note that the clerk's reply included information irrelevant to the location of the gate. Given the context of the board plan recognized from the first utterance, including such information led to a more helpful response.

Finally, knowledge about communication goals, i.e., knowing when and how an utterance relates to previous utterances, is needed. For example, the system should be able to recognize that utterance (3) introduces a goal to clarify the clerk's previous response, and that this goal temporarily interrupts the previous goal of boarding the train to Milan.

In the next section we will present a simplified representation of actions and plans (e.g., [Fikes and Nilsson, 1971; Sacerdoti, 1977]) that supports a model of reasoning typical of general-purpose problem solvers. We will then show how this framework can provide a model of the topic of simple stories and task-oriented dialogues. Following that, we will show how to introduce speech acts into the framework and use them to model the communication process. Finally, we will outline our most recent version of this framework and provide an account of interrupting subdialogues during the conversation process itself.

2. Plans and Goals

In order to be concrete, we will describe a simple representation for actions, plans, and goals to use throughout this paper. This representation is clearly inadequate for any realistic world, but will suffice to make the points in this paper. We assume that the world at any particular time is described by a set of propositions in the first order predicate calculus. In addition, we have a set of action-types defined by conditions that fall into the following three classes:

Preconditions: A set of logical formulas that must be true before the action can successfully be executed;

*This work was supported in part by the National Science Foundation under Grant IST-8504726, and by the Office of Naval Research under Grant N00014-80-C-197. An extended version of this paper will appear in a Special Issue on Natural Language Processing of the *Proceedings of the IEEE*.

Effects: A set of formulas that will be true after the action has been successfully executed; and

Body: A set of actions that describe the decomposition of the action into subactions. Each subaction can itself either be executed or decomposed into subactions.

Intuitively, if the body of (an instantiation of) an action-type is executed in a situation where the preconditions hold, then the action is said to have been executed and the effects will hold. A plan will be represented as a tree of nodes representing action instances, annotated with the relevant preconditions and effects. The tree represents both a hierarchy of actions (i.e., subactions are below their parent action) as well as temporal ordering indicated by reading from left to right across one level of the tree.

Consider a set of propositions, functions and actions that could be used in modeling a train station. We must have predicates such as

HAS(actor, object) -- the actor possesses the object;

ON-BOARD(actor,train) -- the actor is on the train:

AT(actor, object) -- the actor is located next to the object;

IN(actor, city) -- the actor is located in the specified city;

BUY(actor, recipient, object)
Preconditions: HAS(actor, Price(object))
Body: GOTO(actor, recipient)
GIVE(actor, recipient, Price(object))
GIVE(recipient, actor, object)
Effect: HAS(actor, object)

TAKE-TRIP(actor, train, destination)
Preconditions: DESTINATION(train, destination)
Body: BUY(actor, CLERK, Ticket-for(train))
GOTO(actor, train)
GET-ON(actor, train)
Effect: IN(actor, destination)

GOTO(actor, loc)
Preconditions: nil
Effect: AT(actor, loc)

GET-ON(actor, train)
Preconditions: AT(actor, train)
HAS(actor, Ticket-for(train))
Effect: ON-BOARD(actor, train)

GIVE(actor, recipient, object)
Preconditions: HAS(actor, object)
Effect: HAS(recipient, object)

Figure 1: Some Sample Action-Types

and functions such as:

Price(ticket) -- the price of a ticket;

Loc(object) -- the location of an object;

Ticket-for(train) -- the ticket for a train (making the simplification that there is only one ticket for each train).

We must also have a set of action-types, some of which are shown in Figure 1. For example, BUY represents the action of an actor buying something (such as a ticket), while TAKE-TRIP represents the action of taking a train trip, by buying a ticket, going to the appropriate train, and boarding. GOTO represents the action of going to the location of an object. Here we assume that this can always be done (there are no preconditions) and that the action is directly executable (there is no body since the action is not decomposable into subactions). Finally, we have the action of getting on the train and the action of one actor giving something to another actor.

Given propositions representing an initial state, a goal state, and a library of possible actions, a planning algorithm can then be used to construct a plan (sequence of instantiated actions) that achieves the goal state, e.g., IN(A, MILAN). Our planning model needs to utilize two modes of reasoning in order to construct a fully executable plan. The first method, action decomposition, decomposes each action into its subactions until a level of primitive (i.e., non-decomposable) actions is reached. For example, at the most abstract level of detail, a plan to be in Milan is to take a trip by train (call it TR1) to Milan. At the next level of detail, we can decompose the TAKE-TRIP action into its subactions, annotating each subaction with the appropriate preconditions and effects. Finally, if we decompose the BUY action into its subactions, we would have the complete plan decomposition (annotated with preconditions and effects) shown in Figure 2.

The other mode of reasoning needed to construct a fully executable plan, backwards chaining, is used to ensure that the preconditions of each action in a plan are satisfied. If a precondition is not initially true, and is not achieved by an action already in the plan, then we would need to find an action that has this precondition as its effect and introduce it into the plan. For example, if when generating the plan in Figure 2 the agent *did not* have enough money to buy a ticket (the precondition of BUY), an action such as going to the bank would need to be inserted. This action itself might subsequently be decomposed into subactions, creating another action tree. In a more general model of planning we would also need to allow alternate possible decompositions for actions, as well as a more general notion of bodies. Specifically, we could allow subgoals in the body, which would cause the problem solver to be invoked to achieve them much in the same way that unsatisfied preconditions are dealt with. But we will not need to consider these techniques to make the points that follow.

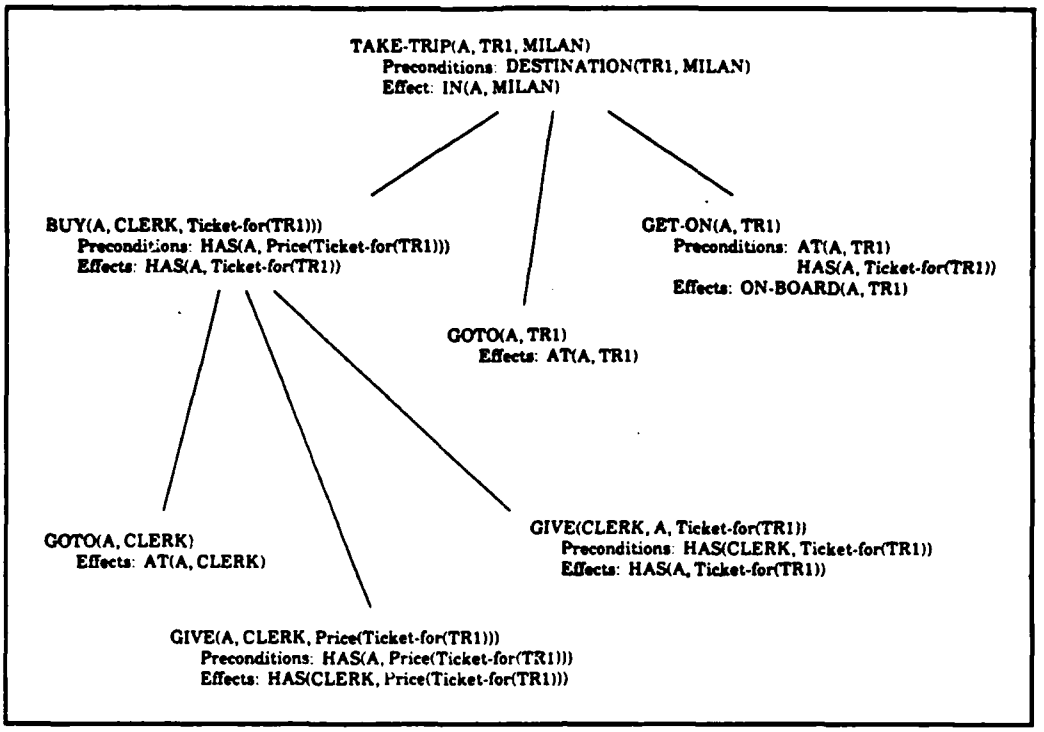


Figure 2: A Plan to Take A Trip to Milan

3. The Use of Plans to Model Topic Structure

If an agent constructs and executes plans, a crucial part of understanding another agent's actions should be to recognize the plans that motivate the actions. Plan recognition can be viewed as the inverse of the process of plan generation just described. Rather than start with a goal and plan a sequence of actions to achieve the goal, we observe an executed action and use our knowledge of other actions to construct a motivating plan and goal. We can then use our knowledge of that plan to model the topic structure of simple stories involving activities, as well as the topic structure in dialogues discussing activities. (The former application is similar to the use of scripts and plans (e.g., [Schank and Abelson, 1977; Wilensky, 1983]) in story understanding.) In particular, we will see how such knowledge can capture our intuition of coherence as well as help analyze such linguistic surface phenomena as referring expressions.

We can best see how knowledge of plans and actions is useful during the interpretation process by looking at some examples. Consider interpreting the sentence "Jack bought a ticket to MILAN and rushed for the train." Disregarding context, the initial semantic interpretation of this sentence might look something like the following:

BUY(JACK1, clerk1, Ticket-for(train1)) &
DESTINATION(train1, MILAN)

for the first half of the conjunct, and for the second:

RUSH-TO(JACK1, train2).

Note that the constants clerk1, train1, and train2 have been introduced as skolem constants, constants that encode the fact that the object referred to is as yet unknown (in contrast to the constants JACK1 and MILAN). Using our knowledge about actions and plans, however, we can further interpret this sentence. For example, we can capture our intuition of coherence by finding the connections between the clauses, and in addition, use our plan knowledge to determine the referents of the noun phrases represented by the skolem constants. We accomplish this by attempting to construct the plan of the actor JACK1 in this story. Starting from the fact that the action BUY(JACK1, clerk1, Ticket-for(train1)) was executed, we search for possible actions that the BUY action could be a subpart of. With the simple library of actions given in Figure 1 there is only one possibility to consider, namely that Jack performed the BUY action as part of the action TAKE-TRIP(JACK1, train1, MILAN), where the derivation of the parameters is as follows: matching

the semantic analysis of the sentence with the BUY action that is in the body of TAKE-TRIP, we find

JACK1 = actor
clerk1 = CLERK
train1 = train

Now, we consider the precondition for TAKE-TRIP, i.e., DESTINATION(train, destination), since if TAKE-TRIP was indeed executed the precondition would have had to be true. We can verify the truth of the precondition by matching it with the second part of the semantic analysis of the sentence, making the additional parameter assignment: destination = MILAN. Thus we conclude that Jack bought the ticket as part of taking a trip to Milan.

With this information in hand we now consider the analysis of the second conjunct, i.e., RUSH-TO(JACK1, train2). Here, we will try to interpret the conjunct as a continuation of the plan we believe to be in progress. If the action type RUSH-TO is defined to be a subtype of the action type GOTO, then we can match the RUSH-TO action with one of the GOTO acts that are part of TAKE-TRIP. Now, there is an implicit assumption in language that there is a temporal ordering imposed such that the BUY action preceded the RUSH-TO action. Thus, we can only match RUSH-TO with a GOTO act that follows the BUY act. This restricts the GOTO to be the second act in the decomposition of TAKE-TRIP. If there had been many such GOTO acts following the BUY act, the algorithm would initially assume it was the first. We match RUSH-TO with the action GOTO(JACK1, train1) and hence conclude that the referent of "the train" (i.e., train2) is the same as train1 introduced in the first conjunct. Thus we have identified the referents appropriately and related the two sentences by virtue of their both being subparts of a common TAKE-TRIP action. Sentences following in the story could be interpreted and integrated with the previous sentences in the same way.

The plan-based model becomes even more useful as we consider natural language dialogue systems. Assume we want to model conversations between two actors who want to cooperate with each other. Then a dialogue such as

A: I want to buy a ticket to Milan. Here's the money.

B: OK. (Hands A the ticket)

A: Where do I go?

B: Gate 7. Better hurry though – the train is about to leave.

can be explained in much the same way as the stories above. For example, A's first utterance both identifies a goal to do a BUY action and indicates execution of BUY's second subaction (recall Figure 1). B can then use knowledge of the decomposition of A's recognized BUY to perform the next and final subaction. Furthermore, just as in the previous example, B can use the library of actions to hypothesize that A is performing BUY as part of the action TAKE-TRIP(A, train1, MILAN). Knowledge of the decomposition of TAKE-TRIP can then be used to capture the coherence of A's question "Now where do I go?" This is because in task-oriented dialogues such as the above, the topic structure naturally follows the execution of the actions in the plan [Grosz, 1977]. Thus B assumes that A's question is related to the next action in TAKE-TRIP, i.e., GOTO(A, train1). Finally, B can use knowledge of TAKE-TRIP to include in the reply not only the information explicitly requested, but also useful information with respect to A's overall goals. That is, helpful behavior such as telling B that the train is leaving can be modeled within a plan framework as follows:

1) B identifies A's plan (e.g., TAKE-TRIP(A, train1, MILAN));
2) B uses this plan to understand and respond to A's explicit utterances;
3) B examines the plan for any obstacles and constructs plans to remove them.

In particular, step 3 could involve B determining that A will not be at the train unless A hurries, and B thus performing some action such as giving A a schedule or telling A to hurry, as done above. Again, note that in a context of recognized plans and actions, potentially ambiguous noun phrases can be used without problem. In other words, while there may be many trains about to leave, TAKE-TRIP constrains the train of B's utterance to mean the train A wants to take to Milan.

Note that to fully describe the model just presented, we need to introduce an ability to plan about one's own utterances. This topic will be the subject of the next section.

4. Plans about Language

We can introduce the need to reason about and perform linguistic actions by taking the role of agent A as A tries to buy a ticket to MILAN. Recall that the decomposition of the BUY action involved three steps:

1) GOTO(A, CLERK);
2) GIVE(A, CLERK, Price(Ticket-for(TR1))) where TR1 goes to MILAN;
3) GIVE(CLERK, A, Ticket-for(TR1)).

There are two major problems that arise when A attempts to execute this plan. The first is that A may not know what the price of a ticket to Milan is, and so cannot execute step (2). The second is that since this is A's plan, there is no reason to suppose that the clerk knows about the plan; the clerk will thus probably not know to execute step (3). Both of these problems can only be solved by using some means of communication. Intuitively, we may solve the first by asking the clerk how much the ticket is, and solve the second by asking the clerk to give A the ticket. In order to formalize this, we must define some actions that correspond to linguistic actions such as "inform" and "ask." Such

actions are usually called speech acts, adopting the terminology used by philosophers (e.g., [Searle, 1969]) who have studied such actions.

Consider defining an act of asking, which we shall call REQUEST. We need to define the effect of REQUEST so that REQUEST affects the goals and plans of another agent. To represent this, we will need a new predicate WANT(agent, action), which is true when an agent intends to perform an action. For the purposes of this paper, we can assume this means that the agent has a plan that contains the action. To incorporate this type of knowledge into our plans we will introduce a precondition on every action, namely that the actor intends to do the action. This condition should be trivially true for actions of one's own in one's plans, since having the action in the plan is equivalent to intending to do the action. For other agents, however, we shall have to explicitly achieve this precondition. Given these additions, a simple formulation of the act REQUEST is

REQUEST(a, b, action)
Preconditions: none
Effects: WANT(b, action)

In a richer framework we would not want to assume that the effect of a REQUEST is that the hearer wants to do the action. We would instead want the effect to be that the hearer knows that the speaker wants the hearer to want to do the act. This second formulation allows for the case where a request can be refused, i.e., it is up to the hearer to "decide" whether to adopt the action since the effect only changes the hearer's beliefs and not the hearer's goals. For a detailed analysis of these issues see [Cohen and Perrault, 1979]. For the purposes of this paper the simpler analysis is sufficient.

The other speech act that we will need for our examples is the inform act, which consists of the speaker telling the hearer the value of one of the functions (e.g., Price(ticket)). To model the effect of this act, we need to introduce a predicate (or modal operator) KNOW-REF(agent, function), which means that the agent knows the value of the function. Various semantics have been suggested for such an operator. In some formulations, KNOW-REF is a modal operator given a possible worlds semantics [Moore, 1980], where an agent knows the value of a function if that function has the same value in all possible worlds. Other theories extend the ontology in such a way that functions and function values can be distinguished and reasoned about [Haas, 1982; McCarthy, 1979]. Since we do not need to address such complications here, we will refer to KNOW-REF informally as a predicate. To integrate this predicate into the representation of plans, we will again need to add additional implicit preconditions on every act, namely that in order to execute any action with functional parameters P1, ..., Pn, the actor must know the value of each of the parameters (e.g., KNOW-REF(A, P1), and so on). We will only list this precondition in our plans when it is currently false. Given these additions, we can define the action of a sincere informing as follows:

INFORM-REF(speaker, hearer, function)
Effects: KNOW-REF(hearer, function)
Preconditions: KNOW-REF(speaker, function)

As with the definition of REQUEST, more complicated definitions are needed if one needs to reason about situations where the hearer doesn't automatically believe what was said. To handle these cases, the effect would have to be that the hearer believes the speaker wants the hearer to know what the value of the function is.

Consider agent A again trying to buy a ticket to MILAN. To execute step (2) above, A needs to achieve the implicit precondition

KNOW-REF(A, Price(Ticket-for(TR1)))

Looking for an action that can achieve this goal, we see that an INFORM-REF act would do the trick, so A plans for the action

INFORM-REF(actor, A, Price(Ticket-for(TR1)))

Checking the preconditions of this action, we see that whoever fills the actor parameter should already know what the price of the ticket is. If we have as part of our initial knowledge about the train domain that the clerk knows such prices, *i.e.*,

KNOW-REF(CLERK, Price(Ticket-for(TR1)))

then by making actor = CLERK we can satisfy the precondition. Now A only needs to satisfy the implicit "want" precondition of the just introduced INFORM-REF, i.e.,

WANT(CLERK,
  INFORM-REF(CLERK, A, Price(Ticket-for(TR1))))

This can be accomplished by having A request CLERK to perform the act. Thus, A can accomplish step (2) by execution of the new subplan

2.1) REQUEST(A, CLERK, INFORM-REF (CLERK, A, Price(Ticket-for(TR1))))
  achieving WANT(CLERK, INFORM-REF (CLERK, A, Price(Ticket-for(TR1))))
2.2) INFORM-REF(CLERK, A, Price(Ticket-for(TR1)))
  achieving KNOW-REF(A, Price(Ticket-for(TR1)))
2.3) GIVE(A, CLERK, Price(Ticket-for(TR1)))

Similarly, we can ensure that step (3) is executed by planning another request action,

REQUEST(A, CLERK,
  GIVE(CLERK, A, Ticket-for(TR1)))

An example dialogue reflecting these speech acts is:

A: How much is a ticket to Milan? (request (2.1))

CLERK: Thirteen Fifty (inform (2.2))

A: Could I have a ticket please (request for (3))

Note that the discussion to this point has said nothing about the mapping of sentences to their speech act forms. In particular, there are many cases when the system will not be able to compute speech act descriptions directly from the input. Consider the widespread use of indirect speech acts [Searle, 1975], utterances where the speaker, if taken literally, says one thing yet actually means another. For example, "Do you know the time?" is literally a yes-no question, but it is usually used as a request for the time (i.e., REQUEST to INFORM-REF the time). In some settings, where the speaker knows the time and the hearer doesn't, it can even be meant as an offer to tell the hearer the time! Thus, instead of computing a speech act from the actual sentence, we will assume that the system will compute a surface speech act form encoding the literal meaning of the sentence out of context. There is not the space to go into this in detail. The interested reader should see [Allen, 1983].

5. Helpful Responses and Sentence Fragments

Just as recognizing a plan from physical actions was useful for modeling topic structure, recognizing the plan underlying an agent's speech acts will be useful for generating an appropriate response. For instance, if taking the role of the CLERK we observe the speech act

REQUEST(A, CLERK,
  GIVE(CLERK, A, Ticket-for(TR1)))

(ignoring for the moment the steps from the surface form to this speech act), with

DESTINATION(TR1, MILAN),

then we could infer from the effect of the REQUEST, i.e., that A wants the clerk to give A a ticket, that A's plan is

TAKE-TRIP(A, TR1, MILAN)

This would be inferred in the same way that we constructed a plan earlier. If we wished to be helpful, we might inspect the plan to see if we could assist A in other ways besides what was explicitly asked for. For instance, since we believe that A will be next performing the act GOTO(A, TR1), we believe A will need to know where TR1 is, since

KNOW-REF(A, Location(TR1))

is an implicit precondition of GOTO(A, TR1). Thus we might plan to perform the action

GIVE(CLERK, A, Ticket-for(TR1))

as requested, but in addition we might perform

INFORM-REF(CLERK, A, Location(TR1))

to satisfy the precondition on the GOTO act. Of course, if we had believed that A already knew where TR1 left from, the precondition would have already been satisfied and we would *not have generated* the additional action. Such a situation would occur in a small country rail station where all trains left from the same single track. Our model thus provides some account of helpful behavior in dialogues, where the participants do not *simply respond to every request* with the minimum effort required.

Because the response is based on the plan and not directly on the actual utterance, the model also suggests a *method for comprehending sentence fragments*. For instance, even if A had said "Milan, please," we could still recognize A's plan using the constraints in the domain. Since the clerk sells tickets, he or she expects most people to be executing some form of a BUY plan when they speak. Of course, they might not be executing a BUY action; they might need directions to somewhere in the station or something else. But in this domain there are a limited number of general plans that the clerk encounters, and of these plans, only a few of them could possibly involve MILAN. For example, while finding the directions to the bathroom has no relation to MILAN, taking a trip both involves MILAN directly and contains buying a ticket as a subpart. Thus the sentence fragment may contain enough information to identify A's plan and the clerk can respond in the same way as before. If no other actions are observed besides the speech act, the clerk would base the response on the first part of the plan that has not been executed. In this case, the act GOTO(A, CLERK) has presumably been executed since its effects are true, but the action

GIVE(A, CLERK, Price(Ticket-for(TR1)))

has not been observed. Examining the preconditions of this action, the clerk finds

HAS(A, Price(Ticket-for(TR1)))

and the implicit precondition

KNOW-REF(A, Price(Ticket-for(TR1)))

Assuming the domain knowledge encodes that passengers *usually have enough money* but often don't know the price, the clerk would pick the latter as the goal to achieve and plan an INFORM-REF act as appropriate. If, on the other hand, the clerk also observes the act

GIVE(A, CLERK, Price(Ticket-for(TR1)))

together with the sentence fragment, then the next step in the plan would involve the clerk giving A a ticket. The clerk would execute this act as the response to the utterance.

The above approach shows how both linguistic and non-linguistic actions can be incorporated into the same formalism to provide a rich theory of the events that occur during a dialogue. It also shows that an appropriate response can be generated from sentence fragments where the initial speech act is not known. In the first case above, the clerk responded as though A's utterance were a question about the price of a ticket to MILAN (i.e., a REQUEST to INFORM-REF), whereas in the second case, when A also gave the clerk some money, the clerk responded as though the utterance was a request to give A a ticket.

6. Plans about Discourse

Implicit in the above discussion was the fact that only one plan (or topic) was ever introduced and referred to by the various speech acts. We can introduce the need to reason about a wide variety of conversational goals including interruption and resumption of various plans by considering an exchange in the train station such as the following:

1) A: I'd like to buy a ticket to Milan please. How much is it?
2) B: Five dollars.
3) A: OK. Here's a ten. By the way, I'd like to find a newsstand. Is there one around here?
4) B: There's one down the corridor there.
5) A: Thanks. Now, exactly when and where do I catch the train?

As in the example above, B can use A's initial utterances along with a library of typical actions to recognize that A is executing the subaction BUY(A, B, Ticket-for(TR1)), with DESTINATION(TR1, MILAN), as part of the action TAKE-TRIP(A, TR1, MILAN). Unlike the previous examples, however, this TAKE-TRIP plan will not be a useful context for interpretating all of A's subsequent utterances. To understand and respond appropriately to A's request to know if there is a newsstand nearby, B will need to recognize that the goal of going to MILAN becomes temporarily irrelevant and that another one of B's goals, namely going to the newsstand to buy a paper, models the new topic (assuming, of course, a richer plan library than previously presented for the examples above). Finally, in order to correctly interpret utterance (5), B will need to recognize the end of the interruption as well as resumption of the previous TAKE-TRIP topic.

The example above just illustrated that in order to manage interruptions such as change of topic, we will need to introduce more sophisticated knowledge about the ways that actions can be related (as well as not related) to a context of previously recognized actions. In the examples of previous sections, a restricted form of this knowledge was implicit in that it was always assumed that a dialogue consisted of utterances that introduced a plan, followed by utterances that continued discussion of the plan previously introduced. In order to generalize our model, however, we will find it necessary to both expand and make explicit this type of knowledge. To do this, a set of plans about the planning process itself, or meta-plans, will be introduced. Meta-plans will be identical to domain plans except for the fact that every meta-plan will always refer to another plan. For example, we will have meta-plans that introduce plans, execute plans, specify parts of plans, debug plans, abandon plans, etc., independently of any domain. While such knowledge loosely corresponds to the various linguistic ways in which utterances can rhetorically be related to one another, formulating this knowledge within our plan-based framework will enable us to both provide formal semantics for such plans as well as use our already existing framework for plan recognition and manipulation. Finally, to support the various relationships of topic suspension and resumption that we will introduce, we will need to introduce a stack of active and suspended plan structures that we will construct and manipulate during the course of a dialogue.

To illustrate in more detail what is meant by a meta-plan, consider defining the previously implicit knowledge that at the beginning of a dialogue the underlying plan needs to be recognized. An extremely simplified meta-plan formulation of this might be as follows:

INTRODUCE-PLAN(speaker, hearer, plan)
Preconditions: nil
Body: INFORM(speaker, hearer, WANT (speaker, goal))
Effects: BEL(hearer, WANT(speaker, plan))
Constraints: SUB-GOAL(goal, plan)

where the representation is analogous to the planning representation given earlier, with the following two exceptions. First, we now need a vocabulary of predicates such as SUB-GOAL for referring to and describing plans. SUB-GOAL(a, b) is defined to be true only if a is below b in a plan tree. Second, we have added a fourth set of defining conditions for action-types called constraints. These are similar to preconditions, except the planner never attempts to achieve a constraint if it is false. Thus, any action whose constraints are not satisfied in some context will not be applicable in that context.

To capture the earlier assumption that during the remainder of a dialogue the speaker will continue execution of this plan, we could similarly add a meta-plan CONTINUE-PLAN(speaker, hearer, plan) with the precondition that PLAN has already been introduced. Then, to understand the initial portion of the dialogue presented above, we could proceed as follows. B matches A's initial utterance, INFORM(A, B, WANT(A, BUY(A, B, Ticket-for(TR1)))) (with DESTINATION(TR1, MILAN)) to the body of INTRODUCE-PLAN. To successfully make the match, however, B has to satisfy INTRODUCE-PLAN's constraint (find a plan for which wanting to buy a ticket is a subgoal), and thus recognizes that A is executing the plan TAKE-TRIP(A, TR1, MILAN).

Then, B can use CONTINUE-PLAN to understand "How much is it?" (and in the earlier model, all subsequent utterances) in the context of this TAKE-TRIP. Note, however, that by understanding each utterance in this new way, we have explicitly recognized not only the underlying domain plan (TAKE-TRIP), but also the relationship of each utterance to this plan (introduction and continuation, respectively). Unfortunately, without any other additions to our theory, these two meta-plans still only support dialogues without interruptions.

As we will see, by adding a plan stack to our model we can also use INTRODUCE-PLAN to model topic change. Furthermore, we can then expand our library of meta-plans and uniformly treat a wide range of other interrupting subdialogues, for example, clarifications and corrections [Litman and Allen, 1986]. The plan stack will be used to monitor execution of a topic and its various interruptions (including interruption of the interruptions, and so on). In other words, a stack of executing and suspended topics will be built and maintained during a dialogue. The original topic will be at the bottom of the stack and the currently executing topic at the top. When a topic is initiated, it will be pushed onto the stack and the previous topic (the previous top of the stack) suspended. When a topic is completed, it will be popped from the stack and the topic below it resumed. For example, a clarification subdialogue would be modeled by a clarification meta-plan that refers to the plan that is the topic of the clarification. When the clarification plan is recognized it is pushed onto the stack, and the previous top, the plan being clarified, is temporarily suspended. When the clarification is completed, the stack is popped and the previous plan resumed.

Obviously a stack metaphor is an idealization for the interruptions found in many conversations. Conversations in which interrupted topics are not always resumed are fairly common. Thus, our plan recognition algorithm will consider conversations following the stack discipline to be an ideal case. If a conversation can be interpreted in a way that follows a stack discipline, this interpretation will always be preferred over any other possibilities. However, when no such choice of conversational interpretation exists, even if our stack metaphor is violated the non-stack interpretation will nevertheless be pursued. Finally, although we have no room to discuss it here, a truly adequate plan recognition system should be able to use various linguistic clues to recognize marked violations of the ideal stack behavior. For example, a phrase such as "never mind" often signals non-resumption of an otherwise expected topic. A treatment of this issue is found in [Litman, 1985].

Our last elaboration of the basic plan recognition framework will be to note that expectations regarding future meta-plans (topic relationships) vary with respect to their coherence with an already existing context. In other words, all other things being equal, given an utterance to interpret and a plan stack representing the previous discourse context, the plan recognizer will prune its search process by preferring (in the following order) meta-plans that

1) continue plans already on the stack, followed by

2) clarifications and corrections of plans already on the stack, and lastly

3) introductions of totally new plans.

When choosing within preferences, the plan recognizer will prefer interpretations following the stack discipline, i.e., relationships with plans on the top of the stack. Intuitively, this ordering corresponds to the observation that people typically expect a new utterance to be related to earlier ones (as captured in the non-interruption assumption of the earlier sections). When an interruption does occur, people still try to relate the interruption to the previous context, for example by viewing it as a clarification or correction of such. Only as a last resort do people, in the unmarked case, appear to assume that a topic has been suddenly and totally changed. As mentioned above, however, default preferences of the plan recognizer can always be overruled by various linguistic markers. An extremely obvious (yet familiar) example of this would be that when a speaker wants to change a topic, he or she often finds it necessary to preface the new discussion with phrases such as "Changing the topic a bit."

In sum, our plan recognition framework has been extended in several important ways, enabling the processing of a wider range of topic relationships between utterances. To do this our model now includes meta-plans encoding discourse relationships and a plan stack, and our plan recognition algorithm has been modified to deal with these additions. In particular, given an utterance to interpret, a library of plausible domain and meta-plans, and the relevant discourse context (i.e., the plan stack), the plan recognizer will now recognize a meta-plan and either relate it to an existing plan in the relevant context or construct a new plan for the meta-plan to be about. The plan recognizer will then output a modified stack reflecting the results of this recognition.

7. Summary

In this paper we have described the uses of plans and goals in natural language systems, from the use of plans as a model of the topic of stories and conversations to the use of plans as an overall theory accounting for the interactions that occur in natural dialogues. In particular, we presented a plan-based model for understanding questions (including sentence fragments) and generating helpful responses, and showed a promising approach for dealing with indirect speech acts. We then expanded our model to deal with extended dialogues where topics may be suspended and later resumed. For more details on these topics, we suggest the following papers: Cohen and Perrault [1979] for an introduction to speech act planning; Allen [1983] and Allen and Perrault [1980] for the use of plan

recognition in question answering systems; and Litman and Allen [1986] for the extension of these models to topic interruption and resumption.

References

Allen, J.F., "Recognizing intentions from natural language utterances," in M. Brady and R.C. Berwick (Eds). *Computational Models of Discourse*. Cambridge, MA: MIT Press, pp. 107-166, 1983.

Allen, J.F. and C.R. Perrault, "Analyzing intention in utterances," *Artificial Intelligence 15*, 3, pp. 143-178, 1980.

Cohen, P.R. and C.R. Perrault, "Elements of a plan-based theory of speech acts," *Cognitive Science 3*, 3, pp. 177-212, 1979.

Fikes, R.E. and N.J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence 2*, 3/4, pp. 189-208, 1971.

Grosz, B.J., "The representation and use of focus in dialogue understanding," SRI Technical Note 151, July 1977.

Haas, A., "Planning mental actions," Ph.D. thesis (and TR 106), Computer Science Dept., University of Rochester, 1982.

Litman, D.J., "Plan recognition and discourse analysis: An integrated approach for understanding dialogues," Ph.D. thesis, Computer Science Dept., University of Rochester, 1985.

Litman, D.J. and J.F. Allen, "A plan recognition model for subdialogues in conversation," to appear, *Cognitive Science*, 1986.

McCarthy, J., "First order theories of individual concepts and propositions," STAN-CS-79-724, Computer Science Dept., Stanford University, 1979.

Moore, R.C., "Reasoning about knowledge and action," Ph.D. thesis, Massachusetts Institute of Technology, 1980.

Perrault, C.R. and J.F. Allen, "A plan based analysis of indirect speech acts," *American Journal of Computational Linguistics 6*, 3, pp. 167-182, 1980.

Reichman, R., "Conversational coherency," *Cognitive Science 2*, 4, pp. 283-328, 1978.

Sacerdoti, E.D. *A Structure for Plans and Behavior*. New York: Elsevier, 1977.

Schank, R.C. and R.P. Abelson. *Scripts, Plans, Goals and Understanding*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1977.

Searle, J.R. *Speech Acts: An Essay in the Philosophy of Language*. New York: Cambridge University Press, 1969.

Searle, J.R., "Indirect speech acts," in P. Cole and J. Morgan (Eds). *Speech Acts*. New York: Academic Press, 1975.

Wilensky, R. *Planning and Understanding*. Reading, MA: Addison-Wesley, 1983.

The HORNE Reasoning System in COMMON LISP

James F. Allen and Bradford W. Miller
Computer Science Department
University of Rochester
Rochester, NY 14627

TR 126 revised

August 1986

HORNE is a programming system that offers a set of tools for building automated reasoning systems. It offers three major modes of inference: (1) a horn clause theorem prover (backwards chaining mechanism); (2) a forward chaining mechanism; and (3) a mechanism for restricting the range of variables with arbitrary predicates.

All three modes use a common representation of facts, namely horn clauses with universally quantified variables, and use the unification algorithm. Also, they all share the following additional specialized reasoning capabilities: 1) variables may be typed with a fairly general type theory that allows intersecting types; 2) full reasoning about equality between ground terms, and limited equality reasoning for quantified terms; and 3) escapes into LISP for use as necessary. This paper contains an introduction to each of these facilities, and the HORNE User's Manual.

This work was supported in part by the Air Force Systems Command, Rome Air Development Center, and the Air Force Office of Scientific Research under Contract F30602-85-C-0008, which supports the Northeast Artificial Intelligence Consortium (NAIC), by the Defense Advanced Research Projects Agency under Grant N00014-82-K-0193, and by the Office of Naval Research under Grant N00014-80-C-0197.

An Overview of the HORNE Reasoning Capabilities

1. Introduction

This is a brief introduction to the major reasoning modes and facilities provided by the HORNE reasoning system. Details on the actual system are contained in the HORNE User's Manual which forms the second half of this report. In this section, we will first discuss the basic reasoning modes, and then outline the specialized reasoning systems embedded in HORNE.

2. The Basic Reasoning Modes

There are three basic reasoning modes. The first two correspond to the antecedent and consequent theorem mechanisms of PLANNER, and are called *forward chaining* and *backward chaining*, respectively. The third is most closely related to reasoning with constraints, and is called *constraint posting*.

Independent of the mode of reasoning, all facts are in the form of horn clauses, which can be viewed as logical implications with a single consequent. Thus

P < Q

read as "if Q then P," is a horn clause, as is

P <

which simply asserts P, and as is

P < Q, R

which should be read as "if Q and R, then P." The following is *not* a horn clause, because there are two consequences:

P, Q < R.

Note that, in more general systems of this type, this would be read as "if R, then P or Q."

A horn clause may contain globally scoped, universally quantified variables which are indicated by a prefix of "?". Thus

(P ?x) < (Q ?x)

is a horn clause that is read as "for any x, if Q of x holds, then P of x holds." Finally, whenever the process of matching two formulas is discussed, we are referring to the full unification algorithm found in resolution theorem-proving

systems extended to unify lists in LISP format. This extension is explained in detail in the HORNE User's Manual.

2.1 Backwards Chaining

This mode provides a PROLOG-like theorem prover. It searches a horn clause that could prove the given goal, and attempts to prove the antecedents of the horn clause. It uses a depth-first, backtracking search. For the reader not familiar with such systems, see [Kowalski, 1979]. As an example, consider the following axioms:

All fish live in the sea.
(1) (LIVE-IN-SEA ?x) < (FISH ?x)
All Cod are fish.
(2) (FISH ?x) < (COD ?x)
All Mackerel are fish.
(3) (FISH ?x) < (MACKEREL ?x)
Whales live in the sea.
(4) (LIVE-IN-SEA ?y) < (WHALE ?y)
Homer is a Cod.
(5) (COD HOMER) <
Willie is a Whale.
(6) (WHALE WILLIE) <

Given these axioms, we can prove Willie lives in the sea as follows, using a straightforward *backtracking search*. We have the goal:

(7) (LIVE-IN-SEA WILLIE)

Rule 1 appears applicable: Unifying (1) with (7) we get

(LIVE-IN-SEA WILLIE) < (FISH WILLIE)

So we have a new subgoal:

(8) (FISH WILLIE)
Rule (2) applies, giving
(FISH WILLIE) < (COD WILLIE),
so we have a new subgoal
(9) (COD WILLIE)
× No rule applies, try (8) again.
Rule (3) applies, giving
(FISH WILLIE) < (MACKEREL WILLIE)
So we have a new subgoal
(10) (MACKEREL WILLIE)
× No rule applies, try (8) again, no more ways to prove (8)
× No rule applies, try (7) again
Rule (4) applies giving
(LIVE-IN-SEA WILLIE) < (WHALE WILLIE)
So we have a new subgoal
(11) (WHALE WILLIE)
Rule (6) asserts (11) as a fact
√ Goal (11) is Proved.
√ Goal (7) is Proved.

2.2 Forward Chaining

The rules for forward chaining are quantified horn clauses augmented with a *trigger*. Such a rule is applied whenever a fact is added that matches (i.e., unifies with) the trigger. In such a case, the reasoner attempts to prove the antecedents of the rule and, if it is successful, asserts the consequence. In general, each of the antecedents is attempted by simple data base lookup only. In other words, the backwards chaining reasoner is not invoked to prove an antecedent. There is an option, however, to invoke the backwards reasoning if desired.

For example, consider maintaining the simple transitive relation < (less than) using forward chaining. The axiom we want to use to ensure the complete DB is

$$\forall x,y,z\ LT(x,y) \& LT(y,z) \supset LT(x,z).$$

To implement this using forward chaining rules, we have the following:

	Trigger	Rule
(12)	(LT ?x ?y)	(LT ?x ?z) < (LT ?x ?y) (LT ?y ?z)
(13)	(LT ?y ?z)	(LT ?x ?z) < (LT ?y ?z) (LT ?x ?y)

Consider the following additions:

(LT B C) triggers rules (12) and (13), but nothing can be proved

(LT A B) triggers (12) ?x ← A, ?y ← B
- proves (LT A B) ✓
- proves (LT B ?z), ?z ← C
- adds (LT A C)
  - triggers (12) ?x ← A, ?y ← C
    - proves (LT A C)
    - fails on (LT C ?z)
  - triggers (13) ?y ← A, ?z ← C
    - proves (LT A C)
    - fails on (LT ?x A)

triggers (13) ?y ← A, ?z ← B
- proves (LT A B)
- fails on (LT ?x A)

As one can see, the rules apply recursively on inferred additions, and the search space generated by the forward chaining rules is completely searched. The forward chainer detects possible infinite loops that could result from adding the same fact twice.

2.3 Constraint Posting

The last facility allows proofs of goals to be delayed for certain predicates until more is known about the arguments to the predicate. In particular, it allows one to delay proving a formula until one of its variables is bound.

This is best illustrated by example. Assume we want to define a predicate of two arguments, ?x and ?y, that is true iff ?x and ?y are bound to different terms. The most common way to implement this in PROLOG systems is to use negation by failure on the EQ predicate, which is simply defined by

(14) (EQ ?x ?x)

Thus EQ forces two terms to unify, and fails if they cannot. Using this, they define

(15) (NOTEQ ?x ?y) < (UNLESS (EQ ?x ?y))

where UNLESS is negation by failure. This formulation gives undesirable results when one of its terms is unbound. In particular, it binds a variable argument to make the terms equal. Thus with the axioms

(16) (P ?x ?y) < (NOTEQ ?x ?y) (R ?y)

(17) (R B)

we could not prove (P A ?y) for the predicate (NOTEQ A ?y) would fail since (EQ A ?y) succeeds by binding ?y to A.

To avoid this, we could define NOTEQ so that it only fails when both arguments are bound. But this would allow incorrect proofs as the variable could later be bound violating the distinctness condition. What is needed is a facility to delay the evaluation of (NOTEQ ?x ?y) until both arguments are bound. We do this by a mechanism called *posting*.

If a literal is POSTED and contains no variables, it is treated as a usual literal. The proof succeeds or fails and the posting has no effect. If the literal does contain a variable, the evaluation of that literal is delayed until the variable is bound. Thus we define a new predicate DISTINCT by

(18) (DISTINCT ?x ?y) < (POST (NOTEQ ?x ?y)).

Now, using a modified axiom (16), namely,

(19) (P ?x ?y) < (DISTINCT ?x ?y) (R ?y)

and the modified definition of NOTEQ as in axioms (20)-(22), i.e., (NOTEQ ?x ?y) is true if either ?x or ?y is not fully grounded (i.e., it is a term containing a variable), or if the two grounded terms cannot be proven to be equal:

(20) (NOTEQ ?x ?y) < (UNLESS (GROUND ?x))

(21) (NOTEQ ?x ?y) < (UNLESS (GROUND ?y))

(22) (NOTEQ ?x ?y) < (UNLESS (EQ ?x ?y))

Given clauses (17) through (22), we can prove (P A ?y), resulting in ?y being bound to B as follows:

Goal: (P A ?y)

Subgoals: (DISTINCT A ?y) (R ?y)

(DISTINCT A ?y) is proven using (18), but the subgoal (NOTEQ A ?y) is not evaluated in the normal manner since ?y is unbound. Instead, the call succeeds and ?y is annotated to be NOTEQ from A.

(R ?y) succeeds from axiom (17) if ?y can be bound to B. The unifier checks (NOTEQ A B), which succeeds, allowing ?y to be bound.

Thus the goal proved is (P A B). Note that DISTINCT, GROUND, and NOTEQ are built-in predicates in HORNE and are defined using these mechanisms.

Let us consider this mechanism in a bit more detail. After a literal Q has been POSTED, its variables are annotated using a form such as

(any ?x (Q ?x))

which is a term that will unify with any term such that Q holds for that term. Thus (any ?x (Q ?x)) unifies with A only if we can prove (Q A).

If there are multiple variables in a posting, each variable is annotated separately, and the constraints on each are checked as each is bound. For example, the trace of the proof of (P ?x ?y) given axioms (17) - (22) is as follows:

Goal: (P ?x ?y)

Rule (19) applies, giving
(P ?x ?y) < (DISTINCT ?x ?y) (R ?y)
Subgoal
(DISTINCT ?x ?y)
Rule (18) applies, giving
(DISTINCT ?x ?y) < (POST (NOTEQ ?x ?y))
Subgoal
(POST (NOTEQ ?x ?y))
succeeds binding ?x ← (any ?x1 (NOTEQ ?x1 ?y1))
?y ← (any ?y1 (NOTEQ ?x1 ?y1))
Proved: (DISTINCT (any ?x1 (NOTEQ ?x1 ?y1)) (any ?y1 (NOTEQ ?x1 ?y1)))
Subgoal
(R (any ?y1 (NOTEQ ?x1 ?y1))
Rule (17) applies
(R B) if we can unify (any ?y1 (NOTEQ ?x1 ?y1)) with B
[We try subproof of (NOTEQ ?x1 B), which succeeds]

Proved: (P (any ?x1 (NOTEQ ?x1 B)) B)

Thus constrained variables may appear in answers. Users may explicitly construct their own constrained variables in queries and assertions as well, if they wish.

Two constrained variables may unify together as long as the combined constraints are provably consistent in a strong sense, i.e., there exists at least

one proof of the combined constraints. For example, if we had the following data base:

(23) (PA A)

(24) (PB B)

(25) (PB A)

(26) (T (any ?x (PA ?x)))

We could prove the goal (T (any ?y (PB ?y)) by unification with (26) as follows: (any ?y (PB ?y)) and (any ?x (PA ?x)) may unify to (any ?z (PB ?z) (PA ?z)) if there is an object such that (PB ?z) and (PA ?z). A subproof of (PB ?z) (PA ?z) is found with ?z ← A. This binding is not used, however, since the desired answer could be something else. The result is

(T (any ?z (PA ?z) (PB ?z))).

If in a later part of a proof, ?z was unified against a constant k, a subproof of (PA k) (PB k) would be done before the unification succeeds.

3. Built-In Specialized Reasoning Systems

There are two built-in specialized reasoning systems provided with HORNE. These provide typing for terms and simple equality reasoning.

3.1 Types

All terms in HORNE may be assigned a type. If a term is not explicitly assigned a type, it is assumed to belong in T-U, the universal type. Variables over a type are allowed, and a special syntax is provided. The variable ?x*DOG, for instance, signifies a variable ranging over all objects of type DOG. Constants and other ground terms can be asserted to be of a certain type using a built-in predicate ITYPE. Thus

(ITYPE A DOG)

asserts that the constant A is of type DOG.

Types in HORNE are viewed as sets of objects, and all the normal set relationships between types can be described. Thus one type may be a subset (i.e., subtype) of another, two types may intersect or be disjoint, and the non-null intersection of two types produces a type that is a subtype of the two original types. All this information is asserted using built-in predicates. For example,

(ISUBTYPE DOG ANIMAL)

asserts that the type DOG is a subset of the type ANIMAL (i.e., all dogs are animals),

(DISJOINT DOG CAT)

asserts that no object can be both a cat and a dog,

(INTERSECTION FAT-CATS CATS FAT-ANIMALS)

asserts that the set of FAT-CATS consists of all cats that are also fat animals, and

(XSUBTYPE (MALES FEMALES) ANIMALS)

asserts that (MALES FEMALES) is a partition of ANIMALS, i.e., that every animal is either a male or a female, and that all males and females are animals.

All direct consequences of these facts are inferred when the axioms are added. For example, if A and B are disjoint, and A1 is asserted to be a subtype of A, then it is inferred that A1 and B are disjoint. This is done by the forward chaining system. During a proof, the partition information is not used. As a result, asserting (XSUBTYPE (a b) c) has the same effect as asserting (ISUBTYPE a c), (ISUBTYPE b c), and (DISJOINT a b). During adding type assertions, however, partition information is used. For example, given the relationship between a, b, and c above, if we assert (ISUBTYPE d c) and (DISJOINT d a), then it will be concluded that (ISUBTYPE d b).

The type reasoner acts during unification. A constant will match a variable of type Tv only if the constant is of type Tv (i.e., the constant is asserted to be of type Tv, or is of type Tvs which is a subtype of Tv). Two variables unify only if the intersection of their types is non-empty. The result is a variable ranging over the intersection of the two types. Thus, complex types may be constructed during a proof. If types T1 and T2 intersect, but no name for the intersection is asserted, then a complex type I(T1 T2), which is their intersection, is constructed when unifying ?x*T1 and ?y*T2.

This type reasoner provides a complete reasoning facility between simple types. For complex types, however, the reasoner may permit some intersections that may not be desired since they are empty. Note that this can be checked for at the end of a proof if desired. Any intersection of more than two types is guaranteed only to be pairwise non-empty. For example, if the complex type I(T1 T2 T3) is constructed by unifying a variable of type I(T1 T2) with a variable of type T3, then it must be the case that I(T1 T2), I(T1 T3), and I(T2 T3) are non-empty. However, there might be no object that is of type I(T1 T2 T3).

The assertions about the types may be incomplete. For example, two types may be introduced where it is not asserted, or is inferrable, that the types intersect or are disjoint. HORNE provides two modes of proof for dealing with these cases. In the strict mode, two types intersect only if they are known to intersect. In the easy-going mode, two types will intersect unless they are known to be disjoint. Easy-going mode is more expensive, but can be useful in many applications, although it may provide conclusions that on closer inspection are not useful since they contain a variable ranging over the empty set.

As an example, the simple fish data base above could be restated in the typed prover as follows:

(1) (ISUBTYPE COD FISH) <

(2) (ISUBTYPE MACKEREL FISH) <

(3) (ITYPE HOMER COD) <

(4) (ITYPE WILLIE WHALE) <

(5) (LIVE-IN-SEA ?x*FISH) <

(6) (LIVE-IN-SEA ?y*WHALE) <

Although this took one more insertion, it also encodes more information (e.g., whales and fish are disjoint). The proof that WILLIE lives in the sea is much shorter in the typed system. It is completed using only two unifications.

Goal: (LIVE-IN-SEA WILLIE)

unifying with (5) fails as WILLIE is not a fish;

unifying with (6) succeeds, ?y ← WILLIE.

Thus Goal is proved.

If we add the following axioms, we can demonstrate more complicated type reasoning. Let us assume that all animals are either fish or mammals.

(7) (XSUBTYPE (FISH MAMMAL) ANIMALS)

This asserts that both FISH and MAMMAL are subtypes of ANIMAL and that they are disjoint. Note that since COD and MACKEREL are subtypes of FISH, these will also now be disjoint from MAMMALS.

(8) (ISUBTYPE WHALE MAMMAL)

This asserts that WHALE is a subtype of MAMMAL, and hence WHALE is disjoint from FISH.

(9) (ISUBTYPE WHALE THINGS-THAT-SWIM)

(10) (ISUBTYPE FISH THINGS-THAT-SWIM)

Note that in asserting that WHALE is a subtype of THINGS-THAT-SWIM, the system then knows that MAMMAL and THINGS-THAT-SWIM intersect.

(11) (BEAR-LIVE-YOUNG ?m*MAMMAL)

(12) (SWIMS-WELL ?t*THINGS-THAT-SWIM)

Now if we try to find something that bears live young and swims well, i.e., find ?x such that

(BEAR-LIVE-YOUNG ?x) (SWIMS-WELL ?x),

we succeed by unifying the first subgoal to (11), causing ?x ← ?m*MAMMAL, and the second subgoal to (12), causing ?m*MAMMAL and ?t*THINGS-THAT-SWIM to be unified, resulting in a complex variable ?y*I(MAMMAL THINGS-THAT-SWIM). Thus the answer is: all things that are both of type MAMMAL and THINGS-THAT-SWIM. If we add

(13) (LARGE ?w*WHALE)

and query for something that bears live young, swims well, and is large, we will end up unifying ?y*I(MAMMAL THINGS-THAT-SWIM) with ?w*WHALE. The result of this is simply ?w*WHALE, since WHALE is a subtype of both MAMMAL and THINGS-THAT-SWIM.

Constrained variables may be typed in the obvious manner. For example

(any ?x*MAMMAL (SWIMS-WELL ?x*MAMMAL))

is a term that will unify with any term t such that t is of type MAMMAL, and (SWIMS-WELL t) is provable. It is interesting to note that the constrained variable system could be used to implement a typed system directly, where a variable ?x*MAMMAL would be replaced by (any ?x (TYPE ?x MAMMAL)). The semantics of the two notations are identical. Types are so common, however, that the special notation for variables is maintained and types are optimized in the implementation.

Unification between a typed constrained variable and a typed variable results in the expected answers. Thus, unifying ?x*MAMMAL with (any ?y*ANIMAL (SWIMS-WELL ?y*ANIMAL)) succeeds with the result (any ?z*MAMMAL (SWIMS-WELL ?x*MAMMAL)). Unifying ?x*ANIMAL with (any ?y*MAMMAL (SWIMS-WELL ?y*MAMMAL) succeeds simply and ?x*ANIMAL is bound to the constrained variable.

Unifying a constrained variable with a term that itself contains variables may introduce new constrained variables. For example, if we are given the fact (P (f A)), then unifying (any ?x (P ?x)) with (f ?w) will produce the term (f (any ?z (P (f ?z)))). This is the correct result since the constrained variable ?x will unify with any term such that (P ?x) is provable. Since (P (f ?z)) is provable (because of the fact (P (f A))), the terms unify. The variable ?w is not bound to A, however, since there may be other terms for which (P (f ?z)) holds as well. Thus (P (f A)) might not be the most general unifier.

These examples are summarized in Figure 1.

Term 1	Term 2	Most General Unifier
(any ?x*MAMMAL (SWIMS-WELL ?x))	WILLIE	WILLIE
(any ?x*MAMMAL (SWIMS-WELL ?x))	?a*ANIMAL	(any ?x*MAMMAL (SWIMS-WELL ?x))
(any ?x*MAMMAL (SWIMS-WELL ?x))	?w*WHALE	(any ?z*WHALE (SWIMS-WELL ?z))
(any ?x (SWIMS-WELL ?x))	(SPOUSE ?a)	(SPOUSE (any ?z (SWIMS-WELL (SPOUSE ?z)))) assuming that the query (SWIMS-WELL (SPOUSE ?a)) succeeds
(any ?x (SWIMS-WELL ?x))	(any ?y (BEAR-LIVE-YOUNG ?y))	(any ?z (SWIMS-WELL ?z) (BEAR-LIVE-YOUNG ?z)) assuming that the query (SWIMS-WELL ?z) (BEAR-LIVE-YOUNG ?z) succeeds

Figure 1: Unification with Constrained Variables

3.2 Typing Functions

Because of the additional complexities involved, a special system is provided for typing functions. This is needed for reasoning about function terms that contain variables. If the only functions used in the system are always fully grounded, the standard type system can be used directly.

For a given function, one can specify the type of the result of the function, plus the types on the arguments of the function. Any function term whose arguments violate these typing restrictions will be flagged as an error. Thus if we define the function SPOUSE to map from PERSON to PERSON, the term (SPOUSE WILLIE) will cause an error, since WILLIE is a WHALE and thus cannot be a PERSON. This function could be defined as follows:

(declare-fn-type 'SPOUSE '(PERSON) 'PERSON),

i.e., the function SPOUSE takes one argument of type PERSON, and produces objects of type PERSON.

Of course, one might like to do better than this, and define SPOUSE to be of type MALE when the argument is FEMALE, and FEMALE when the argument is MALE. Such definitions can be done in HORNE given the following conditions:

1) the function takes a single argument;

2) the function is first declared to the most general type of arguments allowed, and the most general type of objects produced;

3) further declarations are consistent with the other declarations so far;

4) all further declarations have the most general argument type for the specified range type.

In other words,

(declare-fn-type 'SPOUSE '(FEMALE) 'MALE)

is allowed since

1) it is consistent with the initial definition of spouse;

2) every function with argument type FEMALE produces an instance of type MALE;

3) all function instances of type MALE must have an argument type FEMALE.

Similarly, (declare-fn-type 'SPOUSE '(MALE) 'FEMALE) is allowed.

This will produce the appropriate results during unification. Thus if we unify (SPOUSE ?m*PERSON) with ?x*MALE, the result is (SPOUSE ?m*FEMALE), as desired.

One cannot define a further specification that produces instances of a type already used in a specification, but with a different argument type. For example, the following is not allowed:

(declare-fn-type 'fn '(T-U) 'PERSON)

(declare-fn-type 'fn '(MALE) 'MALE)

(declare-fn-type 'fn '(FEMALE) 'MALE) ** ERROR **

since the last declaration violates assumption (4) above. Neither MALE nor FEMALE is the most general argument type producing instances of type MALE.

Function typing does not guarantee that functions fully cover their range type (i.e., they are not necessarily "onto"). For example, given

(declare-fn-type 'G '(T-U) 'ANIMAL)

the query

(EQ (G ?x) ?w*WHALE)

will *fail*, since there is no guarantee that any terms of form (G ?x) are of type WHALE, even though all are of type ANIMAL. Even if there is a known instance of G of type WHALE, such as (EQ (G ABLE) WILLIE), the above proof

will still fail. It is difficult to do otherwise and yet still produce a most general unifier. Some scheme using constrained variables would be possible but would probably be expensive.

3.3 Equality

The system offers full reasoning about equality for ground terms. Thus if you add

(1) (EQ A B) <

(2) (EQ B C) <

(3) (P A) <

you will be able to successfully prove the goal (P B) as well as (P C). Furthermore, given the assertion

(4) (P (f A))

you will be able to successfully prove the goals (P (f B)) and (P (f C)). Adding

(5) (EQ (g A) B)

allows you to prove a potentially infinite class of goals, including (P (g A)), (P (g B)), (P (g C)), (P (g (g A))), (P (g (g B))), etc., to arbitrary depths of nesting of the g function.

An incomplete facility is offered for reasoning about equality for non-ground terms as follows. With a data base of equalities between grounded terms, one can prove an equality statement with variables in it and the variables will be bound appropriately. All possible bindings of the variable are computed and returned in an *any* form so that backtracking to the equality is never needed. Thus if we have

(EQ (f B) G)

(EQ (f A) G)

and we try to prove

(EQ (f ?x) G)

?x will be bound to (any ?x1 (MEMBER ?x1 (A B))). Multiple variables are also handled correctly by this scheme.

A very limited facility is provided for adding equality statements that contain variables. Essentially, these can be used to prove an equality by a single direct unification. Thus if we add

(EQ (f ?x) (g ?x))

(EQ (f ?x) (h ?x))

we will be able to prove

(EQ (f A) (g A)),

(EQ (f A) (h A)), and

(EQ (f (g ?x)) (g (g ?x)),

but not

(EQ (g A) (h A)).

3.4 Structured Types

The REP extension to HORNE supports a hierarchy of structured types akin to frame-based knowledge representations. This facility allows one to associate roles with a type, and it allows subtypes to inherit roles from their supertypes.

Formally, a role is a distinguished function associated with a type. In particular, the function is defined on all objects in the class named by the type. There are two ways to access the values of roles of a given object. The first is by using the appropriate function; the second is by using a special predicate named ROLE. For example, say for the type T-ACTION, we have an "actor" role. Then if A is an object of type action,

(f-actor A)

is the actor of A, as is the value of ?x in

(ROLE A R-ACTOR ?x).

Either one of these constructs can be used to retrieve the actor role. The second method, using the ROLE predicate, is more general, as it allows the user to query role names as well as values. For example, we could find what role ?r an object X plays with A by the query

(ROLE A ?r X).

Certain types may have a set of role names that suffice to uniquely identify each object in that type. In other words, if two objects of that type agree on all their roles, then the objects must be identical. These we shall call *functional types*. For functional types, a function can be defined that maps the set of roles to the object that they identify. For example, if an event of type T-MELT is completely defined by the object melting (R-OBJECT), the time (R-TIME), and the location of melting (R-LOC), then we can define a function

(c-melt ?o*T-PHYS-OBJ ?t*T-TIME ?l*T-LOCATION)

that generates the class of melting events.

Given this informal semantics, we can see that certain relations hold between constructor functions and role functions. In particular, if M is any melting event as defined above, then we know that

M = (c-melt (f-object M) (f-time M) (f-loc M))

even if we do not know the actual values of the three roles of M.

The REP system automatically generates the above function definitions and supports the required equality reasoning between objects, constructor functions, role functions, and the ROLE predicate.

Structured types are declared to the system using two commands, introduced here by example. Full details can be found in Chapter 14 of the manual. The command:

(define-subtype 'T-ACTION 'T-EVENT '(R-ACTOR T-ANIM))

Defines T-ACTION to be a subtype of T-EVENT with the role R-ACTOR defined. All values of R-ACTOR are of type T-ANIM. In addition, T-ACTION will inherit any roles defined with the type T-EVENT. In particular, a function f-actor is defined that maps an object of type T-ACTION to an object of type T-ANIM.

The ROLE predicate is axiomatized such that any object O which is asserted to be the R-ACTOR of some action A will be equal to (f-actor A). Thus if we add

(ROLE A R-ACTOR O)

then

(EQ (f-actor A) O)

will automatically be asserted as well.

On the other hand, the command

(define-functional-subtype 'T-ACTION 'T-EVENT '(R-ACTOR T-ANIM))

would do all of the above, and in addition defines a function C-ACTION that takes an object of type T-ANIM and produces an object of type T-ACTION.

The system is set up so that any instance of type T-ACTION will be equal to its appropriate constructor function. Thus, if we now add

(ITYPE A T-ACTION)

the assertion

(EQ A (c-action (f-actor A)))

would be asserted as well.

With the equality reasoning abilities of HORNE, the system can now integrate all role values as they are asserted later and the appropriate conclusions regarding the equality of objects can be derived. Thus, if we add

(EQ (f-actor A) O) and

(ROLE A R-ACTOR O')

then (EQ O O') will be concluded. Furthermore, the equalities

(EQ A (c-action O))

(EQ (c-action O) (c-action O'))

can be derived as needed during any proof.

Roles are automatically inherited from supertypes at the time the structured type is defined. These inherited roles will appear in constructor functions following the role values that were explicitly defined with the type. An inherited role may be redefined lower in the hierarchy only if the type restriction on the new role definition is a subtype of the original role definition. For example, assuming T-ACTION was a regular (non-functional) subtype of T-EVENT as defined above, if we use

```
(define-functional-subtype 'T-OBJ-ACTION 'T-ACTION
                           '(R-OBJ T-PHYS-OBJ))
```

a constructor function of the form

(c-obj-action ?obj*T-PHYS-OBJ ?a*T-ANIM)

would be defined. On the other hand, the definition

(define-functional-subtype 'T-SING 'T-ACTION '(R-ACTOR T-PERSON))

would be allowed only if T-PERSON were a subtype of T-ANIM. If this were so, a constructor function of the form

(c-sing ?a*T-PERSON)

would be defined.

HORNE User's Manual

Version 28.0, August 1986

TABLE OF CONTENTS

1. INTRODUCTION

HORNE is a Horn-clause-based reasoning system embedded in a LISP environment. Its facilities are called as LISP functions and HORNE programs can themselves call LISP functions. Thus, effective programming in HORNE involves a careful mixture of logic programming and LISP programming. This manual assumes that the user is familiar with the fundamentals of both LISP and Prolog. The naive user should consult Winston and Horn (1981) for an introduction to LISP, and Kowalski (1974; 1979) and Bowen (1979) for PROLOG. The system is fully implemented, and runs in COMMON LISP.

1.1 Using This Manual

Several notational conventions are followed throughout this manual. Function calls that can be made to the HORNE system are shown in italics. HORNE distinguishes between upper and lower case letters. Therefore it is imperative that the reader pay close attention to the case. The usual LISP documentation convention of quoting parameters that are evaluated during function calls is used. For example, in the call

(function-name <arg$_1$> '<arg$_2$>)

<*arg$_2$*>, but not <*arg$_1$*>, is evaluated. Throughout, all functions ending in the letter "q" do not evaluate their arguments, while most other functions do.

1.2 Syntax

The three major classes of expressions in this language are terms, atomic formulas, and axioms. The syntax for these classes are given by the following BNF rules:

```
<axiom>              ::= ( <conclusion> ) |
                         ( <conclusion> <index> ) |
                         ( <conclusion> <index> <list of premises> )
<conclusion>         ::= <atomic formula>
<list of premises>   ::= <premiss> | <premiss> <list of premises>
<premiss>            ::= <variable> | <atomic formula> | /
<index>              :=  <literal atom> | <list of indexes>
<atomic formula>     ::= ( <predicate name> <list of terms> )
<predicate name>     ::= <constant>
<term>               ::= <constant> | <variable > | ( <list of terms> )
<constant>           ::= <literal atom>
<variable>           ::= ? <literal atom>
<list of terms>      ::= <ε> | <term> | <term> <list of terms> |
                         <term> . <term>
<ε>                  ::=
```

An example of an axiom is: ((P ?x) <1 (Q ?x)) where "(P ?x)" is the <conclusion>, "<1" is the index, and "(Q ?x)" is a simple <list of premises>.

This statement is interpreted as follows: the assertion named "<1" signifies that for any x, (Q x) implies (P x). Or, alternately, to prove (P x) for any x, try to prove (Q x).

1.3 Special Symbols

The HORNE system uses two special symbols which should not be used for other purposes:

> "?" indicates a variable will cause the atom following it to be expanded into the internal variable format. This is true only in axioms. The symbol can be used freely in LISP code.

1.4 Running HORNE

From a COMMON-LISP listener do (pkg-goto 'horne) to use HORNE commands, or (pkg-goto 'rep) to use REP and HORNE commands. Commands are exported, so a user package can do :use 'rep to get REP and HORNE commands.

2. BASIC HORNE PROGRAMMING

This section explains how the HORNE database can be modified and examined, and how theorems can be proved.

2.1 Defining and Deleting Predicates

Several simple functions are available for asserting and retracting axioms.

(axioms '<list of axioms>)

Asserts all of the axioms in *<list of axioms>* at the end of the database in the order they appear in the list. Same as *addz*.

(adda '<axiom$_1$> ... '<axiom$_n$>) and *(addaq <axiom$_1$> ... <axiom$_n$>)*

Adds all the axioms to the beginning of the database. *<axiom$_1$>* will precede *<axiom$_2$>* in the database, etc. Warning: This operation is much more expensive than *addz* or *axioms*.

(addz '<axiom$_1$> ... '<axiom$_n$>) and *(addzq <axiom$_1$> ... <axiom$_n$>)*

Adds all the axioms to the end of the database. *<axiom$_1$>* will precede *<axiom$_2$>* in the database.

(retracta '<predicate name>) and *(retractaq <predicate name>)*

Retracts the first axiom in the database that concerns *<predicate name>*.

(retractz '<predicate name>) and *(retractzq <predicate name>)*

Retracts the last axiom in the database that concerns *<predicate name>*.

(retractall '<pattern>) and *(retractallq <pattern>)*

Retracts all the axioms in the database whose conclusions unify with the specified pattern. The predicate name must be specified in the pattern. If an atom is given as a pattern, it will be interpreted as a predicate name and all axioms for that predicate will be deleted. For example, (retractall '(P A ?x)) retracts all axioms whose head unifies with (P A ?x) (e.g., (P ?x ?z), (P ?x B), (P A B)), and (retractall 'P) retracts all axioms for predicate P.

(clear '<index>) and *(clearq <index>)*

Retracts all axioms in the database with an index matching the specific index. This function accepts patterns for complex indexes. Thus (clear '(ff ?x)) would delete all axioms with an index consisting of a two-element list with the first atom being "ff" (e.g., (ff 1), (ff DD), (ff (aa b))).

(clearall)

Deletes all axioms defined by the user.

(reset)

Deletes all axioms, equality information, hashing information, and function type definitions. Essentially restores system to its initial state.

(reset-all-tracing)

Turns off all tracing and warnings user has enabled.

Predicates in HORNE can either have a constant arity or can vary. The addition mechanism assumes that any predicate not previously specified as a varying predicate is constant. To define a predicate with a varying number of arguments, use the function

(declare-varyingq <predname1> ... <prednamen>),

e.g.,

(declare-varyingq or* and*)

The predicate or* defined in Section 5.3 is an example of a predicate that has to be declared to be varying. Only varying predicates allow list matching on their arguments. Thus, for or*, we can use a term of form (or* ?first . ?rest) and the variables will be matched appropriately.

2.2 Examining the Database

The database of axioms can be examined with the following functions:

(printp '<pattern>) and *(printpq <pattern>)*

Pretty prints all of the axioms whose conclusions unify with the pattern, including comments. As with rall, atomic patterns are assumed to be predicate names.

(printi '<index>) and *(printiq <index>)*

Pretty prints all of the axioms that have an index that unifies with the specified index.

(relations)

Returns a list of all the predicate names currently defined in the system. This includes all of the predicate names that are LISP functions.

(indices)

Returns a list of all the indices in use.

(axioms-by-index '<index>)

Returns a list of axiom names associated with the given index. This uses a direct match of the index without unification.

(axioms-by-name-and-index '<pred-name> '<index>)

Returns all the axioms with the given predicate name and the given index. This uses a direct match of the index without unification.

There are also functions for accessing the data base without invoking the prover:

(find-facts '<atomic formula>) and *(find-factsq <atomic formula>)*

Returns all axioms of form *(<conclusion>)* or *(<conclusion> <index>)* that unify with the specified formula. Thus to find all axioms that assert that P is true of something, we could use (find-facts '(P ?x)). If the data base contained the facts

((P A))
((P B) <3)
((P D) <4 (Q R))

then the query would return (((P B) <3) ((P A))).

(find-facts-with-bindings '<atomic formula>)

Same as *find-facts* except that it returns the variable bindings as well in the format ((<axiom> <binding list>)*). For example, with the above three axioms for P, the query (find-facts-with-bindings '(P ?x)) would return

((((P B) <3) ((?x B))) (((P A)) ((?x A)))).

(find-clauses '<atomic formula>)

Returns all axioms whose conclusion unifies with the specified formula. The same restrictions on variable naming as with *find-fact* hold for this function. It would return all three of the above axioms in the query (find-clause '(P ?x).

(get-facts '<atomic formula>)

Same as *find-facts* except that the conclusion must be identical to the specified formula ignoring variable naming, e.g., (get-facts '(P ?x)) with the above three axioms would return NIL.

(get-clauses '<atomic formula>)

Same as *find-clauses* except that the conclusion must be identical to the specified formula ignoring variable naming.

2.3 Proving Theorems

The theorem prover is invoked by calling the LISP function *prove* with a set of formulas that represent the goal clause.

(prove '<atomic formula$_1$> ... '<atomic formula$_n$>)
(proveq <atomic formula$_1$> ... <atomic formula$_n$>)

> Attempts to prove the list of formulas, and returns a bound solution if one is found. This will be a list of the atomic formulas in the same form as given to prove.

Once a proof is completed, you can find out the execution time in seconds by calling *(runtime)*. The answer returned by the last query can be printed using the function *(print-answer)*.

There are variations on the *prove* command that allow multiple answers to be found. These are indicated by an optional first argument as follows:

(prove :query '<atomic formula$_1$> ... '<atomic formula$_n$>)
(proveq :query <atomic formula$_1$> ... <atomic formula$_n$>)

> Prompts the user each time a solution is found, and queries whether to search for another or not.

(prove :all '<atomic formula$_1$> ... '<atomic formula$_n$>)
(proveq :all <atomic formula$_1$> ... <atomic formula$_n$>)

> Does an entire search of the axioms and returns all solutions found. Note that currently if there is an infinite path in the proof tree (e.g., a transitivity axiom) then this function will not return. Will return a list of the lists of the formulas with their variables bound appropriately, e.g.,
>
> ```
> (addzq ((happy joe)<)
>        ((happy mary)<)
>        ((sad frank)<))
> (proveq :all (happy ?x) (sad ?y))
> ```
>
> returns
>
> ```
> (((happy joe) (sad frank)) ((happy mary) (sad frank)))
> ```

(prove <number> '<atomic formula$_1$> ... '<atomic formula$_n$>)
(proveq <number> <atomic formula$_1$> ... <atomic formula$_n$>)

> Finds *<number>* proofs of the goal obtained by evaluating '<formula>. Note that *(prove 1 <formula>)* is equivalent to *(prove '<formula>)*.

Note: Every 500 proof steps the theorem prover prompts the user whether to continue or not. When you see the output "*continue?*", respond with a "y" to continue, "n" to stop. Also at this point, any LISP function can be evaluated and the system will then reprompt whether to continue. See Section 7 to change the number of steps before a prompt.

2.4 Comments

Comments can be added for each predicate name. These are then printed by the various print functions.

(add-comment '<predname> '<comment>)

Adds a comment to the predicate specified (and deletes any existing comment). The comment can be any LISP expression, but it is most convenient to use strings, e.g.,

(add-comment 'loves "This is a comment")

Strings can include carriage returns, so longer comments can be used.

(add-to-comment '<predname> '<comment>)

Extends an existing set of comments with the new comment.

(print-comment '<predname>)

Prints the comments for a predicate.

3. THE PREDICATE EDITOR

The axioms of a single predicate can be defined and modified using the HORNE predicate editor, which is entered with the function (*edita <predicate name>*). An online help facility is provided with the editor using (CNTRL)-HELP. Once the editor has been entered, the following commands are available:

<number$_1$> ... <number$_n$>

p Print the axioms with numbers.

q (Quit) Complete the edit.

u Undo all changes made to the axioms (i.e., complete restart).

a <number>

Add an axiom at indicated position. You will be prompted for the axiom. If index is "z" then axiom is added at the end.

r <number$_1$> ... <number$_n$>

Delete the indicated axioms. The remainder axioms are renumbered.

e <axiom #>

Enter intra-axiom editor mode. Single axioms may be edited using the input editor in this mode. On entering this mode you will be prompted for the number of the axiom to be edited.

m <number$_1$> ... <number$_2$>

Move axiom number *<number$_1$>* to position *<number$_2$>*.

c

(for "cancel") Undoes the last change.

h <command>

Online help facility.

<control> <HELP>

Help for Symbolics input editor.

4. TRACING AND DEBUGGING IN HORNE

The HORNE system provides extensive tracing facilities that operate on the entire proof, or on selected predicates. There are four places where tracing may occur during the processing of a single goal. These are called the *:q*, *:a*, *:b*, and *:r* tracepoints throughout, and are defined as follows:

- The *:q* tracepoint is the point where the goal is first selected by the prover;
- the *:a* tracepoint is the point where a clause is selected in an attempt to prove the goal;
- the *:b* tracepoint is the point where the prover resumes after backtracking (note that the *b* points are a proper subset of the *a* points);
- the *:r* tracepoint is the point where the goal has been proven and the prover is "returning" to consider a new goal.

In every trace function you can explicitly specify which tracepoints you want. If they are not specified, the default is the *:q* and *:r* tracepoints.

4.1 Global Tracing Controls

(htraceall)

When called it turns on a trace of HORNE showing every formula that is about to be proved (i.e., at the *q* tracepoint), as well as indicating when a formula has been proved (i.e., at the *r* tracepoint). It can take the following optional specifications:

(:at <tracepoint>)

Indicates tracing at the specified tracepoints only, e.g., (htraceall (at :q :b)) traces all predicates at the query and backtracking points.

:break

Indicates a break is desired in addition to a trace message. See 4.3 for a description of the break package.

(:using <LISP function>)

Indicates that a user-supplied function should be called at the tracepoint rather than printing a message. See Section 4.4 for details.

These can be combined as you wish. For instance, if you want a break at backtracking points, and a trace of query points, use (htraceall break (at :b)), (htraceall (at :q)).

(unhtraceall)

Turns off all tracing.

4.2 Selective Tracing

The user can trace individual goals by identifying which predicate names are to be traced. The simple form of this function is described first, then further options are introduced.

(htrace '<predspec$_1$> ... '<predspec$_n$>) or
(htraceq <predspec$_1$> ... <predspec$_n$>)

When <predspec> is a simple predicate name (e.g., (htraceq P)), this causes tracing at the *q* and *r* tracepoints of all goals that have the specified predicate name as their head. When <predspec> is a list of form (<predname> <options>*), the user can specify various options as described in Section 4.1. For example, (*htraceq* (*P* (*at* :*q* :*a*))) traces P at the tracepoints :q and :a.

(unhtrace '<predicate name$_1$> ... '<predicate name$_n$>) or
(unhtraceq <predicate name$_1$> ... <predicate name$_2$>)

Turns off selective tracing. If no predicates are specified, all selective tracing is undone.

A similar set of tracing facilities are provided for tracing by the index of clauses rather than the predicate name in the conclusion. In index tracing, however, only the *a* and *b* tracepoints can be specified.

(htraceiq <index-spec$_1$> ... <index-spec$_n$>)

Turns on tracing for the specified index.

An <index-spec> is of the following form:

(<index pattern> <options>)*

An <index pattern> is an expression that may contain HORNE variables. Any clause with one index that unifies with the pattern is traced. For example, *(htraceiq (<1) (<3))* would cause tracing at all a tracepoints that use a clause with index "<1" or "<3," and *(htraceiq ((<G ?x)) ((F ?x) :break))* would cause tracing at all a tracepoints using a clause with an index unifying with (<G ?x), and cause a break at all a tracepoints using a clause with an index unifying with (F ?x).

(unhtraceiq <index$_1$> ... <index$_n$>)

Undoes the above trace commands. If these are called with no arguments, all index tracing is turned off.

The trace messages all involve printing out formulas. To control the I/O behavior one can set limits on how deep a formula will be printed, as well as the length. This is controlled by the global variables:

H$$LENGTH - the length (depth) of formulas to be printed (default is 6).

4.3 The Break Package and Traces of Proofs

Once a proof is interrupted using a break in the trace package, the programmer can look around at what is happening, modify the tracing behavior, etc. To continue the proof, enter *go*. Some useful functions for debugging are:

(goal) -- prints the current formula to be proved.

(top) -- prints the current top of the goal stack.

(stack) -- prints the current goal stack (see below).

(show-proof-trace) -- prints a trace of the proof up to the current point (see below).

<(show-facts)> -- prints the axioms that could directly prove the goal.

<(show-clauses)> -- prints the clauses that could be used to prove the goal.

The goal stack contains the current formula being proved at each level of recursion, plus all the succeeding formulas that need to be proven once the current formula succeeds. Thus if we had the axioms

((A) < (B) (C) (D))
((B) <)
((C) < (E) (F))

and we put a break on the predicate in E (i.e., (htraceq (E break)), in trying to prove A we would find the following stack at the break point:

((E) (F))
((C) (D)).

In other words, we're trying to prove E, after which we will try to prove F. If both succeed then we will have proven C, and will try to prove D.

Any valid LISP expression can also be evaluated while debugging.

After a proof has been found, one can obtain a full trace of the successful proof tree. If multiple proofs are found, a list containing each individual proof is returned. For efficiency reasons, however, a proof trace is not collected unless some predicate is being traced.

(proof-trace)

Returns the successful proof tree(s) of the last call to the prover, or, if called within a proof break, returns the current state of the proof tree. For formatted printing of the trace, you can call *(show-proof-trace)*.

The format of the proof tree is *(<conclusion> <index> <proof-trace of subgoals>)*.

Thus, given the axioms

```
(A <1 B C)
(B <2 D)
(C <3)
(D <4)
```

if we proved the goal A, the proof tree would be

```
(A <1  (B <2 (D <4))
       (C <3))
```

4.4 User Defined Trace Functions

Users can define their own tracing functions for use in the HORNE system. All tracing functions must have the same form: they must be lambda expressions taking two arguments. The first is set to the type of tracepoint (i.e., either *q*, *a*, *b*, or *r*) and the second is the instantiated clause that caused the trace. The default tracer simply prints this information at the terminal after some formatting. For example, we could define our own trace function as follows:

```
(defun ttt
   (tpoint clause)
      (terpri)
      (print (list tpoint clause)))
```

Then given the three axioms:

```
(P ?x) < (Q ?x ?y) (R ?y)
(Q A ?z)
(R B)
```

and the trace command

```
(htraceall (using ttt)),
```

we get the following output during the proof of (P ?d):

```
(q (P ?d))
(q (Q ?d ?y1))
(r (Q A ?y1))
(q (R ?y1))
(r (R B))
(r (P A))
```

5. THE HORNE/LISP INTERFACE

So far, we have seen how the various HORNE facilities can be invoked from within LISP. This section explains how LISP facilities can be used within HORNE.

5.1 Assigning LISP Values to HORNE Variables

There is a simple mechanism for binding a HORNE variable to an arbitrary LISP value. This is accomplished by using the built-in predicate:

(SETVALUE <variable> <LISP expression>)

> This evaluates the *<LISP expression>* as a LISP program and binds the result to the HORNE variable specified. If the variable is already bound, SETVALUE will fail.

(GENVALUE <variable> <LISP expression>)

> This is the same as SETVALUE except that the LISP expression is expected to return a list of values. The variable will be bound to the first value, and if the proof backtracks to this point, to the succeeding values one at a time.

5.2 Predicate Names as LISP Functions

Occasionally it is useful to let a predicate name be a LISP function that gets called instead of letting HORNE prove the formula as usual. The predicate name "NOTEQ," for example, tests its two arguments for inequality by means of a LISP function because it would be impractical to have axioms of the form ((NOTEQ X Y)) for every pair of constants X and Y. These special LISP functions must be macros, or use the &rest argument facility. They receive their argument list from HORNE with all bound variables replaced by their values. To declare such a LISP function to HORNE use

> *(declare-lispfnq <name$_1$> ... <name$_n$>)*

From then on HORNE will recognize those <name>s as LISP functions. LISP functions should only return "t" or "nil" which will be interpreted as true and false respectively. For example, assume we enter the following:

```
(defmacro check (&rest x)
    (terpri)
    (princ "in check, args are:")
    (princ x)
    t)
(declare-lispfnq check)
(addzq ((P ?x ?y) < (check ?x ?y)))
```

Then if we call

```
(proveq (P A B))
```

the LISP function check is called resulting in the output:

in check, args are: (A B).

Since check returns a non nil answer, the LISP call is treated as a success.

Other useful functions for manipulating argument lists within LISP are:

(isvariable '<term>)

Returns the variable name if <term> is an unbound HORNE variable; otherwise it returns nil.

(vartype '<variable>)

Returns the type of the HORNE variable, or nil otherwise.

(bind '<variable> '<value>)

Binds the HORNE variable to the value of the LISP expression. If the first argument is not a HORNE variable, it returns nil. Example: the following LISP function sets the first HORNE argument to 4 if it is a variable:

```
(defmacro SetTo4 (&rest x)
   (cond ((isvariable (car x))
          (bind (car x) (+ 1 3)))))
```

5.3 Using Lists in HORNE

Since HORNE is embedded in LISP, one can use the LISP list facility directly. In fact, the HORNE unifier can be thought of both as operating on logical formulas, and matching arbitrary list structures.

The unifier will handle the dot operator appropriately anywhere except at the top level of non-varying predicates. Thus the following pairs of terms unify with the most general unifier shown:

(a b c)	(a ?x ?y)	with m.g.u. {?x/b, ?y/c}
(a b c)	(a . ?x)	with m.g.u. {?x/(b c)}
(a b c)	(?x . ?y)	with m.g.u. {?x/a, ?y/(b c)}
(a b c)	(a ?x . ?y)	with m.g.u. {?x/b, ?y/(c)}
(a b)	(a ?x .?y)	with m.g.u. {?x/b ?y/nil}
(a)	(a ?x .?y)	does not unify.
(a b)	(?x)	does not unify. (?x) only matches lists of length 1.

List unification is also allowed with varying arity predicates, although the predicate name position cannot contain a variable. Consider the definition of the predicate *or** that is true if any of its arguments is true:

```
(declare-varyingq or*)
((or* ?x .?y) < ?x)          or* is true if the first argument is true
((or* ?x . ?y) <(or* . ?y))  or* is true if or* of all but the first
                                argument is true
```

Thus the call with no arguments, *(or*)*, always fails and each of *(or* (A))*, *(or* (B) (A))*, and *(or* (B) (A) (C))* succeeds if *(A)* is provable.

5.4 Manipulating Answers from HORNE

Once a proof succeeds, these commands can manipulate the answer returned.

(get-binding '<varname>)

Returns the binding for the named variable. For example, *(getbinding '?x)* will return the binding for ?x in the last proof. If multiple solutions were found in the last proof, a list of bindings is returned.

(get-answer)

Returns the answer found in the last query. If multiple answers are found, a list of answers is returned.

6. SAVING AND RESTORING PROGRAMS

These commands allow the user to partially or entirely save his HORNE program and to restore it at a later time.

(get-axioms '<filename>) and *(get-axiomsq <filename>)*

Retrieves the axioms and LISP predicates that have been saved in *<filename>* by one of the save functions below. The names of the predicates defined by this retrieval are put in a list named (concat <filename> 'fns). Thus (*get-axioms xxx*) reads in the predicates in file **xxx**, and sets the variable **xxx**fns to the names of the predicates that were restored from **xxx**.

(save-predicates '<filename> '<list of prednames>)

Saves the axioms and comments for the predicates given in the specified file. LISP predicates declared to HORNE may also be saved. The output is in a pretty format (with "?" for variables). Hashtable info is saved so they can be reconstructed when retrieved.

(save-horne '<filename>)

Does a save-predicates on all the predicates known to the system.

(save-indices '<filename> '<list of indices>)

Saves all axioms with one of the specified indices on the specified file. The output is in pretty format, but no comments are saved. No hashtable info is saved.

(dump-predicates '<filename> '<list of prednames>)

This saves the definitions of the predicates specified in the file in an internal format. Thus reading in the file is considerably faster, but the file is not for human consumption. If the second argument is omitted, all the known predicates are dumped. *Dump-predicates* always saves all the type information even if only a subset of the defined predicates are dumped. Dumped files are compilable by the LISP compiler, whose output can then be loaded into HORNE.

(dump-horne '<filename>)

Dumps entire database of axioms into the specified file.

7. TYPED THEOREM PROVING

The type of a variable is indicated by appending a suffix to the variable indicating its type. Thus *?x*CAT* names a variable *?x* that is of type CAT. The variable *?x*CAT* will unify only with terms that are compatible with the type CAT. The internal format for typed variables is the list *(* # <name> . <type>)* as in *(* 3 ?x . CAT)*.

Types should be viewed as sets, and no restrictions are assumed as to whether sets are disjoint, mutually exclusive, or wholly contained by each other. This information is specified by the user with assertions of the forms:

(ITYPE <individual> <typename>)

Asserts that the individual is of the indicated type, e.g., (TYPE A CAT) asserts that the constant A is of type CAT.

(ISUBTYPE <subtype> <supertype>)

Asserts that the first type is a subclass of the second type, e.g., (SUBTYPE CAT ANIMAL) asserts that CAT is a subclass of ANIMAL.

(DISJOINT <type1> <type2> ... <type n>)

Asserts that all the types mentioned are pairwise disjoint.

(INTERSECTION <newtype> <type1> <type2>)

Asserts that the intersection of type1 and type2 is newtype.

(XSUBTYPE (<type1> <type2> ... <type n>) <super-type>)

Asserts that type1 ... type n is a partition of super-type, i.e., they are all subtypes of super-type, that type1 ... type n are pairwise disjoint, and that the union of type1 ... type n is equivalent to super-type.

7.1 Adding TYPE Axioms

These statements are added to HORNE in the form of axioms by using the regular axiom addition functions *adda*, *addz*, *axioms*, etc. However, two things occur when axioms of these forms are added:

1) The relation between the types named and its implications are added to a matrix which stores the known set relationship between all the types known to the system. Of course what is implied by any statement depends on what is already in the matrix.
2) The statement is added to the axiom list so they can be printed out and edited as normal axioms.

The system that adds a TYPE axiom and its implications to the matrix first checks that the statement is consistent. If the statement contains an inconsistency, an error message is printed and no information is added to the

matrix. For example, if one adds (DISJOINT cats dogs) and then adds (SUBTYPE dogs cats), an error message will be given and information in the second axiom will not be added to the matrix.

In order for the matrix system to derive all implied information, ITYPE axioms should be added after SUBTYPE, XSUBTYPE, DISJOINT, and INTERSECTION axioms. Adding an ITYPE axiom may add or delete other ITYPE axioms implied by the axiom. (In fact, sometimes the axiom that was written might not even be added.) Because of this and the nature of axiom addition, axioms for the predicate ITYPE are always added at the end of the axiom list for ITYPE (e.g., as with using *addz*). This restriction has no effect on the proof procedure, for the order of the atomic ITYPE axioms is irrelevant. *Edita* can be used to reorder the axioms for documentation purposes.

Type restrictions on the arguments to a function term, and on the type of the function term itself, are declared using the form:

(declare-fn-type '<fn-name> (<type1> ... <typen>) '<typename>)
(declare-fn-typeq <fn-name> (<type1> ... <typen>) <typename>)

Asserts that <fn-name> is the name of a function that takes arguments of the types <type1>,...,<typen> and describes objects of type <typename>. For example, (declare-fn-type ADD (NUMBER NUMBER) NUMBER) declares a two-place function ADD, with both arguments of type NUMBER, and which produces an object of type NUMBER.

Single place functions may have multiple declarations subject to strict conditions outlined below:

1) the first declaration is the most general in its argument place and its value;
2) all subsequent declarations define a proper subset of the first definition in both the argument type and the value type;
3) the type of the argument is the most general type that produces values of the specified value type.

Examples and further discussion are found in the system overview, Section 3.2.

Declare-fn-type returns one of three values to indicate the status of the call:

t --	a new definition of a type (or exact repeat of a previous definition)
:compatible --	an additional definition to a single argument function that is compatible with all previous definitions
nil --	improper form of definition or a definition inconsistent with previous definitions

(delete-fn-definition '<function name>)

Removes all previous definitions for the function.

7.2 Deleting TYPE Axioms

In order to delete an axiom about types, one can use one of the HORNE deletion functions (retracta, retractz, edita, retractall, etc.). However, at this point, the prover is disabled. This is because the axiom lists are correct but the matrix has not been changed. In order to restore the matrix and enable the prover to run, use the function:

(recompile-matrix)

Recompiles all the type axioms in the system.

This is an expensive process and should be avoided if possible.

7.3 LISP Interface to Type System

There is a set of LISP functions to access and use the type system independently of HORNE. The most important function returns the type of an arbitrary HORNE term:

(get-type-object '<term>)

Given any HORNE term, this function returns the most specific type of that term. If the term contains one or more variables, it returns the most specific type that includes every instantiation of the term.

(issub '<type1> '<type2>)

Takes any two types and returns t if the types are identical, or if <type1> is a proper subtype of <type2>.

There are functions for inspecting the definitions of function terms (in addition to get-type-object above).

(see-function-definition '<function name>)

Returns the complete type table for the specified function. For single argument function, this may be a tree of the form

(<function type> (<arg type list>) <subtree>*).

For example, the function SPOUSE might have the definition

(PERSON (PERSON) (FEMALE (MALE)) (MALE (FEMALE)))

i.e., SPOUSE of a PERSON is of type PERSON, and SPOUSE of MALE is of type FEMALE, and SPOUSE of FEMALE is of type MALE.

(defined-functions)

Returns a list of all function names that have been declared.

One can examine the TYPE axioms added to the system by using the HORNE functions *printp*, *printi*, etc., but these functions will only show you the base

facts and not all the inferences the system has made. The following functions allow examination of what is in the matrix.

(matrix-relation '<type1 > '<type2 >)

Returns the information that is stored in the matrix for the relationship between the two types.

(type-info '<type >)

Returns a list giving the relationship between the given type and every other type in the system, of the form: ((type rel type1)(type2 rel type) ...) The type you are querying can be in either the first or second slot.

The following are the possible relationships between types:

1) "sb" -- a subset relation holds between the two types.
2) "ss" -- a superset relation holds.
3) "o" -- the types intersect but the overlap is not named.
4) "(ip (list))" or "(p (list))" -- a superset partition relationship holds; the list contains all the partitioning sets of the superset.
5) a list of length 1 -- the item on the list is the name of the intersection of the given types.

(types)

Returns a list of all types known in the system.

7.4 Type Compatibility and an Example

Using the axioms above, HORNE can compute the compatibility of two terms efficiently. Types are *compatible if one is a subtype of the other or if they* overlap. Overlaps occur in two ways: named or unnamed. A named overlap results from an INTERSECTION axiom; an unnamed overlap can be implied from either TYPE axioms or a named overlap. The unification of two typed variables may result in a variable of a complex type of the form *(int type1 type2)* indicating the intersection of the two types. This new type is recognized in the proof as a new type. For example, suppose we have the axioms:

```
(ISUBTYPE cars anything)
(ISUBTYPE person anything)
(ISUBTYPE ford cars)
(ISUBTYPE smallcars cars)
(ISUBTYPE student person)
(ISUBTYPE worker person)
(ITYPE john worker)
(ITYPE john student)    ; note this implies that the types worker and
(INTERSECTION pintos ford smallcars)                student overlap
((want ?x*person ?g*ford) <(fuel-efficient ?g*ford)
                           (wealthy ?x*person))
((fuel-efficient ?f*smallcars) <)
((wealthy ?d*worker) <)
```

We could then query (want ?f*student ?d*ford) and we would get (want ?r*(int student worker) ?u*pintos), pintos being a named overlap while the intersection of the types *student* and *worker* is derived by the prover.

7.5 Tracing Typechecking

In order to trace the typechecking functions, call the function *(trace-typechecking)*. The prover will break during typechecking if this function is called with the form *(trace-typechecking break)*. In order to stop tracing, call *(untrace-typechecking)*.

7.6 Assumption Mode

The default mode for HORNE is to assume that two types whose relationship is not known are not compatible. This can be overridden by the command *(type-assumption-mode)*, in which all unknown relationships are assumed to be unnamed intersections. Alternatively, the mode *(type-query-mode)* will query the user each time two types are found for which there is no known relationship. The function *(normal-type-mode)* returns the system to default mode.

In assumption mode, the format of answers is

((<answer> <type assumptions>)).

For example, given (Q ?x*CAT) and proving (Q ?x*DOG) in assumption mode where no relationship is known between the types CAT and DOG, we get:

(((Q ?x*(int CAT DOG))) ((int CAT DOG)))

Note that if you obtain multiple answers in this mode, the list of assumptions for each answer may refer to assumptions needed for other answers as well.

7.7 Defining a Custom Typechecker

If users wish to design their own type checking facility, the interface between the unifier and the type checking system consists of two LISP functions that can be rewritten. These are:

(typecheck <term> <type>)

Returns *t* if and only if the term is of the appropriate type (or a subtype);

(typecompat <type1> <type2>)

Returns the more specific type. For example,

(typecompat GIRL PEOPLE) returns *GIRL*,
(typecompat GIRL BOY) returns *nil*.

8. EXTENSIONS TO THE UNIFICATION ALGORITHM

The unifier in HORNE has been augmented to allow two types of special unification dealing with equality and restricted variables.

8.1 Equality

The unification algorithm of HORNE has been modified so that when terms do not unify they can be matched by proving that the terms are equal. Any variables in the terms matched will be bound as needed to establish the equality. Equality statements are added to the system by using the axiom EQ. (Note that EQ is of arity 2.) For example:

((EQ (president USA) Ronald-Reagan) <)

expresses a fact that is well known to most Americans. The axiom

(EQ (add-zero 1) 1) <)

expresses an infinite class of equalities. For example, (add-zero (add-zero 1)) equals 1, as does (add-zero (add-zero (add-zero 1))), and so on.

The system provides, in an efficient manner, complete reasoning about fully grounded terms (i.e., terms that contain no variables), and supports partial reasoning about equality assertions containing variables. The current system will allow variables in queries (which may be bound to establish equalities), but variables in equality assertions are restricted in their use. In particular, there is no transitivity reasoning for terms containing variables; e.g., given

(EQ (f ?x) ?x)
(EQ (G ?y) (f ?y))

we can prove (EQ (f A) A), (EQ (f ?z) ?z), and (EQ (G (f ?t)) (f ?t)), but cannot prove (EQ (G A) A), even though it is a logical consequence of the two axioms above.

The information derived from the EQ axioms that are asserted is stored on a pre-computed table which is updated as EQ axioms are added and deleted.

There are two LISP functions for examining the equality assertions:

(equivclass '<ground term>)

Returns a list of all ground terms equal to the <ground term>.

(equivclass-v '<term>)

Returns a list of all terms that could be equal to the term followed by variable binding information.

8.2 The Post-Constraint Mechanism

HORNE allows the user to specify that the proof of an atomic formula be delayed until the terms in it are completely bound. The user does this by enclosing the atomic formula within the lispfn POST, as in the axiom:

((F ?x) < (POST (MEMBER ?x (a very very long list))) (G ?x)).

POST takes an atomic formula as an argument. If the formula is grounded then the proof proceeds as usual. Otherwise the variables in the formula are bound to a function which restricts its value and the proof proceeds as though the proof of the formula succeeded.

Restrictions on variables are implemented by binding the variable to a special form

(any ?newvar (constraint ?newvar)).

Thus, give the above axiom, if we queried (F ?s), the POST mechanism would bind ?s to

(any ?s0001 (MEMBER ?s0001 (a very very long list))).

This use of a special form *any* is similar to the *omega* form used in Kornfeld (1983).

The HORNE unifier has been modified so that it knows about *any*. A term of form (any ?x (R ?x)) will unify with any term that satisfies the constraint (R ?x). Again using the above axiom: after the POST succeeds, the proof continues with the subgoal

(G (any ?s0001 (MEMBER ?s0001 (a very ....].

Now suppose that (G e) is true. Then we can unify these two literals if we can prove

(MEMBER e (a very very long list)).

Note that the constraint will be queried only once its variable is bound. Thus if (G ?c) were true above, the unification would succeed and

(F (any ?s0001 (MEMBER ?s0001 (a very long list))))

would be returned as the result of the proof. If (G (fn ?c)) were true instead, a recursive proof testing whether (MEMBER (fn ?c) (a very very long list)) would be done and, if successful, the final result of the proof would be

(F (fn (any ?z (MEMBER (fn ?z) (a very very long list)))).

During normal tracing, any subproofs due to the post constraint mechanism are not traced. If tracing is desired for these proofs, call *(htrace-post-proof)*. To set it back to the default of no tracing, call *(unhtrace-post-proof)*.

8.3 Interaction Between Systems

The equality system and the POST mechanism use each other as can be shown by the following example.

(EQ (child-of Adam) Abel)
(EQ (child-of Eve) Abel)

Then we can unify (child-of ?x) with Abel, resulting in ?x being bound to

(any ?x0001 (MEMBER ?x0001 (Adam, Eve))).

Thus we have restricted the values that ?x can take on to Adam or Eve. It should be noted that MEMBER must take equality into account; that is, in the example, the *any* term should unify with the term (First-man) given (EQ (First-man Adam)).

9. THE FORWARD CHAINING FACILITY

The prover has a forward production system in which the addition of new axioms adds new facts that are implied by the existing axioms. The general form of forward axioms are as follows:

((trigger) (list of conclusions) index (list of conditions)).

After a HORNE axiom is added to the database it is checked to see if it matches any trigger pattern. A trigger must be an atomic formula, but cannot be a LISP predicate. If it matches, then using the binding list of the match the system tries to show that the conditions associated with the trigger are in the database. Note that the system does not try to prove the conditions (unless specified), but simply checks that they are in the database. If all the conditions can be shown to be in the database then each of the conclusions in the conclusion list is added to the HORNE axiom list using the bindings collected in the process. LISP predicates can be used in the conditions and in the conclusions, where they are called as in the backwards chaining system. The value returned by a LISP predicate in the conclusion list is ignored. In adding a conclusion another trigger may be fired. To prevent infinite looping the forward chaining system will not add axioms that are already in the database.

9.1 Defining Forward Production Axioms

(addf '<atomic formula> ['<atomic formula> ...) '<index>
['<atomic formula> ...))
(addfq <atomic formula> (<atomic formula> ...) <index>
(<atomic formula> ...))

Adds the forward production axiom to the end of the data base, e.g., adding the following

```
(addf '(e ?d) '((w ?d)) 'r '((r ?d))))
(addaq ((r d) s))
(addaq ((e ?f) j))
```

will result in the axiom ((w d) r) being added to the database.

9.1.1 Options to addf and addfq

(addf :all ['<atomic formula> ...) '<index> ['<atomic formula> ...))
(addfq :all (<atomic formula) ...) <index> (<atomic formula> ...))

Using the atom "all" for the trigger adds a separate forward-chaining axiom for each of the atomic formulas in the condition list with that condition as the trigger. Thus each of the conditions is a trigger, e.g.,

```
(addf :all '((eq ?y ?z)) '<1 '((eq ?y ?x) (eq ?x ?z)))
```

adds the following forward chaining axioms to the system:

```
1. (eq ?y ?x) ((eq ?y ?z)) <1 ((eq ?y ?x) (eq ?x ?z))
2. (eq ?x ?z) ((eq ?y ?z)) <1 ((eq ?y ?x) (eq ?x ?z))
```

Given these, the following addition:

```
(addaq ((eq w e) 1)
(addaq ((eq r w) 1)
```

causes the axiom ((eq r e) <1) to be added to the system.

(addf '<atomic formula> (<atomic formula> ...) '<index> ())
(addfq <atomic formula> (<atomic formula> ...) <index> ())

Using "()" for the conditions list makes it such that whenever the axiom is triggered it will assert its conclusions.

(addf '<atomic formula> (<atomic formula> ...) '<index>
(<atomic formula>) ...)
(addfq <atomic formula> (<atomic formula> ...) <index>
(<atomic formula>) ...)

This option allows a lispfn to occupy the position of the predicate name in any of the conditions. The lispfn succeeds if it returns a non nil value.

(addf '<atomic formula> (<atomic formula> ...) '<index>
((:prove <atomic formula>) ...))
(addfq <atomic formula> (<atomic formula> ...) <index>
((:prove <atomic formula>) ...))

The prove option allows any of the conditions to call the theorem prover to prove the condition. (Note that normally conditions are not proved but just shown to be in the data base). The condition is true if the atomic formula can proved by the theorem prover. Any variables bound in the proof will be passed on to the next condition.

(retract-forward 'form) and *(retract-forwardq form)*

These delete the forward-chaining axioms specified by the given form, which is either a pattern or a predicate name. If the form is a predicate name, all forward-chaining axioms that have the given predicate name as their trigger name are deleted. Otherwise all forward-chaining axioms whose trigger unifies with the given pattern are deleted. Note that if the form is a pattern the car of the pattern must be an atom.

The system does not perform truth maintenance; i.e., axioms entered into the data base due to a forward-chaining axiom are not removed when the axiom is removed.

9.2 Examining Forward Production Axioms

(printf 'form) and *(printfq form)*

These functions pretty print all axioms whose triggers are specified by the form argument, which can be either a predicate name or a pattern. If it is a predicate name, all forward-chaining axioms with the given trigger name will be printed. Otherwise all forward-chaining axioms whose

trigger matches with the given pattern will be pretty printed. Note that if the form is a pattern the car of the pattern must be an atom.

(printc 'form) and *(printcq form)*

These functions pretty print all axioms whose conclusions are specified in the form argument. The form argument can be either a predicate name or a pattern. If it is a predicate name then all forward-chaining axioms that have as a member of thier conlusion list an atomic formula with the given predicate name will be pretty printed. Otherwise all forward-chaining axioms who have a member of their conclusion list that unifies with the given pattern will be pretty printed.

(triggers)

Returns a list of all the predicate names which are trigger names for forward-chaining axioms.

9.3 Tracing Forward Chaining

Because the forward-chaining mechanism is defined in HORNE, the standard tracing functions (e.g., htraceall) are useable for debugging forward-chaining axioms. In addition, the following trace facilities are provided.

(trace-assertions)

This causes the system to print out all axioms that are asserted by the forward chaining system. The system default is that this tracing is on.

(untrace-assertions)

Stops the tracing of assertions made by the forward chaining system.

(trace-forward)

Causes the system to print out the trigger and rule of any forward-chaining axiom that has been triggered.

(untrace-forward)

Undoes the effects of "trace-forward."

9.4 I/O

I/O for forward production rules are handled by the I/O functions documented in Section 6 (Saving and Restoring Programs). An exception is the function "save-indices," which cannot be used to save forward chaining rules.

9.5 Editing Forward Chaining Axioms

(editf '<predicate name>)

The above call will get you into an interactive editor for forward-chaining axioms. The actual editor is the same as the regular axiom editor described in Section 3.

9.6 Examples

The first example shows the use of forward chaining for a simple equality system. The rules capture the transitivity and symmetric properties of equality. The rules are:

```
(addf :all '((MYEQ ?y ?z)) 'p '((MYEQ ?y ?x) (MYEQ ?x ?z)))
(addf '(MYEQ ?s ?d) '((MYEQ ?d ?s)) 'p '())
```

If we now add

```
(addaq ((MYEQ w e) k))
```

the following axioms are also asserted by the system:

```
((MYEQ e w) p)
((MYEQ w w) p)
((MYEQ e e) p)
```

If we now add

```
(addaq ((MYEQ r e) k))
```

then the following are also asserted:

```
((MYEQ e r) p)
((MYEQ r w) p)
((MYEQ w r) p)
```

The second example involves forward chaining rules that are used to maintain consistency in a data base for a simple blocks world. Here the chaining rules call LISP functions to delete axioms.

```
(addf '(pickup ?d)  '((holding ?d)
                      (RETRACT (ontable ?d))
                      (RETRACT (clear ?d))
                      (RETRACT (handempty)))

                    'index

                    '((ontable ?d)
                      (clear ?d)
                      (handempty)))

(addaq ((ontable block1) k)
       ((clear block1) k)
       ((handempty) k))
```

If we now add

```
(addaq ((pickup block1) k))
```

then the axiom ((holding block1) index) becomes true and the predicates (ontable block1) (clear block1) and (handempty) are deleted from the data base.

10. BUILT-IN PREDICATES

This section documents the built-in predicates that are already defined in HORNE.

(ASSERT-AXIOMS <list of axioms>)

Adds the specified axioms to the data base at the end of the axiom list for the specified predicate. Thus, this performs a similar function to addz but is callable from HORNE and returns t. All logic variables in the new axioms that are bound in the current environment will be replaced by their values before the new axioms are added.

(ATOM? <term>)

Succeeds if *<term>* is an atom.

(BOUND ?x)

Succeeds only if *?x* is not a variable. It succeeds on any other non-grounded term. For example, *(bound (f ?x))* succeeds. Equivalent to but faster than *(UNLESS (VAR ?x))*.

(DISTINCT <term$_1$> <term$_2$>)

Succeeds if both terms are fully grounded, but to different atoms. If a term is not fully grounded, this posts a constraint on the variable(s) and succeeds.

(EQ <term$_1$> <term$_2$>)

Succeeds if <term$_1$> equals <term$_2$> (i.e., they unify) (see Section 8.1).

(FAIL)

This predicate is always false.

(FIND-FACTS <atomic formula>)

Same as the LISP function find-facts in Section 2.2.

(GENVALUE <variable> <LISP expression>)

Sets the HORNE variable <variable> to first value in list returned by evaluating the <LISP expression>. Other values are used for backtracking (see Section 5.1).

(GROUND <term$_1$>)

Succeeds if term$_1$ is a fully grounded term, i.e., it contains no variables.

(IDENTICAL <term$_1$> <term$_2$>)

Succeeds if <term$_1$> and <term$_2$> are structurally identical, i.e., if they unify without assignment of variables or the equality mechanism. For example, (IDENTICAL A A) succeeds, and (IDENTICAL A ?x) fails.

(MEMBER <term$_1$> <list>)

Succeeds if <term$_1$> is equal (i.e., HORNE equality) to a term in the list.

(NOTEQ <term$_1$> <term$_2$>)

Succeeds if both <term$_1$> and <term$_2$> are fully grounded, but to different values. Otherwise it fails.

(RETRACT <term$_1$>)

Retracts all axioms whose head unifies with <term$_1$>.

(RPRINT <term$_1$> ... <term$_n$>)

The values of <term$_1$> through <term$_n$> are printed on successive lines.

(RTERPRI)

Prints a line feed.

(SETVALUE <variable> <LISP expression>)

Sets the HORNE variable *<variable>* to the value of the LISP expression *<LISP expression>*. Any logic variables in <LISP expression> are replaced by their logic bindings before LISP evaluation (see Section 5.1).

(UNLESS <atomic formula>)

Succeeds only if the call *(proveq <atomic formula>)* fails. This gives us proof by failure. Note that variables change in interpretation in the UNLESS function, e.g., if we are given the fact that (P A) is true, then

(UNLESS (P B)) will succeed,
(UNLESS (P A)) will fail as expected.

But (UNLESS (P ?x)) also fails, since (P ?x) can be proven.

(VAR <variable>)

Succeeds only if *<variable>* is an unbound variable.

CUT

The cut symbol. It has no effect until HORNE tries to backtrack past it, and then the prover immediately fails on the subproblem it was working on. An alternate definition: cut always succeeds, and when executed, removes all choice points in the proof from the point at which the predicate which appears in the head of the axiom containing the cut was selected to the current point of the proof.

11. HASHING

A hashtable can be declared for a predicate name whether it currently has axioms asserted for it, or will have axioms asserted later. It can also be used to redefine an already existing hashtable for the predicate. The hashtable allows the axioms for a predicate to be stored according to the values of the arguments to the predicate. They can currently only be used on argument positions that do not allow equality reasoning. For example, consider a one-place predicate P with hashing on its argument into three buckets. If we have asserted the facts (P A), (P B), (P C), (P D), (P (f A)) and (P (g ?x)), the hashed structure might look like the following (ignoring efficiency encodings):

```
                bucket 1 → (P A)
                bucket 2 →(P B), (P D)
P               bucket 3 → (P C)
                function bucket → (P (f A)), (P (g ?x))
                variable bucket → (P A), (P B), (P C), (P D), (P (f A)), (P (g ?x))
```

Now if we query (P A), we would hash on A to bucket 1 and just unify (P A) with those axioms there, i.e., only (P A). Similarly, for (P E), if hashing on E gives bucket 3, then (P E) would be unified only with (P C). Any complex argument, such as (P (g B)), will be checked against the special *function bucket*, i.e., (P (f A)) and (P (g ?x)). Finally, any query with a variable, e.g., (P ?y), will be matched against the *variable bucket* which contains the complete axiom list.

As one can see, if equalities were allowed on terms in the argument position, this structure might fail. For example, given B = F, if we query (P F), and hashing on F gives bucket 1, then (P F) will be checked only against (P A) and would fail.

Hash tables are defined as follows:

(define-hashtable <predicate name>)

For forward chaining axioms, the trigger can be hashed using the function

(define-hashed-trigger <predicate name>).

For both of these uses, the system then prompts for paths through a formula to where the hashing should take place, and for the size of the buckets for each hash. The simple options for paths are as follows:

<number>

Hash on n^{th} argument to predicate.

(i <number>)

Hash on first atom found by successively taking CARs on the n^{th} argument to predicate.

Arbitrary paths may be built by specifying a sequence of CARs and CDRs starting from the predicate name. Thus the path *(CAR CDR)* is equivalent to the first argument. The path *(CAR)* would give the predicate name. The only other possibility in a path is to specify an arbitrary number of CARs, specified as CAR* in the path. Thus entering *(CAR* CDR CDR)* is equivalent to *(i 2)*.

The minimum number of buckets in a hashtable is 3: one for variables, one for lists (i.e., functions), and one for atoms. The number of buckets for atoms is the only size under programmer control. Thus, entering a 5 when prompted will produce 5 buckets for atoms.

A sample session that hashes a predicate MYPRED on the form of its second argument (into 10 buckets), and on some other arbitrary position in the third argument (into five buckets) follows:

→ *(define-hashtable MYPRED)*
Enter path spec: *2*
Hashtable size? ("q" to respecify path) *10*
Enter path spec: *(CAR* CAR CDR CAR CDR CDR)*
Hashtable size? ("q" to respecify path) *5*
Enter path spec: *q*
Hashtable defined.

The hashing facility can be set up directly from a LISP function, without the user interaction, using the following functions:

(H-setup-hashtable <name> <path> <#buckets> <size>)

where <name> is the predicate name to hash on, <path> specifies the argument to hash on, <#buckets> is the number of buckets to use, and <size> is the expected number of entries to be made for the predicate.

(H-setup-hashed-trigger <name> <path> <#buckets> <size>)

Defined the same as *H-setup-hashtable* except that it is used for the forward chaining axioms.

12. CONTROLS ON HORNE

The following global variables affect the behavior of HORNE:

H$$LIMIT

The number of steps HORNE can take before asking the user whether it should continue. Default value is 500. To continue, simply enter *y*, to terminate enter *n*. You can enter debug mode by entering *d*, after which typing *go* gets you back to the question whether to continue.

H$$PARTITION$CHK

The mechanism that adds information to the TYPE matrix does extensive consistency checking involving XSUBTYPEs. If no XSUBTYPE axioms are present the consistency testing is wasted. If this flag is set to nil then the testing is turned off. Default value is "t".

The following functions also control the behavior of HORNE:

(warnings)

Enables the printing of warning messages at the user's terminal. By default, warning messages are printed.

(nowarnings)

Disables the printing of warning messages. By default, warning messages are printed.

13. EXAMPLES

13.1 A Simple Example

The following is a simple session with HORNE:

```
*(addzq ((HAPPY ?person ?item) <
            (DESIRABLE ?item)
            (CAN-AFFORD ?person ?item))
        ; you can afford items if you have money
        ((CAN-AFFORD ?person ?item) <
            (HAS-MONEY ?person))
        ; but love is for free
        ((CAN-AFFORD ?person Sweetheart) <)
        ((DESIRABLE Newsuit) <)
        ((DESIRABLE Caviar) <)
        ((DESIRABLE Sweetheart) <)
        ((HAS-MONEY Sam))  )

*(htraceall)
; prove JOHN can be happy even if he has no money
*(proveq (HAPPY JOHN ?why))

(q-1) (HAPPY JOHN ?why)
      (q-2) (DESIRABLE ?why)
      (r-2) (DESIRABLE Newsuit)
      (q-2) (CAN-AFFORD JOHN Newsuit)
            (q-3) (HAS-MONEY JOHN)
            ; note, backtracking to (q-2) (DESIRABLE ?why)
      (r-2) (DESIRABLE Caviar)
      (q-2) (CAN-AFFORD JOHN Caviar)
            (q-3) (HAS-MONEY JOHN)
            ; backtracking again to (q-2) (DESIRABLE ?why)
      (r-2) (DESIRABLE Sweetheart)
      (q-2)(CAN-AFFORD JOHN Sweetheart)
      (r-2) (CAN-AFFORD JOHN Sweetheart)
(r-1) (HAPPY JOHN Sweetheart)
; end of trace, the value returned is:
((HAPPY JOHN Sweetheart))
```

13.2 The Same Example with Posting

```
*(addzq ((HAPPY ?person ?item) <
                (POST (DESIRABLE ?item))
                (CAN-AFFORD ?person ?item))
          ((CAN-AFFORD ?person ?item) <
                (HAS-MONEY ?person))
          ((CAN-AFFORD ?person Sweetheart))
          ((DESIRABLE Newsuit) <)
          ((DESIRABLE Caviar) <)
          ((DESIRABLE Sweetheart) <)
          ((HAS-MONEY Sam))   )

*(htraceall)

*(proveq (HAPPY JOHN ?why))

(q-1) (HAPPY JOHN ?why)
      (q-2) (POST (DESIRABLE ?why))
      (r-2) (POST (DESIRABLE (any ?why6 ((DESIRABLE ?why6)))))
      (q-2) (CAN-AFFORD JOHN (any ?why6 ((DESIRABLE ?why6)))))
            (q-3) (HAS-MONEY JOHN)
            ; in trying the second axiom for CAN-AFFORD, we must
            prove (DESIRABLE Sweetheart) to unify Sweetheart
            with (any ?why6 ...)
      (r-2) (CAN-AFFORD JOHN Sweetheart)
(r-1) (HAPPY JOHN Sweetheart)
((HAPPY JOHN Sweetheart))
```

The only difference between this proof and the proof in 13.1 is when the predicate DESIRABLE is proved. In the first, we would backtrack through all values until one was found that succeeded. In the second, the rest of the proof is done first, and then when a value for ?why is found, it is checked to see if we can prove it is DESIRABLE.

13.3 An Example Using Types

This example uses a type hierarchy with two types, PROFESSOR and MUSICIAN, that intersect with the subtype MUSICAL-PROFESSOR.

; The type hierarchy

```
(addzq ((ISUBTYPE PROFESSOR PEOPLE))
       ((ISUBTYPE MUSICIAN PEOPLE))
       ((INTERSECTION  MUSICAL-PROFESSOR
                       PROFESSOR
                       MUSICIAN)))
```

; The axioms:

all professors teach, and all musicians sing
someone is happy if they teach and sing

```
(addzq ((TEACH ?p*PROFESSOR))
       ((SING ?m*MUSICIAN))
       ((HAPPY ?p) < (TEACH ?p) (SING ?p)))
```

; Here we could add hundreds of professors and musicians, and a few musical-professors.

```
(addzq ((ITYPE JACK MUSICAL-PROFESSOR)))
```

Now we can prove the following:

Is Jack Happy? yes.

```
(proveq (HAPPY JACK))

(q-1)  (HAPPY JACK)
       (q-2) (TEACH JACK)
       (r-2) (TEACH JACK)
       (q-2) (SING JACK)
       (r-2) (SING JACK)
(r-1)  (HAPPY JACK)
```

Who is happy? All musical professors.

```
(q-1)  (HAPPY ?x)
       (q-2) (TEACH ?x)
       (r-2) (TEACH ?y*PROFESSOR)
       (q-2) (SING ?y*PROFESSOR)
       (r-2) (SING ?z*MUSICAL-PROFESSOR)
(r-1)  (HAPPY ?z*MUSICAL-PROFESSOR)

((HAPPY ?z*MUSICAL-PROFESSOR))
```

14. THE REP SYSTEM

The REP system supports reasoning about structured types (see Section 3.4 of the introduction). The following naming conventions, while not necessary, are used to distinguish the different kinds of objects:

T- ... -- a type name

R- ... -- a rolename

f- ... -- the function named by a rolename

c- ... -- a constructor function

To define a subtype with roles, there are two options, depending on whether the objects of the new type are fully determined by the set of roles defined. Both of these enforce the restriction that the new type must be a subtype of an existing type. A type T-U is predefined as the root of the type hierarchy. The following sections describe how to define roles on the type hierarchy and how to retrieve role information about objects.

14.1 Defining Roles in the Type Hierarchy

(define-subtype '<type> '<supertype> (<rolename type>))*
(define-subtypeq <type> <supertype> (<rolename type>))*

Defines *<type>* as a subtype of *<supertype>* and defines the indicated type restricted roles for the new type. In addition, *<type>* inherits any roles from *<supertype>*. An inherited role may be redefined only if its new type restriction is a subtype of the inherited type restriction, e.g.,

(define-subtype 'T-ACTION 'T-U '(R-ACTOR T-ANIM))

defines T-ACTION to be a subtype of T-U, with a role R-ACTOR defined and restricted to be of type T-ANIM. This is roughly equivalent to adding:

(ISUBTYPE T-ACTION T-U)

and defining f-actor by

(declare-fn-typeq f-actor (T-ACTION) T-ANIM)

In addition, define-subtype sets up some internal data structures to maintain the role inheritance in an efficient manner.

(define-functional-subtype '<type> '<supertype> (<rolename type>))*
(define-functional-subtypeq <type> <supertype> (<rolename type>))*

This defines *<type>* in the same manner as the define-subtype function, but in addition defines a constructor function for the type. Thus, given the definition of T-ACTION above,

(define-functional-subtype 'T-EAT 'T-ACTION '(R-OBJ T-FOOD))

would define T-EAT to be a subtype of T-ACTION with roles R-OBJ and R-ACTOR (inherited), and would define the function f-obj for the R-OBJ role and a constructor function c-eat. This is roughly equivalent to adding:

(FUNCTIONAL T-EAT)

(ISUBTYPE T-EAT T-ACTION)

where the functions are defined by

(declare-fn-typeq f-obj (T-EAT) T-FOOD)
(declare-fn-typeq c-eat (T-FOOD T-ANIM) T-EAT)

The definition of f-actor for T-ACTION will apply as needed to instances of T-EAT.

The REP system provides a convenient abbreviated form for defining instances of structured types.

(define-instance '<instance> '<type> (<rolename value>))*
(define-instanceq <instance> <type> (<rolename value>))*

This defines *<instance>* to be an ITYPE of *<type>* and defines the indicated roles of *<instance>* to have the indicated values. For example,

(define-instance 'E1 'T-EAT '(R-OBJ F1 R-ACTOR JOE))

is equivalent to adding

(ITYPE E1 T-EAT)
(ROLE E1 R-OBJ F1)
(ROLE E1 R-ACTOR JOE)

Either way of asserting this information will cause the following equalities to be derived:

(EQ (c-eat F1 JOE) E1)
(EQ (f-obj E1) F1)
(EQ (f-actor E1) JOE)

14.2 Retrieving in the REP System

The REP system provides a general facility for providing information about any object defined. This is provided by the function

(retrieve-def '<object>)

which returns a description of the object in the following formats:

If the *<object>* is a type, it returns a list of the form

(TYPENAME <list of immediate supertypes>
<list of roles defined>
<type restrictions on roles>)

For example, given the definition of T-OBJ-ACTION above, retrieve-def would return

(TYPENAME (T-ACTION) (R-OBJ R-ACTOR)
(T-PHYS-OBJ T-ANIM))

Given a rolename, retrieve-def returns

(ROLENAME <list of types using that role>)

Given a function name, retrieve-def returns

(FUNCTIONNAME (<type restrictions on args>) <type of result>)

Given a free variable, retrieve-def returns

(VARIABLE <type restriction>)

Given an object, retrieve-def returns

(CONSTANT <type> (<role> <value>)*)

For example, if A is an instance of T-OBJ OBJ-ACTION with the R-OBJ role set to O1 and R-ACTOR set to (f-actor A2), then (retrieve-def 'A) would return

(CONSTANT T-OBJ-ACTION (R-OBJ O1) (R-ACTOR (f-actor A2)))

Finally, given a function containing unbound variables, retrieve-def will return as much information as it can derive using the basic format for constants, but differing in the first atom, i.e., it returns

(FUNCTION <type> (<role> <value>)*)

For example, given all the assertions in Section 14.1, we would retrieve the following:

```
> (retrieve-def 'T-EAT)
    (TYPENAME (T-ACTION) (R-OBJ R-ACTOR)
                         (T-FOOD T-ANIM))

> (retrieve-def 'R-OBJ)
    (ROLENAME (T-EAT))

> (retrieve-def 'R-ACTOR)
    (ROLENAME (T-EAT T-ACTION))

> (retrieve-def 'f-obj)
    (FUNCTIONNAME (T-EAT) T-FOOD)

> (retrieve-def 'c-eat)
    (FUNCTIONNAME (T-FOOD T-ANIM) T-EAT)

> (retrieve-def '?x*T-EAT)
    (VARIABLE T-EAT)

> (retrieve-def 'E1)
    (CONSTANT T-EAT (R-OBJ F1 R-ACTOR JOE))

> (retrieve-def '(f-obj E1))
    (CONSTANT T-FOOD)
```

```
> (retrieve-def '(c-eat (f-obj E1) JACK))
      (CONSTANT T-EAT (R-OBJ (f-obj E1) R-ACTOR JACK))

> (retrieve-def '(c-eat ?x*FOOD JACK))
      (FUNCTION T-EAT (R-OBJ ?x*FOOD R-ACTOR JACK))

> (define-instance 'E2 'T-EAT (R-ACTOR JACK))
      E2

> (retrieve-def 'E2)
      (CONSTANT T-EAT (R-OBJ (f-obj E2) R-ACTOR JACK))
```

14.3 Examples

```
;     A REP system transcript, slightly modified for readability.
;     Lines 1 to 13 define a type hierarchy and some instances.

1.  (DEFINE-SUBTYPEQ T-PHYS-OBJ T-U)

2.  (DEFINE-SUBTYPEQ T-LEGAL-PERSONS T-U)

3.  (DEFINE-SUBTYPEQ T-HUMANS T-LEGAL-PERSONS)

4.  (DEFINE-SUBTYPEQ T-COMPANIES T-LEGAL-PERSONS)

5.  (DEFINE-SUBTYPEQ T-RELATION T-U)

6.  (DEFINE-FUNCTIONAL-SUBTYPEQ T-BUILDS-RELATION
        (R-AGT T-LEGAL-PERSONS) (R-OBJ T-PHYS-OBJ))

7.  (DEFINE-SUBTYPEQ T-AUTOMOBILES T-PHYS-OBJ)

8.  (DEFINE-SUBTYPEQ T-MUSTANGS T-AUTOMOBILES)

9.  (DEFINE-SUBTYPEQ T-MODEL-TS T-AUTOMOBILES)

10. (DEFINE-INSTANCEQ I-GM T-COMPANIES)
        (((ITYPE I-GM T-COMPANIES)))

11. (DEFINE-INSTANCEQ I-FORD T-COMPANIES)
        (((ITYPE I-FORD T-COMPANIES)))

12. (DEFINE-INSTANCEQ I-OLD-BLACK T-MUSTANGS)
        (((ITYPE I-OLD-BLACK T-MUSTANGS)))

13. (DEFINE-INSTANCEQ I-LIZZY T-MODEL-TS)
        (((ITYPE I-LIZZY T-MODEL-TS)))

;     Now we add a fact that ford builds all mustangs using the "builds" relation
;        defined above in step 6
```

```
14. (addzq ((holds (c-builds i-ford ?m*t-mustangs))))

;   A trivial proof that ford builds "old black" (defined in line 12)

15. (proveq (holds (c-builds i-ford i-old-black)))
        ((HOLDS (C-BUILDS I-FORD I-OLD-BLACK)))

;   Here we explicitly build an instance of the relation that ford builds
;       old-black, using the define-instanceq function. The actual axioms
;       added to the system follow.

16. (define-instanceq i-b-o-b t-builds (r-agt i-ford) (r-obj i-old-black))
        (((ITYPE I-B-O-B T-BUILDS)))

17. (printiq i-b-o-b)
        ((ITYPE I-B-O-B T-BUILDS)      I-B-O-B)
        ((ROLE I-B-O-B R-AGT I-FORD)      I-B-O-B)
        ((ROLE I-B-O-B R-OBJ I-OLD-BLACK)      I-B-O-B)
        ((EQ I-FORD ( F-AGT I-B-O-B))      I-B-O-B)
        ((EQ I-OLD-BLACK ( F-OBJ I-B-O-B))      I-B-O-B)
        ((EQ I-B-O-B ( C-BUILDS ( F-AGT I-B-O-B) ( F-OBJ I-B-O-B)))    I-B-O-B)

;   Now we can prove that relation i-b-o-b also holds (i.e., it unifies with fact
;       added in step 14)

18. (proveq (holds i-b-o-b))
        ((HOLDS I-B-O-B))

;   Now we define a relation that ford builds lizzy (19), and then assert that
;       this relation holds (20):

19. (define-instanceq i-build-lizzy1 t-builds (r-agt i-ford) (r-obj i-lizzy))
        (((ITYPE I-BUILD-LIZZY1 T-BUILDS)))

20. (addzq ((holds i-build-lizzy1)))

;   Now we can find the company that builds lizzy using the constructor
;       function for T-BUILDS.

21. (proveq (holds (c-builds ?c*t-companies i-lizzy)))
        ((HOLDS (C-BUILDS I-FORD I-LIZZY))

;   Now we happen to define another build relation that turns out also to be
;       that Ford builds lizzy as well.

22. (define-instanceq i-build-lizzy2 t-builds (r-agt i-ford))
        (((ITYPE I-BUILD-LIZZY2 T-BUILDS)))

23. (addzq (ROLE i-build-lizzy2 R-OBJ i-lizzy))

;   This relation can then also be shown to be hold:
```

```
24. (proveq (holds i-build-lizzy2))
        ((HOLDS I-BUILD-LIZZY2))

;   Given this database, the following queries involving the ROLE predicate
;       can be made

;   find all relations involving I-LIZZY in any way

25. (prove :all '(role ?r*t-relation ?n i-lizzy))
        ((ROLE I-BUILD-LIZZY1 R-OBJ I-LIZZY)
            (ROLE I-BUILD-LIZZY2 R-OBJ I-LIZZY))

;   find all relations that involve OLD-BLACK in the R-OBJ role

26. (prove :all '(role ?r*t-relation r-obj i-old-black))
        ((ROLE I-B-O-B R-OBJ I-OLD-BLACK))

;   find all relations involving automobiles in any role

27. (prove :all '(role ?r*t-relation ?n ?a*t-automobiles))
        ((ROLE I-BUILD-LIZZY1 R-OBJ I-LIZZY)
            (ROLE I-BUILD-LIZZY2 R-OBJ I-LIZZY)
            (ROLE I-B-O-B R-OBJ I-OLD-BLACK))
```

INDEX OF FUNCTIONS

(define-instanceq <instance> <type> (<rolename value>))* -- Sect. 14.1

(define-subtype '<type> '<supertype> (<rolename type>))* -- Sect. 14.1

(define-subtypeq <type> <supertype> (<rolename type>))* -- Sect. 14.1

(delete-fn-definition '<function name>) -- Sect. 7.1

(DISJOINT $<type_1>$ $<type_2>$ ... $<type_n>$) -- Sect. 7

(DISTINCT $<term_1>$ $<term_2>$) -- Sect. 10

(dump-horne '<filename>) -- Sect. 6

(dump-predicates '<filename> '<list of prednames>) -- Sect. 6

(editf '<predname>) -- Sect. 9.5

(edita <predicate name>) -- Sect. 3

(EQ $<term_1>$ $<term_2>$) -- Sect. 10

(equivclass '<ground term>) -- Sect. 8.1

(equivclass-v '<term>) -- Sect. 8.1

(FAIL) -- Sect. 10

(find-clauses '<atomic formula>) -- Sect. 2.2

(find-facts '<atomic formula>) and *(find-factsq <atomic formula>)* -- Sect. 2.2

(find-facts-with-bindings '<atomic formula>) -- Sect. 2.2

(GENVALUE <variable> <LISP expression>) -- Sect. 5.1, Sect. 10

(get-answer) -- Sect. 5.4

(get-axioms '<filename>) and *(get-axiomsq <filename>)* -- Sect. 6

(get-binding '<varname>) -- Sect. 5.4

(get-clauses '<atomic formula>) -- Sect. 2.2

(get-facts '<atomic formula>) -- Sect. 2.2

(get-type-object '<term>) -- Sect. 7.3

(goal) -- Sect. 4.3

(GROUND $<term_1>$) -- Sect. 10

(H-setup-hashed-trigger <name> <path> <#buckets> <size>) -- Sect. 11

(H-setup-hashtable <name> <path> <#buckets> <size>) -- Sect. 11

(htrace '$<predspec_1>$ ... '$<predspec_n>$) -- Sect. 4.2

(htrace-post-proof) -- Sect. 8.2

REFERENCES

Bowen, K.A., "PROLOG," *Proc., ACM Annual Conference*, October 1979.

Kornfeld, W.A., "Equality for Prolog," *Proc., 8th IJCAI*, Karlsruhe, W. Germany, August 1983.

Kowalski, R.A., "Predicate logic as programming language," *Proc., IFIP 74*, Amsterdam, North Holland, 1974.

Kowalski, R.A. *Logic for Problem Solving*. New York: Elsevier-North Holland, 1979.

Steele, G.L. *COMMON LISP*. Digital Press, 1984.

Winston, P.H. and Horn, B.K.H. *LISP*. Reading, MA: Addison-Wesley, 1981.

Appendix A-3

THE LOGIC OF PERSISTENCE

Henry A. Kautz

Department of Computer Science
University of Rochester
Rochester, New York 14627

ABSTRACT

A recent paper [Hanks1985] examines temporal reasoning as an example of default reasoning. They conclude that all current systems of default reasoning, including non-monotonic logic, default logic, and circumscription, are inadequate for reasoning about persistence. I present a way of representing persistence in a framework based on a generalization of circumscription, which captures Hanks and McDermott's procedural representation.

1. Persistence

The frame problem is that of representing a dynamic world so that one can formally infer the facts whose truth values are not changed by a given action. A temporal world model allows one to assert that various actions occur at various times, and to be silent about other times. When one reasons with such a model, the frame problem is generalized to the persistence problem: given that no relevant action, or perhaps no action at all, occurred over a stretch of time, one may need to infer that certain facts do not change their truth values over that time. In other words, one needs to represent the "inertia" of the world, the moment to moment persistence of many of its properties.

Examples of persistence abound in everyday reasoning. Sitting in my office, I can infer that my car is in the parking lot, because that is where I left it this morning. [Hanks1985] examines the following example, here simplified. Assume a simple linear, discrete model of time, containing instants 1, 2, 3, etc. At time 1 John is alive, and a gun aimed at John is loaded. At time 3 the gun is fired. We know that if the gun is loaded when it is fired, John will die at the next moment of time. We would like to conclude that John is not alive at time 4. In order to do so, we must make the persistence inference that the gun stays loaded from times 1 to 3. (See figure 1.)

This report describes work done in the Department of Computer Science at the University of Rochester. It was supported in part by National Science Foundation grant DCR-8502481

2. Problems with Default Reasoning

We would like to find some simple rule of default inference which captures persistence reasoning. Hanks and McDermott describe "obvious" solutions to the persistence problem using Reiter's default logic, McCarthy's circumscription operation, and McDermott and Doyle's non-monotonic logic. In default logic, for example, one would include an rule which stated that if a fact held at a time T1, and it was consistent that it held over an interval immediately following time T1, then infer that it does hold over T1. In the circumscriptive approach, one could define a "clipping event" which occurs whenever a fact changes truth value. Persistence is indirectly asserted by circumscribing (minimizing) the predicate which holds of all clipping events.

While intuitively appealing, these approaches do not work. The basic problem, Hanks and McDermott point out, is that default inferences are not prioritized by each system. For example, applying default rules in different orders yields different extensions: in circumscription, many different models of the axioms may be minimal in the "clipping" predicate. Yet only some of these extensions or minimal models correspond to the intuitive understanding of persistence.

Consider the gun example. The axioms have a minimal model (or corresponding extension) in which the fact ALIVE persists, but the fact LOADED is (mysteriously) clipped between times 1 and 3. (See figure 2.) Therefore simply circumscribing clipped (or adding default rules) does not sanction inferences about persistence.

3. A Procedural Solution

Hanks provides a temporal-assertion management program which computes persistences. Hanks's program functions by computing persistences in temporal order, from the past to the future. For example, the persistence of LOADED is computed before the persistence of ALIVE, and so the program concludes that John dies. The program reflects our intuitions in many cases because it captures the temporal order of causality: the gun being loaded can cause John to die, and so has precedence over it.

Hanks is not optimistic about the ability of any default logic to handle this reasoning properly: "...If a significant part of defeasible reasoning can't be represented by default logics, and if in the cases where the logics fail we have no better way of describing the reasoning process than by a direct procedural characterization (like our program or its inductive definition), then logic as an AI representation language begins to look less and less attractive." Such pessimism may be premature. It is possible to represent many kinds of ordered defaults in an declarative representation. We show how this can be done in a circumscriptive framework.

4. Model Theory

The semantics of circumscription are based on the idea of minimal entailment. One statement entails another if all models of the first are also models of the second. Suppose a partial order is defined over class of models. The minimal models of a statement are those which have no strict predecessor in the partial order. Then one statement minimally entails another if all minimal models of the first are also models of the second.

McCarthy's original formulation of circumscription [McCarthy1980] defined the partial order over models in terms of the extension of some predicate, say P. A model M1 would be less than a model M2 if the extension of P in M1 is a subset of its extension in M2, and M1 and M2 are otherwise the same. Newer work [McCarthy1985] has refined this definition, largely concentrating on the role of the non-circumscribed predicates in the minimization. But many other variations on circumscription are possible.

Let facts (such as LOADED) be represented by terms, and the atom

Hold(t,f)

be used to assert that fact f holds at time t. The predicate Clip holds of a time and a fact if the fact becomes false at that time; otherwise, the truth-value of the fact persists from the earlier instant. That is:

$$\text{Hold}(t,f) \supset (\text{Hold}(t+1,f) \oplus \text{Clip}(t+1,f))$$

(The symbol $\oplus$ represents exclusive or.) Suppose we are given some assertions about when various facts hold. We wish to define a partial order over models of these sentences which reflects our intuitions about persistence. "Good" models, Hanks and McDermott suggest, are ones in which earlier facts persist as long as possible, and so should fall at the beginning of the ordering. Where M1 and M2 are models, M1 is as good or better than M2 if every clipping in M1 is either matched by an identical clipping in M2, or by an earlier clipping in M2 (possibly of some different fact) which does not also appear in M1.

The less than or equal relation between models is formally defined as follows. Where M1 is a model and P is a predicate, the expression M1[P] yields the extension of P in M1. The extension of a binary predicate such as Clip is a set of pairs, where the pair of x and y is written $\langle x, y \rangle$. Models can be compared only if they interpret constant, function, and predicate symbols other than Clip or Hold in the same way. In particular, this means that the models agree on the predicate "<", which is used to order time instances. Because models may be compared even if they do not agree on the predicate Hold, that predicate (as well as Clip) is said to *vary* during the minimization.

$M1 \leq M2$ if and only if

(i) M1 and M2 have the same domain

(ii) Every constant, function, and predicate symbol other than Clip and Hold receives the same interpretation in M1 and M2.

(iii) The following (meta-theoretic) statement is true:

$$\begin{aligned}&\langle f, t\rangle \in M1[\text{Clip}] \supset \\ &\quad \langle f, t\rangle \in M2[\text{Clip}] \vee \\ &\quad \exists\, t', f' \,.\, \langle f', t'\rangle \in M2[\text{Clip}] \,\&\, \\ &\qquad \langle f', t'\rangle \notin M1[\text{Clip}] \,\&\, \\ &\qquad \langle t', t\rangle \in M1[<]\end{aligned}$$

The final clause in this formula means that the time instant t' is before the time instant t.

A model M1 is strictly better than M2 ($M1 < M2$) just in case $M1 \leq M2$ and it is not the case that $M2 \leq M1$. From this definition one can prove that if $M1 < M2$, then (in terms of the Clip predicate) M1 and M2 are identical up to some time t': at t', the set of clippings in M2 properly includes the set of clippings in M1.

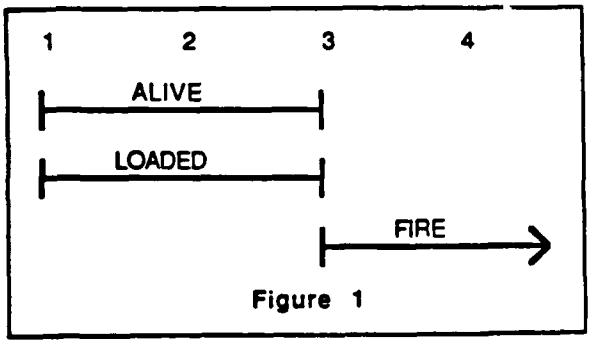


Figure 1

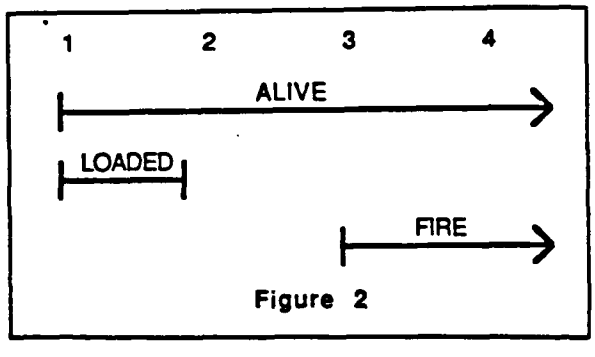


Figure 2

The minimal models are those M such that there is no M' such that M'<M. It is important to understand that the set of models is only *partially* ordered; there will be many minimal models.

The set of minimal models may be empty if there is an infinite chain of models, M1 > M2 > M3 > ... . This can occur if we minimize an existential statement of the form, "f will eventually be clipped", with no upper bound placed on the time of clipping. The preference order will attempt to postpone the clipping for an infinite period of time. This problem does not occur if such an unknown time is given a (skolem) constant name, however, due to the fact that constants and functions do not vary in the minimization.

5. Proof Theory

McCarthy's circumscription formula is a statement in 2nd-order logic which entails those statements true in all the minimal models of a predicate. The following persistence circumscription formula entails those statements true in models minimal in the partial order defined above. Let K(Clip,Hold) be our initial set of temporal assertions. We write an expression such as K(Foo,Bar) to stand for the set of sentences obtained by substituting the predicates Foo and Bar for every occurrence of Clip and Hold in K(Clip,Hold) respectively. The variables c and h range over predicates.

$$
\begin{aligned}
&\forall\ c,h\ .\\
&\quad \{K(c,h)\ \&\\
&\qquad \forall\ t,f\ .\ c(t,f) \supset\\
&\qquad\quad [Clip(t,f) \vee\\
&\qquad\qquad \exists\ t2,f2\ .\ t2<t\ \&\ Clip(t2,f2)\ \&\ \neg c(t2,f2)]\}\\
&\quad \supset \forall\ t,f\ .\ Clip(t,f) \equiv c(t,f)
\end{aligned}
$$

The formula can be informally understood as follows. Suppose that c and h are arbitrary predicates which satisfies all the constraints placed by the knowledge base on the predicates Clip and Holds respectively. Furthermore, suppose whenever c holds of a particular time and a particular fact, then either Clip also holds of that time and fact, or Clip holds of some earlier time and fact which are not in the extension of c. The conclusion is that c and Clip are identical; the second alternative is never the case. There cannot be a predicate which satisfies all the constraints on Clip, yet allows some fact to persist for a longer time, without having to clip some other fact at that time. The predicate-variable h was introduced in order to allow Holds to vary during the minimization of Clip.

In order to use this formula, we must select particular instantiations for the variables c and h, such that the initial set of assertions K(Clip,Hold) entails the main antecedent (in curly braces) Typically c is instantiated as a lambda expression which enumerates the the desired set of clippings. The variable h is instantiated by a lambda expression which describes which and when facts hold in the corresponding minimal models. Modus ponens then allows us to conclude that the extension of Clip is precisely the desired set of clippings: $Clip(t,f) \equiv c(t,f)$.

It is possible to show that this formula is valid in all models minimal in the above sense. As with standard circumscription, the formula is inconsistent if there are no minimal models. [Lifschitz1985] develops a generic circumscription-like formula based on pre-orders. The formula above is easy to express in Lifschitz' compact and elegant notation.

6. Example

The gun example illustrates the use of persistence circumscription. K(Clip,Hold) is the following set of statements. Not shown are unique name axioms, such as LOADED ≠ ALIVE, etc.

$$Hold(t,f) \supset (Hold(t+1,f) \oplus Clip(t+1,f))$$

$$Hold(t,FIRE)\ \&\ Hold(t,LOADED) \supset \neg Hold(t+1,LOADED)\ \&\ \neg Hold(t+1,ALIVE)$$

$$Hold(1,LOADED)$$

$$Hold(1,ALIVE)$$

$$Hold(3,FIRE)$$

The goal is to prove that ¬Hold(4,ALIVE). (A more complete set of axioms would also state that if something is a fact, and it does not hold at a time, then its negation holds at that time. This complication would not materially change our solution.)

Our intuitions tell us that the only (required) clipping event occurs at time 4, when both LOADED and ALIVE become false (as in figure 1). The instantiation for c is therefore:

$$c = \lambda\ t,f\ .\ t=4\ \&\ (f=LOADED \vee f=ALIVE)$$

When do various facts hold? Again referring to figure 1, we see that LOADED and ALIVE hold between times 1 and 3, and FIRE begins holding at time 3 (and persists thereafter). For h we can thus choose:

$$
\begin{aligned}
h = \lambda\ t,f\ .&\\
&(f=LOADED \supset 1\le t\le 3)\ \&\\
&(f=ALIVE \supset 1\le t\le 3)\ \&\\
&(f=FIRE \supset t\ge 3)\ \&\\
&(f=LOADED \vee f=ALIVE \vee f=FIRE)
\end{aligned}
$$

These expressions are placed in the persistence circumscription formula, which is then simplified. This involves proving that the main antecedent of the formula:

$$\{K(c,h) \;\&\; \forall t,f\,.\,c(t,f) \supset [Clip(t,f) \vee \exists t2,f2\,.\,t2<t \;\&\; Clip(t2,f2) \;\&\; \neg c(t2,f2)]\} \supset \forall t,f\,.\,Clip(t,f) \equiv c(t,f)$$

must be true, where c and h are defined as above. This is done by showing (i) that K(Clip,Hold) entails K(c,h), and (ii) that K(Clip,Hold) entails:

$$\star \quad \forall t,f\,.\,c(t,f) \supset [Clip(t,f) \vee \exists t2,f2\,.\,t2<t \;\&\; Clip(t2,f2) \;\&\; \neg c(t2,f2)]$$

The first part of the proof involves substituting c and h for Clip and Holds in the initial knowledge base and simplifying, which is straightforward but tedious. For example, the formula Hold(1,LOADED) becomes h(1,LOADED), which is:

$$[\lambda t,f\,.\,(f=LOADED \supset 1\leq t\leq 3) \;\&\; (f=ALIVE \supset 1\leq t\leq 3) \;\&\; (f=FIRE \supset t\geq 3) \;\&\; (f=LOADED \vee f=ALIVE \vee f=FIRE)]\,(1,LOADED)$$

This expression reduces to:

$$(LOADED=LOADED \supset 1\leq 1\leq 3) \;\&\; (LOADED=ALIVE \supset 1\leq 1\leq 3) \;\&\; (LOADED=FIRE \supset 1\geq 3) \;\&\; (LOADED=LOADED \vee LOADED=ALIVE \vee LOADED=FIRE)$$

which, given the unique name axioms mentioned above, is a tautology.

The second step, as mentioned above, is to show that K(Clip,Hold) entails the statement marked with a ($\star$). The antecedent of ($\star$) is false, and the statement therefore true, except when t=4, and f=LOADED or f=ALIVE. Therefore we must show that:

$$Clip(4,LOADED) \vee \exists t2,f2\,.\,t2<4 \;\&\; Clip(t2,f2) \;\&\; \neg c(t2,f2)$$

and

$$\dagger \quad Clip(4,ALIVE) \vee \exists t2,f2\,.\,t2<4 \;\&\; Clip(t2,f2) \;\&\; \neg c(t2,f2)$$

Consider the sentence involving ALIVE, marked with a ($\dagger$). We can show this statement is true by showing that if the second main disjunct is false, then the first disjunct must be true. So suppose that

$$\exists t2,f2\,.\,t2<4 \;\&\; Clip(t2,f2) \;\&\; \neg c(t2,f2)$$

is false. This means that there is no clipping event before time 4. K(Clip,Hold) includes the statements Hold(1,ALIVE) and Hold(1,LOADED). The axiom

$$Hold(t,f) \supset (Hold(t+1,f) \oplus Clip(t+1,f))$$

can therefore be applied for times t=1 and t=2, giving the conclusion

$$Hold(3,ALIVE) \;\&\; Hold(3,LOADED)$$

Since Hold(3,FIRE), the axiom about firing loaded guns tells us that $\neg$Hold(4,ALIVE). Since Hold(3,ALIVE), we finally conclude that Clip(4,ALIVE), the first disjunct of ($\dagger$), is true. Therefore ($\dagger$) is true. The sentence (just before ($\dagger$)) involving LOADED can be proven in a similar manner.

Thus the statement ($\star$) is true, the main antecedent of the instantiated persistence circumscription formula is true, and so

$$Clip(t,f) \equiv c(t,f)$$

Since c(4,ALIVE), it must the case that Clip(4,ALIVE), and so $\neg$Hold(4,ALIVE).

Discussion

Several morals can be drawn from this exercise. One is that in reasoning about time, and probably most other applications, default inferences must be properly ordered. Another is that we may need to step beyond the incremental progression of circumscriptive techniques, from predicate circumscription, to circumscription with variables, to formula circumscription, and view circumscription as a general framework for expressing inference in terms of various classes of minimal models. A final moral is that by thinking about default inference in terms of relationships between models, we may more readily see the inadequacies of our own purported solutions.

The particular formulas just presented do not solve in the persistence problem in general. Recall the example using persistence to infer that my car is in the parking lot. Suppose I learn at time 1000 that my car is gone. Using the techniques just described, I can infer that the car was in the parking lot up to the shortest possible time before I

knew it was gone. This is clearly an unreasonable inference. Someone could have stolen it five minutes after I left it there; I have no reason to prefer an explanation in which it vanished five seconds before I glanced out my office window. The inadequacy is ontological: we can't handle persistence properly until we have a richer theory of causation. The purely temporal solution often works because the flow of time reflects the order of physical causation. When the full story of causation is told, we then require an efficient algorithm for performing the necessary deductions, such as Hanks's, and a clear model theory, such as that provided by generalized circumscription, to explain and justify the whole process.

References

Hanks1985.
Steve Hanks and Drew McDermott, "Temporal Reasoning and Default Logics," YALEU/CSD/RR #430, Yale University, Department of Computer Science, Oct 1985.

Lifschitz1985.
Vladimir Lifschitz, "Some Results on Circumscription," in *Proceedings from the Non-Monotonic Reasoning Workshop*, AAAI, Oct 1985.

McCarthy1985.
John McCarthy, "Applications of Circumscription to Formalizing Common Sense Knowledge," in *Proceedings from the Non-Monotonic Reasoning Workshop*, AAAI, Oct 1985.

McCarthy1980.
John McCarthy, "Circumscription -- A Form of Non-Monotonic Reasoning," *Artificial Intelligence*, vol. 13, no. 1, pp. 27-38, 1980.

Appendix A-4

GENERALIZED PLAN RECOGNITION

Henry A. Kautz
James F. Allen

Department of Computer Science
University of Rochester
Rochester, New York 14627

ABSTRACT

This paper outlines a new theory of plan recognition that is significantly more powerful than previous approaches. Concurrent actions, shared steps between actions, and disjunctive information are all handled. The theory allows one to draw conclusions based on the class of possible plans being performed, rather than having to prematurely commit to a single interpretation. The theory employs circumscription to transform a first-order theory of action into an action taxonomy, which can be used to logically deduce the complex action(s) an agent is performing.

1. Introduction

A central issue in Artificial Intelligence is the representation of actions and plans. One of the major modes of reasoning about actions is called plan recognition, in which a set of observed or described actions is explained by constructing a plan that contains them. Such techniques are useful in many areas, including story understanding, discourse modeling, strategic planning and modeling naive psychology. In story understanding, for example, the plans of the characters must be recognized from the described actions in order to answer questions based on the story. In strategic planning, the planner may need to recognize the plans of another agent in order to interact (co-operatively or competitively) with that agent.

Unlike planning, which often can be viewed as purely hypothetical reasoning (i.e. if I did A, then P would be true), plan recognition models must be able to represent actual events that have happened as well as proposing hypothetical explanations of actions. In addition, plan recognition inherently involves more uncertainty than in planning. Whereas in planning, one is interested in finding any plan that achieves the desired goal, in plan recognition, one must attempt to recognize the particular plan that another agent is performing. Previous plan recognition models, as we shall see, have been unable to deal with this form of uncertainty in any significant way.

A truly useful plan recognition system must, besides being well-defined, be able to handle various forms of uncertainty. In particular, often a given set of observed actions may not uniquely identify a particular plan, yet many important conclusions can still be drawn and predictions about future actions can still be made.

For example, if we observe a person in a house picking up the car keys, we should be able to infer that they are going to leave the house to go to the car, even though we cannot tell if they plan to drive somewhere, or simply to put the car in the garage. On the basis of this information, we might ask the person to take the garbage out when they leave. To accomplish this, a system cannot wait until a single plan is uniquely identified before drawing any conclusions. On the other hand, a plan recognizer should not prematurely jump to conclusions either. We do not want to handle the above example by simply inferring that the person is going to put the car into the garage when there is no evidence to support this interpretation over the one involving driving to the store.

In addition, a useful plan recognizer in many contexts cannot make simplistic assumptions about the temporal ordering of the observations either. In story understanding, for example, the actions might not be described in the actual order that they occurred. In many domains, we must also allow actions to occur simultaneously with each other, or allow the temporal ordering not to be known at all. Finally, we must allow the possibility that an action may be executed as part of two independent plans. We might, for example, make enough pasta one night in order to prepare two separate meals over the next two days.

None of the previous models of plan recognition can handle more than a few of these situations in any general way. Part of the problem is that none of the frameworks have used a rich enough temporal model to support reasoning about temporally complex situations. But even if we were able to extend each framework with a general temporal reasoning facility, there would still be problems that remain. Let us consider three of the major approaches briefly, and discuss these other problems.

The explanation-based approaches, outlined formally by [Cha85] all attempt to explain a set of observations by finding a set of assumptions that entails the observations. The problem with this is that there may be many such sets of assumptions that will have this property, and the theory says nothing as to how to select among them. In practice, systems based on this framework (e.g. [Wil83]) will over-commit, and select the first explanation found, even though it is not uniquely identified by the observations. In addition, they are not able to handle disjunctive information.

The approaches based on parsing (e.g. [Huf82, Sid81]) view actions as sequences of subactions and essentially model this knowledge as a context-free rule in an "action grammar". The primitive (i.e. non-decomposable) actions in the framework are the terminal symbols in the grammar. The observations are then treated as input to the parser and it attempts to derive a parse tree to explain the observations. A system based on this model would suffer from the problem of over-commitment unless it generates the set of possible explanations (i.e. all possible parses). While some interesting temporal aspects in combining plans can be handled by using more powerful grammars such as shuffle grammars, each individual plan can only be modelled as a sequence of actions. In addition, every step of a plan must be observed -- there is no capability for partial observation. It is not clear how more temporally complex plans could be modelled, such as those involving simultaneous actions, or how a single action could be viewed as being part of multiple plans.

The final approach to be discussed is based on the concept of "likely" inference (e.g. [All83, Pol84]). In these systems a set of rules is used of the form: "If one observes act A, then it may be that it is part of act B". Such rules outline a search space of actions that produces plans that include the observations. In practice, the control of this search is hidden in a set of heuristics and thus is hard to define precisely. It is also difficult to

This work was supported in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, and the Air Force Office of Scientific Research, under Contract No. F30602-85-C-0008, and the National Science Foundation under grant DCR-8502481.

ittach a semantics to such rules as the one above. This rule does not mean that if we observe A, then it is probable that B is being executed, or even that it is possible that B is being executed. The rule is valid even in situations where it is impossible for B to be in execution. These issues are decided entirely by the heuristics. As such, it is hard to make precise claims as to the power of this formalism.

In this paper we outline a new theory of plan recognition that is significantly more powerful that these previous approaches in that it can handle many of the above issues in an intuitively satisfying way. Furthermore, there are no restrictions on the temporal relationships between the observations. Another important result is that the implicit assumptions that appear to underly all plan recognition processes are made explicit and precisely defined within a formal theory of action. Given these assumptions and a specific body of knowledge about the possible actions and plans to be considered, this theory will give us the strongest set of conclusions that can be made given a set of observations. As such, this work lays a firm base for future work in plan recognition.

Several problems often associated with plan recognition are not considered in the current approach, however. In particular, beyond some simple simplicity assumptions, the framework does not distinguish between a priori likely and non-likely plans. Each logically possible explanation, given the assumptions, is treated equally within the theory. It also can only recognize plans that are constructed out of the initial library of actions defined for a particular domain. As a result, novel situations that arise from a combination of existing plans may be recognized, but other situations that require generalization techniques, or reasoning by analogy cannot be recognized.

2. A New View of Plan Recognition

It is not necessary to abandon logic, or to enter the depths of probabilistic inference, in order to handle the problematic cases of plan recognition described above. Instead, we propose that plan recognition be viewed as ordinary deductive inference, based on a set of observations, an action taxonomy, and one or more simplicity constraints.

An action taxonomy is an exhaustive description of the ways in which actions can be performed, and the ways in which any action can be used as step of a more complex action. Because the taxonomy is complete, one can infer the disjunction of the set of possible plans which contain the observations, and then reason by cases to reduce this disjunction.

An action taxonomy is obtained by applying two closed-world assumptions to an axiomatization of an action hierarchy. The first assumption states that the known ways of performing an action are the only ways of performing that action. The assumption is actually a bit more general, in that it states the known ways of *specializing* an action are the only ways. Each time an abstract action is specialized, more is known about how to perform it. For example, because the action type "throw" specializes the action type "transfer location", we can think of throwing as a way to transfer location.

The second assumption states that all actions are purposeful, and that all the possible reasons for performing an action are known. This assumption is realized by stating that if an action A occurs, and P is the set of more complex actions in which A occurs as a substep, then *some* member of P also occurs.

These assumptions can be stated using McCarthy's circumscription scheme. The action hierarchy is transformed by first circumscribing the ways of specializing an act, and then circumscribing the ways of using an act. The precise formulation of this operation is described in section 6 below.

The simplicity constraints become important when we need to recognize a plan which integrates several observations. The top of the action taxonomy contains actions which are done for their own sake, rather than as steps of more complex actions. When several actions are observed, it is often a good heuristic to assume that the observations are all part of the same top level act, rather than each being a step of an independent top level act. The simplicity constraint which we will use asserts that as few top level actions occur as possible. The simplicity constraint can be represented by a formula of second-order logic which is similar to the circumscription formula. In any particular case the constraint is instantiated as a first-order formula which asserts "there are no more than n top level acts", which n is a particular constant choosen to be as small as possible, and still allow the instantiated constraint to be consistent with the observations and taxonomy.

While one can imagine many other heuristic rules for choosing between interpretations of a set of observed actions, the few given here cover a great many common cases, and seem to capture the "obvious" inferences one might make. More fine grained plan recognition tasks (such as strategic planning) would probably require some sort of quantitative reasoning.

3. Representing Action

The scheme just described requires a representation of action that includes:

--the ability to assert that an action actually occurred at a time;

--a specialization hierarchy;

--a decomposition (substep) hierarchy.

Action instances are individuals which occur in the world, and are classified by action types. The example domain is the world of cooking, which includes a very rich action hierarchy, as well as a token bit of block stacking. (See figure 1.) The broad arrows indicate that one action type is a specialization of another action type, whereas the thin arrows indicates the decomposition of an action into subactions. We will see how to represent this information in logic presently. The diagram does not indicate other conditions and constraints which are also part of an action decomposition. Instances of action types are also not shown. We introduce particular instances of actions using formulas such as

$$\#(E9, \text{makePastaDish})$$

to mean that E9 is a real action instance of type MakePastaDish. (The symbol # is the "occurs" predicate.) The structure of a particular action can be specified by a set of role functions. In particular, the function τ applied to an action instance returns the interval of time over which the action instance occurs. Other roles of an action can also be represented by functions; e.g., Agent(E9) could be the agent causing the action, and Result(E9) could be the particular meal produced by E9. (For simplicity we will assume in this paper that all actions are performed by the same agent.) To record the observation of the agent making a pasta dish at time I7, one would assert:

$$\exists\, e\,.\ \#(e, \text{makePastaDish}) \ \&\ \tau(e) = I7$$

Action types need not all be constants, as they are here; often it is useful to use functions to construct types, such as Move(x,y). For simplicity, all the actions used in the examples in this paper use only constant action types.

Action specialization is easy to represent in this scheme. In the cooking world, the act of making a pasta dish specializes the act of preparing a meal, which in turn specializes the class of top level acts. Specialization statements are simply universally-quantified implications. For example, part of the hierarchy in figure 1 is represented by the following axioms:

[1] $\forall\, e\,.\ \#(e, \text{PrepareMeal}) \supset \#(e, \text{TopLevelAct})$

[2] $\forall\, e\,.\ \#(e, \text{MakePastaDish}) \supset \#(e, \text{PrepareMeal})$

[3] $\forall\, e\,.\ \#(e, \text{MakeFettucciniMannara}) \supset \#(e, \text{MakePastaDish})$

[4] $\forall\, e\,.\ \#(e, \text{MakeFettucini}) \supset \#(e, \text{MakeNoodles})$

[5] $\forall e . \#(e, \text{MakeSpaghetti}) \supset \#(e, \text{MakeNoodles})$

[6] $\forall e . \#(e, \text{MakeChickenMarinara}) \supset \#(e, \text{MakeMeatDish})$

The first statement, for example, means that any action instance which is a PrepareMeal is also a TopLevelAct.

The decomposition hierarchy is represented by implications which assert necessary (and perhaps sufficient) conditions for an action instance to occur. This may include the fact that some number of subactions occur, and that various facts hold at various times [All84]. These facts include the preconditions and effects of the action, as well as various constraints on the temporal relationships of the subactions [All83a].

For the level of analysis in the present paper, we do not need to distinguish the minimal necessary set of conditions for an action to occur, from a larger set which may include facts which could be deduced from the components of the act. It is also convenient to eliminate some existentially quantified variables by introducing a function S(i,e) which names the i-th subaction (if any) of action e. (The actual numbers are not important; any constant symbols can be used.) For example, the makePastaDish action is decomposed as follows:

[7] $\forall e . \#(e, \text{MakePastaDish}) \supset$
$\exists t_1 . \#(S(1,e), \text{MakeNoodles}) \wedge$
$\#(S(2,e), \text{Boil}) \wedge$
$\#(S(3,e), \text{MakeSauce}) \wedge$
$\text{Object}(S(2,e)) = \text{Result}(S(1,e)) \wedge$
$\text{hold}(\text{noodle}(\text{Result}(S(1,e))), t_1) \wedge$
$\text{overlap}(T(S(1,e)), t_1) \wedge$
$\text{during}(T(S(2,e)), t_1)$

This states that every instance of MakePastaDish consists of (at least) three steps: making noodles, boiling them, and making a sauce. The result of making noodles is an object which is (naturally) of type noodle, for some period of time which follows on the heels of the making. (Presumably the noodles cease being noodles after they are eaten.) Furthermore, the boiling action must occur while the noodles are, in fact, noodles. A complete decomposition of MakePastaDish would contain other facts, such that result of the MakeSauce act must be combined at some point with the noodles, after they are boiled.

The constraint that all the subactions of an action occur during the time of the action is expressed for all acts by the axiom:

[8] $\forall i,e . \text{during}(T(S(i,e)), T(e))$

It is important to note that a decomposable action can still be further specialized. For example, the action type MakeFettuciniMarinara specializes MakePastaDish and adds additional constraints on the above definition. In particular, the type of noodles made in step 1 must be fettucini, while the sauce made in step 3 must be marinara sauce.

A final component of the action hierarchy are axioms which state action-type disjointedness. Such axioms are expressed with the connective "not and", written as ∇:

[9] $\forall e . \#(e, \text{MakeFettuciniAlfredo}) \nabla \#(e, \text{MakeFettuciniMarinara})$

This simply says that a particular action cannot be *both* an instance of making fettucini Alfredo *and* an instance of making fettucini Marinara. Disjointedness axioms can be compactly represented and used in resolution-based inference using techniques adapted from [Ten86].

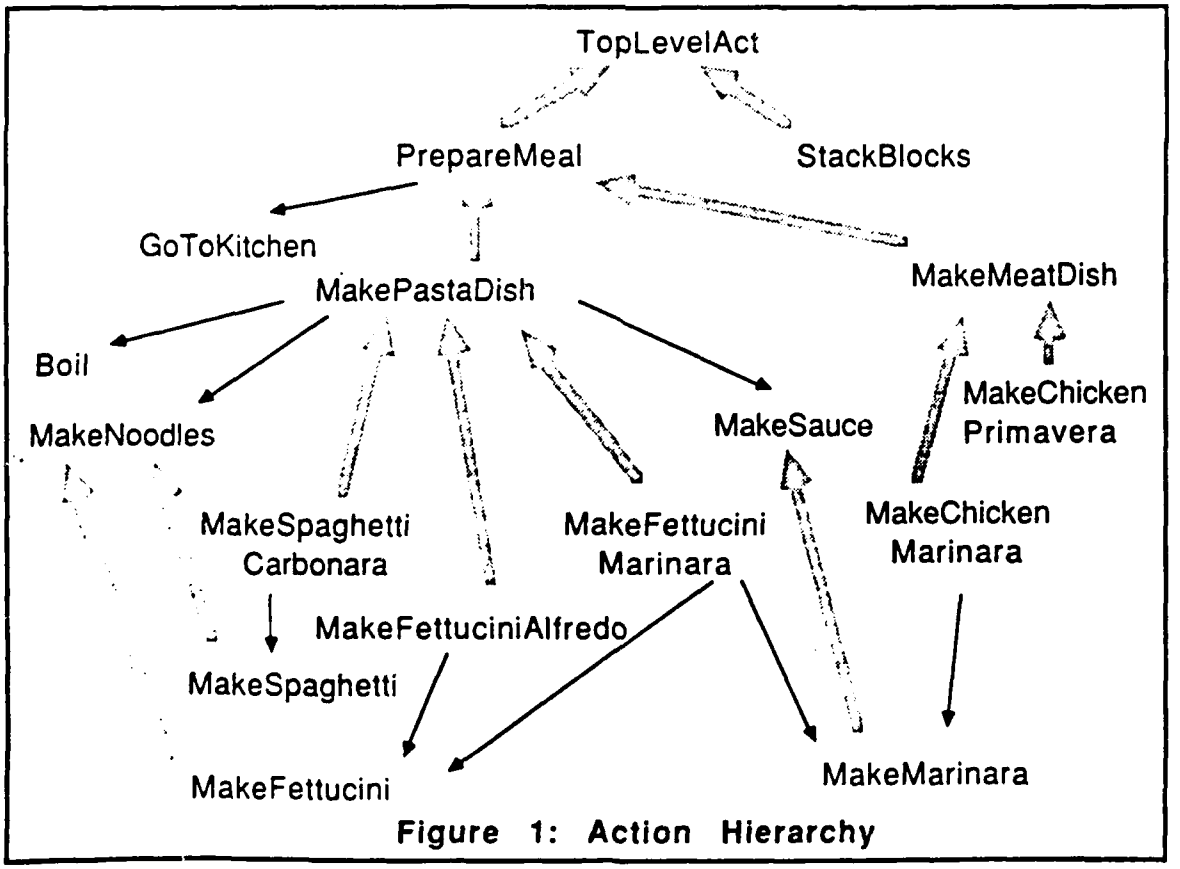


Figure 1: Action Hierarchy

4. Creating the Taxonomy

The assumptions necessary for plan recognition can now be specified more precisely by considering the full action hierarchy presented in figure 1. Let KB1 be the set of axioms schematically represented by the graph, including the axioms mentioned above. KB1 will be transformed into a taxonomy by applying the completeness assumptions discussed above. The result of the first assumption (all ways of specializing an action are known) is the database KB2, which includes all of KB1, together with statements which assert specialization completeness. These include the following, where the symbol ⊕ is the connective "exclusive or".

[10] ∀ e . #(e, TopLevelAct) ⊃
#(e, PrepareMeal) ⊕
#(e, StackBlocks)

[11] ∀ e . #(e, PrepareMeal) ⊃
#(e, MakePastaDish) ⊕
#(e, MakeMeatDish)

[12] ∀ e . #(e, MakePastaDish) ⊃
#(e, MakeFettuciniMarinara) ⊕
#(e, MakeFettuciniAlfredo) ⊕
#(e, MakeSpaghettiCarbonara)

[13] ∀ e . #(e, MakeMeatDish) ⊃
#(e, MakeChickenMarinara) ⊕
#(e, MakeChickenPrimavera)

[14] ∀ e . #(e, MakeNoodles) ⊃
#(e, MakeFettucini) ⊕
#(e, MakeSpaghetti)

These state that every top level action is either a case of preparing a meal, or of stacking blocks; that every meal preparation is either a case of making a pasta dish or making a meat dish; and so on, all the way down to particular, basic types of meals. Not all actions, of course, are specializations of TopLevelAct. For example, axiom [14] states that every MakeNoodles can be further classified as a MakeFettucini or as a MakeSpaghetti, but it is not the case that any MakeNoodles can be classified as a TopLevelAct.

The second assumption asserts that the given decompositions are the only decompositions. KB2 is transformed to the final taxonomy KB3, which includes all of KB2, as well as:

[15] ∀ e . #(e, MakeNoodles) ⊃
∃ a . #(a, MakePastaDish) & e = S(1,a)

[16] ∀ e . #(e, MakeMarinara) ⊃
∃ a . [#(a, MakeFettuciniMarinara) & e = S(3,a)] ∨
[#(a, MakeChickenMarinara) & e = S(3,a)]

[17] ∀ e . #(e, MakeFettucini) ⊃
∃ a . [#(a, MakeFettuciniMarinara) & e = S(1,a)] ∨
[#(a, MakeFettuciniAlfredo) & e = S(1,a)]

Axiom [15] states that whenever an instance of MakeNoodles occurs, then it must be the case that some instance of MakePastaDish occurs. Furthermore, the MakeNoodles act which is required as a substep of the MakePastaDish is in fact the given instance of MakeNoodles. Cases like this, where an action can only be used in one possible super-action, usually occur at a high level of action abstraction. It is more common for many uses for an action to occur in the taxonomy. The given hierarchy has two distinct uses for the action MakeMarinara, and this is captured in axiom [16]. From the fact that the agent is making Marinara sauce, one is justified in concluding that an action instance will occur which is *either* of type MakeFettuciniMarinara or of type MakeChickenMarinara.

All these transformations can easily be performed automatically given an action taxonomy of the form described in the previous section. The formal basis for these transformations is described in section 6.

5. Recognition Examples

We are now ready to work through some examples of plan recognition using the cooking taxonomy. In the steps that follow, existentially-quantified variables will be replaced by fresh constants. Constants introduced for observed action instances begin with E, and those for deduced action instances being with K. Simple cases typical of standard plan recognition are easily accounted for. In this section, we shall consider an extended example demonstrating some more problematic cases.

Let the first observation be disjunctive: the agent is observed to be either making fettucini or making spaghetti, but we cannot tell which. This is still enough information to make predictions about future actions. The observation is:

[18] #(E1, MakeFettucini) ∨ #(E1, MakeSpaghetti)

The abstraction axioms let us infer up the hierarchy:

[19] #(E1, MakeNoodles) abstraction axioms [4], [5]

[20] #(K01, MakePastaDish) decomposition axiom [15], and existential instantiation

[21] #(K01, PrepareMeal) abstraction axiom [2]

[22] #(K01, TopLevelAct) abstraction axiom [1]

Statement [20] together with [7] lets us make a prediction about the future: a Boil will occur:

[23] #(S(2,K01), boil) &
Object(S(2,K01)) = Result(E1) &
after(T(S(2,K01)),T(E1))

Thus even though the particular plan the agent is performing cannot be exactly identified, specific predictions about future activities can still be made.

The previous step showed how one could reason from a disjunctive observation up the abstraction hierarchy to a non-disjunctive conclusion. With the next observation, we see that going up the decomposition hierarchy from a non-disjunctive hierarchy can lead to a disjunctive conclusion. Suppose the next observation is:

[24] #(E3, MakeMarinara)

Applying axiom [16], which was created by the second completeness assumption, leads to the conclusion:

[25] #(K02, MakeFettuciniMarinara) ∨
#(K02, MakeChickenMarinara)

The abstraction hierarchy can again be used to collapse this disjunction:

[26] #(K02, MakePastaDish) ∨ #(K02, MakeMeatDish)

[27] #(K02, PrepareMeal)

[28] #(K02, TopLevelAct)

At this point the simplicity constraint comes into play. The strongest form of the constraint, that there is only one top level action in progress, is tried first:

[29] ∀ e1,e2 . #(e1, TopLevelAct) &
#(e2, TopLevelAct) ⊃ e1 = e2

Together with [22] and [28], this implies:

[30] K01 = K02

Substitution of equals yields:

[31] #(K02, MakePastaDish)

One of the disjointedness axioms from the original action hierarchy is:

[32] ∀ e . #(e, MakePastaDish) ⊃ #(e, MakeMeatDish)

Statements [30], [31], and [6] let us deduce:

[33] ¬#(K02, MakeMeatDish)

[34] ¬#(K02, MakeChickenMarinara)

Finally, [34] and [25] let us conclude that only one plan, which contains both observations, is occurring:

[35] #(K02, MakeFettuciniMarinara)

Temporal constraints did not play a role in these examples, as they do in more complicated cases. For example, observations need not be received in the order in which the observed events occurred, or actions might be observed in an order where the violation of temporal constraints can allow the system to reject hypotheses. For example, if a Boil act at an unknown time were input, the system would assume that it was the boil act of the (already deduced) MakePastaDish act. If the Boil were constrained to occur before the initial MakeNoodles, then the strong simplicity constraint (and all deductions based upon it) would have to withdrawn, and two distinct top level actions postulated.

Different top level actions (or any actions, in fact) can share subactions, if such sharing is permitted by the particular domain axioms. For example, suppose every PrepareMeal action begins with GotoKitchen, and the agent is constrained to remain in the kitchen for the duration of the act. If the agent is observed performing two different instances of PrepareMeal, and is further observed to remain in the kitchen for an interval which intersects the time of both actions, then we can deduce that both PrepareMeal actions share the same initial step. This example shows the importance of including observations that certain states hold over an interval. Without the fact that the agent remained in the kitchen, one could not conclude that the two PrepareMeal actions share a step, since it would be possible that the agent left the kitchen and then returned.

6. Closing the Action Hierarchy

Our formulation of plan recognition is based on an explicitly asserting that the action hierarchy (also commonly called the "plan library") is complete. While the transformation of the hierarchy into a taxonomy can be automated, some details of the process are not obvious. It is not correct to simply apply predicate completion, in the style of [Cla78]. For example, even if action A is the only act which is stated to contain act B as a substep, it may not be correct to add the statement

$$\forall e . \#(e.B) \supset \exists e1 . \#(e1.A)$$

if there is some act C which either specializes or generalizes B, and is used in an action other than A. For example, in our action hierarchy, the only *explicit* mention of MakeSauce appears in the decomposition of MakePastaDish. But the taxonomy should *not* contain the statement

$$\forall e . \#(e, \text{MakeSauce}) \supset \exists a . \#(a, \text{MakePastaDish}) \wedge e = S(3,a)$$

because a particular instance of MakeSauce may also be an instance of MakeMarinara, and occur in the decomposition of the action MakeChickenMarinara. Only the weaker statement

$$\forall e . \#(e, \text{MakeSauce}) \supset \exists a . [\#(a, \text{MakePastaDish}) \wedge e = S(3,a)] \vee [\#(a, \text{MakeChickenMarinara}) \wedge e = S(3,a)]$$

is justified. It *would* be correct, however, to infer from an observation of MakeSauce which is known *not* to be an instance of MakeMarinara that MakePastaDish occurs.

We would like, therefore, a clear understanding of the semantics of closing the hierarchy. McCarthy's notion of minimal entailment and circumscription [McC85] provides a semantic and proof-theoretic model of the process. The implementation described in section 7 can be viewed as an efficient means for performing the sanctioned inferences.

There is not space here to fully explain how and why the circumscription works; more details appear in [Kau85]. A familiarity with the technical vocabulary of circumscription is probably needed to make complete sense of the rest of this section. Roughly, circumscribing a predicate minimizes its extension. Predicates whose extensions are allowed to change during the minimization are said to *vary*. All other predicates are are called *parameters* to the circumscription. In anthropomorphic terms, the circumscribed predicate is trying to shrink, but is constrained by the parameters, who can choose to take on any values allowed by the original axiomatization. For example, circumscribing the predicate p in the theory:

$$\forall x . p(x) \equiv q(x)$$

where q acts as a parameter does *nothing*, because q can "force" the extension of p to be arbitrarily large. On the other hand, if q *varies* during the circumscription, then the circumscribed theory entails that the extension of p is *empty*.

As demonstrated above, the first assumption states that the known ways of specializing an action are all the ways. Let us call all action types which are not further specialized *basic*. Then another way of putting the assumption is to say that the "occurs" predicate, #, holds of an instance and an *abstract* action type only if it *has to*, because # holds of that instance and a basic action type which specializes the abstract type. So what we want is to circumscribe that part of # which applies to non-basic action types. This can be done by adding a predicate which is true of all basic action instances, and to let this predicate act as a parameter during the circumscription of #. In our example domain, the following two statements, which we will call Ψ, define such a predicate.

$$\text{Let } \Psi = \{ \forall x . \text{basic}(x) \equiv x = \text{MakeFettuciniMarinara} \vee x = \text{Boil} \vee \ldots .$$

$$\forall e,x . \#\text{basic}(e,x) \equiv \#(e,x) \wedge \text{basic}(x) \}$$

KB1 is the set of axioms which make up the original action hierarchy. KB2 is then defined to be the circumscription of # relative to KB1 together with Ψ, where all other predicates (including #basic) act as parameters.

$$KB2 = \text{Circumscribe}(KB1 \cup \Psi, \#)$$

Interestingly, the process works even if there are many levels of abstraction hierarchy above the level of basic actions. Note that basic actions (such as MakeFettuciniMarinara) may be decomposable, even though they are not further specialized.

The second assumption states that any non-top-level action occurs only as part of the decomposition of some top-level action. Therefore we want to circumscribe that part of # which applies to non-top-level actions. This can be done by adding a predicate to KB2 which is true of all top-level action instances, and circumscribing # again. The predicate #basic added above must be allowed to vary in the circumscription.

$$\text{Let } \Phi = \{ \forall e . \#\text{toplevel}(e) \supset \#(e,\text{TopLevelAct}) \}$$

$$KB3 = \text{Circumscribe}(KB2 \cup \Phi, \#, \#\text{basic})$$

As before, the process can percolate though many levels of the action decomposition hierarchy. Note that the concepts basic action and top-level action are not antonyms; for example, the type MakeFettuciniMarinara is basic (not specializable), yet any instance of it is also an instance of TopLevelAct.

Circumscription cannot be used to express the simplicity constraint. Instead, one must minimize the *cardinality* of the extension of #, after the observations are recorded. [Kau85] describes the cardinality-minimization operator, which is similar, but more powerful than, the circumscription operator.

7. Implementation Considerations

The formal theory described here has given a precise semantics to the plan recognition reasoning process by specifying a set of axioms from which all desired conclusions may be derived deductively. Although no universally-applicable methods are known for automating circumscription, by placing reasonable restrictions on the form of the action hierarchy axioms, we can devise a special-purpose algorithm for computing the circumscriptions. As a result, in theory we could simply run a general purpose theorem prover given the resulting axioms to prove any particular (valid) conclusion. In practice, since we often don't have a specific question to ask beyond "what is the agent's goal?" or "what will happen next?", it is considerably more useful to design a specialized forward chaining reasoning process that essentially embodies a particular inference strategy over these axioms.

We are in the process of constructing such a specialized reasoner. The algorithm divides into two components: the preprocessing stage and the forward-chaining stage. The preprocessing stage is done once for any given domain. The two completeness assumptions from in the previous section are realized by circumscribing the action hierarchy. The result of the circumscription can be viewed as an enormously long logical formula, but is quite compactly represented by a graph structure.

The forward-chaining stage begins when observations are received. This stage incorporates the assumption that as few top-level acts as possible are occurring. As each observation is received, the system chains up both the abstraction and decomposition hierarchies, until a top-level action is reached. The intermediate steps may include many disjunctive statements. The action hierarchy is used as a control graph to direct and limit this disjunctive reasoning. After more than one observation arrives, the system will have derived two or more (existentially instantiated) constants which refer to top-level actions. The simplicity assumption is applied, by adding a statement that some subsets of these constants must be equal. Exclusive-or reasoning now propagates down the hierarchy, deriving a more restrictive set of assertions about the top-level acts and their subacts. If an inconsistency is detected, then the number of top-level acts is incremented, and the system backtracks to the point at which the simplicity assumption was applied.

This description of the implementation is admittedly sketchy. Many more details, including how the temporal constraint propagation system integrates with the forward-chaining reasoner, will appear in a forthcoming technical report.

8. Future Work

Future work involves completing the theoretical foundation, and building a test implementation.

The theoretical work includes a formal specification of the form of the action taxonomy so that its circumscription can always be effectively computed. Theorems guaranteeing the consistency and intuitive correctness of the circumscription will be completed.

More complex temporal interactions between simultaneously occurring actions will be investigated. We will show how the framework handles more complicated examples involving step-sharing and observations received out of temporal order (e.g. mystery stories). It will probably be necessary to develop a slightly more sophisticated simplicity constraint. Rather than stating that as few top-level actions occur as possible, it is more realistic to state that as few top-level actions as possible are occurring at any one time. In addition, observations of *non*-occurrences of events (e.g. the agent did *not* boil water) are an important source of information in plan recognition. Non-occurrences integrate nicely into our framework.

Many of the subsystems that are used by the plan recognizer (such as a temporal reasoner [All83a], and a lisp-based theorem prover which handles equality [All84a]) have been developed in previous work at Rochester, and construction of the complete implementation is under way.

References

All83. James F. Allen, "Recognizing Intentions from Natural Language Utterances," in *Computational Models of Discourse*, ed. M. Brady, MITP, 1983.

All83a. James F. Allen, "Maintaining Knowledge About Temporal Intervals," *Communications of the ACM*, no. 26, pp. 832-843, Nov 1983.

All84. James F. Allen, "Towards a General Theory of Action and Time," *Artificial Intelligence*, vol. 23, no. 2, pp. 123-154, July 1984.

Cha85. Eugene Charniak and Drew McDermott, *Introduction to Artificial Intelligence*, Addison Wesley, Reading, MA, 1985.

Cla78. K.L. Clark, "Negation as Failure," in *Logic and Databases*, ed. J. Minker, Plenum Press, New York, 1978.

Huf82. Karen Huff and Victor Lesser, "KNOWLEDGE-BASED COMMAND UNDERSTANDING: An Example for the Software Development Environment," Technical Report 82-6, Computer and Information Sciences University of Massachusetts at Amherst, Amherst, MA, 1982.

Kau85. Henry A. Kautz, "Toward a Theory of Plan Recognition," TR162, Department of Computer Science, University of Rochester, July, 1985.

McC85. John McCarthy, "Applications of Circumscription to Formalizing Common Sense Knowledge," in *Proceedings from the Non-Monotonic Reasoning Workshop*, AAAI, Oct 1985.

Pol84. Martha E. Pollack, "Generating Expert Answers Through Goal Inference," PhD Thesis Proposal, Department of Computer Science, University of Pennsylvania, August 1983, January 1984. DRAFT

Sid81. Candace L. Sidner and David J. Israel, "Recognizing Intended Meaning and Speakers' Plans," *IJCAI*, 1981.

Ten86. Josh D. Tenenburg, "Reasoning Using Exclusion: An Extension of Clausal Form," TR 147, Department of Computer Science, University of Rochester, Jan 1986.

Wil83. Robert Wilensky, *Planning and Understanding*, Addison-Wesley, Reading, MA, 1983.

All84a. James F. Allen, Mark Giuliano, and Alan M. Frisch, "The HORNE Reasoning System," TR 126 Revised, Computer Science Department, University of Rochester, Sept 1984.

A Formal Logic of Plans in Temporally Rich Domains

RICHARD PELAVIN AND JAMES F. ALLEN

This paper outlines a temporal logic extended with two modalities that can be used to support planning in temporally rich domains. In particular, the logic can represent planning environments that have assertions about future possibilities in addition to the present state, and plans that contain concurrent actions. The logic is particularly expressive in the ways that concurrent actions can interact with each other and allows situations where either one of the actions can be executed, but both cannot, as well as situations where neither action can be executed alone, but they can be done together. Two modalities are introduced and given a formal semantics: HNEV expresses simple temporal possibility, and IFTRIED expresses counterfactual-like statements about actions.

I. Introduction

This paper presents a formal logic that provides a foundation for a theory of plans in temporally rich domains. Such domains may include actions that take time, concurrent actions, and the simultaneous occurrence of many actions at once. This also includes domains with external events, i.e., actions by other agents and natural forces, that the planner may need to interact with in order to prevent some event, insure the successful completion of some event, or to perform some action enabled by the event.

In general, a planning problem can be specified by giving a description of desired future conditions (the goal), a partial description of the scenario in which the goal is to be achieved (which we will call *the planning environment*), and for each action that the planning agent can execute, a specification of its effects and the conditions under which it can be executed (which we will call *the action specifications*). The planner must find a collection of temporally related actions (the plan) that can be executed and if executed would achieve the goal in any scenario described by the planning environment. For example, consider a goal to go to the bank before it closes at 3:00 without getting wet even though it is going to rain. The planning environment might be the following: at 2:30, the time of planning, the agent is at home, an umbrella is in the house, it will start to rain before 3:00, and the agent will get wet if it is outside without an umbrella while it is raining. The actions that the agent can perform might include "walking from home to the bank" which takes 10 min and "taking an umbrella" which can be done as long as, prior to execution, there is an umbrella at the same location as the agent.

Our formal language must be able to express sentences describing the planning environment, the goal, and the action specifications. We will see that this logic diverges from traditional approaches because we are considering planning problems where the world may be affected by events other than the agent's actions. Such a logic must allow us to represent statements about external events that will occur during plan execution and statements describing the interaction between a plan and the external world in which it is being executed. In addition, we are considering plans with concurrent actions and, therefore, our logic must be able to represent concurrent actions and their interactions, such as resource conflicts. By treating concurrency, we will be able to reason about a robot agent that has multiple effector devices that can be operating simultaneously. Furthermore, we can treat plans to be jointly executed by a group of cooperating agents that are simultaneously performing their own part of the plan.

Situation Calculus and the State-Based Planning Paradigm

One of the most successful approaches to representing events and their effects in Artificial Intelligence has been situation calculus [12]. In this logic an event is represented by a function that takes a situation, i.e., an instantaneous snapshot of the world, and returns the situation that would result from applying the event to its argument. Events can be combined to form sequences. The result function for the sequence $e_1; e_2; \cdots; e_n$ is recursively defined as the result of executing $e_2; \cdots; e_n$ in the situation that results from applying e_1 to the initial situation.

Situation calculus has given rise to the state-based planning paradigm which has the following form: Given a set of sentences describing conditions that are initially true and a set of sentences describing goal conditions to be achieved, a sequence of actions must be found that when applied to any situation where the initial conditions hold yields a situation where the goal conditions hold. This approach is limited since the description of the planning environment con-

Manuscript received May 1, 1985; revised April 1, 1986. This research was supported in part by the Rome Air Development Center under Grant SU 353-9023-6 and in part by the National Science Foundation under Grant DCR-8502481.

The authors are with the Department of Computer Science, The University of Rochester, Rochester, NY 14627, USA.

0018-9219/86/1000-1364$01.00 © 1986 IEEE

sists only of statements about the initial situation, i.e., the situation that holds just prior to plan execution. As a result, this framework is adequate only for planning problems where all changes in the world result from the planning agent's actions. Conditions that are true in the initial situation will remain true until the agent performs an action that negates it, and thus the future is uniquely determined by the initial situation and the sequence of actions performed starting from this situation.

This simple type of planning environment is not suitable for planning problems where the world may be affected by external events, as well as by the planning agent's actions. Examples of conditions that we want to handle that cannot be represented in the simple state-based model include:

- The bank is going to open at 9:00 and will close at 3:00.
- It will possibly start raining any time between 3:00 and 4:00.
- If the agent is outside without an umbrella while it is raining, the agent will get wet.
- It is possible that the can of paint sitting in the doorway will be knocked over if the agent does nothing about it.

The type of goals considered in state-based planning are also very limited. Goals are just conditions that must hold at the completion of plan execution. In this research, goals may be any temporally qualified propositions describing conditions in the future of planning time. Thus goals might involve avoiding some condition while performing some task, preventing an undesirable condition that possibly will happen, or the achievement of a collection of goals to be done in some specified order. Examples of these types of goals are:

- Do not damage the tape heads while repairing the tape deck.
- Preventing Tom from entering the room.
- Get to the gas station on the way to driving to school (ordered goals).

More precisely, goals are temporally qualified propositions that partition the set of possible futures into the set of possible futures where the goal holds versus the set of possible futures where the goal does not hold. This rules out goals such as "finding the best way to achieve . . ." which presupposes some utility measure or precedence ordering relating the possible futures. Our notion of goals are just black and white. Either a future is good, i.e., the goal holds in it, or it is bad, i.e., the goal does not hold in it.

Finally, in the state-based approach, the type of plans that can be handled are limited to sequences of actions to be executed in the initial situation. Therefore, plans containing concurrent actions and plans that have an execution time that starts later than the initial situation are not treated. These restrictions come about because it is impossible to represent concurrent actions in situation calculus. Secondly, in situation calculus planning problems, there is no need to treat plans with execution times that will start later than the initial situation. Since in this paradigm all changes in the world are due to the planning agent, it makes no difference as to whether a plan is immediately executed or whether it is executed at a later time; the world will not change until the plan is started.

Since we will be handling plans with concurrent actions, our logic must represent the interactions between concurrent actions, some examples being:

- There is only one burner working and only one pan can be placed on it at one time. Thus only one dish can be cooked at a time.
- The agent can always carry one grocery bag to the car, but can only carry two when it is not icy out.
- The agent can move forward at any time, move backward at anytime, but cannot do both simultaneously.

Automated Planning Systems

Most domain-independent planners are essentially state-based planners, or limited extensions of the state-based approach. All these systems model actions as functions that transform one instantaneous state into another. The ancestor of all these systems is the STRIPS planner [6].

The type of planing problems handled by STRIPS is exactly what we have described as the state-based planning paradigm. The major contribution made by STRIPS is the so called "STRIPS assumption" to handle the frame problem, that is, the problem of representing that an action only effects a small part of the world. Thus if situation s_1 is the result of applying action a_1 in situation s_0, then s_0 and s_1 will be very much alike. In STRIPS, an action is defined as ordered triplet consisting of a precondition list, add list, and delete list, each member of these lists being atomic formulas. An action may be applied in any state s where all its preconditions hold yielding a new state that is computed from s, by adding only the formulas on the add list, and deleting all the formulas on the delete list. Implicit in this treatment is that any formula that is not explicitly asserted in a state is taken to be untrue. This allows one to avoid explicitly specifying negated atomic formulas.

Also in this class are the nonlinear hierarchical planners descending from NOAH [16]; the fact that they are nonlinear and hierarchical does not make them any more expressive than STRIPS since these terms refer to the control strategy, rather than the representation of plans. While one might argue that the notion of a procedural network gives us added representational power since it embodies a notion of hierarchical plans that are partially ordered, formally they are just descriptions of the set of linear sequences formed by atomic actions that can be decomposed from the hierarchical descriptions and meet the partial ordering.

There have been a number of domain-independent planning systems that have handled a larger class of planning problems than STRIPS. Wilkins' SIPE [20] and Vere's DEVISER [19] are two of the most sophisticated types. SIPE can handle plans with concurrent actions. The collection of actions making up a plan are only partially ordered. Any two actions where one is not ordered before the other are considered to be in parallel branches. Wilkins introduced the notion of *resources* to reason about the interaction of parallel actions. A resource is defined as an object that an action uses during its execution. If two actions share the same resource, they cannot be executed in parallel. Thus ordering constraints are imposed (if possible) to insure that any two actions that share a resource are not in parallel branches.

Wilkins points out that although resources used in his system are very useful, they are still quite limited in that

they do not allow one to treat things such as money or computational power as resources. They also cannot be used to encode actions that conditionally interact, such as the example we posed earlier where the agent can carry two bags only if it is not icy out.

Vere's DEVISER system can represent a planning environment where there are assertions about future events. In this system, a time line is modeled by the nonnegative real numbers, zero being the time of planning and positive real numbers being future times. The system can represent that an event not under the control of the planning agent will occur starting at some specified time and ending at another time. One can specify that some goal condition must hold between two time points. Vere calls the time points in which a goal is to be achieved the goal's (time) window. A goal can be the conjunction of two or more conditions to be achieved simultaneously within one window, or conditions to be achieved at different times, thus each condition has its own specified window.

Vere's treatment of time is in terms of an absolute scale and does not provide a general representation allowing relative temporal orderings. For example, it does not handle goals such as: get to the gas station before getting to school. Furthermore, there is no way to represent that two actions must be nonoverlapping. If two actions, a_1 and a_2 in a plan being constructed must be nonoverlapping, one of the actions must be constrained to end before the other begins. In many cases, the choice as to whether a_1 is constrained to be earlier than a_2, or a_2 is constrained to be earlier than a_1, must be made arbitrarily. If a bad choice is made, the system would have to backtrack. The treatment of external events is also limited. One cannot represent that some external event will start any time between 2:00 and 2:15, the exact start and finish time must be given.

Both Vere's system and Wilkins' system can be viewed as *ad hoc* extensions to the simple state-space planning paradigm. They use a conception of actions that arises directly from situation calculus; namely, that actions transform one state into another. This is manifested in that both systems specify actions by giving what is essentially precondition-add-delete lists and using a STRIPS-like assumption to handle the frame problem. It will be argued that the STRIPS assumption is inappropriate for any planning system that treats either external events or plans with concurrent actions. By explicitly designing a logic that represents concurrent events, we have arrived at a conceptually different way of looking at the frame problem and specifying frame assumptions. This will be described later after the logic is introduced.

An Overview of the Approach

Recently, Allen [3] and McDermott [13] have presented logics of events and time where events are not simply treated as functions from state to state. In both these treatments, a global notion of time is developed that is independent of the agent's actions. In Allen's logic, there are objects that denote temporal intervals which are chunks of time in a global time line. An Event is equated with the set of temporal intervals over which the change associated with the event takes place. Thus there is a notion of what is happening while an event is occurring. Secondly, there may be a number of events that are occurring over the same interval. Therefore, one can treat concurrent actions by asserting that two actions occur over intervals that overlap in time.

McDermott also equates an event with the set of intervals over which the event is occurring, but his notion of intervals is slightly different. While Allen's logic can only express statements about what is actually true, McDermott's logic is based on a world model where there may be many possible futures realizable from a particular world-state (an instantaneous snapshot of the world). A world model consists of a collection of world-states and a "future relation" that arranges the world-states into a tree-like structure that branches into the future. Each branch, which McDermott calls a chronicle, represents a possible complete history of the world extending infinitely in time. A global time line is associated with the real number line and each world-state is mapped to its time of occurrence. Because world-states in different chronicles may map to the same time, one cannot equate the intervals over which events occur with temporal intervals in the global time line. Specifying that an event occurs over a temporal interval does not tell which chronicles it occurs in. Instead, intervals are associated with totally ordered convex sets of world-states (with respect to the future relation). In other words, intervals are contiguous blocks of world-states that lie along some chronicle.

Both logics can be used to describe the outside world, but without extension neither can be used to represent what can and cannot be done by the planning agent. This is essential for a planning system where one must reason about the effects of the various actions that can be executed.

Our formal logic of plans is based on Allen's temporal logic, extended with a modality to express what the agent can and cannot do and a modality that represents future possibility. We introduce a new type of object called a plan instance that refers to an action at a particular time done in a particular way. These objects are discussed in the next section. Following this we give a brief description of Allen's logic and show why it must be extended with these two modalities. We first extend Allen's logic with the inevitability operator arriving at a logic that is similar to McDermott's [13] and Haas's [9] and show why this is insufficient to represent what can and cannot be done. We then introduce a modality called IFTRIED that represents what can and cannot be done. IFTRIED expresses counterfactual-like statements of the form "if the agent were to attempt to execute plan instance *pi* then *P* would be true. Examples are presented illustrating how different types of goals may be represented. This is followed by a section that describes the conditions under which two plan instances, concurrent or not, can be jointly executed and how plan instance interactions, such as resource conflicts, may be represented. We then present a simple planning example which leads into a discussion on how the frame problem manifests itself in our logic. Finally, we give the semantic model and interpretation for our two modalities.

The following notational conventions will be used throughout the rest of the paper. Sentences and terms in our formal language, which is a quantified modal language, will be given in LISP-type notation. For example, the formula (*P a*) refers to the unary predicate *P* with argument *a*. All variable terms will be prefixed with a "?." We will assume that all free variables in a statement are implicitly bound by a universal quantifier. The logical connectives

will be specified by: *AND* for conjunction, *OR* for conjunction, *IF* for material implication, *IFF* for equivalence, ∀ for the universal quantifier, and ∃ for the existential quantifier. The formula $(= t_1 t_2)$ will be used to mean that t_1 and t_2 denote the same object and $(\neq t_1 t_2)$ will be used to mean that t_1 and t_2 denote distinct objects. When we discuss the semantics for this logic in Section V, the functions and relations in the semantic model will be given in the conventional notation in logic (e.g., "$f(x)$," "$R(a, b)$," etc.).

II. Preliminaries

Plan Instances

In our theory we introduce objects called *plan instances* that refer to an action at a particular execution time done in a particular way. A plan instance's execution time is a temporal interval specifying the time over which the plan instance would occur if executed, not just a time point specifying the time that execution would begin. If we wanted to formally introduce plans, they would be functions from intervals to plan instances.

In situation calculus, one could get away with being vague as to whether an action refers to a behavior done in a particular way or whether it refers to a whole class of behaviors. This is because there is no notion of what is happening during the time of execution; as long as two particular behaviors have the same preconditions and effects they are indistinguishable in situation calculus. In a more general model, however, this distinction becomes conspicuous since one can represent what is happening while a plan instance is being executed. Consider a simple scenario where there are two paths of equal length going from location *A* to location *B*. Let us refer to these paths as P_1 and P_2. When talking about the effects of "go from *A* to *B* during interval *I*," one must be clear as to whether it refers to a particular way of going from *A* to *B* during *I* (i.e., whether it refers to going down path P_1, or going down path P_2), or whether it refers to taking either path during *I*. If it refers to a particular behavior, one can simply talk about its effects. If on the other hand, "go from *A* to *B* during *I*" refers to a class of behaviors, one must distinguish between saying "no matter how it is done, EFF will be true" and saying "there is a way that it is done such that EFF is true." In this example, it is correct to say that there is a way for an agent to do "go from *A* to *B* during *I*" such that this agent is on path P_1 during *I*, but incorrect to say that no matter how an agent does "go from *A* to *B* during *I*," this agent is on path P_1 during *I*. Plan instances are defined as "particular ways of doing something" since this is more primitive; later on, plan types referring to "classes of behaviors" can be introduced, defined in terms of plan instances.

Any two plan instances can be composed together to form a more complex plan instance. A composite plan instance occurs iff both its component parts occur. By composing plan instances, plan instances containing concurrent actions can be formed, as well as plan instances that are essentially sequences of actions, and plan instances that contain two components with gaps separating their execution times. In particular, plan instances that have concurrent actions can be formed by composing two plan instances that have execution times that overlap in time.

It is important to point out that a plan instance is not a complete specification of the agent's behavior over its time of execution. If this were so, a plan instance that was the composition of two distinct plan instances with overlapping times could never occur under any circumstances. For example, consider the plan instances "the agent is grasping object1 in its right hand during interval *I*" and "the agent is grasping object2 in its left hand during interval *I*." If plan instances are taken to be complete specifications then we could equivalently describe these plan instances as "during interval *I*, the only action the agent performs is grasping object1 in its right hand" and "during interval *I*, the only action the agent performs is grasping object2 in its left hand." Clearly, these two plan instances could never occur together and therefore their composition could never occur.

The STRIPS Assumption

Implicit in the STRIPS assumption is that actions are complete specifications of the agent's behavior over their time of execution (Georgeff [8] makes a similar point). This leads to problems in planning problems with concurrent actions and external events. We can paraphrase the STRIPS assumption as saying: if condition C holds at a time just prior to action a_1's execution, and a_1's effects do not negate C, then C will be true at a time immediately following a_1's execution. This presents a problem if we have two concurrent actions and we apply the STRIPS assumption to them separately. Suppose at a particular time t_0, P_1 is true. Consider two actions, a_1 and a_2, having the same duration and both having the preconditions that P_1 is true. Let the effects of a_1 be that P_1 is negated and the effects of a_2 be that P_2 is made true. Using the STRIPS assumption to compute the effects of executing a_1 starting at t_0, we get that P_1 is negated at a time immediately following a_1's execution which we will call time t_2. Similarly, using the STRIPS assumption to compute the effects of executing a_2 starting at t_0, we get that both P_1 and P_2 are true at t_2, the time immediately following a_2's execution. This, of course, is a contradiction, P_1 and its negation cannot both hold at the same time (i.e., at t_2).

Problems also arise if the STRIPS assumption is applied in a planning environment where there are external events. For example, suppose that at time t_0 it is asserted that it is raining outside. Consider action a_1 that can be applied at time t_0 and if it is executed will complete at time t_2. If we assume that it is out of the agent's control as to whether or not it is raining, the effects of a_1 will not negate the condition "it is raining." Thus by the STRIPS assumption, if a_1 is executed starting at time t_0, the result is that it is raining out at time t_2. This is unacceptable since, independent of the agent's actions, it might stop raining sometime before t_2.

Automated planning systems that use the STRIPS assumption avoid the above problems by restricting the type of planning problems that can be handled. Clearly, the "concurrent action problem" does not arise if plans are linear sequences of actions. If plans are treated as sets of partially ordered actions, the concurrent action problem can be avoided by making the assumption that the plan will be linearized before execution. Wilkins' system [20] does not make this assumption; if two actions are not ordered then it is possible that they will be executed in parallel. He, therefore, had to introduce the "resource mechanism" to handle conflicts between concurrent actions. As previously men-

tioned, this mechanism only handles interactions of the form: action a_1 and action a_2 share the same resource. There is no general notion of parallel action interactions.

Clearly, the "external event problem" does not arise if we are planning in a world where all changes are caused by the planning agent. On the other hand, in a system such as Vere's [19] that represents external events, one must be careful in specifying the planning environment. All external events that affect the value of any external property (i.e., a condition out of the agent's control) that is specified in the initial situation must also be included in the planning environment description. For example, if it is asserted that "the bank is open" holds in the initial situation, it is necessary to include the external event that negates this property at the time when the bank is closing. If this is not done, we would get the spurious result that the precondition that the bank is open is always satisfied.

To trace the root of the problem with the STRIPS assumption, let us look at a state-based representation, such as situation calculus, for which the STRIPS assumption was originally intended. Implicit in any state-based representation is that each action is a complete specification of what goes on from one situation to the next. One could re-interpret $s_1 = \text{RESULT}(a_1, s_0)$ (i.e., s_1 is the result of applying action a_1 in s_0) as saying that the result of doing only a_1 in s_0 results in situation s_1. The fact that most things do not change in going from s_0 to s_1 is not really due to the fact that a_1 is done, but instead to the fact that all actions other than a_1 are not done. Most things stay the same because of these nonoccurrences. That is, given any property p holding in situation s_0, there is a set of actions $\{a_{p_1}, \cdots, a_{p_n}\}$ (possibly empty) that make this property false upon execution in situation s_0. If a_1 does not belong to this set then p will also hold in the situation $\text{RESULT}(a_1, s_0)$ due to the fact that none of the actions in $\{a_{p_1}, \cdots, a_{p_n}\}$ are executed.

Thus the problem seems to be that the conclusion made by the STRIPS assumption is attributed to the execution of the action, not to the nonoccurrence of any action that can negate p. In effect, the STRIPS assumption is hiding the real reason why a property remains true from situation to situation. We will explicitly treat plan instances that refer to nonoccurrences and do not have to resort to a STRIPS assumption. In a later section, we discuss how the frame problem manifests in our logic and show the use of treating nonoccurrences. Georgeff [8] also presents an approach for handling the frame problem without appealing to the STRIPS assumption. His approach is similar to ours only in the fact that he also treats actions as partial specifications over their time of occurrence (although he does not describe it in these words).

Temporal Intervals, Properties, and Event Instances

The starting point for our logic is the treatment of action and time described in [3]. This logic is cast as a sorted first-order logic with terms denoting temporal intervals, events, properties (static conditions that hold or do not hold over intervals), and objects in the domain. We will slightly modify this theory by considering other event instances which refer to an event at a particular time. Thus we start with a sorted first-order language with terms denoting temporal intervals, event instances, plan instances (which also belong to the event instance sort), properties, and objects in the domain. A small number of predicates are introduced to specify the temporal relation between intervals, to specify that a property holds over an interval, and to specify that an event instance occurs. We also introduce symbols to denote the function that specifies an event instance's time of occurrence and the function that composes two plan instances. We now give the syntax the fragment which we will refer to as interval logic.

There are thirteen different ways in which two intervals can be temporally related. These are described in detail in [1] and so will not be repeated here. In this paper we will just introduce interval relation predicates as needed as they come up in the examples. Two interval relations that will be used frequently are: (PRIOR i_1 i_2) which means the interval denoted by i_1 is before or immediately precedes (meets) the interval denoted by i_2 and (ENDS-BEFORE i_1 i_2) which means that the interval denoted by i_1 ends before or at the same time as the interval denoted by i_2. The HOLDS predicate is used to assert that a property holds over some interval. The formula (HOLDS p i) means that the property denoted by p holds over the interval denoted by i. It is a theorem that if a property holds over an interval, then that property holds over any interval contained in this interval. The OCC predicate is used to specify that an event instance occurs. The formula (OCC ei) means that the event instance denoted by ei occurs. There is no need to say that it occurs over some interval since there is a time of occurrence associated with each event instance. We will use terms of the form "$e@i$" to refer to an event instance or plan instance whose time of occurrence is denoted by i. The function term (TIME-OF ei) denotes the time of occurrence associated with the event instance denoted by ei, thus we have (= #TIME-OF $e@i$ i)). Finally, the function term (COMP pi_1 pi_2) denotes the plan instance composed of the plan instances denoted by pi_1 and pi_2. This is the plan instance that occurs iff its component parts both occur together. The time of occurrence of a composite plan instance is the smallest interval that contains both its components' times of occurrence. Throughout the rest of this paper we will be more cavalier in treating the use-mention distinction and, for example, will use "plan instance pi" in place of "the plan instance denoted by pi."

In laying out this theory of time, Allen did not distinguish an actual time and as a consequence there is no formal notion of the present, past, or future in this theory. For simplicity, we will assume that the specification of the planning environment includes the designation of the interval that represents the time of planning. Throughout the rest of this paper we will use "future plan instance" to refer to a plan instance with an execution time later than the time of planning, use "past condition" to refer to conditions that hold earlier than the time of planning, use "current time" to mean "the time of planning," etc.

Allen's treatment of time can be characterized as a linear time logic. Temporal assertions are only about what is actually true; there is no notion of what will possibly happen or what possibly happened. When planning, the agent must be able to reason about how its actions can bring about different future possibilities. Thus a logic to be used for planning must be able to represent some form of possibility. We will extend interval logic to achieve this end.

III. A Logic for Planning

We extend interval logic by introducing a temporal modality, INEV, referred to as the inevitability operator. The modal statement (INEV *i P*) means that at interval *i*, statement *P* is inevitable, or equivalently, regardless of what possible events occur after *i*, *P* is true. If intervals i_1 and i_2 finish at the same time, then (INEV i_1 *P*) is true iff (INEV i_2 *P*) is true. This is because the same events that are in the future of i_1 are in the future of i_2. The possibility operator POS is the dual of INEV for a fixed time. Thus it is defined in terms of INEV which is given by:

(POS *i P*) = $_{def}$(NOT (INEV *i* (NOT *P*))).

Theorems involving INEV are given in Fig. 1. INEV1 can be roughly restated as saying if it is possible that property

```
INEV1)
   (IF (ENDS-BEFORE ?h ?i)
       (IFF (POS ?i (HOLDS ?p ?i:h))
            (INEV ?i (HOLDS ?p ?i:h))))

INEV2)
   (IF (ENDS-BEFORE (TIME-OF ?e:) ?i)
       (IFF (POS ?i (OCC ?e:))
            (INEV ?i (OCC ?e:))))

INEV3)
   (IF (PRIOR ?i1 ?i2)
       (IF (INEV ?i1 P)
           (INEV ?i2 P)))

INEV4)
   (IF (INEV ?i (IF P Q))
       (IF (INEV ?i P) (INEV ?i Q)))

INEV5)
   (IF (INEV ?i P) P)

INEV6)
   (IF (INEV ?i P)
       (INEV ?i (INEV ?i P)))

INEV7)
   (IF (POS ?i P)
       (INEV ?i (POS ?i P)))

where P and Q are any sentences in the logic with no free occurrences of ?i
```

Fig. 1. The axiomatization of INEV.

p held in the past or holds in the present, then it is inevitable that property *p* held in the past or holds in the present. INEV2 makes a similar claim about event instances with past or present times of occurrence. INEV3 captures the fact that if a statement is inevitable at time *i*, then it is inevitable at all later times. INEV4 says that material implication distributes out of INEV. INEV5 says that what is inevitable at any time is actually true. INEV6 and INEV7 stem from the fact that INEV is an S5 modal operator for a fixed time point. We also have the rule of inference: if *P* is a theorem then (INEV I_p *P*) is a theorem, for all interval terms I_p.

An important property of the INEV operator is: if interval logic statement IL_0 is entailed by the interval logic statements $IL_1 \cdots IL_m$ then (INEV *i* IL_0) is entailed by (INEV *i* IL_1) $\cdots$ (INEV *i* IL_n) for any interval term *i*. This is very useful when doing proofs in our planning examples. Many times, we will have interval logic statements nested within an inevitability operator and must be able to derive other statements of the form: (INEV *i IL*) where *IL* is an interval logic statement. Haas [9] who has an operator similar to INEV elaborates on this argument and claims that most of the modal reasoning needed to do planning (in his system) requires only first-order theorems applied within some modal context. The same argument can be used in our case.

Interval logic extended with the INEV operator can be viewed as a variant of a future branching time logic. These are temporal logics in which one can make statements about future possibility. Typically, these are modal languages with semantic models that consist of a collection of world states (i.e., instantaneous snapshots of the world) arranged in a tree-like structure that "branches into the future." Each branch is a possible history of the world, that is, a totally ordered set of states extending infinitely in time. The set of branches passing through some world-state are all the possible futures realizable from this world-state.

The logic consisting of interval logic and the INEV operator, however, does not have instantaneous world-states. Instead, the semantic model contains world-histories which are complete histories of the world extending throughout time and in the object language, there are temporal interval terms that refer to particular times in a world-history. All statements are interpreted with respect to a world-history. The truth value of a nonmodal statement (i.e., an interval logic statement) at world-history h_0 is only dependent on the event instances that occur and properties that hold in h_0. Thus interval statements are about what is actually true. The truth value of a modal sentence at h_0 depends on world-histories that are accessible from the h_0. This will be discussed in detail in the last section of this paper. Suffice to say that the accessibility relation in terms of which we interpret INEV relates world-histories that have a common past.

Interval logic augmented with the INEV modality can represent statements describing what is inevitable at time *i*, what is possible at time *i*, and also, what is actually true. This poses a slight problem. What does it mean to say that some course of events will possibly happen while also asserting that another course of events will actually happen? One answer to this is that possibilities are what could have happened. If this is the case, why should a planner worry about something that is possibly true if it is asserted that it is actually false?

Thomason [8] describes a formal technique for avoiding the above problem (although his motivation for developing the machinery is different from ours). We will not present this method here, but will do something that for our purposes is equivalent. We will assume that the description of the planning environment consists entirely of modal statements. Thus there will be no assertions about something that is actually false but possibly true. In solving a planning problem we will be looking for a plan instance *pi* such that at time *i*, it is inevitable that *pi* achieves the goal, not simply possible that *pi* achieves the goal.

Before going on, we should note that a term denotes the same object in all possible worlds, thus if two terms are actually equal (unequal) then at all times it is inevitable that they are equal (unequal). Secondly, if a temporal relation between two intervals is actually true, then at all times it is inevitably true. This is because intervals refer to chunks of time in a global time line. Thus their temporal relationship

is invariant over different possible worlds. If intervals did not refer to a global time line, we could not talk about different ways the world could have been at some particular time. As a consequence, it is not necessary to embed equality statements, i.e., formulas of the form $(= t_1\ t_2)$ and $(\neq t_1\ t_2)$, and statements that temporally relate two intervals, i.e., formulas such as (PRIOR $i_1\ i_2$) and (ENDS-BEFORE $i_1\ i_2$), within the POS or INEV modality.

Achieving a Goal

Consider a planning problem where at planning time, which we will denote by I_p, we want to achieve goal G, where G is an interval logic statement describing desired future conditions. A future plan instance must be found that achieves the goal under all possible future conditions as described by the planning environment. Thus we are looking for a future plan instance *pi* such that in any possible future where *pi* occurs, G is also true. This is equivalent to saying that it is inevitable (at I_p) that if *pi* occurs then G is true:

(INEV I_p (IF (OCC *pi*) G)).

One might argue that it is impossible to find a plan instance that works under all possible circumstances, but this is not our objective. We are just looking for a plan instance that works assuming that the planner's view of the world (i.e., the planning environment and the action specifications) is correct. Whether the planning environment describes a great number of possibilities, or just a few of the very likely ones is not of immediate concern here.

If at time I_p, it is inevitable that *pi* does not occur (which would be the case if *pi* was an "impossible plan instance," i.e., one that never can be executed under any circumstance), the above statement would vacuously hold. Therefore, any plan instance *pi* under consideration must also meet the condition that there is a possible future where it occurs. This is simply stated as:

(POS I_p (OCC *pi*)).

This condition, however, does not insure that *pi* can be executed regardless of possible circumstances out of the agent's control or guarantee that *pi* contains all the steps needed for execution.

Consider the following planning problem. The goal is to get into a particular room sometime during the interval I_G. The plan instance WALK-IN-ROOM@I_w refers to the action of walking through the doorway into the room during interval I_w, where we are assuming that I_w ends at a time during I_G. In order to perform this plan instance the door must be unlocked at a time just prior to I_w. Let us suppose that it is possible the door is locked at this time and possible that it is unlocked at this time. Also assume that it is impossible for the robot to perform an action to unlock the door (or to get someone else to unlock the door), if it happens to be locked.

In this scenario, it is possible that the plan instance WALK-IN-ROOM@I_w occurs, thereby achieving the goal, since it is possible that the door is unlocked. We would not, however, be satisfied with such a plan instance, since whether or not it can be done is dependent on a condition out of the agent's control; namely, whether or not the door happens to be locked. What is needed is a plan instance that can be executed under any possible circumstances (as described by the planning environment) out of the agent's control. Thus in the above example, we would be looking for a plan instance that can be executed regardless of whether the door is locked or not.

A logic that represents the above example must be able to represent statements such as: "it is out of the agent's control as to whether or not the door is locked" and to express conditions such as: "under all possible external conditions, plan instance *pi* can be executed." Interval logic augmented with the INEV operator is insufficient in itself to represent these statements. The INEV operator may be used to represent that some future condition is possible, but cannot attribute the cause of the possibility to external factors, the agent's actions, or a combination of both factors.

Finding a plan instance that can be executed under all possible external circumstances, however, is still not sufficient. An even stronger condition that must be satisfied is that the plan instance contains all steps needed for execution (with respect to what is possible at planning time). This can be clarified by the following example. During planning time, which we will denote by I_p, the agent is standing by a locked safe and by a table on which the safe's key is resting. The goal is to open the safe (at some time in the near future). The agent can perform the plan instance OPEN-SAFE@I_o as long as it has the key grasped in its hand just prior to execution time I_o. The agent can also perform GRASP-KEY@I_g which corresponds to grasping the safe's key at a time immediately after planning time and results in the key being grasped just prior to I_o. This plan instance can be performed as long as the agent is by the table on which the key is resting just prior to execution time (I_g). In this example, we assume this condition happens to hold. Thus it is inevitable at planning time that this condition holds since the present is inevitable.

In this case, it is under the agent's control to enable the conditions under which OPEN-SAFE@I_o can be executed. By executing GRASP-KEY@I_g, the agent can bring about the conditions under which OPEN-SAFE@I_o can be executed. Thus it is possible that OPEN-SAFE@I_o occurs. We would not, however, want the planner to simply return that OPEN-SAFE@I_o achieves the goal, leaving out that it must be done in conjunction with some other plan instance. Instead, we would want the planner to return a plan instance such as (COMP GRASP-KEY@I_g OPEN-SAFE@I_o). At planning time I_p, this composite plan instance contains all the steps needed for execution since the conditions under which GRASP-KEY@I_g can be executed are inevitably true (at I_p) and it is inevitably true that OPEN-SAFE@I_o can be executed if GRASP-KEY@I_g is also executed. If we had a different scenario where the condition "the agent is grasping the key just prior to interval I_o" was inevitable, we would have concluded that OPEN-SAFE@I_o could simply be executed alone.

In describing the plan instances in both examples, we gave conditions under which each of them can be executed. If it is possible at planning time I_p that the conditions under which plan instance *pi* can be executed will not hold, and it is not in the agent's control to bring about these conditions, then we do not want to conclude that at time I_p, *pi* can be executed. Even if these conditions can be made true by executing some other plan instance, we consider a plan

instance containing *pi* to be under specified unless it also contains a plan instance that enables these conditions. Thus in planning to achieve a goal *G* at time I_p, we are looking for a plan instance *pi* such that i) it is inevitable at I_p if *pi* occurs then *G* is true and ii) it is inevitable at I_p the conditions under which *pi* can be executed hold.

To capture the notion of "conditions needed for execution," our theory will make a distinction between plan instance attempts and plan instance occurrences. The conditions under which *pi* can be executed will be equated with the conditions under which attempting *pi* leads to *pi* occurring. As an example attempting OPEN-SAFE@I_o might associated with moving one's arm during interval I_o in such a way that the key being grasped is twisted in the safe's lock (resulting in the safe being opened). This arm twisting movement could be done regardless of whether or not the agent was grasping the key, but only when it is done while the agent is grasping a key, would we say that the agent is opening the safe with the key.

The IFTRIED Modality

We extend our language to include the modal operator IFTRIED. The sentence (IFTRIED *pi P*) is taken to mean that if plan instance *pi* were to be attempted then *P* would be true, where *P* is any sentence in our extended language. This is a counterfactual modality that is used to make assertions about what would result if plan instance *pi* were to be executed, not about what is actually true. Thus it is consistent to assert that (IFTRIED *pi P*) and (NOT *P*) are both true. We must also point out that both arguments to IFTRIED are temporally qualified. The first argument being a plan instance has an associated time of occurrence. The second argument is a sentence in the logic and thus is either an interval logic statement and hence is temporally qualified or is a modal statement containing (temporally qualified) interval logic statements.

The IFTRIED operator is related to INEV by the following axiom schema, which states that if a property *P* is inevitable at some time *t*, then it will remain true no matter what plan is attempted after *t*

IFTRIED1)
(IF (PRIOR ?*t* (TIME-OF ?*pi*))
(IF (INEV ?*t P*) (IFTRIED ?*pi P*))).

From this axiom schema and the two axioms, INEV1 and INEV2, stating that the past and present are inevitable, we get the desired result that attempting a plan instance has no effect on earlier properties and events.

The sentence (IFTRIED *pi* (OCC *pi*)) means that if plan instance *pi* were to be attempted then it would occur, or what we equivalently say: "*pi* is executable." For convenience, we will define the predicate EXECUTABLE in our object language by

(EXECUTABLE *pi*) $=_{def}$ (IFTRIED *pi* (OCC *pi*)).

At time I_p, a plan instance *pi* contains all the steps needed for execution iff in all possible futures, if *pi* were to be attempted it would occur. This is expressed by

(INEV I_p (EXECUTABLE *pi*)).

We will also introduce the notion of "choosibility" which can be expressed in terms of IFTRIED. We say that a condition *P* is choosible at time I_p iff there exists a plan instance with execution time after I_p which if attempted would result in *p* being true. The definition is given by:

(CHOOSIBLE I_p *P*) $=_{def}$
(∃ ?*pi* (AND (PRIOR I_p (TIME-OF ?*pi*))
(IFTRIED ?*pi P*)).

Saying that *P* is choosible at time I_p means that there is something the agent could have done (starting after I_p) to make *P* true. By using CHOOSIBLE, we can succinctly state that some condition is out of the agent's control and state that a condition can be made true regardless of external conditions. To express that at time I_p, the agent cannot affect whether or not proposition *P* is true, we state that it is inevitable at I_p that if *P* happens to come out true then it is not choosible at I_p that *P* is untrue, and similarly, it is inevitable at I_p, if *P* comes out to be untrue, then it is not choosible at I_p that *P* is true. For example, the following states that at all times, the agent has no control as to whether it is raining out:

(INEV ?*ie*
(AND (IF (HOLDS (raining) ?i_1)
(NOT (CHOOSIBLE I_p
(NOT (HOLD (raining) ?i_1))))
(IF (NOT (HOLDS (raining) ?i_1))
(NOT (CHOOSIBLE I_p
(HOLDS (raining) ?i_1)))))).

To express "regardless of possible external conditions after I_p, *P* can be made true," we state that regardless of what future possibilities arise, it is in the agent's control to make *P* true. This is expressed by

(INEV I_p (CHOOSIBLE I_p *P*)).

It is illustrative to compare the IFTRIED modality with Moore's RES operator [14] which is a modal operator that captures the result function in situation calculus. Moore equated possible worlds with situations allowing him to integrate a theory of action based on situation calculus with a theory of belief based on possible world semantics. For the comparison here, we will only talk about the RES modality.

Statements in Moore's language are interpreted with respect to a world at a particular time, not with respect to an entire world history as in our logic. Thus statements are about what is currently true and are not temporally qualified. The sentence (RES a_1 *P*) means that currently, action a_1 can be executed and if executed then *P* will be true where *P* is either a sentence in first-order logic, or contains modal operators. Now, it is important to note that neither argument to RES is temporally qualified. Secondly, RES is a "tense shift" operator. (RES a_1 *P*) is making an assertion about what will be true in a situation in the future of the current time. This contrasts to (IFTRIED *pi P*) where both its arguments are temporally qualified and they may have any temporal relation; *P* need not be about a time in the future of *pi*'s occurrence.

In [15] we show how the RES operator can be viewed as a special case of the IFTRIED modality. Very roughly, a situation calculus view of the world is modeled in interval logic by discrete intervals where intervals associated with situations are interleaved with intervals associated with action

occurrences. If we assume that interval I_0 denotes the current time, we can translate (RES a_1 P), where P is a nonmodal, to (IFTRIED a_1@i_1 (AND (OCC a_1@I_1) (HOLDS P I_2)) where we are assuming that i_2 immediately follows i_1 which immediately follows I_0.

IV. Examples

Representing Different Types of Goals

This formalism can easily express goals such as avoiding some condition while performing some task, achieving a collection of goals to be done in some specified order, and preventing an undesirable condition that possibly will happen. In this section, we will examine an example of each of these goals. The simplest example concerns a simple sequence of goals. Suppose at planning time I_p, the goal is to be at school at some time $I_{at\text{-}s}$, while stopping at the gas station on the way. A plan instance pi (in the future of planning time I_p) must be found such that the following holds:

```
(INEV I_p
   (AND (EXECUTABLE pi)
        (IFTRIED pi
          (∃ ?i(AND (HOLDS (at agt school) I_at-s)
                    (PRIOR ?i I_at-s)
                    (HOLDS (at agt gas-station) ?i)))))).
```

The above statements says: under *all* circumstances possible at time I_p, both pi is executable and if pi were to be attempted (and thus would occur since it is executable), the agent would be at school during $I_{at\text{-}s}$, and at the gas station prior to this. Similarly, avoiding some condition while achieving another is easily represented by specifying that some condition does not hold during the execution of the plan.

Prevention problems pose a number of problems as other authors [3], [9], [13] have noted. It does not make sense to say that an occurrence is prevented, if it is not possible in the first place. Furthermore, the reason why the occurrence is possible must not be due to the agent's actions alone. This last qualification has been overlooked by these authors. Consider a simple scenario where if an open paint can is sitting in the doorway and an agent happens to walk through the doorway, the paint can will be knocked over spilling the paint on the floor. For simplicity, we will only talk about a particular spilling occurrence at time I_{sp} which will occur iff it is immediately preceded by an occurrence of an agent walking through the doorway while the can is sitting in the doorway:

```
(INEV ?ie
   (IFF (OCC paint-spills@i_sp)
        (∃ ?i_wtd ?agt
           (AND
           (MEETS ?i_wtd i_sp)
           (OCC (walks-through-door ?agt)@?i_wtd)
           (HOLDS (position paint-can doorway) ?i_wtd))))).
```

The above statement says: at all times it is inevitable that paint-spills@i_{sp} occurs iff immediately prior to i_{sp} there is an agent that walks through the door while a can of paint is in the doorway.

Now, at planning time I_p, we want to find a plan instance pi that under all possible future conditions is executable and if it is attempted then paint-spills@i_{sp} will not occur:

```
(INEV I_p
   (AND (EXECUTABLE pi)
        (IFTRIED pi (NOT (OCC paint-spills@i_sp))))).
```

Even if the above statement is true, however, we cannot yet claim that the planning agent can prevent the occurrence of (spills paint)@i_{sp}. To begin with, it might be the case that is inevitable that the paint can is not in the doorway immediately prior to i_{sp} or inevitable that no one will walk through the doorway at a time immediately prior to i_{sp}. If either of the above holds, then it is inevitable that the paint can will not spill. Thus in order to conclude that we prevented the paint can from spilling, it must be the case that it possibly spills, that is, the following must be true:

```
(POS I_p (OCC paint-spills@i_sp)).
```

Even this is not sufficient to conclude that we prevented this occurrence. It might be the case that the only way that the paint can will be in the doorway is if the planning agent puts it there, or the only agent that possibly can spill the paint is the planning agent. To represent that paint-spills@i_{sp} possibly occurs because of external forces, we must introduce a special plan instance that corresponds to the planning agent being inactive over a period of time, which we will denote by do-nothing@i_{dn}. We then check if it is possible that paint-spills@i_{sp} occurs even if the agent does nothing up until the time when paint-spills@i_{sp} would complete:

```
(IF (AND (MEETS I_p ?i_dn)(ENDS-BEFORE i_sp ?i_dn))
   (POS I_p (IFTRIED do-nothing@?i_dn
                    (OCC paint-spills@i_sp)))).
```

Further discussion of these issues can be found in [15].

Composing Two Plan Instances and Plan Instance Interactions

One of the primary goals in developing this logic was to represent plan instance interactions and to provide a formal basis for determining which plan instances can be executed together. In state-based planners, the system determines which linear combinations of actions achieve the goal and which sequence of actions can be executed together. In these planners, the interactions of interest involve one action enabling another by bringing about the other's preconditions, or one action's effects interfering with another's preconditions. All these interactions, however, concern actions that are linearly ordered. In this section, we will consider the interactions between concurrent plan instances and show how to determine whether they can be executed together and if so under what conditions.

Let us first consider the relation between the conditions under which a composite plan instance is executable and the conditions under which each of its components are executable. What it means to say that a composite plan instance is executable is that if both components were attempted together then both components would occur. There are a number of cases to consider. It might be the case that both components are executable when taken alone, but they cannot be executed together since they interfere with each other. Such is the case if two plan instances share the same resource or if two plan instances are alternative

choices, one of which can be performed at one time. Thus it is incorrect to assert that if both pi_1 and pi_2 are executable, then (COMP pi_1 pi_2) is executable. The converse of this statement does not hold either. It might be the case that (COMP pi_1 pi_2) is executable while pi_1 is not because the occurrence of pi_2 brings about the conditions under which pi_1 is executable. It might also be the case that (COMP pi_1 pi_2) is executable, but neither pi_1 or pi_2 is executable alone. Such is the case if pi_1 and pi_2 are "truly parallel actions," ones that must be executed together. An example of this is where an object is lifted by applying pressure to two ends of the object, one hand at each end. If pressure were applied to only one end, the result would be a pushing action, not part of a lifting action.

A general theorem will be given that relates (COMP pi_1 pi_2) to its component parts, regardless of their temporal relations. Before presenting this general theorem, however, it will be clearer to first look at special cases concerning the relation between pi_1 and pi_2.

We begin by considering the case where pi_1 and pi_2 do not have overlapping execution times. Without loss of generality assume that pi_1 is before pi_2. Clearly, whether or not a plan instance occurs is not affected by the attempt of another plan instance with a later execution time. This is captured by the following theorem:

INTERACTION1)
(IF (PRIOR (TIME-OF ?pi_1) (TIME-OF ?pi_2))
(IFF (OCC ?pi_1)
(IFTRIED ?pi_2 (OCC ?pi_1))))

Since pi_1 is before pi_2, attempting pi_2 has no effect on whether or not pi_1 occurs. Therefore, if (COMP pi_1 pi_2) is executable then pi_1 must also be executable. It need not be the case, however, that pi_2 is executable. The execution of pi_1 may bring about the conditions under which pi_2 is executable. Alternatively, pi_2 may be executable, but it would not be if pi_1 were to occur. This provides justification for the following theorem:

INTERACTION2)
(IF (PRIOR (TIME-OF ?pi_1) (TIME-OF ?pi_2))
(IFF (EXECUTABLE (COMP ?pi_1 ?pi_2))
(AND (EXECUTABLE ?pi_1)
(IFTRIED ?pi_1 (EXECUTABLE ?pi_2)))))

This theorem says that for any nonoverlapping plan instances pi_1 and pi_2, pi_1 being the earlier one, the composite plan instance (COMP pi_1 pi_2) is executable iff pi_1 is executable and if pi_1 were to be attempted (and thus would occur because it is executable), then pi_2 would be executable.

If pi_1 is earlier than pi_2, but the two plan instances overlap in time, the consequent in the above theorem might not hold while the antecedent does; there are cases where the statements "pi_1 is executable" and "if pi_1 were to be attempted then pi_2 would be executable," are both true but their composition is not executable, since pi_2 interferes with the successful completion of pi_1. For example, consider the function term (walk home store)@l_w which denotes the plan instance where the agent walks from home to the store during interval l_w. This plan instance is executable as long as the agent is at home just prior to execution (i.e., at some time that immediately precedes l_w). The effects of this plan instance are, that the agent is outside during execution and is at the store at a time following execution. Consider also the plan instance (stay-at home)@l_s which refers to the action of staying at home during interval l_s. This plan instance is executable as long as the agent is at home just prior to execution and its effects are that the agent is at home for the duration of its execution. If we assume: i) (stay-at home)@l_s is executable, ii) l_s is earlier than but overlaps l_w, and iii) it is impossible to be at two places at once, then it follows that the earlier plan instance (stay-at home)@l_s is executable, and if the plan instance (walk home store)@l_w were attempted, it would be executable, but their composition is not executable since it is impossible to be outside and at home at the same time.

In general, if plan instance pi_1 is executable, but the conditions prohibit both pi_1 and pi_2 from occurring together, then we have the following:

(IFTRIED pi_1 (OCC pi_1))
(IFTRIED pi_1 (IFTRIED pi_2 (NOT (OCC pi_1))))

The first statement is simply the definition of (EXECUTABLE pi_1). The second statement says: If pi_1 were to be executed (and thus would occur), we would get to a scenario where if pi_2 were to be attempted, it would preclude pi_1 from occurring. In other words, pi_2 interferes with pi_1.

If, on the other hand, i) pi_1 is executable, ii) the conditions under which pi_1 and pi_2 can occur together hold, and iii) if pi_2 were to be attempted, pi_2 would be executable, then we would have the following:

(IFTRIED pi_1 (OCC pi_1))
(IFTRIED pi_1 (IFTRIED pi_2 (AND (OCC pi_1) (OCC pi_2))))

This can be read as saying that if pi_1 were to be attempted (and thus would occur since it is executable), we get to a scenario where attempting pi_2 would result in both pi_2 and pi_1 occurring. In this case, both plan instances can be performed together, and we want to conclude that the composite plan instance is executable.

In the general case, we have the following theorem:

INTERACTION3)
(IF (IFTRIED ?pi_1
(IFTRIED ?pi_2 (AND (OCC ?pi_1) (OCC ?pi_2))))
(IFTRIED (COMP ?pi_1 ?pi_2)
(AND (OCC ?pi_1) (OCC ?pi_2))))

This theorem says that if attempting plan instance pi_1 would result in a scenario where attempting pi_2 would result in both plan instances occurring, then their composition is executable (i.e., if it were attempted, both of its components would occur). Notice that there are no temporal restrictions relating pi_1 and pi_2. Secondly, this general theorem provides for the case where neither pi_1 nor pi_2 are executable alone, but they are executable together. This would be the case if attempting pi_1 alone would not result in pi_1 occurring, but the attempt would make it so that pi_2 is executable, which in turn would answer the conditions for pi_1's successful completion. There is, in fact, an even more general theorem about interaction that is not necessary for the purposes of this paper, which is discussed in [15].

Concurrent Plan Instance Interactions

In this section, a few examples are presented to illustrate how concurrent plan instance interactions may be repre-

sented and how these statements lead to conclusions about whether or not the composition of these plan instances are executable. We start with a simple resource conflict example. Consider a very simple case where there is a stove on which only one pan can be placed at a time. Let the function term (heating *pn*)@I_H denote the plan instance where the pan *pn* is being heated on the stove during the interval I_H. The fact that only one pan can be heated at a time is captured by:

HEAT1)
(INEV ?*ie*
(IF (AND (OCC (heating ?pn_1)@?i_{H1})
(OCC (heating ?pn_2)@?i_{H2}))
(OR (DISJOINT ?i_{H1} ?i_{H2})
(AND (=? pn_1?pn_2)(=?I_{H1} ?I_{H2})).

This says that under all possible circumstances (i.e., it is inevitable at all times), if there are two occurrences of a pan being heated on a burner then either their times of occurrence are not overlapping or they refer to the same plan instance (two function terms are equal if all their argument terms are equal). This relationship between plan instances is what Lansky [10] calls a behavioral constraint; it directly relates two plan instances (actions) instead of implicitly relating two plan instances by use of action precondition–effect lists.

Treating such an example with precondition–effect lists would be awkward and inefficient. We would have to introduce a property associated with "burner being used." The action, "heat pan *pn*" could not simply be modeled by a simple action specified by a precondition–effect list, instead it would have to be modeled by two simple actions that must be performed consecutively (note: Vere's system [19] has such a facility). The reason for this is that the effect of "heat pan *pn*" is that the burner is in use during execution, not before or after execution. We would then model this action by two consecutive actions a_1 and a_2, where a_1's effect is that the burner is in use and a_2's effects are that the burner is free. As previously noted, Wilkins' SIPE [20] has a special mechanism for treating a limited class of resource conflicts. This mechanism could be used to solve the above example without recourse to the "in use" properties.

In order for the behavioral constraint HEAT1 to be useful, it must lead to the deduction that a composite plan instance consisting of two overlapping plan instances using the same burner but different pans is not executable. This is easily shown. Let us consider plan instances (heating *pn*1)@I_{H1} and (heating *pn*2)@I_{H2} in the case where *pn*1 and *pn*2 denote different objects, and the intervals, I_{H1} and I_{H2}, overlap in time. We want to prove that the composite is not executable, i.e.,

PRV1)
(NOT (IFTRIED (COMP (heating *pn*1)@I_{H1}
(heating *pn*2)@I_{H2})
(AND (OCC (heating *pn*1)@I_{H1})
(OCC (heating *pn*2)@I_{H2})))))).

From axiom HEAT1, the fact that I_{H1} and I_{H2} overlap in time, and that *pn*1 and *pn*2 are unequal, it logically follows that PRV1 is true. A sketch of this proof is given in Fig. 2.

Our next example concerns two plan instances that cannot occur together if certain external conditions hold. Sup-

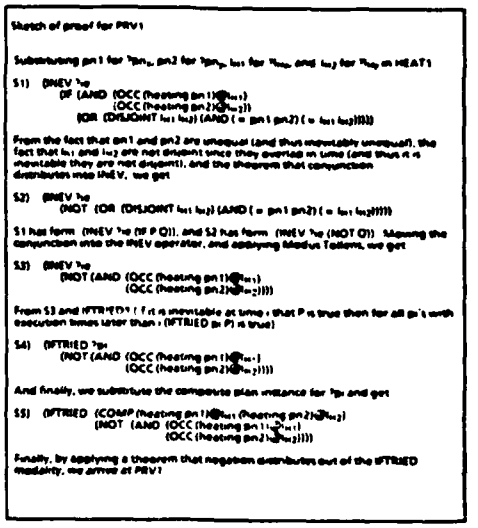


Fig. 2.

pose that the agent cannot carry two grocery bags from the supermarket to the car (without slipping) if it is icy out. Consider the scenario where at planning time I_p, it is possible that it is going to be icy out during I_{wt}, and also possible that it is not icy out during I_{wt}. This is represented by

ICY-ST1)
(AND (POS I_p (HOLDS (icy) I_{wt})
(POS I_p (HOLDS (not (icy)) I_{wt}))).

We also state that it is impossible for the agent to affect whether or not it is icy out:

ICY-ST2)
(INEV ?*ie*
(AND (IF (HOLDS (icy) ?*ii*)
(NOT (CHOOSIBLE ?*ie*
(NOT (HOLDS (icy) ?*ii*)))))
(IF (HOLDS (not (icy)) ?*ini*)
(NOT (CHOOSIBLE ?*ie*
(NOT (HOLDS (not (icy)) ?*ini*))))))).

Consider the two function terms, (carry *bag*1)@I_c and (carry *bag*2)@I_c, which refer to the plan instances where the agent takes each bag from the shopping cart and carries it to the car during interval I_c which we assume is during I_{wt}. Also assume that *bag*1 and *bag*2 denote distinct objects. Finally, assume that under all circumstances, if it is icy out during I_{wt}, then it is impossible that both plan instances occur together. This is represented by:

ICY-ST3)
(INEV ?*ie* (IF (HOLDS (icy) I_{wt})
(NOT (AND (OCC (carry *bag*1)@I_c)
(OCC (carry *bag*2)@I_c))))).

From ICY-ST3 above, the fact that it is possible that it is icy

out (ICY-ST1), and the fact that the agent cannot perform any action that would prevent it from being icy (ICY-ST2), we can prove that it is possible at I_p that there is nothing the agent can do to make (carry *bag*1)@I_c and (carry *bag*2)@I_c occur together.

PRV2)
(POS I_p
(NOT (CHOOSIBLE I_p (AND (OCC (carry *bag*1)@I_c)
(OCC (carry *bag*2)@I_c))))).

The sketch of the proof of PRV2 is given in Fig. 3.

Sketch of the proof of PRV2

PRV2) (POS I_p
(NOT (CHOOSIBLE I_p (AND (OCC (carry bag1)@Ic)
(OCC (carry bag2)@Ic)))))

We use the two following theorems which we give without proof

CH1 (IF (INEV ?i (IF P Q))
(INEV ?i (IF (NOT (CHOOSIBLE ?i (NOT P)))
(NOT (CHOOSIBLE ?i (NOT Q))))))

POS1 (IF (AND (INEV ?i (IF P Q)) (POS ?i P))
(POS ?i Q))

We first prove that it is possible (at I_p) that it is icy out (during I_{out}) and in this case the agent cannot choose to make it not icy out. Thus is derived using the first conjunct of ICY-ST1

S1) (POS I_p (HOLDS (icy) I_{out}))

using ICY-ST2 substituting I_{out} for ?i1 and I_p for ?i2, and the theorem that conjunction distributes out of INEV

S2) (INEV I_p (IF (HOLDS (icy) I_{out}) (NOT (CHOOSIBLE I_p (NOT (HOLDS (icy) I_{out})))))))

then applying POS1 substituting I_p for ?i, (HOLDS (icy) I_{out}) for P and (NOT (CHOOSIBLE I_p (NOT (HOLDS (icy) I_{out})))) for Q, giving us the desired result

S3) (POS I_p (NOT (CHOOSIBLE I_p (NOT @HOLDS (icy I_{out}))))))

Next, we prove that it is unavoidable (at I_p) that if it is not choosible that it is not icy out, then it is not choosible that we can carry both bags together. This is derived using ICY-ST3 substituting I_p for ?i2

S4) (INEV I_p (IF (HOLDS (icy) I_{out})
(NOT (AND (OCC (carry bag1)@Ic) (OCC (carry bag2)@Ic))))))

and then using CH1 substituting I_p for ?i, (HOLDS (icy) I_{out}) for P, and (NOT (AND (OCC (carry bag1)@Ic) (OCC (carry bag2)@Ic))) for Q, giving us the desired result

S5) (INEV I_p
(IF (NOT (CHOOSIBLE (NOT (HOLDS (icy) I_{out}))))
(NOT (CHOOSIBLE (AND (OCC (carry bag1)@Ic) (OCC (carry bag2)@Ic)))))))

Finally, from S3, S5 and POS1 substituting I_p for ?i and making obvious substitutions for P and Q, we prove prv2

Fig. 3.

This example could not be handled by existing nonlinear planners, because only two types of interactions can be handled: 1) If action a_1 interferes with action a_2's preconditions then either a_1 is ordered after a_2, or another action is inserted between a_1 and a_2 in order to restore a_2's preconditions, and 2) If two actions have effects that contradict each other, then one of the actions must be ordered before the other to avoid the possibility of trying to execute them together and thus failing. Now, in the example above, the plan instances, (carry *bag*1)@I_c and (carry *bag*2)@I_c, do not really contradict each other; the fact that both plan instances occur is consistent with the material implication stating that if it is icy out, then the two plans do not both occur. Thus a naive implementation would allow these two plan instances to simultaneously occur. From the fact that (carry *bag*1)@I_c and (carry *bag*2)@I_c both occur, and this material implication, we would conclude that it is not icy out during interval I_{out}. Clearly, this is not desired. Instead, we would want the planning system to conclude that the two plan instances can be executed together only if it happens not to be icy out during I_{out}.

This problem manifests itself differently in the case where the agent can bring about the conditions under which two plan instances can be executed together. Let us suppose that action a_1 and a_2 share the same type of resource and it is under the agent's control how many of these resources are present. Now, it is not a contradiction that a_1 and a_2 occur simultaneously. A naive implementation might return a plan where a_1 and a_2 are executed simultaneously, while failing to include a plan step that guarantees that two resources are present while a_1 and a_2 are being executed. Thus we get a plan that does not have all the steps needed for execution.

These problems can be avoided in a state-based system by explicitly introducing properties associated with resources in use, but as we have previously mentioned this becomes quite cumbersome and would lead to an inefficient search space. Properties and precondition lists are useful, however, for specifying the conditions under which actions taken alone can be executed. Thus an adequate representation needs to handle both behavioral constraints and properties that serve as preconditions. We have just discussed how behavioral constraints can be expressed. Saying that if property *pr* holds over interval *i*, then plan instances *pi*'s preconditions hold (i.e., *pi* is executable) is simply expressed by:

(INEV ?*ie* (IF (HOLDS *pr i*) (EXECUTABLE *pi*)).

Most theories can express only precondition properties. Lansky [10] describes a theory that allows behavioral constraints, but it is difficult to treat properties. To define a property one must know all the possible actions and then precompute the patterns of execution that result in the property being true and untrue.

A Simple Planning Problem

In this section, we present a simple planning example that leads into a discussion on how the frame problem manifests itself in our logic. Consider the following example. Suppose at planning time I_p, the planning agent, which we denote by *agt*, is at home and its goal is to get to the bank sometimes during I_C which is an interval that immediately follows I_p. The set of sentences that describe the planning environment are given by:

PE1) (MEETS I_p I_C)
PE2) (INEV I_p (HOLDS (*at agt home*) I_p)).

The interval logic statement describing the goal of getting to the bank at a time during interval I_C is given by

(∃ ?i_b (AND (DURING ?i_b I_C) (HOLDS (*at agt* bank) ?i_b))).

In the rest of this example we will use G to refer to this interval logic statement describing the goal conditions.

For our simple example, we introduce a class of plan instances that refer to walking from one building location to another. We will let the function term (walk *bldg*1 *bldg*2)@I_{walk} denote the action of walking from building *bldg*1 to building *bldg*2 during interval I_{walk}. The term will

denote a plan instance for all arguments, *bldg*1 and *bldg*2, that denote building locations and all interval terms I_{walk}, but unless the duration of I_{walk} is greater than or equal to the minimal time it takes the agent to walk from *bldg*1 to *bldg*2, it denotes an impossible plan instance. These are *plan instances that are never executable under any circumstances*. For simplicity, we ignore the case where *bldg*1 and *bldg*2 denote the same building or are so far apart that it is impossible to walk from one to the other. If we have two plan instances, (walk *bldg*1 *bldg*2)@I_{w1} and (walk *bldg*1 *bldg*2)@I_{w2}, where I_{w2} and I_{w1} start at the same time, but I_{w1} has a shorter duration, they differ in the rate that the agent walks from *bldg*1 to *bldg*2.

If (walk *bldg*1 *bldg*2)@I_{walk} is not an impossible plan instance, then it is executable as long as the agent is at *bldg*1 just prior to execution time (i.e., I_{walk}). Furthermore, this connection between (walk *bldg*1 *bldg*2)@I_{walk} and the conditions under which it is executable is inevitable at all times. Thus we have the following:

```
EXC-WALK)
  (IF (< = (minimal-time ?bldg1 ?bldg2)
           (DURATION ?i_walk))
      (INEV ?ie
  (IF (AND (HOLDS (at agt ?bldg1) ?i_at-bldg1)
           (MEETS ?i_at-bldg1 ?i_walk))
      (EXECUTABLE
        (walk ?bldg1 ?bldg2)@?i_walk))))
```

where the function term (minimal-time *bldg*1 *bldg*2) denotes the minimal time it takes for the agent to walk from *bldg*1 to *bldg*2.

The effects of (walk *bldg*1 *bldg*2)@I_{walk} are that the agent is outside during the time of execution and will be at *bldg*2 iately after execution. The connection between a plan instance and its effects is inevitable at all times. Thus we have the following:

```
EFF-WALK)
  (INEV ?ie
    (IF (OCC (goto ?bldg1 ?bldg2)@?i_walk)
      (∃ ?i_at-bldg2
        (AND (HOLDS (at agt ?bldg2) ?i_at-bldg2)
             (MEETS ?i_walk ?i_at-bldg2))
             (HOLDS (at agt outside) ?i_walk)))).
```

Given the sentences describing the planning environment and the sentences describing the action specifications, we are looking for a future plan instance such that at the time of planning, it is inevitable that it is executable and if it occurs the goal conditions will obtain. Thus we want to find a plan instance *pi* that has execution time that is later then I_p and it logically follows from the sentence describing the planning environment and the action specifications that the following is true:

```
GOAL)
  (INEV I_p (AND (EXECUTABLE pi) (IFTRIED pi G)))
```

where

```
G = def (∃ ?i_b (AND (DURING ?i_b I_C)
                     (HOLDS (at agt bank) ?i_b)).
```

In order for it to be possible to achieve the goal by executing a plan instance of the form (walk *bldg*1 *bldg*2)@I_{walk}, the minimal time it takes to get from home to the bank must be less than the duration of I_C (since the plan instance must be performed during this interval). We will assume that this condition holds and therefore the planning environment is described by PE1, PE2 along with:

PE3) (< (minimal-time home bank) (DURATION I_C)).

It can be shown that GOAL can be satisfied by any plan instance *pi* belonging to the set:

```
{(walk home bank)@?i_w
   |  (MEETS I_p ?i_w)
      and (< = (minimal-time home bank)
      (DURATION ?i_w))
      and (DURING ?i_w I_C)}.
```

That such an interval exists is guaranteed by PE3 along with an axiom in interval logic concerning the existence of intervals.

These constraints on $?i_w$ result from intersecting the conditions needed to guarantee that it is inevitable (at I_p) that if (walk home bank)@$?i_w$ occurs the goal condition will hold and the conditions needed to guarantee that it is inevitable that the plan instance is executable. The constraint that $?i_w$ finishes before I_C insures that the agent does not arrive at the bank at a time later than I_C, while the conditions $?i_w$ immediately follows I_p and has a duration greater or equal to (minimal-time home bank) insure that the plan instance is executable.

These constraints may be derived by using a control strategy similar to the backward chaining strategy employed to solve a single conjunct in nonlinear planning such as described in Allen and Koomen [2]. We find a set of plan instances each of which would achieve the goal if executed. Call this set S_0. We then see if there is any plan instance (or class of plan instances) belonging to S_0 which are executable. If this is the case we are done. Otherwise, we must try to compose a plan instance *pi* (or class of plan instances) belonging to S_0 with another that achieves the conditions needed for *pi* to be executable. If this composite plan instance is executable, we are done, else we repeat the process looking for a plan instance that achieves the conditions needed for the composite plan instance to be executable, etc.

Now, as we mentioned, by constraining $?i_w$ so that it immediately follows I_p and has duration greater than or equal to the minimal time it takes to go from the home to the bank, we find a class of executable plan instances that achieve the goal. For the sake of illustration, suppose that we must constrain $?i_w$ so that it is strictly after I_p instead. In this case, it does not follow that it is inevitable at planning time that (walk home bank)@i_w is executable. The reason is that (walk home bank)@i_w is executable only if the agent is at home immediately prior to i_w. Although it is inevitable that the agent is at home during I_p, we cannot prove that it is inevitable that the agent is at home (or any other location for that matter) at any time later than I_p.

If we want to execute (walk home bank)@i_w for i_w strictly after I_p, then we must compose this plan instance with another plan instance whose effect is that the agent is at home just prior to i_w. For this purpose, we introduce a class of plan instances that refer to staying at the same location for any period of time. We will use the function term (stay-at

bldg)@i_{ss} to denote the plan instance where the agent stays in the building *bldg* during the interval i_{ss}. The conditions under which it is executable are simply that the agent is in the building just prior to execution.

EXC-STAY)
(INEV ?*ie* (IF (AND (HOLDS (at *agt* ?*bldg*) $?i_{ss\text{-}bldg}$)
(MEETS $?i_{ss\text{-}bldg}$ $?i_{stay}$))
(EXECUTABLE (stay-at ?*bldg*)@$?i_{stay}$))).

The effects of (stay-at *bldg*)@i_{ss} is that the agent is at *bldg* during the time of execution:

EFF-STAY)
(INEV ?*ie* (IF (OCC (stay-at ?*bldg*)@$?i_{stay}$)
(HOLDS (at *agt* ?*bldg*) $?i_{stay}$))).

By executing (stay-at home)@$?i_{st}$ where $?i_{st}$ immediately precedes $?i_{w}$, we can achieve the conditions under which (walk home bank)@$?i_{w}$ is executable. If $?i_{st}$ immediately follows I_p, then (stay-at home)@$?i_{st}$ is executable. From this we get

(IF (AND (MEETS I_p $?i_{st}$)
(MEETS $?i_{st}$ $?i_{w}$)
(< = (minimal-time home bank)
(DURATION $?i_{w}$)))
(INEV I_p
(EXECUTABLE (COMP (stay-at home)@$?i_{st}$)
(walk home bank)@$?i_{w}$)))).

This is derived using the theorem INTERACTION1 which states: if pi_1 is prior to pi_2, pi_1 is executable, and if pi_1 were executed, then pi_2 would be executable, then (COMP pi_1 pi_2) is executable. By also adding the constraint that $?i_{w}$ completes before I_C, we can insure that the composite plan instance (COMP (stay-at home)@$?i_{st}$) (walk home bank)@$?i_{w}$) achieves the goal of being at the bank sometime during I_C.

The Frame Problem

In the example above, we saw that if (walk home bank)@i_{w} has an execution time that does not immediately follow planning time, we have to introduce another plan instance whose effect is that (at *agt* home) is true immediately prior to i_{w}. A plan instance of the form (stay-at home)@i_{s} served this purpose. The property (at *agt* home) holds at planning time and the execution of (stay-at home)@i_{s} simply maintains this property up to the time when (walk home bank)@i_{w} is to be executed.

The fact that we have plan instances that maintain properties contrasts with the traditional approach in nonlinear planning. In the nonlinear planning paradigm, if we introduce an action a_1 into the plan and this action has preconditions that are satisfied by conditions that hold in the initial situation, a "ghost node" is created, indicating that it is not necessary (at least at this stage) to explicitly introduce another action to achieve a_1's preconditions [16]. If the system finds another action a_2 in the plan whose effects negate a_1's preconditions, the system would try to order this action to follow a_1. If this ordering is not possible, the system would have to introduce a third action following a_2 and before a_1 that restores a_1's preconditions, or remove a_1 or a_2 from the plan. This strategy can be seen as an implementation of the STRIPS assumption.

In effect, these ghost nodes correspond to our maintenance plan instances, although they are not given the same status as actions and are treated differently by the nonlinear planner. For example, one does not introduce a ghost node into a plan, just like one introduces actions. They result as a side effect of linking an action to an earlier state where one of its preconditions holds. So one cannot explicitly choose between maintaining a condition to achieve a precondition as opposed to introducing an action that explicitly makes the precondition true, in the case where the precondition holds at an earlier state. The first alternative is automatically tried first (which turns out to be a good heuristic).

As we previously described, the use of the STRIPS assumption (in the guise of ghost nodes) leads to problems if we assert that some property, which the agent cannot affect, is true in the initial situation (and in the case of Vere's system, does not include all scheduled external events that affect this property). Since there will be no actions whose effects violate an external property p, an action whose precondition is p can be ordered anywhere in the plan.

This problem arises because a ghost node can be created for all properties. In our formalism, we do not have the same problem because there will only be maintenance plan instances for properties completely in the agent's control. Thus we might have a maintenance plan instance for "the agent stays in the same location," but clearly would not have a maintenance plan instance for a property such as "the gym is locked." Thus even though the property "the gym is locked" holds during planning time and there are no assertions about whether "the gym is locked" holds in the future, there is no plan instance that can be executed that makes this condition true at any time in the future. The agent is at the mercy of its environment.

Now, the fact that we have maintenance plan instances does not give us a simple solution to the frame problem. The frame problem manifests itself in a different form in our logic. What we do get though is a uniform treatment of the frame problem. In a system such as Wilkins' that uses the STRIPS assumption but allows concurrent actions, he has one mechanism for finding conflicts caused by one action's effects interfering with another's preconditions and another for finding conflicts between concurrent actions, this being the resource mechanism. In our logic, both conflicts can be seen to be of the same form and can be treated by the same mechanism, as follows.

We have already shown that if we want to execute pi_1 and pi_2 together, then we must prove that the following is true:

(INEV I_p (IFTRIED (COMP pi_1 pi_2)
(AND (OCC pi_1) (OCC pi_2)))).

To prove that a plan instance, say pi_1, does not interfere with a later one's preconditions, say pi_2, we show that pi_1 can be executed together with the plan instance(s) that bring about pi_2's preconditions. Thus the problem of determining whether a plan instance interferes with another one's preconditions reduces to the problem of determining whether two plan instances are executable when taken together. The following "blocks-world" example will help to clarify. Consider the plan instance (grasp *blk* 1)@i_1 which refers to grasping block *blk* 1 during interval i_1. This plan instance is executable as long as the property (clear *blk* 1) holds just prior to i_1. Also suppose that property (clear *blk* 1) is as-

serted to be true at planning time and (keep-clear *blk* 1)@i_{kc} is a plan instance that maintains property (clear *blk* 1) from the time of planning time up to the beginning of i_1. Thus the plan instance (COMP (keep-clear *blk* 1)@i_{kc} (grasp *blk* 1)@i_1) is executable. Now, consider some plan instance a_0@i_0 that is prior to i_1 and it is inevitable (at planning time) that it is executable. If we want to execute a_0 along with the composite plan instance above, we must prove that (COMP a_0@i_0 (keep-clear *blk* 1)@i_{kc} (grasp *blk* 1)@i_1)) is executable. Note that, since COMP is associative, without loss of generality we can let COMP take any number of arguments.

Since (keep-clear *blk* 1)@i_{kc} guarantees the executability of (grasp *blk* 1)@i_1, and (grasp *blk* 1)@i_1 cannot interfere with a_0@i_0 since it occurs after a_0@i_0, the above is executable only if the plan instance (COMP a_0@i_0 (keep-clear *blk* 1) @i_{kc})) is executable. As a result, reasoning about how (grasp *blk* 1)@i_1 and a_0@i_0 interact is reducible to reasoning about how the two overlapping instances (keep-clear *blk* 1)@i_{kc} and a_0@i_0 interact.

Thus both forms of interactions involve determining whether it is inevitable at planning time that two overlapping plan instances are executable together. As we discussed earlier, two plan instances that overlap in time are executable together only if they do not interfere with each other, i.e.,

(IF (OCC pi_2) (IFTRIED pi_1 (OCC pi_2))).

It is at this step that we need "frame axioms" or some nondeductive method for solving the frame problem. Typically, when describing an action, one only mentions its effects, not what it does not affect, but to reason about the interaction of overlapping plan instances, we must be able to determine what a plan instance does not affect. Thus we could explicitly encode frame axioms of the form

(INEV I_ρ (IF C_i (IFTRIED *pi* C_i)))

to specify the temporally qualified conditions C_i that are not affected by *pi*'s execution.

From a pragmatic standpoint, however, specifying a large number of frame axioms might lead to an inefficient implementation. Such a point has been made by Wilkins [20]. An alternative approach is to restrict the form of sentences allowed in specifying the planning environment and action specifications and use a default-like assumption to compute what an action does not affect. Such is done by adopting the STRIPS assumption where the effects of an action are limited to a list specifying the properties that are negated and a list specifying the properties that are made true. This precludes the use of disjunctive effects among other things. Dean [5] handles the frame problem in a temporally rich domain by appealing to a "persistence assumption." This is a rule of the form: once a property is made true, it remains true until some other property makes it false. To implement this mechanism, one must be able to efficiently compute when a set of properties are inconsistent when taken together. This is done by restricting the way properties can be logically related. We are currently studying several such default reasoning techniques and attempting to characterize the range of problems that each technique can handle.

V. The Semantics

The formal semantics for this logic can be characterized as "possible-world semantics." The basic components of a model structure are a set of world-histories, which are complete histories of the world (our variety of possible worlds), and two accessibility relations. Each sentence is interpreted with respect to a world-history within some model structure. The truth value of a nonmodal sentence (i.e., interval logic sentence) at world-history h_0 is only dependent on the properties that hold and event instances that occur in h_0. The truth value of the sentence (INEV *i P*) at h_0 depends on the world-histories that are possible with respect to h_0 and share a common past up until the end of interval *i*. The truth value of the sentence (IFTRIED *pi P*) at h_0 depends on the "closest" world-histories to h_0 where the plan instance denoted by *pi* is attempted. The interpretation of counterfactual statements in terms of a "closeness" accessibility relation derives from Lewis' [11] and Stalnaker's [17] work on conditionals. To get at a more concrete notion of what a plan instance attempt is, we appeal to Goldman's theory of actions [7]. This is described in the next section. Following this, we present the model structure and give the interpretation for the two modal statements (INEV *i P*) and (IFTRIED *pi P*). We must also note that the interpretation of a term is constant over world-histories, that is, terms are treated as rigid designators. The interpretation for the rest of the language (i.e., atomic formulas, and sentences related by the first-order connectives) is omitted since this is straightforward. A detailed presentation of the semantics is given in [15].

Goldman's Theory of Actions and Basic Generators

In Goldman's theory of action, he defines what it means to say that one act token (an action at a particular time performed by a particular agent) *generates* another, under specified conditions. Roughly put, the statement "act token a_1 generates act token a_2" holds whenever it is appropriate to say that a_1 can be done by doing a_2. Associated with each generation relation, "a_1 generates a_2" is a set of conditions C^* such that C^* are necessary and sufficient conditions under which if a_1 occurs then a_2 occurs.

Goldman also introduces the concept of a *basic* action token. A basic action token is a primitive action token in the sense that every nonbasic action token is generated by some basic action token (or some set of basic action tokens executed together) and there is no action token, more primitive, that generates a basic action token. They can be thought of as the fundamental building blocks of which all action tokens are composed. In Goldman's work, basic action tokens are associated with particular body movements that can be done "at will" as long as certain "standard conditions" hold. As an example, moving one's arm counts as a basic action since it can be done at will as long as no one is holding the arm down, the agent is not paralyzed, etc.

In our theory, we equate "plan instance *pi* is attempted" with "*pi*'s basic generator occurs." In each model, a subset of the plan instances are designated as being basic and a basic generator function is specified that associates every plan instance with the basic plan instance that generates

it. If the plan instance is basic, then it generates itself. Each plan instance has a unique generator since we are defining a plan instance as an action at a particular time done in a particular way. It is not necessary to provide for a plan instance that is generated by a set of basic plan instances, not just a single one, because the set of basic plan instances are closed under composition. Our conception of "basic" differs from Goldman's treatment in that we allow basic plan instances that are more abstract than collections of body movements at specified times. For example, in modeling a game of chess, we might take our basic plan instances to be simple chess moves such as moving the queen from Q_1 to Q_3 at a particular time, etc. The standard conditions would be that the move is legal by the rules of chess. There is no benefit in looking more closely at a chess move and saying that it is generated by the arm movement that physically moves the piece. Even though each plan instance has a basic generator associated with it, one can make assertions about a plan instance in the object language without knowing its generator. This is analogous to making assertions about physical objects in some first-order language while not knowing all its exact features as captured by the semantic model.

The Model Structure

Formally, a model is a 12-tuple of the form:

$$<H, D, OBJ, INT, MTS, PROP, EI, PI, BPI, BGEN, R, F_{ct}>$$

The constituents of this tuple are described as follows:

H a nonempty set of world-histories.

D a nonempty set of domain individuals. This is partitioned into four disjoint subsets *OBJ*, *INT*, *PROP*, and *EI* which are given as follows:

OBJ a nonempty set of objects that existed at any time in any world-history.

INT a nonempty set of temporal intervals.

The relation *MTS* is defined over the set of intervals. $MTS(i_1, i_2)$ means that interval i_1 meets interval i_2 to the left (i_1 immediately precedes i_2). Allen and Hayes [4] present an axiomatization of the MTS relation; these axioms will be adopted here. All other interval relations (such as overlaps, is before, is contained in, etc.) can be defined in terms of MTS as long as we assume that for each interval, there exists another that meets it to the left, and one that is met by it to the right. We also must stipulate that the intersection of any overlapping intervals belongs to the set of intervals and that the concatenation of any two intervals belongs to the set of intervals. The relation $IN\text{-}OR\text{-}EQ(i_1, i_2)$ will be defined as being true when interval i_1 is contained in i_2 or when the intervals are equal. We also define the function $COVER(i_1, i_2)$ which yields the smallest interval that contains both i_1 and i_2

PROP is a nonempty set of properties. Each element of *PROP* is a set containing elements of the form: $<i, h>$ where i belongs to *INT* and h belongs to H.

Property *pr* holds during interval *i* in world-history *h* iff $<i, h>$ belongs to *pr*. For convenience, we define the function $HOLDS(pr, i)$ which yields the set of world-histories where property *pr* holds over interval *i*:

$$HOLDS(pr, i) =_{def} \{h \mid <i, h> \in pr\}.$$

To capture the constraint: if a property holds over an interval, it holds over any properly contained interval, we have the following:

HOLD1)

For all properties (pr) and intervals (i_1 and i_2),
If $IN\text{-}OR\text{-}EQ(i_2, i_1)$
then $HOLDS(pr, i_1) \subseteq HOLDS(pr, i_2)$.

EI is a nonempty set of event instances. Each element of *EI* is an ordered pair of the form: $<i, h\text{-}set>$ where i belongs to *INT* and $h\text{-}set$ is a subset of H. The time of occurrence of event instance $<i, h\text{-}set>$ is i, and $<i, h\text{-}set>$ occurs only in world-histories belonging to $h\text{-}set$.

For convenience, we define the function TIME-OF(*ei*) which yields event instance *ei*'s time of occurrence.

For all event instances $(<i, h\text{-}set>)$,

$$TIME\text{-}OF(<i, h\text{-}set>) =_{def} i.$$

Similarly, we define the function OCC(*ei*) which yields the set of world-histories where event instance *ei* occurs.

For all event instances $(<i, h\text{-}set>)$,
$OCC(<i, h\text{-}set>) =_{def} h\text{-}set$.

PI is a nonempty set of plan instances. *PI* is a subset of *EI*.

The set *PI* is closed under plan instance composition. The composition of two plan instances occurs iff both its components occur, and its time of occurrence is the smallest interval that contains both of its components' times of occurrence. For convenience, we define the composition function *CMP* by:

for all pl. inst. $(<i_1, h\text{-}set_1>$ and $<i_2, h\text{-}set_2>)$,
$CMP(<i_1, h\text{-}set_1>, <i_2, h\text{-}set_2>) =_{def}$
$<COVER(i_1, i_2), h\text{-}set_1 \cap h\text{-}set_2>$

To state that *PI* is closed under *CMP* we have:

PI1)

For all plan instances (pi_1 and pi_2),
$CMP(pi_1, pi_2) \in PI$.

BPI is nonempty set of basic plan instances. *BPI* is a subset of *PI*. *BPI* is also closed under composition, thus we have:

BPI1)

For all basic plan instances (bpi_1 and bpi_2),
$CMP(bpi_1, bpi_2) \in BPI$.

BGEN is a one-place function with domain *PI* and range *BPI*. For every plan instance pi, $BGEN(pi)$ is its basic generator. If pi is a basic plan instance then $pi = BGEN(pi)$.

In all world-histories, if a plan instance occurs then its basic generator also occurs. Thus we have the constraint:

BGEN1)
For all plan instances (pi),
$OCC(pi) \subseteq OCC(BGEN(pi))$.

A plan instance and its generator have the same time of occurrence giving us the constraint:

BGEN2)
For all plan instances (pi),
$TIME\text{-}OF(pi) = TIME\text{-}OF(BGEN(pi))$.

Finally, we have the constraint that the generator of a composite plan instance is equal to the composition of the generators of the plan instance's components:

BGEN3)
For all plan instances (pi_1 and pi_2),
$BGEN(CMP(pi_1, pi_2)) = CMP(BGEN(pi_1), BGEN(pi_2))$.

The R Accessibility Relation and Interpretation of INEV

The truth value of the sentence (INEV *i P*) directly depends on the *R* accessibility relation, a three-place relation taking an interval and two world-histories as arguments. $R(i, h_0, i_1)$ can be read as: h_1 is an alternate way the future might have unfolded with respect to h_0 at time i. The interpretation of (INEV *i P*) is given by

For every world-history (h_0), sentence (*S*),
and interval term (i) (INEV *i S*) is true at h_0 iff
for every world-history (h_1)
if $R(V(i), h_0, h_1)$ then *S* is true at h_1

where $V(i)$ is the interval (member of *INT*) that the term i denotes.

The constraints we place on *R* are as follows:

If $R(i_1, h_0, h_1)$ is true then h_0 and h_1 share a common past up until the end of interval i_1. We therefore impose the following constraints:

R1)
For all world-histories (h_0 and h_1),
properties (p) and intervals (i_1 and i_2),
if $R(i_1, h_0, h_1)$ and ENDS-BEFORE(i_2, i_1) then
$h_0 \in HOLDS(p, i_2)$ iff $h_1 \in HOLDS(p, i_2)$.

R2)
For all world-histories (h_0 and h_1),
event instances (ei) and interval (i_1),
if $R(i_1, h_0, h_1)$ and ENDS-BEFORE(TIME-OF(ei), i_1)
then $h_0 \in OCC(ei)$ iff $h_1 \in OCC(ei)$

where ENDS-BEFORE(i_1, i_2) means that i_1 ends before i_2 or the intervals end at the same time.

An alternative world-history at time i_1 is an alternative at all earlier times.

R3)
For all world-histories (h_0 and h_1),
properties (p) and intervals (i_1 and i_2),
if $R(i_1, h_0, h_1)$ and ENDS-BEFORE(i_2, i_1) then $R(i_2, h_0, h_1)$

R is an equivalence relation for a fixed time.

R4) (reflexive)
For all world-histories (h_0) and intervals (i),
$R(i, h_0, h_0)$ is true.

R5) (symmetric)
For all world-histories (h_0 and h_1) and intervals (i),
if $R(i, h_0, h_1)$ then $R(i, h_1, h_0)$.

R6) (transitive)
For all world-histories (h_0, h_1, and h_2) and intervals (i),
if $R(i, h_0, h_1)$ and $R(i, h_1, h_2)$ then $R(i, h_0, h_2)$.

The Selection Function F_{ct}

The truth value of the sentence (IFTRIED *pi P*) directly depends on the selection function F_{ct}. This function has domain $BPI \times H$ and range 2^H. If basic plan instance *bpi*'s standard conditions hold in world-history h, then, $F_{ct}(bpi, h)$ is the set of "closest" world-histories to h where *bpi* occurs. If the standard conditions do not hold, $F_{ct}(bpi, h)$ is set equal to $\{h\}$. The approach of giving semantics to counterfactuals in terms of a "closeness" accessibility relation follows from the work of Stalnaker [17] and Lewis [11]. Very roughly (using Stalnaker's formalization), the counterfactual "If *A* then *C*" is true at world w_0, if *C* is true in the closest world to w_0 where antecedent *A* is true (if such a world exists). The reason for having a "closeness" measure on possible worlds seems to stem from a pragmatic principle on how one evaluates counterfactuals. This is best captured by the following test proposed by Frank Ramsy:

> Suppose that you want to evaluate the counterfactual "If *A* then *C*." First you hypothetically add the antecedent *A* to your stock of beliefs and make the minimal revision required to make *A* consistent. You then consider the counterfactual to be true iff the consequent *C* follows from this revised stock of beliefs.

What one can say about a general notion of "closeness" is quite limited. Most matters of "closeness" are decided by pragmatics, not semantics. In both Stalnaker's and Lewis' approaches, only a few, mostly obvious constraints (such as if *A* is true at w_0 then the closest world to w_0 where *A* is true is w_0 itself) are placed on the "closeness" relation. In our theory, we are only treating counterfactuals of a particular form: "If *pi* were to be attempted *P* would be true." This allows us to impose additional constraints on our closeness relation F_{ct} that arise from the specific nature of these counterfactuals and their intended use: to reason about which actions can be physically executed together. Our intuitive picture of "closeness" is as follows. Roughly, if h_1 is a closest world-history to h_0 where basic plan instance *bpi* occurs (and not equal to h_0), then h_1 differs solely on the account of executing *bpi*, or any basic action that

physically cannot be done in conjunction with *bpi*, or any basic plan instance whose standard conditions are violated by *bpi*. There are alternative conceptions of "closeness," but we argue in [15], that this conception is the most appropriate for reasoning about what actions possibly can be done together.

The interpretation of (IFTRIED *pi S*) is given by:

For every world-history (h_0), sentence (*S*)
and plan instance term (*pi*),
(IFTRIED *pi S*) is true at h_0 iff
for every world-history (h_1)
if $h_1 \in F_{cf}(BGEN(V(pi)), h_0)$ then *S* is true at h_1

where *V*(*pi*) is the plan instance (member of *PI*) denoted by the term *pi*.

This can be read as saying that (IFTRIED *pi P*) is true at world-history h_0 iff in all closest world-histories to h_0 where *pi*'s basic generator occurs, *P* is true

In this paper, we only present the most general properties that F_{cf} must have. In [15] we discuss additional constraints that may be imposed on F_{cf} and discuss under what conditions they are appropriate. For convenience, we define F'_{cf} by extending F_{cf} to take world-history sets as its second argument

$F'_{cf}(bpi, hS) =_{def} U_{h \in hS} F_{cf}(bpi, h)$.

The constraints we impose are as follows:

There will always be at least one closest world-history

FCL1)
For every world-history (*h*)
and basic plan instance (*bpi*), $F_{cf}(bpi, h) \neq \emptyset$.

If a basic plan instance *bpi* occurs in a world-history *h* then there is only one closest world-history where *bpi* occurs and it is *h*. In both Stalnaker's and Lewis' models, we have the analogous constraint that if proposition *A* holds in world-history w_0, then the closest world-history to w_0 where *A* holds is w_0 itself.

FCL2)
For every world-history (*h*) and basic plan instance (*bpi*),
if $h \in$OCC(*bpi*) then $F_{cf}(bpi, h) = \{h\}$.

The following constraint relates a composite basic plan instance to its component parts:

FCL3)
For every world-history (*h*)
and basic plan instances (bpi_1 and bpi_2),
if $F'_{cf}(bpi_2, F_{cf}(bpi_1, h)) \subseteq$ OCC(*CMP*(bpi_1, bpi_2))
then $F_{cf}(CMP(bpi_1, bpi_2), h) = F'_{cf}(bpi_2, F_{cf}(bpi_1, h))$.

This can be explained as follows. Let *S* be the set of world-histories that are reached by first going to a closest world-history where bpi_1 occurs and then going to a closest world-history where bpi_2 occurs. If both bpi_1 and bpi_2 occur in every world-history belonging to *S* (or equivalently, if *CMP*(bpi_1, bpi_2) occurs in every world-history belonging to *S*), then the closest world-histories to *h* where the composition *CMP*(bpi_1, bpi_2) occurs is exactly *S*.

The next constraint says: for all world-histories *h* and times *t*, there is always a basic plan instance *bpi*, later than *i*, such that the closest world-history where *bpi* occurs is *h* itself.

FCL4)
For all world-histories (*h*), and intervals (*i*),
there exists a $bpi \in BPI$ such that *PRIOR*(*i*, TIME-OF(*bpi*))
and $F_{cf}(bpi, h) = \{h\}$

where PRIOR(i_1, i_2) is defined as: interval i_1 immediately precedes or is before i_2.

The following constraint captures the fact that if a set of world-histories *S* is reachable by first going to the closest world-history where bpi_1 occurs and then going to the closest world-history where bpi_2 occurs, then there is a third world-history bpi_3 that reaches exactly the world-histories in *S* and has a time of occurrence equal to the smallest interval that contains both bpi_1's and bpi_2's times of occurrence:

FCL5)
For every world-history (*h*)
and plan instances (bpi_1 and bpi_2),
there exists a basic plan instance (bpi_3)
such that TIME-OF(bpi_3)) = TIME-OF(*CMP*(bpi_1, bpi_2))
and $F_{cf}(bpi_3, h) = F'_{cf}(bpi_2, F_{cf}(bpi_1, h))$.

F_{cf} and *R* are related by the following constraints:

R-CL-1)
For all world-histories (h_0 and h_1)
basic plan instances (*bpi*) and intervals (*i*),
if $h_1 \in F_{cf}(bpi, h_0)$ and PRIOR(*i*, TIME-OF(*bpi*))
then *R*(*i*, h_0, h_1).

R-CL-1 can be read as saying that if h_1 is a closest world-history to h_0 where *bpi* occurs, then h_1 is possible with respect to h_0 at all times up until the beginning of *bpi*'s execution time. Thus h_0 and h_1 share a common past up until the beginning of *bpi*'s execution time.

The following constraint leads to the fact that if h_1 and h_0 share a common past up until the end of *bpi*'s execution time, then any closest world-history to h_0 can be matched with a closest world-history to *h*, that it share a common past with until the end of *bpi*.

R-CL-2)
For all world-histories (h_0, h_1, h_{cf0}, h_{cf1})
and basic plan instances (*bpi*)
if *R*(TIME-OF(*bpi*), h_0, h_1) and $h_{cf0} \in F_{cf}(bpi, h_0)$
then there exists a world-history (H_{cf1}) such that
$h_{cf1} \in F_{cf}(bpi, h_1)$ and *R*(TIME-OF(*bpi*), h_{cf0}, h_{cf1}).

VI. Conclusion

The logic presented in this paper extended Allen's linear time logic with the INEV modality which expresses temporal possibility and IFTRIED which is a counterfactual-like modality that can be used to represent sentences describing how an agent can affect the world. This extends the class of planning problems typically handled by the situation calculus. In particular, a planning environment can be represented that has assertions about the past, present, and future, not just assertions about the current state. Plan instances can contain concurrent actions and have any ex-

ecution time; they are not restricted to be sequences of actions to be executed in the current situation. Finally, any temporal statement can be a goal statement; we are not restricted to goals that describe conditions that must hold just after plan execution.

The IFTRIED modality can be used to specify which propositions can and cannot be affected by the robot's actions. Without making extensions, neither McDermott's or Allen's logic could encode this type of statement. We have shown how to use IFTRIED to represent that some condition cannot be affected by the agent's actions, such as whether or not it is raining, and to represent that some condition could always be made true regardless of the external circumstances.

By nesting the IFTRIED operator, we can represent how plan instances interact with each other. We presented sufficient conditions that guarantee that two plan instances can be executed together. Other forms of interaction can also be represented. For example, the logic can represent that two plan instances can be separately executed, but cannot be executed together. This might be the case if the two plan instances had concurrent execution times and shared the same resource.

In the last section, we presented the model structure which consists of a set of possible world-histories related by the R relation which is used to interpret INEV and the F_{cf} function which is used to interpret IFTRIED. F_{cf} embodies the notion of "closeness." The approach of interpreting a counterfactual-like modality in terms of "closeness" accessibility relation derives from Lewis' and Stalnaker's semantic theories of conditionals.

References

[1] *J. F. Allen, "Maintaining knowledge about temporal intervals," Commun. ACM*, vol. 26, no. 11, pp. 832–843, Nov. 1983.

[2] J. F. Allen and J. A. Koomen, "Planning using a temporal world model," in *Proc. 8th. Int. Joint Conf. on Artificial Intelligence* (Karlsruhe, W. Germany, Aug. 1983), pp. 711–714.

[3] *J. F. Allen, "Towards a general theory of action and time," Artificial Intell.*, vol. 23, no. 2, pp. 123–154, 1984.

[4] J. F. Allen and P. J. Hayes, "A common-sense theory of time," in *Proc. 9th Int. Joint Conf. on Artificial Intelligence* (Los Angeles, CA, Aug. 1985).

[5] T. Dean, "Planning and temporal reasoning under uncertainty," in *Proc. IEEE Workshop on Knowledge-Based Systems* (Denver, CO, 1984).

[6] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intell.*, vol. 2, pp. 189–205, 1971.

[7] A. I. Goldman, *A Theory of Human Action*. Englewood Cliffs, NJ: Prentice-Hall, 1970.

[8] M. P. Georgeff, "A theory process," SRI Int., unpublished manuscript, 1985.

[9] A. Haas, "Possible events, actual events, and robots," *Comput. Intell.*, vol. 1, no. 2, pp. 59–70, 1985.

[10] A. L. Lansky, "Behavioral specifications and planning for multiagent domains," Tech. Rep. 360, SRI Int., 1985.

[11] D. K. Lewis, *Counterfactuals*. Cambridge, MA: Harvard Univ. Press, 1973.

[12] J. McCarthy and P. J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," in B. Meltzer and D. Miche, Eds., *Machine Intelligence, Vol. 4*. New York, NY: Elsevier, pp. 463–502, 1969.

[13] D. McDermott, "A temporal logic for reasoning about processes and plans," *Cogn. Sci.*, vol. 6, no. 2, pp. 101–155, 1982.

[14] R. C. Moore, "Reasoning about knowledge and action," Tech. Rep. 191, SRI Int., 1980.

[15] R. N. Pelavin, "A formal logic that supports planning with external events and concurrent actions," Ph.D. dissertation, Computer Sci. Dep., U. Rochester, expected in 1986.

[16] E. D. Sacerdoti, *A Structure for Plans and Behavior*. New York, NY: Elsevier, 1977.

[17] R. Stalnaker, "A theory of conditionals," in W. L. Harper, R. Stalnaker, and G. Pearce, Eds., *IFS*. Dordrecht, The Netherlands: Reidel, 1981, pp. 41–55.

[18] R. H. Thomason, "Indeterminist time and truth value gaps," *Theoria*, vol. 36, pp. 264–281, 1970.

[19] S. A. Vere, "Planning in time: Windows and durations for activities and goals," Res. Rep., Jet Propulsion Lab., Pasadena, CA, Nov. 1981.

[20] D. Wilkins, "Domain independent planning: Representation and plan generation," Tech. Note 226, SRI Int., May 1983.

Appendix A-6

PLANNING WITH ABSTRACTION

Josh Tenenberg
Department of Computer Science
University of Rochester
Rochester, NY 14620
josh@rochester

Abstract

Intelligent problem solvers for complex domains must have the capability of reasoning abstractly about tasks that they are called upon to solve. The method of abstraction presented here allows one to reason analogically and hierarchically, making both the task of formalizing domain theories easier for the system designer, as well as allowing for increased computational efficiencies. It is believed that reasoning about concepts that share structure is essential to improving the performance of automated planning systems by allowing one to apply previous computational effort expended in the solution of one problem to a broad range of new problems.

I. Introduction

Most artificial intelligence planning systems explore issues of search and world representation in toy domains. The blocks world is such a domain, with one of its salient and unfortunate characteristics being that all represented objects (blocks) are modeled as being perfectly uniform in physical features. We would like to model a richer domain, where objects bear varying degrees of similarity to one another. For instance, we might wish to model blocks and trunks, which are both stackable but of different sizes and weights, or boxes and bottles, which are both containers but of different shape and material. As a consequence of solving problems in this richer domain, we will want plans to solved problems to be applicable to new problems based upon the similarites of the objects to be manipulated. So, for instance, a plan for stacking one block on top of another will be applicable to a similar trunk stacking in terms of its gross features, but will differ at more detailed levels. We will present a representation for plans of varying degrees of abstraction based upon a hierarchical organization of both objects and actions that provides a qualitative similarity metric for problems posed to the planner. This plan representation has the following property. When a plan is expressed at a high level of abstraction, it will apply to a wide class of problems but with little search information, while, when expressed at lower levels of abstraction it will apply to fewer problems but give increasing amounts of information with which to guide search.

II. Object and Action Abstraction

A common way to represent physical objects is within a taxonomic hierarchy [Hendrix 1979]. A graphic example of part of one such hierarchy is given in figure 1. An arc from node *v* to node *w* indicates a subset relation between these types. Therefore, all objects of type *v* are also objects of type *w*, and inherit all properties provable of type *w*. We will call *w* an abstraction of *v*, and *v* a specialization of *w*. These taxonomies enable us to make assertions about a class of objects that we need not repeat for all of its subclasses. So, for instance, if it is asserted that all supportable objects can be stacked, then it need not be asserted separately that blocks can be stacked, boxes can be stacked, and trays can be stacked. It suffices to assert that blocks, boxes and trays are all supportable objects. This structure is not strictly a tree, which means that each object can be abstracted along several different dimensions, with the effect that every node inherits all of the properties of every other node from which there is a path. For example, a Bottle is both a Container and a Holdable object , since there are paths in the graph from Bottle to both Holdable and Container. Note that this structure admits no exceptions. We prefer instead to weaken those assertions we can make of a class in order to preserve consistency.

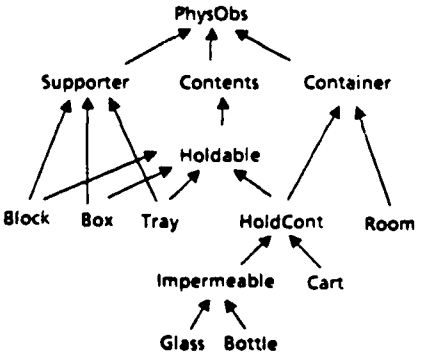


figure 1

We would like to represent actions similarly. Typically [McCarthy and Hayes 1969], actions are represented in terms of those conditions that suffice to hold before the performance of the action (called *preconditions*) that ensure that the desired effects will hold after the performance of the action. However, an inherent inefficiency with this is that many actions share preconditions and effects which must be specified separately for each action, providing no means with which to determine which actions are similar and hence replaceable by one another in analogous problems.

What we will do alternatively is to provide an action taxonomy, by grouping actions into inheritance classes. An example of a partial action hierarchy is given in figure 2. The boxed nodes denote actions and the dotted arcs between actions denote inheritance. As with the object hierarchy, if there is an

This work was supported in part by the National Science Foundation, grants DCR-8405720 and IST-8504726.

inheritance arc from action v to action w, we say that v is a specialization of w, and w is an abstraction of v. The solid arcs from a literal into an action denote necessary preconditions for that action, and the solid arcs from an action to a literal denote effects of that action. Each action inherits all preconditions and effects from every one of its abstractions. So, for instance, *CarriedAloft(x)* is a precondition of *placeIn(x,y)* inherited from *put(x,y)*, and *In(x,y)* is an effect of *placeIn(x,y)* inherited from *contain(x,y)*. As we proceed down this graph from the root node traversing inheritance nodes *backward*, by collecting the preconditions for each action encountered, we are adding increasing constraints on the context in which the action may be performed in order to have the desired effects. At the source nodes, which represent the primitive actions, the union of all of the preconditions on every outgoing path constitute a sufficient set of preconditions. An action can only be applied if its sufficient set of preconditions are all satisfied in the current state. The sufficient set of preconditions for *placeInBox* has been italicized, and is exactly the union of those preconditions for each action type on all paths from *placeInBox* to *contain*. Additional action hierarchies we might have are *remove* with specializations *pourOut* and *liftOut*, and the hierarchy *open*, with specializations *openDoor* and *removeLid*.

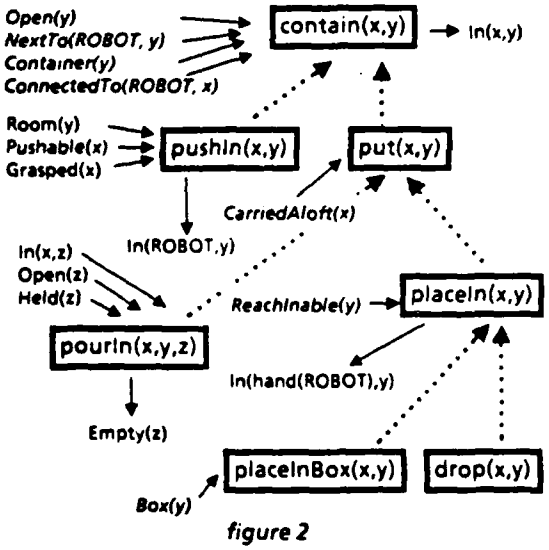


figure 2

III. Plan Abstractions

Planning involves finding a temporally ordered sequence of primitive actions which when applied with respect to the temporal ordering from a given initial state produces a state of the world in which the desired goals hold, and for which the sufficient preconditions for each primitive action must be satisfied by the state in which the action is performed. In this paper, a *total* temporal ordering of actions will be used for simplicity, although the ideas presented here can be extended to more general temporal orderings (partial orderings [Sacerdoti 77], concurrent actions [Allen 84]). Such a totally ordered sequence of actions will be called a *primitive plan*, or simply a *plan*. Finding plans to solve given problems involves searching for a state of the world which satisfies our goals from those states of the world which are possible from the initial state through the performance of one or more actions. To reduce this search, we will make use of plans that have already been found for solving previous problems. In order to use saved plans, the similarity between the previous problem solved by this plan and the current problem we wish to solve must be evaluated. We describe *plan graphs* which are a means for performing this evaluation. These plan graphs are generalizations of triangle tables [Fikes, Hart, and Nilsson 1972].

Using *explanation based generalization* [Mitchell, Keller, and Kedar-Cabelli 1985] techniques, from a primitive plan a plan graph is constructed which embeds the causal structure of the primitive plan such that the purpose of each plan step can be determined. Each action is represented not only as a primitive, but as a path from a primitive to an abstract action taken from an action hierarchy, where the causal structure enables us to determine which hierarchy to choose. Given a new problem, this plan graph is searched for its most specific subgraph whose causal structure is consistent with the new problem. This subgraph represents an abstract plan for the new problem, and will be used as a guide for finding the primitive plan for this problem. If an action in the original plan cannot be applied due to some difference between the old and the new problem, such as a difference in corresponding objects manipulated, (e.g., a ball in one case, a box in the other) we can replace this action by choosing another which is a different specialization of the same abstraction (*pickupBall* replacing *pickupBox*, both of which are specializations of *pickup*). Thus many problems can be solved by performing search within the constraints of the abstract plan we have retrieved for this problem, rather than having to perform an unconstrained global search.

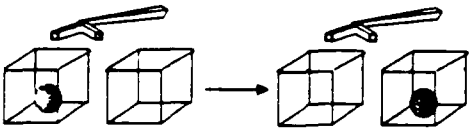
A plan graph of a plan will have nodes for each action in the primitive plan, and nodes with directed arcs for each precondition and effect of these actions. If an effect of an action satisfies a precondition of another, this will appear as an arc from the first action, to its effect, to the second action. These *causal chains* establish the purpose of each action in terms of the overall goal of the plan. We will formally define plan graphs in two stages. The first stage includes only the causal structure, while the second incorporates abstractions.

A plan graph $G = (V,E)$ for primitive plan P is a directed acyclic graph where V and E are defined as follows. The set of vertices is partitioned into two subsets V_p and V_a, precondition nodes and action nodes. Likewise, E is partitioned into two two subsets E_c and E_s, causal edges and specialization edges. For every action in P, there is a node in V_a labeled by its corresponding action. If *p* is an effect of action *a* in P, then there is a corresponding node in V_p labeled *p*, and the edge (a,p) is in E_c, and for every action *b* in P that this instance of *p* satisfies there is an edge (p,b) in E_c. For example, if action A1 establishes condition K which is a precondition of action A2, then (K,A2) is in E_c if and only if there does not exist action A3 that occurs after A1 but before A2 that *clobbers* K (establishes $\neg$K). Clearly any precondition of each action that is not satisfied by a previous action must be satisfied by the initial state. For every such precondition *p* there is a corresponding node in V_p labeled *p*, and for every action *a* in P for which this instance of *p* is a precondition, there is an edge (p,a) in E_c. Each action node in E_a is additionally labeled by a number indicating its temporal order, the n^{th} action labeled by *n*.

This graph will be specified further by the addition of action abstractions, but note that as it stands it is similar to a graph

version of triangle tables [Fikes, Hart, and Nilsson 1972], and fulfills much the same function. We can use the same technique as used in triangle tables for generalizing a plan by replacing all constants in the action and precondition nodes by variables, and redoing the precondition proofs to add constraints on variables in different actions of the plan that should be bound to the same object (see previous reference for details). These constraints will have to be added to the graph as additional preconditions, but are left off in our examples for clarity. The preconditions for this plan graph are the set of source nodes (nodes with no incoming arcs), and the goals of this plan graph are the set of sink nodes (nodes with no outgoing arcs). This graph has the property that any subset of its goals can be achieved from any initial situation in which we can instantiate all of the preconditions by applying each of the actions in order. A plan graph for the problem in figure 3 of moving a ball from one box to another is given in figure 4 (nodes representing preconditions satisfied by the initial state rather than by a previous action are not included in this figure). This graph will be altered to include abstract actions in a straightforward fashion.

figure 3



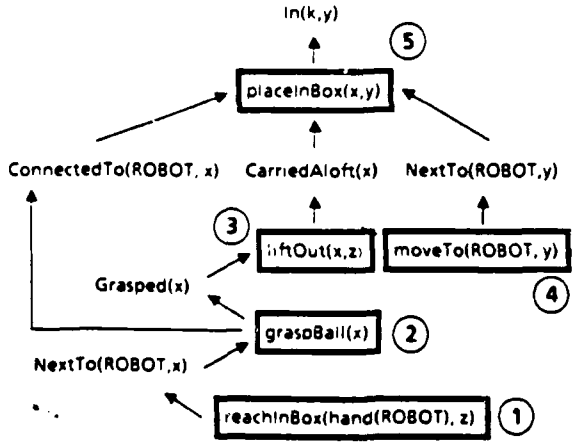


figure 4

Figure 5 is an example of the altered plan from figure 4 (the outlined subgraph of figure 5 will be explained later). This alteration is done as follows. For each primitive action A_0 in V_a, we will add nodes to V_a labeled A_1, A_2, ..., A_n (where n may be different for each primitive action) and edges to E_t labeled (A_0, A_1), (A_1, A_2), ..., (A_{n-1}, A_n), where there exists some action hierarchy such that A_i is an abstraction of each A_k for $k<i$, and A_n satisfies at least one effect p for which there exists a node in V_p labeled p and an edge in E_c labeled (A_0, p). More simply, we add an abstraction *path* from an action hierarchy to the plan graph. We then redirect each precondition arc (p, A_0) to point to the highest abstraction A_i for which there is an arc (p, A_i) in the chosen action hierarchy. In other words, p is a precondition of abstraction A_i, but not of any abstraction of A_i. For instance, *placeInBox* is replaced by the abstraction sequence *placeInBox, placeIn, put, contain*, and the preconditions *NextTo* and *ConnectedTo* are redirected to *contain*, while *CarriedAloft* is redirected to *put*. We additionally redirect effect arcs (A_0, p) such that the effects come from the highest abstraction A_i for which there is an arc (A_i, p) in the chosen action hierarchy. In other words, p is an effect of abstraction A_i, but not of any abstraction of A_i. Turning again to figure 5, the effect arc into *ConnectedTo* is redirected to come from *attachToAgent*, since this will be an effect of every specialization of this abstraction, and the effect arc to *Grasped* is redirected to come from *grasp*. We will additionally add temporal numberings to each abstraction on a path from each primitive action (although the examples will only number the highest abstractions for each action).

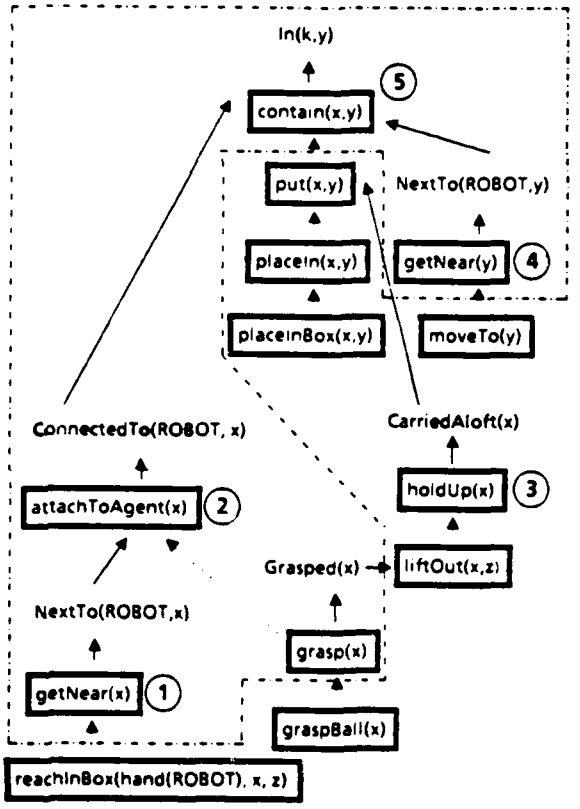


figure 5

The primitive action nodes of this plan graph indicate the primitive plan that solves the problem for which the plan was constructed. The distance between an action node and one of the goals of the entire plan graph along its shortest causal chain is a rough measure of the significance of the action to the overall plan. The shorter the distance, the more likely this action or an abstraction of it will be required in a similar problem; the greater the distance, the less likely this action will be useful in a similar problem. This plan can thus be abstracted by one or both of the following: removing causal chains from one or more precondition nodes, and removing specialization paths from one or more action nodes. Each resultant *partial plan graph* represents a plan with some of the detail unspecified.

More formally, a partial plan graph P of plan graph P' is any subset of the nodes and arcs of P' such that no source nodes are action nodes, at least one sink node (goal) of P' is in P, and these will be the only sink nodes in P, and for every node in P, there exists at least one path from this node to a sink node (unless that node is itself a sink node). Additionally, if b is an action node, then every node p for which there exists an arc (p,b) in P' will be added to P along with this arc. We will additionally "mark" each source node in P that was also a source node in P'. This mark indicates that this precondition is satisfied by the initial state of the original problem, as opposed to being satisfied by the performance of a previous action. The reason for marking these nodes will be explained later. From this definition, there will be several partial plan graphs that can be constructed from a given plan graph. The subgraph outlined by the dotted line in figure 5 is one exapmle. As before, the preconditions of a partial plan graph are the formulas attached to the source nodes (not included in the given figures), while the goals of each partial plan graph are the formulas attached to the sink nodes.

Figure 7 is the plan graph for a plan to solve the problem from figure 6. Here a box must be moved between rooms. In both this problem and that of figure 3, the goal is to move an object from one container to another. This draws analogies between rooms and boxes, which are both containers according to our object hierarchy, and between placing objects in boxes and pushing objects into rooms, which are both containment actions, according to our action hierarchy. At an abstract level, the plan of attaching the object to the agent, and moving the agent from one container to the other suffices for both problems, and in fact this is the abstract plan represented by the identical partial plan graph that is outlined by the dotted line in both figures 5 and 7. So although the problems that these graphs solve are different, at this level of abstraction they are identical.

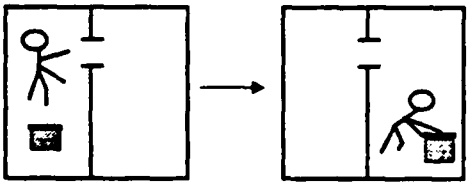


figure 6

We can generalize from this in that for any partial plan graph P of plan graph P', there will exist a set Π of plan graphs for which P will be a partial plan graph of each of them. That is, P will describe each primitive plan of each element from this set at some level of abstraction. We will use the symbol Π_P to denote the largest such set. For instance, if we label the outlined partial plan graph of figure 5 K, then the graphs of figures 5 and 7 are in Π_K. We will say that the primitive plan of each member of Π_P is an *expansion* of the partial plan graph P. The more general P is, that is, the smaller a subgraph of P' it is and hence the more abstract each of its constituent actions are and the smaller its causal chains, the larger will be the cardinality of Π_P. We will say that partial plan graph P *solves* problem Q if and only if there exists an element of Π_P whose primitive plan solves Q for some instantiation of all of its variables by ground terms.

Given a partial plan graph P and a problem instance Q that P solves, we can find an expansion of P that solves Q by only searching for specializations of the abstract actions of P without

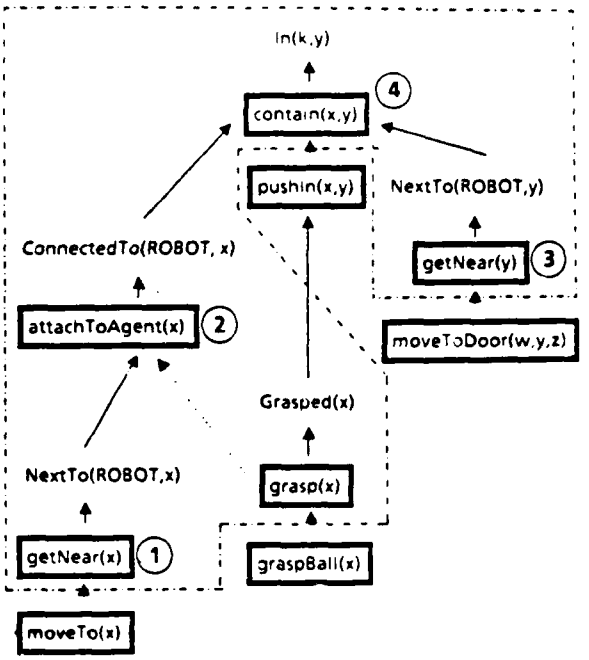


figure 7

having to backtrack through the actions of P itself. P thus serves as an abstract guide to solving Q. So, for instance, given the partial plan graph outlined in figure 5, we can find the remainder of the primitive plan (those actions not inside the dotted line) that solves the problem from figure 6 by only having to do local search. By this, we mean that for any non-primitive action in this partial plan graph, (such as *contain*, in figure 7), we follow arcs backward through that abstract action's specialization tree (figure 2 in this case) until we find a primitive action whose preconditions are all satisfied by the state in which it is executed (*pushIn*, in this example). If no such primitive exists, then additional primitives must be inserted in this plan to establish the sufficient preconditions for some specialization, where these inserted actions do not clobber preconditions of any of the already established succeeding actions.

Unfortunately, we cannot in general know if a given partial plan solves a given problem instance unless we perform the possibly unbounded local search for the primitive plan that verifies this. It may not be possible to find specializations of each extant abstract action without rearranging some of the actions, and therefore backtracking through and altering the partial plan graph itself. Although we are not guaranteed certainty, we can still use the plan graphs as a heuristic for search. We will define a partial plan graph P as being *applicable* to a problem Q if and only if the goals of Q are a subset of the goals of P, and the marked preconditions of P are a subset of the conditions that hold in the initial state of Q. Recall that we marked all of those preconditions in a partial plan graph that were satisfied by the initial state of the original problem. Applicability thus means that the current initial state satisfies the same preconditions *at this level of abstraction* as the original initial state.

Suppose we wish to find a primitive plan for problem Q consisting of an initial state and a set of goals (for simplicity we

will assume that this goal is a single literal). Additionally suppose that the goals of plan graph P are the same as those of Q. We will attempt to find the most specialized partial plan graph P' of P for which an expansion exists that will solve Q, even though it is possible that no such P' exists. We will do this by traversing P *backward* from its goal node through the causal and specialization arcs, considering increasingly larger partial plans of P. We will continue this traversal as long as the partial plan represented by all of the paths pursued is still applicable to Q, stopping when we can no longer traverse any arc and still have applicability of the current partial plan to problem Q. The size of the partial plan that we have constructed is thus a qualitative measure of similarity between the original problem and the current one. If there are only insignificant differences, the partial plan may be equivalent to the entire plan graph. If the differences between the problems are large, this may result in a graph of only a few actions expressed at high levels of abstraction. But given the exponential nature of searching through combinatorial spaces, knowing the temporal ordering of even a few of the action abstractions that will eventually appear specialized in our plan may help significantly.

IV. Previous Research

Abstraction in planning is typically viewed in terms of *decompositional abstraction* as used in NOAH-like planners [Sacerdoti 1977]. In these planners, action A is an abstraction of actions B,C,D if the latter actions are each steps in the performance of action A. This type of abstraction is thus orthogonal to inheritance abstraction presented here.

ABSTRIPS [Sacerdoti 1974], although using different techniques, shares some important similarities. ABSTRIPS is an iterative planner, where increasingly large subsets of preconditions of each action are considered at each successive iteration. The developed plan at each level is then used to guide search at more detailed levels, where the satisfaction of emergent preconditions is attempted locally, similar to what is done in this paper.

Of even greater similarity, but within a different domain, is the work presented in [Plaisted 1981], who uses abstraction within a theorem prover. He details how a desired proof over a set of clauses can be obtained by first mapping the clause set to a set of abstract clauses, obtaining a proof in this (hopefully simpler) space, and then using this proof as a guide in finding the proof in the original, detailed space. His mapping process and abstract proof are similar to our search for an abstract plan within our saved plan space - but rather than constructing an abstract plan for each new problem, we attempt to appropriate one from a previously solved problem.

V. Conclusion

The primary motivation for using abstraction was so that search for solutions to new problems can be improved by using solutions to old problems. We believe that this approach can be used to these ends in a domain in which objects are distinguishable at various levels of detail. We will try matching abstract plans to problems that have the same goals. Any such new problem whose initial state does not contain *all* of the preconditions of the original initial state will thus not match the abstract plan at every level, but will likely do so at some level. The partial plan graph still provides two important functions in this case. First, it ignores "unimportant" preconditions at the most general levels, where the importance of a precondition is determined by the height at which it appears in the action hierarchy. Second, the search space of the new problem can be explored along those paths that do not match the original problem, while attempting to leave intact those paths that do match.

We must point out that the abstraction described in this paper has not been implemented for even a small domain. In fact, one of the obstacles to doing such an implementation is that one may likely only see benefits in a large domain. Thus, there will be little point to use this method as a representation for the vanilla blocks world. An additional issue is in the choice of problems that the system will encounter. One can always construct problem sequences given as input to the problem solving system such that the abstractions in the model will optimize performance. By the same token, one can always construct problem sequences where the abstractions will give quite poor performance. The ultimate test of a set of abstractions will therefore be empirical in that they must be cost-effective (in terms of some resource measure) only as compared with other problem solvers (human or machine) for a given domain. We can make no such claims for the *particular* abstractions of the limited physical world domain illustrated in this paper. The importance of this work is in how we can structure knowledge for solving problems in domains that are far richer than the ones in which the current generation of planners have approached. It is believed that inheritance abstraction will be a powerful technique in this endeavor.

Special thanks to my advisor, Dana Ballard, whose energy, knowledge, piercing insights and trust have made it all worthwhile, to Leo Hartman, who always seems to have an answer when an answer is needed, and to Jay Weber, who will hopefully solve the questions of how we go about *constructing* abstraction hierarchies.

References

[Allen 84]
Allen, J.F., "Towards a General Theory of Action and T[illegible]", Artificial Intelligence 23:123 - 154, 1984.

[Fikes, Hart, and Nilsson 1972]
Fikes, R., Hart, P. , and Nilsson, N., "Learning and executing generalized robot plans", *Artificial Intelligence* 3:251 - 288, 1972

[Hendrix 1979]
Hendrix, G.C., "Encoding Knowledge in Partitioned Networks" in, Associative Networks, ed. Findler, N.V. 1979

[McCarthy and Hayes 1969]
McCarthy, J., and Hayes, P., "Some philosophical problems forom the standpoint of artificial intelligence", In B.Meltzer and D. Michie (editors), *Machine Intelligence 4*, 1969.

[Mitchell, Keller and Kedar-Cabelli 1985]
Mitchell, T., Keller, R. and Kedar-Cabelli, S., "Explanation Based Generalization: A Unifying View", Rutgers Computer Science Dept. ML-TR-2, 1985.

[Plaisted 1981]
Plaisted, D., "Theorem Proving with Abstraction", *Artificial Intelligence* 16:47-108, 1981

[Sacerdoti 1974]
Sacerdoti, E., "Planning in a hierarchy of abstraction spaces", *Artificial Intelligence* 5:115 - 135, 1974

[Sacerdoti 1977]
Sacerdoti, E. A structure for plans and behavior American Elsevier Publishing Company, New York, 1977

Appendix A-7

Constraint Propagation Algorithms for Temporal Reasoning

Marc Vilain — BBN LABORATORIES, 10 MOULTON ST., CAMBRIDGE, MA 02238

Henry Kautz — UNIVERSITY OF ROCHESTER, COMPUTER SCIENCE DEPT., ROCHESTER, NY 14627

Abstract: *This paper considers computational aspects of several temporal representation languages. It investigates an interval-based representation, and a point-based one. Computing the consequences of temporal assertions is shown to be computationally intractable in the interval-based representation, but not in the point-based one. However, a fragment of the interval language can be expressed using the point language and benefits from the tractability of the latter.*[1]

The representation of time has been a recurring concern of Artificial Intelligence researchers. Many representation schemes have been proposed for temporal reasoning; of these, one of the most attractive is James Allen's algebra of temporal intervals [Allen 83]. This representation scheme is particularly appealing for its simplicity and for its ease of implementation with constraint propagation algorithms.

Reasoners based on this algebra have been put to use in several ways. For example, the planning system of Allen and Koomen [1983] relies heavily on the temporal algebra to perform reasoning about the ordering of actions. Elegant approaches such as this one may be compromised, however, by computational characteristics of the interval algebra. This paper concerns itself with these computational aspects of Allen's algebra, and of a simpler algebra of time points.

Our perspective here is primarily computation-theoretic. We approach the problem of temporal representation by asking questions of complexity and tractability. In this light, this paper examines Allen's interval algebra, and the simpler algebra of time points.

The bulk of the paper establishes some formal results about the temporal algebras. In brief these results are:

- Determining consistency of statements in the interval algebra is NP-hard, as is determining all consequences of these statements. Allen's polynomial-time constraint propagation algorithm is sound but not complete for these tasks.
- In contrast, constraint propagation is sound and complete for computing consistency and consequences of assertions in the time point algebra. It operates in $O(n^3)$ time and $O(n^2)$ space.
- A restricted form of the interval algebra can be formulated in terms of the time point algebra. Constraint propagation is sound and complete for this fragment.

Throughout the paper, we consider how these formal results affect practical Artificial Intelligence programs.

[1]This research was supported in part by the Defense Advanced Research Projects Agency, under contracts N00014-85-C-0079 and N-00014-77-C-0378.

The Interval Algebra

Allen's interval algebra has been described in detail in [Allen 83]. In brief, the elements of the algebra are *relations* that may exist between intervals of time. Because the algebra allows for indefiniteness in temporal relations, it admits many possible relations between intervals (2^{13} in fact). But all of these relations can be expressed as *vectors* of definite *simple relations*, of which there are only thirteen.[2] The thirteen simple relations, whose definitions appear in Figure 1, precisely characterize the relative starting and ending points of two temporal intervals. If the relation between two intervals is completely defined, then it can be exactly described with a simple relation. Alternatively, vectors of simple relations introduce indefiniteness in the description of how two temporal intervals relate. Vectors are interpreted as the disjunction of their constituent simple relations.

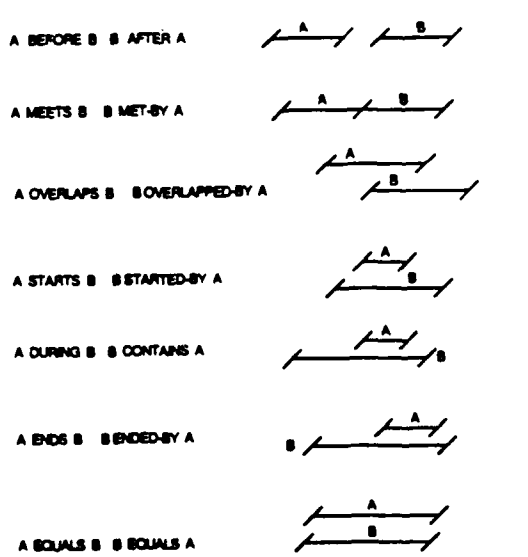


Figure 1: Simple relations in the interval algebra

Two examples will serve to clarify these distinctions (please refer to figure 2). Consider the simple relations *BEFORE* and *AFTER*: they hold between two intervals that strictly follow each other, without overlapping or meeting. The two differ by the order of their

[2]In fact, these thirteen simple relations can be in turn expressed in terms of universally and existentially quantified expressions involving only one truly primitive relation. For details, see [Allen & Hayes 85].

arguments: today John ate his breakfast *BEFORE* he ate his lunch, and he ate his lunch *AFTER* he ate his breakfast. To illustrate relation vectors, consider the vector *(BEFORE MEETS OVERLAPS)*. It holds between two intervals whose starting points strictly precede each other, and whose ending points strictly precede each other. The relation between the ending point of the first interval and the starting point of the second is left ambiguous. For instance, say this morning John started reading the paper before starting breakfast, and he finished the paper before his last sip of coffee. If we didn't know whether he was done with the paper before starting his coffee, at the same time as he started it, or after, we would then have:

PAPER *(BEFORE MEETS OVERLAPS)* COFFEE

Returning to our formal discussion, we note that the interval algebra is principally defined in terms of vectors. Although simple relations are an integral part of the formalism, they figure primarily as a convenient way of notating vector relations. The mathematical operations defined over the algebra are given in terms of vectors; in a reasoner built on the temporal algebra, all user assertions are made with vectors.

Two operations, an addition and a multiplication, are defined over vectors in the interval algebra. Given two different vectors describing the relation between the same pair of intervals, the addition operation "intersects" these vectors to provide the least restrictive relation that the two vectors together admit. The need to add two vectors arises from situations where one has several independent measures of the relation of two intervals. These measures are combined by summing the relation vectors for the measures. For example, say the relation between intervals *A* and *B* has been derived by two valid measures as being both

V_1 = *(BEFORE MEETS OVERLAPS)*
V_2 = *(OVERLAPS STARTS DURING)*.

To find the relation between *A* and *B*, that is implied by V_1 and V_2, the two vectors are summed:

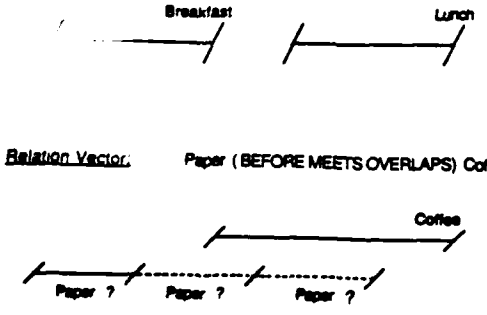


Figure 2: Examples of simple relations and relation vectors

$V_1 + V_2$ = *(OVERLAPS)*.

Algorithmically, the sum of two vectors is computed by finding their common constituent simple relations.

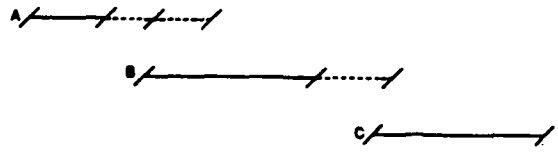
Multiplication is defined between pairs of vectors that relate three intervals *A*, *B*, and *C*. More precisely, if V_1 relates intervals *A* and *B*, and V_2 relates *B* and *C*, the product of V_1 and V_2 is the least restrictive relation between *A* and *C* that is permitted by V_1 and V_2. Consider, for example, the situation in Figure 3. If we have

V_1 = *(BEFORE MEETS OVERLAPS)*
V_2 = *(BEFORE MEETS)*

then the product of V_1 and V_2 is

$V_1 \times V_2$ = *(BEFORE)*

As with addition, the multiplication of two vectors is computed by inspecting their constituent simple relations. The constituents are pairwise multiplied by following a simplified multiplication table, and the results are combined to produce the product of the two vectors. See [Allen 83] for details.



R<A,B> = (BEFORE MEETS OVERLAPS)
R<B,C> = (BEFORE MEETS)

R<A,C> = (BEFORE)

Figure 3: Intervals whose relations are to be multiplied

Determining Closure in the Interval Algebra

In actual use, Allen's interval algebra is used to reason about temporal information in a specific application. The application program encodes temporal information in terms of the algebra, and asserts this information in the database of the temporal reasoner. This reasoner's job is then to compute those temporal relations which follow from the user's assertions. We refer to this process as completing the closure of the user's assertions.

In Allen's model, closure is computed with a constraint *propagation* algorithm. The operation of this forward-chaining algorithm is driven by a queue. Every time the relation between two intervals *A* and *B* is changed, the pair *<A, B>* is placed on the queue. The algorithm, shown in Figure 4 operates by removing pairs from the queue. For every pair *<A, B>* that it removes, the algorithm determines whether the relation between *A* and *B* can be used to constrain the relation between *A* and other intervals in the database, or between *B* and these other intervals. If a new relation can be successfully constrained, then the pair of intervals that it relates is in turn placed on the queue. The process terminates when no more relations can be constrained.

As Allen suggests [Allen 83], this constraint propagation algorithm runs to completion in time polynomial with the number of intervals in the temporal database. He provides an estimate of $O(n^2)$ calls to the *Propagate* procedure. A more fine-grained analysis reveals that when the algorithm runs to completion, it will have performed $O(n^3)$ multiplications and additions of temporal relation vectors.

Theorem 1: Let *I* be a set of *n* intervals about which *m* assertions have been added with the Add procedure. When invoked, the Close procedure will run to completion in $O(n^3)$ time.

Proof: *(Sketch[3])* A pair of intervals *<i,j>* is entered on

[3]Most of the theorems in this paper have rather long proofs. For this reason, we have restricted ourselves here to providing only proof sketches.

```
/* Let Table be a two-dimensional array, indexed by intervals, in
   which Table[i,j] holds the relation between intervals i and j.
   Table[i,j] is initialized to (BEFORE MEETS ... AFTER), the
   additive identity vector consisting of all thirteen simple relations;
   except for Table[i,i] which is initialized to (EQUAL).
   Let Queue be a FIFO data structure that will keep track of those
   pairs of intervals whose relation has been changed.
   Let Intervals be a list of all intervals about which
   assertions have been made. */

To  Add(R<i,j>)
  /* R<i,j> is a relation being asserted between i and j.*/

  begin
    Old ← Table[i,j];
    Table[i,j] ← Table[i,j] ∗ R<i,j>;
    If Table[i,j] ≠ Old
      then Place <i,j> on Fifo Queue;
    Intervals ← Intervals ∪ {i, j};
  end;


To Close
/* Computes the closure of assertions added to the database. */

  While Queue is not empty do
    begin
      Get next <i,j> from Queue;
      Propagate(i,j);
  end;


To Propagate(I,J)
/* Called to propagate the change to the relation between
   intervals I and J to all other intervals. */

  For each interval K in Intervals do
    begin
      Temp ← Table[I,K] ∗ (Table[I,J] x Table[J,K]);
      If Temp = 0
        then (signal contradiction);
      If Table[I,K] ≠ Temp
        then Place <I,K> on Queue;
      Table[I,K] ← Temp;
      Temp ← Table[K,J] ∗ (Table[K,I] x Table[I,J]);
      If Temp = 0
        then (signal contradiction);
      If Table[K,J] ≠ Temp
        then Place <K,J> on Queue;
      Table[K,J] ← Temp;
  end;
```

Figure 4: The constraint propagation algorithm

Queue when its relation, stored in *Table[i,j]*, is non-trivially updated. It is easy to show that no more than $O(n^2)$ pairs of intervals *<i,j>* are ever entered onto the queue. This is because there are only $O(n^2)$ relations possible between the *n* intervals, and because each relation can only be non-trivially updated a constant number of times.

Further, every time a pair *<i,j>* is removed from *Queue*, the algorithm performs O(n) vector additions and multiplications (in the body of the Propagate procedure). Hence the time complexity of the algorithm is $O(n \cdot n^2) = O(n^3)$ vector operations.

The vector operations can be considered here to take constant time. By encoding vectors as bit strings, addition can be performed with a 13-bit integer *AND* operation. For multiplication, the complexity is actually $O(|V_1| \cdot |V_2|)$, where $|V_1|$ and $|V_2|$ are the "lengths" of the two vectors to be multiplied (i.e., the number of simple constituents in each vector). Since vectors contain at most 13 simple constituents, the complexity of multiplication is bounded, and the idealization of multiplication as operating in constant time is acceptable.

Note that the polynomial time characterization of the constraint propagation algorithm of Figure 4 is somewhat misleading. Indeed, Allen [1983] demonstrates that the algorithm is sound, in the sense that it never infers an invalid consequence of a set of assertions. However, Allén also shows that the algorithm is incomplete: he produces an example in which the algorithm does not make all the inferences that follow from a set of assertions. He suggests that computing the closure of a set of temporal assertions might only be possible in exponential time. Regrettably, this appears to be the case. As we demonstrate in the following paragraphs, computing closure in the interval algebra is an NP-hard problem.

Intractability of the Interval Algebra

To demonstrate that computing the closure of assertions is NP-hard, we first show that determining the consistency (or satisfiability) of a set of assertions is NP-hard. We then show that the consistency and closure problems are equivalent.

Theorem 2: Determining the satisfiability of a set of assertions in the interval algebra is NP-hard.

Proof: *(Sketch)* This theorem can be proven by reducing the 3-clause satisfiability problem (or 3-SAT) to the problem of determining satisfiability of assertions in the interval algebra. To do this, we construct a (computationally trivial) mapping between a formula in 3-SAT form and an equivalent encoding of the formula in the interval algebra.

Briefly, this is done by creating for each term *P* in the formula, and its negation *~P*, a pair of intervals, P and NOTP. These intervals are then related to a "truth determining" interval MIDDLE: intervals that fall before MIDDLE correspond to *false* terms, and those that fall after MIDDLE correspond to *true* terms. The original formula is then encoded into assertions in the algebra; this can be done (deterministically) in polynomial time.

The encoding proceeds clause by clause. For each clause $P \vee Q \vee R$, special intervals are created. These intervals are related to the literals' intervals P, Q, and R in such a way that at most two of these intervals can be before MIDDLE (which makes them false). The other (or others) can fall after MIDDLE (which makes them true).

It can then be shown that the original formula has a model just in case the interval encoding has one too. Satisfiability of a 3-SAT formula could thus be established by determining the satisfiability of the corresponding interval algebra assertions. Since the former problem is NP-complete, the latter one must be (at least) NP-hard.

The following theorem extends the NP-hard result for the problem of determining satisfiability of assertions in the interval algebra to the problem of determining closure of these assertions.

Theorem 3: The problems of determining the satisfiability of assertions in the interval algebra and determining their closure are equivalent, in that there are polynomial time-mappings between them.

Proof: *(Sketch)* First we show that determining closure follows readily from determining consistency. To do so, assume the existence of an oracle for determining the consistency of a set of assertions in the interval algebra. To determine the closure of the assertions, we run the oracle thirteen times for each of the $O(n^2)$ pairs *<i,j>* of intervals mentioned in the assertions. Specifically, each time we run the oracle on a pair *<i,j>*, we provide the oracle with the original set of assertions and the additional

assertion *i (R) j*, where R is one of the thirteen simple relations. The relation vector that holds between *i* and *j* is the one containing those simple relations that the oracle didn't reject.

To show that determining consistency follows from determining closure, assume the existence of a closure algorithm. To see if a set of assertions is consistent, run the algorithm, and inspect each of the $O(n^2)$ relations between the *n* intervals mentioned in the assertions. The database is inconsistent if any of these relations is the inconsistent vector: this is the vector composed of no constituent simple relations.

The two preceding theorems demonstrate that computing the closure of assertions in the interval algebra is NP-hard. This result casts great doubts on the computational tractability of the algebra, as no NP-hard problem is known to be solvable in less than exponential time.

Consequences of Intractability

Several authors have described exponential-time algorithms that compute the closure of assertions in the interval algebra, or some subset thereof. Valdes-Perez [1986] proposes a heuristically pruned algorithm which is sound and complete for the full algebra. The algorithm is based on analysis of set-theoretic constructions. Malik & Binford [1983] can determine closure for a fraction of the interval algebra with the exponential *Simplex* algorithm. As we shall show below, their method is actually more powerful than need be for the fragment that they consider.

Even though the interval algebra is intractable, it isn't necessarily useless. Indeed, it is almost a truism of Artificial Intelligence that all interesting problems are computationally at least NP-hard (or worse)! There are several strategies that can be adopted to put the algebra to work in practical systems.

The first is to limit oneself to small databases, containing on the order of a dozen intervals. With a small database, the asymptotically exponential performance of a complete temporal reasoner need not be noticeably poor. This is in fact the approach taken by Malik and Binford to manage the exponential performance of their *Simplex*-based system. Unfortunately, it can be very difficult to restrict oneself to small databases, since clustering information in this way necessarily prevents all but the simplest interrelations of intervals in separate databases.

Another strategy is to stick to the polynomial-time constraint propagation closure algorithm, and accept its incompleteness. This is acceptable for applications which use a temporal database to notate the relations between events, but don't particularly require much inference from the temporal reasoner. For applications which make heavy use of temporal reasoning, however, this may not be an option.

Finally, an alternative approach is to choose a temporal representation other than the full interval algebra. This can be either a fragment of the algebra, or another representation altogether. We pursue this option below.

A Point Temporal Algebra

An alternative to reasoning about intervals of time is to reason about points of time. Indeed, an algebra of time points can be defined in much the same way as was the algebra of time intervals. As with intervals, points are related to each other through relation vectors which are composed of *simple point relations*. These primitive relations are defined in Figure 5.

As with the interval algebra, the point temporal algebra possesses addition and multiplication operations. These operations, whose tables are given in Appendix , mirror the operations in the interval algebra. Addition is used to combine two different measures of the relation of two points. Multiplication is used to determine the relation between two points *A* and *B*, given the relations between each of *A* and *B* and some intermediate point *C*.

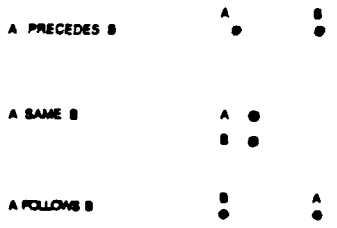


Figure 5: Simple point relations

Computing Closure in the Point Algebra

As was the case with intervals, determining the closure of assertions in the point algebra is an important operation. Fortunately, the point algebra is sufficiently simple that closure can be computed in polynomial time. To do so, we can directly adapt the constraint propagation algorithm of Figure 4. Simply replace the interval vector addition and multiplication operations with point additions and multiplications, and run the algorithm with point assertions instead of interval assertions.

As before, the algorithm runs to completion in $O(n^3)$ time, where *n* is the number of points about which assertions have been made. As with the interval algebra, the algorithm is sound: any relation that it infers between two points follows from the user's assertions. This time, however, the algorithm is complete. When it terminates, the closure of the point assertions will have been correctly computed.

We prove completeness by referring to the model theory of the time point algebra. In essence, we consider any database over which the algorithm has been run, and construct a model for any possible interpretation of the database. If the database is indefinite, a model must be constructed *for each* possible resolution of the indefiniteness.[4]

We choose the real numbers to model time points. A model of a database of time points is simply a mapping between those time points and some corresponding real numbers. The relations between time points are mapped to relations between real numbers in the obvious way. For example, if time point *A* precedes time point *B* in the database, then *A*'s corresponding number is less than *B*'s.

Theorem 4: The constraint propagation algorithm is complete for the time point algebra. That is, a model can be constructed for any interpretation of the processed database.

Proof: *(Sketch)* We first note that the algorithm partitions the database into one or more partial order graphs. After the algorithm is run, each node in a graph corresponds to a *cluster* of points. These are all points related to by the vector *(SAME)*; note that the algorithm computes the transitive closure of *(SAME)* assertions. Arcs in the graph either indicate precedence (the vectors *(PRECEDES)* or *(PRECEDES SAME)*, or their inverses) or disequality (the vector *(PRECEDES FOLLOWS)*). At the bottom of each graph is one or more "bottom" nodes: nodes which are preceded by no other node.

Further, when the algorithm has run to completion the

[4] This demonstrates completeness in the following sense. If there were an interpretation of the processed database for which no model could be constructed, the algorithm would be incomplete. It would have failed to eliminate a possible interpretation prohibited by the original assertions.

graphs are all consistent, in the following two senses. First, all points are linearly ordered: there is no path from any point in a graph back to itself that solely traverses precedence arcs (time doesn't curve back on itself). Second, no two points that are in the same cluster were asserted to be disequal with the *(PRECEDES FOLLOWS)* vector. If the user had added any assertions that contradicted these consistency criteria, the algorithm would have signalled the contradiction.

Note that all of the preceding properties can be shown with simple inductive proofs by considering the algorithm and the addition and multiplication tables.

The model construction proceeds by picking a cluster of points (i.e., a node) at the "bottom" of some graph and assigning all of its constituent points to some real number. The cluster is then removed from the graph, and the process proceeds on with another real number (greater than the first) and another cluster (either in the same graph or in another one). The process is complicated somewhat because some clusters may be "equal" to other clusters (their constituent points may be related by some vector containing the *SAME* relation). For these cases it is possible to "collapse" several (zero, one, or more) of these clusters together, and assign their constituent points to the same *real number*. Some *other clusters* may be "disequal". For these, we must just make sure never to "collapse" them together. Because the choice of which "bottom" node to remove and which clusters to collapse is non-deterministic, the model construction covers all possible interpretations of the database.

Relating the interval and point algebras

The tractability of the point algebra makes it an appealing candidate for representing time. Indeed, many problems that involve temporal sequencing can be formulated in terms of simple points of time. This approach is taken by any of the planning programs that are based on the situation calculus, the patriarch of these being *STRIPS* [Fikes & Nilsson 71].

However, as many have pointed out, time points as such are inadequate for representing many real phenomena. Single time points by themselves aren't sufficient to express natural language semantics [Allen 84], and they are very inconvenient (if not useless) for modelling many natural events and actions [Schmolze 86]. For these tasks, an interval-based time representation is necessary.

Fortunately, many interval relations can be encoded in the point algebra. This is accomplished by considering intervals as defined by their endpoints, and by encoding the relation between two intervals as relations between their endpoints. For example, the interval relation

A *(DURING)* B

can be encoded as several point assertions

A- *(FOLLOWS)* B-
A+ *(PRECEDES)* B+
A- *(PRECEDES)* A+
B- *(PRECEDES)* B+,

where *A-* denotes the starting endpoint of interval *A*, *A+* denotes its finishing endpoint, and similarly for *B*.

This scheme captures all unambiguous relations between intervals, that is all relations that can be expressed using vectors that contain only one simple constituent. It can also capture many ambiguous relations, but not all. One can represent ambiguity as to the pairwise relation of endpoints, but one can not represent ambiguity as to the relation of whole intervals. The vector *(BEFORE MEETS OVERLAPS)* for example can be encoded as point assertions, but the vector *(BEFORE AFTER)* can not. See Figure 6.

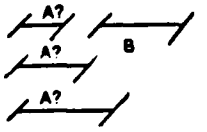
INTERVAL VECTOR	POINT TRANSLATION	ILLUSTRATION
A (BEFORE OVERLAPS MEETS) B	A- (PRECEDES) B- A- (PRECEDES) A+ A+ (PRECEDES) B+ B- (PRECEDES) B+	
A (BEFORE AFTER) B	*No equivalent point form*	

Figure 6: Translation of interval algebra to point algebra

The fragment of the interval algebra that can be translated to the point algebra benefits from all the computational advantages of the latter. In particular, the polynomial-time constraint propagation algorithm is sound and complete for the fragment. This is the interval representation method that Simmons uses in his geological reasoning program [Simmons 83, and personal communication].

This fragment of the interval algebra is also the one used by Malik and Binford [1983] in their spacio-temporal reasoning program. In their case, though, reasoning is performed with the exponential *Simplex* algorithm. This use of the general *Simplex* procedure is not strictly necessary, though, since the problem could be solved by the considerably cheaper constraint propagation algorithm.

Although many applications may be able to restrict their interval temporal reasoning to the tractable fragment of the interval algebra, some applications may not. One program that requires the full interval algebra is the planning system of Allen and Koomen [1983] that we referred to above. In this system, actions are modeled with intervals. For example, to declare that two actions are non-overlapping, one asserts

ACT_1 *(BEFORE MEETS MET-BY AFTER)* ACT_2

As we just showed, this kind of assertion falls outside of the tractable fragment of the interval algebra. In a planner with this architecture, this representation problem can be dealt with either by invoking an exponential temporal reasoner, or by bringing to bear planning-specific knowledge about the ordering of actions.

Consequences of These Results

Increasingly, the tools of knowledge representation are being put to use in practical systems. For these systems, it is often crucial that the representation components be computationally efficient. This has prompted the Artificial Intelligence community to start taking seriously the performance of AI algorithms. The present paper, by considering critically the computational characteristics of several temporal representations, follows this recent trend.

What lessons may we learn from analyses such as this? Of immediate benefit is an understanding of the computational advantages and disadvantages of different representation languages. This permits informed decisions as to how the representation components of application systems should be structured. We can better understand when to use the power of general representations, and when to set these general tools aside in favor of more application-specific reasoners.

A close scrutiny of the ongoing achievements of Artificial Intelligence enables a better understanding of the nature of AI methods. This process is crucial for the maturation of our field.

Appendix: Algebraic Operations in the Point Algebra

Addition and multiplication are defined in the point algebra by the two tables in Figure 7. These operations both have constant-time implementations if the relation vectors between time points are encoded as bit strings. With this encoding, both operations can be performed by simple lookups in two-dimensional (8 x 8) arrays. Alternatively, addition can be performed with an even simpler 3-bit logical *AND* operation.

+	<	<=	>	>=	=	-=	?
<	<	<	0	0	0	<	<
<=	<	<=	0	=	=	<=	<=
>	0	0	>	>	0	>	>
>=	0	=	>	>=	=	>	>=
=	0	=	0	=	=	0	=
-=	<	<	>	>	0	-=	-=
?	<	<=	>	>=	=	-=	?

x	<	<=	>	>=	=	-=	?
<	<	<	?	?	<	?	?
<=	<	<=	?	?	<=	?	?
>	?	?	>	>	>	?	?
>=	?	?	>	>=	>=	?	?
=	<	<=	>	>=	=	-=	?
-=	?	?	?	?	-=	?	?
?	?	?	?	?	?	?	?

Key to symbols:

0	is	(), the null vector
<	is	(PRECEDES)
<=	is	(PRECEDES SAME)
>	is	(FOLLOWS)
>=	is	(SAME FOLLOWS)
=	is	(SAME)
-=	is	(PRECEDES FOLLOWS)
?	is	(PRECEDES SAME FOLLOWS)

Figure 7: Addition and multiplication in the time point algebra

References

[Allen 83] Allen, J. F.
Maintaining Knowledge About Temporal Intervals.
Communications of the ACM 26(11):832-843, November, 1983.

[Allen 84] Allen, J. F.
Towards a General Theory of Action and Time.
Artificial Intelligence 23(2):123-154, 1984.

[Allen & Hayes 85] Allen, J. F. and Hayes, P. J.
A Common-Sense Theory of Time.
In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 528-531. The International Joint Conference on Artificial Intelligence (IJCAI), Los Angeles, CA, August, 1985.

[Allen & Koomen 83] Allen, James F., and Koomen, Johannes A.
Planning Using a Temporal World Model.
In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 741-747. The International Joint Conference on Artificial Intelligence (IJCAI), Karlsruhe, W. Germany, August, 1983.

[Fikes & Nilsson 71] Fikes, R., and Nilsson, N.J.
STRIPS: A new approach to the application of theorem proving to problem solving.
Artificial Intelligence 2:189-208, 1971.

[Malik & Binford 83] Malik, J. and Binford, T. O.
Reasoning in Time and Space.
In *Proceedings of the Eighth Int'l. Joint Conference on Artificial Intelligence*, pages 343-345. The International Joint Conference on Artificial Intelligence (IJCAI), Karlsruhe, W. Germany, August, 1983.

[Schmolze 86] Schmolze, J. G.
Physics for Robots: Representing Everyday Physics for Robot Planning.
PhD thesis, The University of Massachusetts, Amherst, 1986.

[Simmons 83] Simmons, R. G.
The Use of Qualitative and Quantitative Simulations.
In *Proceedings of the Third National Conference on Artificial Intelligence (AAAI-83).* The American Association for Artificial Intelligence, Washington, D.C., August, 1983.

[Valdes-Perez 86] Valdes-Perez, R. E.
Spatio-Temporal Reasoning and Linear Inequalities.
1986.
Unpublished A.I Memo, Massachusetts Institute of Technology Artificial Intelligence Laboratory.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.