Proceedings from the

“From Formal Systems to Practical Systems”

October 27-29, 1988

Editors:
Josh Tenenberg        Jay Weber        James Allen

Department of Computer Science
University of Rochester

Supported by the
Northeast Artificial Intelligence Consortium,
the
American Association for Artificial Intelligence,
and the
University of Rochester Computer Science Department
Preface

A considerable amount of money and research effort has been expended toward the construction of planning agents. One can scarcely consider any intelligent endeavor which does not require the purposeful composition of actions (physical or mental) in subservience of an agent's ultimate goals. On the surface, then, it might appear surprising that there is little agreement as to what planning is, what representations are sufficient for adequately reasoning about change and persistence, and what control strategies will efficiently employ the scarce resources available to the agent. However, for those actively engaged in planning research, there is little surprise about the lack of planning universals; it has become evident that planning encompasses issues of profound import that have been the subject of inquiry in linguistics, philosophy, mathematics and psychology for considerably longer than the 30 years of research in planning. These issues include reasoning about concurrency, duration, and change, definitions of meaning and reference in language (both formal and natural), notions of truth, uncertainty, probability, evidence, knowledge, intention, preference and rationality.

It is also true that we cannot afford to wait for definitive solutions to these formal problems before building systems of practical consequence. Funding agencies, the commercial marketplace, and the competitive efforts of our colleagues will oblige many of us to transfer knowledge from the theoretical to the practical, and likewise, from the practical back to the theoretical. It is likely that the ultimate measure of our formalisms is the extent to which they can be transferred to practical systems. It is hoped that this workshop will serve to facilitate this transfer and to hasten the rate at which it is attempted and successfully achieved.

We consider this workshop unique in the rejection of a specific focus such as "Representing Knowledge and Action", or "The Frame Problem" (without casting aspersions on such workshops). In addition, we consider planning not simply in its relation to automated agents, but also with respect to natural language understanding, scheduling, control, plan recognition, and multi-agent tasks. One of the risks of such an approach is the loss of focus; perhaps the range of concepts and topics is too broad to be successfully integrated. We happily take this risk, however (and, thankfully, so do our funding sources), given the potential synergistic benefits of bringing together a large variety of researchers from across the planning spectrum. There seems to be sufficient accumulated social and scientific history within this discipline to begin to chart the interplay between theory and practice.

In particular, we view the following questions as being of crucial importance, many of which serve as the focus of the technical talks:

- How do various applications place constraints on the planning mechanism?
- How far into the future does a planner need to reason for different applications?
- What real-time constraints are placed on the planner's performance?
- How do the interactions between real agents depend on reasoning about each other's plans?
The workshop is organized into five parts: an overview of planning motivations (consisting of four separate introductory talks), and four panels. For each panel we chose a panel organizer to whom we provided a general topic. The organizer was responsible for choosing panel members, and for giving specific instructions to these panelists as to the actual content of their presentations. We recommended that the panels consist of a position presented by each panelist, followed by a commentary showing how the positions relate to each other and to the panel theme. The results were quite different for each panel, largely due to the diverse ways in which the panel concept evolved for each organizer. We are pleased, however, that many of the positions represent the authors' recent thoughts and work in progress. To encourage this, all contributions were by invitation and not review.

We believe the theme of this workshop to be both timely and important. However, it did not originate with us, but rather with Northrup Fowler of RADC. We can say with confidence that without his original suggestion and continued inputs this workshop would not have taken place. In addition, we received a considerable amount of help in shaping the workshop from Austin Tate. To our other speakers we are additionally grateful, especially to those who in addition to volunteering to speak accepted some of the organizational burden.

We would also like to acknowledge the people at NAIC and AAAI that made this workshop financially possible, and the following graduate students who gave their time: Lou Hoebel, Nat Martin, Keri Jackson, Leo Hartman, Hans Koomen, Michael McInerny, Steve Feist, Steve Whitehead, David Traum, and Elizabeth Hinkleman. We additionally thank the contribution by the Computer Science department staff, in particular Patricia Armstrong, who took care of an unimagineable host of difficulties with humor and aplomb.

Josh Tenenberg
Jay Weber
James Allen
# Contents

Introduction: Motivations for Planning 3  
Robotics and Planning (Stanley J. Rosenschein) 4  
Natural Language and Plan Reasoning (James Allen) 6  
Knowledge Based Scheduling: A Review (Mark S. Fox) 10  
Controlling Problem Solving Through Planning (Victor R. Lesser) 12  

Panel: Planning and Execution 15  
Approaches to Planning and Execution Systems (Austin Tate) 16  
Situated Control Rules (Mark Drummond) 18  
Truth Maintenance-Based Planning with Error Recovery (Paul Morris) 34  
Action Networks (Nils J. Nilsson) 36  
Planning and Execution (Stanley J. Rosenschein) 69  

Panel: Micro vs. Macro Planning 71  
Planning in the Large (Mark S. Fox) 72  
Multiagent Planning and Scheduling (Amy Lansky) 74  

Panel: Planning and Plan Recognition 77  
Plan Recognition (Henry A. Kautz) 78  
Probability, Language, and Plan Recognition (Eugene Charniak) 79  
Metrics for Explanation Simplicity (C. Raymond Perrault) 81  
Formal Theories of Plan Recognition (Henry A. Kautz) 82  
Towards Useable Plan Recognition (Bradley A. Goodman) 91  

Panel: Reactive vs. Strategic Planning 99  
From Planning to Instruction Use (David Chapman) 100  
An Embedded Reasoning and Planning System (Michael P. Georgeff) 105  
On the Value of Goals (Thomas Dean and Michael P. Wellman) 129  
Planning and Behaving (Drew McDermott) 141
Introduction:

Motivations for Planning

Chair:
Josh Tenenberg, University of Rochester

Members:
James Allen, University of Rochester
Mark Fox, Carnegie Mellon University
Victor Lesser, University of Massachusetts
Stan Rosenschein, Teleos Research
Abstract

For two fields as closely related as Robotics and AI Planning, it is remarkable how little contact there has been at the level of fundamental theory and programming methods. This talk explores some of the reasons for this state of affairs in the hope of stimulating future research at the boundaries between the fields. Among the factors considered are:

1. differences of "cultural background" e.g., an orientation toward discrete vs. continuous mathematics or towards formal philosophy vs. control engineering;

2. different tastes about which tasks are of interest, e.g., tasks in which the agent's behavior depends only on physical characteristics of the immediate environment (such as grasping an irregularly shaped cup) vs. tasks in which the agent's behavior depends on qualitative aspects of the world spatially and temporally removed from the immediate situation (such as locating an administrator to ask about the summer course schedule.)

3. different assumptions about the amount and kind of information available, with many robotic control techniques requiring high-precision, quantitative parameters (positions, velocities, shapes, forces) before they can be applied at all, whereas techniques of AI planning and reasoning, when applied to representations with this degree of precision and asked to "reason" rather than "calculate," become hopelessly bogged down; and

4. a mindset in each field that assumes that certain modular capabilities ("reasoning," "visual recognition," "grasping") can be imported from the other field as primitives, whereas, in fact, the modularization of these capabilities is highly problematic.

Recently there has been growing interest in topics that bear on the connection between AI and robotics. Some AI planning researchers, for instance, disenchanted with the high computational complexity of traditional symbolic inference techniques, have been exploring more direct methods of linking sensing and acting, and robotics has emerged as an attractive test domain for these new methodologies. The talk highlights some of the substantive areas where a joint attack by roboticists and AI researchers could prove fruitful:

- real-time operation
- hierarchical models of action
- the interface between the numeric and the symbolic, e.g., a deeper conceptual integration of "motion planning" with "AI planning"
- qualitative modeling of physical systems
- the control of multiple degrees of freedom
- statistical strategies for overcoming uncertainty
Natural Language and Plan Reasoning

James Allen
Department of Computer Science
University of Rochester

Introduction

Natural Language understanding and generation provide a rich set of motivations for plan reasoning. In fact, I would claim that any planning problem that can be posed can be recast as a natural language understanding problem. In this short note I want to outline three particular uses of plan reasoning in natural language understanding. While I will not explicitly consider issues of natural language generation, I think the need for plan reasoning in order to generate appropriate language should be clear from the examples based on understanding. The three, ordered in terms of increasing distance from traditional planning models are based on analyzing causality, motivation, and the problem solving process itself.

Causal Analysis

As claimed above, almost any planning problem can be recast as a language understanding problem simply by describing the problem in language. More importantly, in order to recognize the appropriate meanings of sentences requires a causal analysis exactly as found in plan reasoning systems. For example, consider the following two sentences:

He opened the door with a key.

He opened the bottle of pills with a key.

In the first, the usual interpretation is that the key was used to unlock the door and thus enable the actor to open it. In the second, it is more likely that the key was used as an instrument to pry off the lid. To arrive at these solutions, an NL system must solve the following problems, which are exactly the type of problems given to a plan reasoning system:

How do you use a key to open a door?

How do you use a key to open a pill bottle?

In the first case, knowledge about how doors are opened would contain information about locks, and the fact that locks are operated by keys, etc. In the second, the problem is harder to represent - but general knowledge about pill bottles and methods of opening such containers may contain the technique of prying off the top with an instrument (such as a key). But the point here is, once you solve the planning problem you've solved a large part of the NL understanding problem.

As one more example, to understand the following story, the NL system must essentially have a solution to the Yale shooting problem (among other things):
The villian loaded his gun and hid in the bushes. As the crown prince walked by, he aimed and pulled the trigger. The next day, war broke out.

There is an interesting difference in emphasis between the plan reasoning required in an NL system and that found in a traditional planner. An NL system uses plan reasoning to explain observed actions and situations, whereas a planner typically reasons in order to solve a goal. This difference has motivated much of the work in plan recognition, but I think that the examples above show that the same fundamental problems arise in both. In fact, I would claim that as so-called "planning" and "plan recognition" systems become more sophisticated, the differences between the two will virtually disappear.

**Motivation Analysis**

The next set of examples show that plan reasoning is required, or at least is the best model we have so far, for reasoning about the intentions underlying language. For example, the sentence "Do you have a watch?" has at least three common interpretations:

1. it is a request for the time, and could be replied to by "Yes. It's 11:30".
2. it is a yes/no question about whether the hearer has a watch and could be replied to by "Yes, I have one at home".
3. it is an offer to give/lend the hearer a watch, and could be replied to by "No, I don't, thanks for offering".

A natural language dialog system must first be able to generate these different readings, and be able to identify the appropriate one in a given context. The best solution to this problem so far has resulted from formulating the analysis as a plan reasoning problem. In this approach, utterances are formalized as actions called speech acts. The prerequisites and effects for speech acts are defined in terms of the conversant's different beliefs and goals. In particular, speech acts allow one agent to influence the beliefs and goals of another. For example, the act of S informing H that it is raining would be defined so that

- if a sincere inform, then S believes that it is raining;
- if a successful inform, then H will believe that it is raining (or at least believe that S believes this is so).

Without going into detail, a plan formulation could be used directly to identify the following prerequisites for the three interpretations of "Do you have a watch".

As a request:
the speaker doesn't know the time
the speaker believes the hearer has a watch

As a yes/no question:
the speaker doesn't know if the hearer has a watch
As an offer:
the speaker believes the hearer doesn't have a watch
the speaker is willing to give the hearer the watch.

This analysis is not precoded for each sentence. Rather it is produced by a plan reasoning process that starts with an initial literal interpretation of the sentence. Except for the fact that the plan reasoner needs to reason about other agents' knowledge, and make inferences based on their knowledge, the plan reasoning is otherwise fairly straightforward. For instance, the request interpretation is generated from the literal yes/no question by reasoning about what could be enabled if the hearer had a watch:

- having a watch, which enables
- looking at the watch, which has the effect of
- knowing the time, which enables
- telling the requestor the time, which has the effect
- the requestor knows the time.

There is a serious question about how different interpretations are evaluated. Unlike in planning, where a single solution to a problem may suffice, the plan reasoning for speech acts must identify which of the possible interpretations was intended by the speaker. This is typical of plan recognition problems.

## Analyzing Problem Solving

Another set of understanding problems seems to require an explicit model of the problem solving process itself. This is easiest to demonstrate using simple narratives. For example, consider the following continuations after the sentence:

I went to buy a ticket at the theater, but found that I didn't have enough money.

1. I asked the person behind me for a dollar.
2. I asked the teller where the nearest bank was.
3. I offered the teller a coupon for a free car wash.
4. I walked behind the theater and found an open door.
5. I decided to go home and catch the later show.
6. I decided to go to a bar instead.

Each of these is a plausible continuation. To analyze each, however, requires that one can represent the original plan, the problem in the plan, and the various ways in which a person might deal with the situation. Existing problem solvers suggest a rich model for this. In particular, the agent might attempt to solve the problem directly by getting more money (1 & 2) or abandon part of the plan (buying the ticket) and achieve the goal in another way
(i.e. bartering for the ticket (1) or sneaking in (4)), or the agent might abandon the specific goal and pursue another goal with similar effect (ie. see film later (5), or go to a bar (6)).

This form of reasoning seems to require an explicit model of the problem solving process and thus moves far beyond what existing planners might do. The problems it addresses, namely dealing with failure in execution, are of primary concern to recent research in plan-directed execution.

Conclusion

Natural language is an excellent application for a wide range of planning problems. While initial work has primarily focussed on plan recognition processes, it is easy to find problems that require goal achievement, and that require a sophisticated model of plan execution and repair. Because complex situations are much easier to describe in language than to create in a physical realization, natural language plan reasoning systems typically deal with many issues in advance of their consideration in traditional planning applications. For example, it was the natural language applications that forced the integration of explicit models of beliefs, desires and intentions into the planning literature and first started to address the issues in multi-agent cooperation.

In addition, once actions such as speech acts are introduced, many representation issues arise that can be avoided if all actions are purely physical movement. In particular, there is no fixed set of physical realizations for a speech act - so the action cannot be defined in terms of some set of “primitive” transformations. Speech acts can only be defined relative to the speaker’s and hearer’s beliefs and intentions. In general, this leads to the view that particular actions cannot be defined independently of the context in which they appear. For example, consider the action of signalling a left turn in a car by putting your arm out the window. All instances of moving your arm in such a way are not necessarily signalling actions. Rather the setting must be appropriate and the conventions established before the arm movement counts as a signal.

Natural language applications produce a new set of representation issues that can often be avoided in simpler, (primarily causal) domains. These issues rapidly become relevant as planning work moves towards more realistic domains with multiple agents, external events, and must explicitly reason about and control its execution in real time.
Knowledge Based Scheduling: A Review

Mark S. Fox
Robotics Institute and Computer Science Department
Carnegie Mellon University

Extended Abstract

We define scheduling to be the assignment of temporal intervals to activities for multiple jobs such that they

- obey the temporal restrictions of production processes and the capacity limitations of a set of shared resources, and
- balance a conflicting set of preferential concerns (e.g., meeting deadlines, minimizing work-in-process time, etc.)

Scheduling differs from planning in that a one or more (alternative) plans are known a priori. The application of Artificial Intelligence techniques to scheduling and control problems is becoming wide spread. Rather than there being a single knowledge based approach, each scheduling problem poses a unique set of requirements resulting in widely varying problem solving architectures. The goal of this presentation is to review the problem solving and knowledge representation techniques, used to provide both predictive and reactive scheduling in a variety of situations chosen from the domain of factory scheduling: including FMS, flow and job shops. Approaches to be reviewed include:

- Mathematical programming: Mathematical programming tends to focus on optimization techniques. Simplex and karmarkar methods are linear programming techniques that find optimal solutions to models composed of linear inequalities. They provide good performance on problems with hundreds of variables and constraints. Scheduling problems cannot be represented using linear models. Instead nonlinear models or integer approximations must be used. Solution techniques primarily focus on reducing the problem to a linear approximation that can be solved and using the solution to guide a more general search process (e.g., Lagrangian relaxation). Search is performed using branch and bound techniques, a generalization of A*. Relatively little success has been achieved using these approaches due to the complexity of the scheduling problems.

- Dispatch simulation: Operations research has taken a second tack to scheduling which does not focus on optimization. Instead myopic (local) heuristics are used to make scheduling systems. Essentially a simulation is performed of jobs moving through a plant. As a resource becomes free, a heuristic is used to select the job to work on next. Global optimization of a single criterion such as tardiness is attempted via local selections. Examples of local heuristics include choosing the job with the earliest due date (EDD) or shortest processing time (SPT).
Knowledge Based Scheduling: A Review

- Expert systems: A common approach to solving the scheduling problem in industry is to identify and extract expertise from human schedulers. In cases where the scheduling problem is of low complexity and fairly static, this approach works. But in most cases, the scheduling problem is so complex that there does not exist a person whose performance is worth emulating.

- Constraint labelling: CSP techniques are beginning to be applied to scheduling. It takes two forms. The first is to recast the scheduling problem as finding assignments to a set of variables where each variable is an activity and its assignment is its position in a queue for a machine. Predcence constraints between activities restrict the set of allowable position assignments.

  A second approach is to treat the assignments to the activity variables as time intervals. Using CPM-like windowing techniques across temporal relations, time bounds for activities can be refined.

- Constraint-directed search: There are four types of constraint directed search that have been investigated:

  Constraint-based heuristic search is primarily a heuristic search technique that uses constraints as both generators and tests.

  Hierarchical constraint-based search reformulates a problem into a hierarchy of simpler versions whose solutions are encapsulated in the form of constraints and are used to guide search at the next level.

  Macro-Oppportunistic search opportunistically switches between a job and resource perspective when making scheduling decisions. If a job is of high priority, then scheduling will focus on scheduling the activities of the job. If there is a high degree of contention for resources, then scheduling will focus on scheduling only the activities that require that resource.

  Micro-Oppportunistic search takes opportunism one step further to the point where switching occurs among individual activities. The basic scheduling look is to analyze the problem in order to select the next activity to schedule.

- Distributed negotiation: A market approach is taken where an agent representing a job calls for bids on activities it would like to have performed. Other agents representing resources such as machines bid on each of the activities. The job agent selects the appropriate bids.
Controlling Problem Solving Through Planning

Victor R. Lesser
Computer Science Department
University of Massachusetts

Extended abstract

Using planning as a way of controlling a problem solver is not a new idea [1, 2, 3, 4], but this approach to control is still not well known. The control problem involves deciding what actions the problem solver will take, and where each action applies particular knowledge to particular data. As AI techniques are applied to harder problems which involve complex, multi-dimensional search spaces and where there are many possible approaches to tackling the problem solving, and where the problem solver is interacting in real-time with its environment and other problem solvers, sophisticated control based on planning will be required. By making control explicit in terms of a problem-solving plan, the following apply:

• decisions from a more global perspective can be made about when to revise or terminate problem solving,

• estimates for the time to complete problem solving can be developed which is important for real-time control,

• external problem-solving resources such as a modifiable sensor can be controlled in the context of a control plan to improve problem solving.

• effective coordination among cooperating problem-solvers can be facilitated by sharing problem-solving plans, and

• finally, the system becomes more explainable and modifiable,

Because there are relatively few examples, a specific planning technology has not emerged as the most appropriate for control. I will present a number of different examples of the uses of planning for control to provide a context for workshop discussion. It is hoped that in this way some intuitive understanding can be gained about what the unique problems are in exploiting planning technology for use in control.

The first system to be discussed is the blackboard control architecture developed by B. Hayes-Roth, called BB1 [4]. As part of this architecture, she introduced generic control knowledge sources (Initialize-Prescription, Update-Prescription and Terminate-Prescription) that implement hierarchical plans for control. The plans were extremely simple—a sequence of actions with a specification of the objective of the plan. (Clancey, in his work on Neomycin [1], uses a similar hierarchical plan structure for control, called metatasks.) Plans are used to implement high-level strategies for problem solving. These plans are not constructed dynamically (i.e., built in scripts) but are dynamically instantiated based on either data-directed conditions coming from the current state of problem solving on the blackboard or subgoals being posted by higher level plans. The expansion of these plans is incremental.
based on the completion of the previous step in the plan. The action steps of the plan do not result in the direct execution of specific knowledge sources but rather affect the evaluation function used to choose the next knowledge source to execute from the agenda of currently executable knowledge sources. One of the interesting aspects of this work is that multiple plans can be simultaneously active, the evaluation function taking into account the possible contradictory criterion for what next action to choose.

A more dynamic view of planning for control occurs in the work of Durfee and Lesser [2] on control of an interpretation system. They view the control component as not only resolving uncertainty about what actions will lead to desirable solutions but also about what solutions will be desirable. Whereas most planners are given a set of well-defined goals, a problem solver’s control planner must often resolve uncertainty about the goals to achieve. By specifying the characteristics of likely solutions (defining more specific long-term goals) in advance, control plans can be developed to process only the most useful low-level data. Thus, they view an important first step in applying planning for control as being the development of a mechanism for constructing the specific goals to be achieved. By abstracting the problem-solving state they show how the planner can recognize, at a gross level of detail, competing and compatible solutions and can roughly predict the importance (likely of being the correct interpretation) and the expense of developing potential solutions.

Based on this information, they develop a high-level plan to resolve the uncertainty about which competing solution is more likely and simultaneously develop the specific characteristics of the solution. In forming this plan they used heuristics such as "work in common areas," "work in inexpensive areas," and "work in distinctive competing areas" to order the steps in the plan. A plan represents specific actions for the near future and more general actions for the distant future. The detailed plans are constructed based on the planner having a model of the pre- and post-conditions of the potential actions available. By forming detailed plans only for the near future, the problem-solver does not waste time planning for situations that never arise; however, by sketching out the entire plan, details for near-term can be based on a long-term view. As problem solving proceeds, the plan must be monitored (based on the model of actions) and repaired when necessary and new actions for the near future are added incrementally. Thus, plan formation, monitoring, modification and execution are interleaved.

Durfee and Lesser [5, 6] also show how such problem-solving plans can be used for coordination among cooperating experts. They use not only the plans but also rough estimates of the time required for high-level steps in the plan to develop coordinated plans among experts. Again the issue is a scheduling problem of what sequence to do the plan steps in order to facilitate cooperative behavior. One of the interesting characteristics of this ordering process is the fact that by doing certain steps before others you can either speed up or slow down the time required to complete subsequent steps. They also use these plans to develop a communication plan about how and when results should be integrated. Another interesting aspect of this generation of coordinated plans is how to treat the time estimates given for plan steps. One approach is to go strictly by the estimates assuming best case; this leads to very tight and effective coordination if the estimates are upper-bounds but also a lot of overhead to do replanning in case they are underestimates. An alternative approach is to plan for a certain amount of under-estimation and build this into the coordination plans; in this case, playing off less overhead for replanning against less effective coordination.
It is not possible to draw conclusions about the type of planning technology most appropriate for control from these few examples. However, one important issue is that of uncertainty in the results of problem-solving actions. This leads to approaching the planning task incrementally, doing detailed planning of long sequences of actions seems inappropriate; instead develop a high-level plan then refine incrementally as needed. Another issue seems to be the idea of merging plans and then reordering of plan steps based on relationships among steps of different plans. How much does this relate to NOAH's least commitment strategy? Finally, what is the space that you develop control plans in; what are the objectives and goals that are used by the planner to develop control plans?

References


Panel:
Planning and Execution

Chair:
Austin Tate, AIAI, Edinburgh University

Members:
Mark Drummond, NASA AMES
Paul Morris, Intellicorp
Nils Nilsson, Stanford University
Stan Rosenschien, Teleos Research
Approaches to Planning and Execution Systems

Austin Tate
AIAI
Edinburgh University

Abstract

The individual members of the panel will describe their recent work as it relates to the interplay between AI planning and execution systems. To provide a framework in which workshop participants can consider the presentations, consider the following breakdown of aspects of planning and execution.

There are a range of approaches to the construction of systems which deal with planning and execution. Some have distinct components for the reduction of task specifications and goals to plans of action and for the reliable execution of those plans in an uncertain world. Others seek a more uniform architecture where there is little or no explicit distinction in the phases. Some systems seek to take a hierarchical decomposition approach to planning and produce more detailed plans which take into account execution problems (by adding lower level information, adding error handling paths, etc.). Other systems take an execution system focus working in terms of task achieving behaviours and seek to push this to increasingly higher planning levels. The panel members will cover an interesting range of these options.
Questions to consider with respect to Planning & Execution Systems

- Are there similarities or differences in the planning architecture and the execution system architecture?
- What is the means of communication each way between planning & execution systems?
- What is the level of provision for feedback between the execution system & planner?
- What mechanisms are present for synchronising the work of the planner and execution systems?
- What level of knowledge is available to the planner & execution system?
  - knowledge about how to plan or execute
  - knowledge about the application domain
  - knowledge about the specific task(s) being handled
  - knowledge about the state of the environment
- How is such knowledge described?
- Are real-time constraints allowed for?
- How autonomous can the execution agent be?
- How is uncertainty in the execution environment handled?
- Are asynchronous changes to the task allowed for?
- When to plan and when to act?

Input from other Workshop members

The panel will welcome your constructive input to the discussion. Some prior consideration of the presentation of other work with respect to the structure outlined above would be especially welcome.
Situated Control Rules

Mark Drummond
NASA Ames Research Center

Abstract

This paper presents plan nets and situated control rules. Plan nets are a language for describing the basic behavioural abilities of a robot. Situated control rules are the result of planning, and are used to constrain the behaviours produced by plan nets. Situated control rules are generated by a temporal projection component if there is time to do so. Each situated control rule characterizes the performance of possible actions in terms of the robot's current environment. Situated control rules do not view plan execution as sequence following, and so provide a better means of handling action failure. Action can be taken without advance planning, simply by executing the plan net. If situated control rules exist, they can be used to achieve global goals by making local decisions. The planned behaviour produced by a system using this approach is a product of interactions between the plan net and any situated control rules that have been generated.

1 Introduction

Programming a robot involves more than simply programming a computer. Modern robots are typically under computer control, so programming a robot almost always involves programming a computer. But a robot introduces problems that are qualitatively different from any found in traditional computer programming. Robot programming involves realizing actions in a physical environment. Physical environments bring uncertainty into the otherwise deterministic world of the computer. It is the effective management of this uncertainty that makes robot programming so hard.

AI planning has often studied robot programming problems. The blocks world and rooms world domains in the planning literature are used to express high-level robot programming problems. In these domains, plans are viewed as programs for controlling the behaviour of a robot. Typically, AI planning systems use operators to describe the effects of domain actions. Plans are expected to be total orders or partial orders of operators. Thus, the business of planning is seen as one of sequence formation, and plan execution is seen as sequence following.

There are at least two serious problems with viewing plans as sequences of operators. First, there is no sure way to realize the actions that a plan's operators describe. Executing an instruction inside the computer controlling a robot may or may not realize the desired action in the environment. There is no way to ensure correct action realization. In contrast, when programming only a computer we rely on descriptions of machine instruction behaviour provided by the machine's manufacturer. If the machine fails to realize these descriptions, we should complain to the manufacturer. The computer is expected to live up to the promises of its action descriptions. Not so with an external physical environment. For a robot, the connections between the operators and the actions they describe is probabilistic; the action realization mechanism cannot be guaranteed.
Since there is no way to guarantee action realization, plans cannot be expressed as sequences of operators. Actions cannot necessarily be realized in sequence; explicit checks must be carried out to determine whether or not actions have been taken as specified in the plan. This means that a plan must be a structure that describes operators in terms of the global conditions under which their realization is appropriate. STRIPS triangle tables are one way of doing this (Nilsson, 1984). So are Schoppers' (1987) Universal Plans. The relationship between triangle tables, universal plans, and the work reported here is described in detail in section 5. The central idea is enough for now: plans are not sequences to be executed; instead, plans are specifications of what must be done in specific situations.

The second problem with viewing a plan as an operator sequence is that not all behaviour can be planned: low-level behavioural competence must be informed by plans, not defined in terms of them. A robot must be able to act without plans. This point has been well made by Agre and Chapman (1988). When there is time to plan, behaviour can be more goal-directed. But when situations warrant, action must be taken without advance planning. A plan should be only one of many possible resources to action. Plans must inform action, and should not be solely responsible for generating it.

2 Overview

This paper presents plan nets and Situated Control Rules (SCRs). Plan nets are a language for describing the basic behavioural abilities of a robot. SCRs are the result of planning, and are used to constrain the behaviours produced by plan nets. SCRs are generated by a temporal projection component if there is time to do so. Each SCR characterizes the performance of possible actions in terms of the robot's current environment. SCRs do not view plan execution as sequence following. Action can also proceed without advance planning, simply by executing the plan net. Planned behaviour produced by the system is a product of interactions between the plan net and any SCRs that exist.

A system-level sketch is given in figure 1. Both planned and unplanned behaviour can produced by the execution component. This component accepts a plan net, and reads it as a nondeterministic program to be executed. It also accepts data from sensors describing the state of its environment. Planned behaviour is the result of SCR input. The execution component will always check to see if any SCRs exist that deal with the current situation. If there is an SCR that applies to the current situation, its advice about what to do next will be heeded. If there are no applicable SCRs, unplanned execution is still possible. Without reference to the SCR input from the plan projector, the execution component can simply select and attempt to execute an action that the plan net describes as enabled in the current state.

The plan projector accepts the same plan net as the execution component. It also accepts goals, read as non-effective specifications of desired future behaviours. The plan projector analyzes the plan net in terms of these goals. Critical choice points in the future are recognized, and SCRs are generated for each. A critical choice point is a situation during execution where it is possible for the execution component to take a locally possible action that does not lead to a globally acceptable solution.

The process of SCR synthesis takes time. In some cases, the execution component will act before the projection process is complete. This only affects the overall quality of the
system's behaviour, and not its basic ability to act and plan independently. The hope is that the projector can always remain a step or two ahead of the execution component. If this does not happen, then the system must sometimes perform "real-time backtracking", to get out of unforeseen difficulties (like painting itself into a corner).

In order to focus the discussion this paper considers a simple blocks world problem that demands extreme robustness on the part of the execution component. The example problem is only used to present plan nets and SCRs. Our approach does not depend on any specific aspects of the example. This problem is presented in section 3. Section 4 presents plan nets and SCRs. Section 5 compares this to other work in the field.

3 A Planning and Control Problem

Consider the following plan generation and execution problem (Fox & Kempf, 1988). We are given a table on which to assemble three blocks in a row; block A on the left (location 1), block B in the middle (location 2), and block C on the right (location 3). The blocks are initially available for placement, and there is an unspecified means for transporting each block to its target location on the table. The exact means for moving the blocks does not matter: given that a block is available it may be deposited. The overriding constraint is that block B cannot be deposited last. Once A and C are in place, there is not enough clearance to place B. B must be swept in from the left or from the right, and cannot be placed if A and C are already down. We call this the BNL (B Not Last) problem.

The BNL problem is under the control of a particular plan execution component. A plan must be produced for the execution component which is as flexible as possible. If a block can be deposited then the plan must so instruct the executor. If a block cannot be deposited according to the constraints then the plan must prevent the executor from
attempting to deposit the block. The execution component must never deadlock in a state from which no progress is possible. This would happen, for instance, if $A$ were in place, and $C$ was placed next. $B$ could then not be placed last.

It takes four totally ordered plans or three partially ordered plans to represent a solution to the BNL problem. See figure 2. Block placements are indicated by \( \downarrow \)'s. Thus, \( \downarrow A \) indicates the placement of block $A$ on location 1. The three partial orders given characterize four total orders. Partial orders are more expressive in the sense that a partial order can contain a set of total orders in its extension. There is some overlap in the extensions of the partial orders given. All three partial orders are necessary however; if any two of the partial orders are passed on to the execution component it is not possible to realize all legal assembly strategies.

If one partial order is thought of as one plan then there is no one plan that can be given to the execution component to solve the BNL problem. For instance, suppose that the executor is given the partially ordered plan which constrains $C$ to be deposited last, preceded in any order by $A$ and $B$ (the partial order on the left of figure 2). This plan does not specify that block $C$ can be deposited first, even though such an action is consistent with the problem definition.

There are different possible assembly strategies, and no one partial order can describe all of them. To deal effectively with this problem, we must be able to represent and reason about disjunction within a plan.

We can deal with disjunction by defining a plan to be a set of partial orders; unfortunately, this does not solve the entire problem. Given the set of 3 partial orders necessary, how does the execution component index the appropriate “planned” action? The plan is a set of partial orders and the set is read as a disjunction. We expect that the execution component will simply perform actions in a sequence compatible with one of the partial orders it has been given. As explained in section 1, this will not work in general.

The view of plan execution as sequence following will not work, since actions often fail. Actions fail when the execution component cannot realize the action that an operator describes. Action failure is a real and common problem. Simply having an operator which characterizes an action is no guarantee of being able to make that action occur when required.

Our approach does not depend on the view of plan execution as sequence following. Instead, the execution component indexes appropriate operators by its current environmental conditions. This approach handles the problem of action failure. If an action is attempted and fails, the environment will reflect the results of the action that was actually realized.

\[ \text{Figure 2: The three partial orders needed for the BNL problem} \]
The state of the environment following the failure can be used to determine the appropriate action to perform next. If the "current" operator is successfully translated into physical action, then the next action to perform is simply one of those that follow in the relevant partial order. Successful execution is also captured by our indexing approach: when actions are realized as predicted, the environment will be in the state predicted by the planner, and the preconditions of the next operator will actually hold. So when actions occur as predicted, the indexing approach generates the same behaviour as the sequence following approach. But when actions fail, the sequence following approach fails also.

The problem is even harder when there can be parallel actions. Consider the initial situation for the BNL problem, where all the blocks are available. According to the constraints, blocks A and B could be placed in any order, or in parallel. Similarly, blocks B and C could be placed in parallel. If B is placed first, then A and C could be placed in parallel. This sort of contingent causal independence makes disjunctive action indexing even more difficult. It is exactly this combination of difficulties that we address.

3.1 Hand-coding is Not the Answer

Some might suggest that we simply write a program for the BNL assembly controller and forget about the problem. "Planning is too hard", some say, and so we should hand-code all the behaviours we would like to see the system exhibit. This is impossible in general as there will often be information available only at plan execution time, when the skills of the behaviour-designer are no longer available. The system must be able to synthesize and compose its own behaviours, and this means doing planning.

4 Our Approach

4.1 The Nature of the Search

The current paradigm for plan generation begins with NOAH (Sacerdoti, 1975) and NonLin (Tate, 1977); this approach has been extended by others (Vere, 1981; Wilkins, 1984, Currie & Tate, 1985), but the idea remains the same. A planner searches through a space of incomplete plans, each of which is partially ordered. Each plan is incomplete in the sense that it must be further refined to be considered acceptable. Transitions through the space of partial plans correspond to plan refinements; typically, operator introductions, variable bindings and additional operator orderings. The choices in this search space have very little to do with choices open to the execution system regarding the next action to perform.

We define the search space to be a state-space structure wherein the nodes describe domain states and the transitions describe the occurrence of possible domain actions. This search space is chronologically organized in the sense that choice in the search space corresponds to, and subsequently directs, the choice of next relevant domain action.

4.2 Using a Search Space Generator

We use a type of Condition/Event System (Reisig, 1985) called a plan net as a search space generator. A plan net is a bipartite directed graph where the two types of nodes
Situated Control Rules

are interpreted as conditions and events. The links between nodes describe relationships of causation and enablement. Events cause conditions and conditions enable events. Formally, a plan net is defined as a triple \( N = (P, O, F) \), where \( P \) is a set of non-negated literals and \( O \) is another set of non-negated literals; \( P \) and \( O \) must be disjoint: \( P \cap O = \emptyset \). \( F \) expresses precondition and “add-list” information: \( F = (P \times O) \cup (O \times P) \). Each \( p \in P \) is a predicate which can be true or false in the domain, and each \( o \in O \) is an operator which denotes an event. \( F \) is composed of two relations: \((P \times O)\), interpreted as enables; and \((O \times P)\) interpreted as causes.

A plan net for the BNL problem is graphically rendered in figure 3. Operators are drawn as squares; predicates are drawn as circles; the enablement relation is drawn as arrows from circles to squares; and the causation relation is drawn as arrows from squares to circles. As previously mentioned, all event preconditions are deleted so those preconditions that remain true after the occurrence of an event must be explicitly included as postconditions as well. This is not a tremendously general solution to the frame problem. There are other types of net which do not require this simple (and limited) approach. See Drummond (1986) for an approach based on the use of domain constraints, where the conditions that are made false by the occurrence of an event are a function of those that are true to begin with. The particular approach taken to the frame problem does not have any impact on what follows.

The plan net for the BNL problem (figure 3) has four operators. \textit{deposit(A)} is an operator which denotes the event of placing block \( A \) on the table at location 1. The preconditions for this event are \textit{available(A)} and \textit{spaceat(1)}. The first precondition is true when block \( A \) is available for placement, and the second is true when there is space for \( A \) at location 1. Similar to the operator \textit{deposit(A)}, we have \textit{deposit(C)} at the bottom of the figure. Its preconditions are that \( C \) is available, and that there is space at 3 for \( C \). The postcondition of depositing \( A \) is that \textit{inplace(A)} will be true; similarly \textit{inplace(C)} is made true by \textit{deposit(C)}. The placement of \( A \) means that there is no longer space at location 1; similarly for block \( C \) and location 3.

Block \( B \) is treated differently from \( A \) and \( C \). There are two ways that \( B \) can be placed at location 2: it can be swept in from the left or swept in from the right. The operator \textit{leftdeposit(B)} denotes the event of sweeping \( B \) in from the left, through the space occupied (eventually) by block \( A \). This is why there must be space at 1 for the left deposition of \( B \). The predicate \textit{spaceat(1)} is also a postcondition of \textit{leftdeposit(B)} since the condition remains true after the event is finished. The right deposition of \( B \), denoted by the operator \textit{rightdeposit(B)} is similar. It requires that there be space at location 3, in order to sweep \( B \) into place. Either method of placing \( B \) results in the condition \textit{inplace(B)} being true.

4.3 Net Execution and Analysis

A plan net is a program for the execution component to run. It can contain loops and disjunction. The plan net is a causal account of domain events, their preconditions and results, and is amenable to direct execution. To make this work out elegantly plan nets must be hierarchically structured. For this paper we assume the approach taken in the STRIPS Planex system (Nilsson, 1984), where \textit{Intermediate Level Actions} and \textit{Low Level Actions} are provided as control routines. These routines validate the planner’s high-level action abstractions. Our future work will extend plan nets so that these lower levels of behaviour can also be represented.
Figure 3: A plan net for the BNL problem
It is not enough to simply execute a net. We must occasionally plan. To plan, we need to be able to reason about the future. And to reason about the future, we need a way of applying operators to states to reason about what events do. We can define a function \textit{do} which models the occurrence of an event in a domain state. This corresponds to the operator application mechanism in systems like STRIPS (Fikes & Nilsson, 1971), and is the computational realization of our solution to the frame problem. Of course, \textit{do} can be defined in many different ways.

From an initial domain state we can define a state space called a \textit{projection} which describes the occurrence of possible events. Each transition between states is constructed using the function \textit{do}. The projection serves two purposes. First, it acts as a manager of the system's \textit{search space} (as per section 4.1); second, it acts as the system's account of the various possible domain futures.

\subsection*{4.4 Exploiting Causal Independence}

Events are sometimes causally independent from each other. We could say that a set of operators (each of which denotes an event) is \textit{free from interference} if each operator pair in the set is causally independent. Net Theory gives us a way of specifying the conditions under which this is so. The appropriate definition changes with the approach taken to the frame problem. With the simple approach taken here, two operators are independent if their precondition and postcondition sets are disjoint. More expressive operator application mechanisms complicate the definition. In practice, the definitions often do not get very complex. For instance, it is easy to give an appropriate definition for STRIPS operators, the AI planning industry standard.

The definition of \textit{do} can be extended to exploit causal independence. Typically, \textit{do} is defined as a function from a state and operator to another state. It is a simple matter to give an extended definition which applies to a state and a set of operators. Given a state in the projection, and given a set of operators proven to be free from interference in that state, it is possible to apply \textit{do} to all operators at once. Construction of the projection can continue from the state which results. This allows projection to exploit event independence when possible, giving \(O(n^2)\) analysis cost in the best case, where \(n\) is the number of events being analyzed. It also allows projection to proceed even when events are not independent, giving \(O(n!)\) analysis cost in the worst case. This can be contrasted with the standard planning approach, where the mechanism of goal achievement \textit{assumes} pair-wise causal independence, and fails when this is not so. Synergistic and conditional effects cannot be handled with traditional goal achievement methods (Chapman, 1985).

Extending the definition of \textit{do} to cover sets of operators has implications for the structure of the projection. Typically, state-space structures are defined such that state-transitions describe the occurrence of single actions. Not so here. With \textit{do} defined on sets of operators, transitions are also no longer labelled with individual operators; instead, transitions are labelled with sets of operators. Operators in a set labelling a transition will have been proven causally independent in the state that the transition comes from. The transition describes the joint occurrence of all the events characterized by the operators in the labelling set. This approach does not present any epistemological or formal difficulties. See Drummond (1986) and Reisig (1985) for appropriate definitions.
4.5 The BNL projection

A projection is given for the BNL plan net in figure 4. States are drawn as ovals, and labeled as follows: "-" indicates the initial state, where all blocks are available for placement; "A-" indicates the state where only block A has been placed; "AB-" indicates the state where blocks A and B have been placed; etc. The final state at the bottom of the figure has all three blocks in place: "ABC". Possible transitions between states are indicated by arcs in the figure. To save space operator names are not written out in full; instead they have been shortened to the name of the block being moved, where possible. Thus, \{A\} is shorthand for \{deposit(A)\}, and \{C\} is shorthand for \{deposit(C)\}. \{LB\} replaces \{leftdeposit(B)\} and \{RB\} replaces \{rightdeposit(B)\}.

Seven transitions are possible from the initial state. The placement of A and the right deposit of B are causally independent, so there is a transition from the initial state that is labeled with the set \{A, RB\}. The set specifies that the indicated actions can be performed in either order. Similarly, block C and the left deposit of B are independent. Two of the transitions from the initial state deal exclusively with the individual placement of block B: it may be swept in from the left or from the right; with no other constraints operating, the exact means of depositing B makes no difference. This is an exhaustive projection, but such exhaustivity is not necessary. A projection can be heuristically narrowed to whatever extent the planner feels necessary.

The placements of A and C are also causally independent in the initial state. Notice however that the transition labeled with \{A, C\} leads to the state "A-C", from which no further progress is possible. It is this state that makes the BNL problem interesting. Not all assembly sequences will arrive at the final state, so some care is required during execution. The projection indicates that there are some operator sequences that achieve the required goals, and that there are some sequences which do not. It is not the business of the net and projection to say what should happen, but only to say what could happen.

4.6 Controlling the Search

It should also go without saying (but will not) that the entire projection need not be constructed. As with all search problems, there are two state spaces. First, there is an implicit space defined by the initial state and next-state function. There is also an explicit sub-space that is actually explored by the program. Typically, the hope is that a solution can be found in an explicit sub-space which is as small as possible. To achieve this we need heuristic guidance. Means-Ends Analysis (MEA) is one source of such guidance. We use MEA as a heuristic to guide search, and not as way of defining the search space itself. This is a different approach than is commonly taken (Chapman, 1985), where MEA is used to define the implicit search space.

Our use of MEA can reduce the space so that solutions are no longer found. Incompleteness is a concern even in simple problems such as register swapping. Systems that define their search to be through a space of partial plans can in principle solve register swapping problems. But the performance of these systems on even small problems throws into doubt the utility of defining search this way.
Figure 4: The BNL plan net projection
4.7 A Language for Expressing Goals

We base our goal language on the work of Ben-Ari et al (1981) in branching temporal logic. Goals come in the form of temporally scoped expressions which post constraints on acceptable projection paths. Given the obvious definition of the satisfaction of simple formulae involving conjunction, disjunction and negation in particular projection states, we can define goals of maintenance, achievement and obtainment over state sequences in the projection. We define satisfaction for a goal $G$ with respect to a sequence of states that begins with state $S[i]$ as shown below.

Maintain $G$ : $S[i] \models \text{maint}(G) \equiv (\forall j \geq i) \ S[i] \models G$
Achieve $G$ : $S[i] \models \text{ach}(G) \equiv (\exists j \geq i) \ S[j] \models G$
Obtain $G$ : $S[i] \models \text{obt}(G) \equiv (\exists j \geq i) \ s[j] \models G \land (\forall k > j) \ s[k] \models G$

A sequence of states through the projection defines a path. The truth of a goal can be evaluated in a state, with respect to different projection paths rooted in that state. There are three possible answers to the question “is goal $G$ true in state $S$”. $G$ might be necessarily true, possibly true or necessarily false. Each of these cases demands a different response on the part of the planner.

A goal will be necessarily true in a projection state if it is true for all paths that are rooted in that state. If a goal is necessarily true then no extra work is required: all possible domain actions, as described by the plan net, already satisfy the goal.

A goal will be possibly true in a projection state if it is true for some of the paths rooted in the state and false for some others. Possible truth indicates that some extra work is required of the planner. Since not all executions satisfy the goal, only those executions that do must be permitted. The way we permit selected executions is presented below, in section 4.8.

A goal $G$ will be necessarily false in a projection state $S$ if there are no paths rooted in $S$ which make $G$ true. In such a situation, there is nothing the planner can do to make the goal true: the goal cannot be satisfied, no matter what the execution component does.

If the implicit projection space does not include at least one path that makes all required goals true then there is no way of synthesizing a satisfactory plan. On the other hand, if all paths through the implicit projection space make all goals true then no plan (and no planning) is necessary: all actions satisfy the goals. Such luck is unlikely in practice. More typically, there will be some projection paths which achieve the goals and some other paths which do not. In situations of possible truth it is necessary to constrain the behaviours generated by the plan net to be only those that satisfy the goals. The planner must transform the statement of possibility into a promise of necessity.

4.8 Handling Critical Choices

Goals in our language are non-effective specifications of the behaviours that a plan net must produce. These must be translated into effective advice about what the execution component should do next.
The BNL plan net of figure 3 over-generates. It allows $B$ to be deposited last. This is not surprising: the business of a net is to say what can happen, and the business of the goals is to help say what the system (or its user) wants to happen. We can analyze the net through a projection, and can use the projection to determine operator application sequences that satisfy all given goals. Successful operator application sequences can be lifted from paths through the projection.

There is a single goal for BNL: $ach(inplace(A) \land inplace(B) \land inplace(C))$. This goal is only possibly true from the initial state in the projection (see figure 4). In total, there are three states in the projection from which the goal is only possibly true. We call states in which a goal is only possibly true critical choice points. The critical choice points in the BNL projection are, graphically: "\_", "$A\_$" and "\_C". There is only one state in which the goal is necessarily false: "$A_C". This state is the one which makes the BNL problem interesting (to the extent that it is). The goal of $A$, $B$ and $C$ all being in place is necessarily true in all other projection states.

There are three critical choice points in the BNL projection. From the initial state, "\_", the execution component could simultaneously deposit $A$ and $C$. This would force it immediately into the state "$A_C"$, from which no further progress is possible. Another state in which an execution mistake can be made is "$A_\_$". Here, $C$ can be deposited, and once again, this leads to the dead-end state "$A_C". The third state in which an execution error can be made is "\_C". From here it is possible to deposit block $A$, once again leading to "$A_C". All three of these states are critical choice points. In each, the execution component might make the incorrect choice for the action to perform next.

The problem this: the execution component does not have access to global information in the plan net projection. It must make local decisions about what to do next, based only on information regarding what it is possible to do next. Of course, what is possible might not be in line with what is desired. To circumvent this problem, information must be communicated to the execution component regarding projection paths that satisfy the goals. This information must be structured in a way that informs the executor about what to do in terms of directly-sensible environmental conditions. The information cannot be in the form of operator sequences. Objections to the sequence-following approach have already been raised.

We represent advice about what to do in a situation as Situated Control Rules (SCRs). The antecedent of an SCR is a well-formed-formula that can be evaluated in the system's current domain state. If the antecedent of an SCR evaluates to true, the rule is applicable. The consequent of an SCR is a set of sets of operators: anyone of the component sets can be performed. Performance of the action is guaranteed to be consistent with the system's goals. The consequent of an SCR is a set of all the sets of operators proven independent in, and applied from, the projection state in which the SCR's preconditions hold.

A set of SCRs encodes the planned behaviour and also expresses heuristic search information. This is a direct result of using a chronologically organized search space: choices in search correspond directly to choices of which action to perform.

Think of the process of SCR synthesis as one of translating a causal model into associational rules. If there is sufficient time, such a translation can provide the "eventuality" guarantees that all planners seek. Without sufficient time, action can still be taken, simply by executing the plan net itself. Actions taken in this way will have no guarantee of satis-
fying the system's goals, but as has been argued endless times in the literature, it is often better to have acted and lost than to have never acted at all.

4.9 The BNL Situated Control Rules

Remember that we are demanding extreme robustness on the part of the BNL execution component. If something can be done, our plan must say what to do. Of course the size of the plan will be proportional to the demands for robustness. If a single assembly strategy were sufficient, fewer SCRs would be needed. Since we want maximum robustness, we need to cover all critical choice points. This means we need three SCRs. Recall the states that must be covered by the SCRs: "__", "A___" and "___C". The three SCRs for the BNL problem are presented and discussed below. Numbers to the left of a rule are used for reference.

1: \( \text{available}(A) \land \text{available}(B) \land \text{available}(C) \land \text{spaceat}(1) \land \text{spaceat}(2) \land \text{spaceat}(3) \Rightarrow \{ \{ \text{deposit}(A) \}, \{ \text{leftdeposit}(B) \} \}, \{ \text{rightdeposit}(B) \}, \{ \text{deposit}(C) \}, \{ \text{deposit}(A), \text{rightdeposit}(B) \}, \{ \text{leftdeposit}(B), \text{deposit}(C) \} \) 

This first rule covers the state "__", and indicates that when all the blocks are available, \( A \) may be deposited by itself; that \( C \) may be deposited by itself; that \( B \) may be swept in from the left or from the right; that \( A \) and the right deposition of \( B \) are independent; and that \( C \) and the left deposition of \( B \) are independent. Independence means freedom from interference.

One might think that the set \{\text{deposit}(A), \text{rightdeposit}(B)\} entails that \{\text{deposit}(A)\} is also acceptable by itself. This is not always the case. If the projection has only considered maximally parallel event occurrences, then states which result from arbitrary event inter­leavings might not have been analyzed. For our example, these intermediate states have been considered, and this is communicated by explicitly listing the individual actions that may be performed in these states.

2: \( \text{inplace}(A) \land \text{available}(B) \land \text{available}(C) \land \text{spaceat}(2) \land \text{spaceat}(3) \Rightarrow \{ \{ \text{rightdeposit}(B) \} \} \)

This second rule covers the state "A__". It permits depositing \( B \) only through a right sweep. The other possibility for immediate execution from this state is \text{deposit}(C). The SCR does not permit this, since depositing \( C \) generates a state in which the goal is necessarily false.

3: \( \text{available}(A) \land \text{available}(B) \land \text{inplace}(C) \land \text{spaceat}(2) \land \text{spaceat}(1) \Rightarrow \{ \{ \text{leftdeposit}(B) \} \} \)

The third SCR covers the state "___C". Symmetric with rule 2, it precludes the deposition of \( A \) when that would lead to necessary failure. In the state covered by this rule, the only safe action to perform is \text{leftdeposit}(B); that is, \( B \) must be swept in from the left.
4.10 Project Status

This work is just starting at the NASA Ames Research Center. Our ideas are being implemented on a Sun 3/60 workstation in Common LISP. We plan to test the ideas on scenarios taken from the Mars Rover Sample Return mission, but we are currently looking at simpler problems, such as BNL.

5 Comparison With Other Work

Suchman's (1987) ideas on situated action have had significant impact on research within AI. Her study of action-in-context has provided insight on the relationship between planned and unplanned activity. Schoppers' (1987) work on universal plans has motivated parts of our approach. His view of planning as the synthesis of reactions to situations has affected our work, but there are significant differences. Universal plans are typically seen as a compiled response to all possible situations. SCRs do not need to cover all possible situations. When SCRs exist, they can be used to improve performance. When they do not exist, action can still be taken. In contrast, universal plans seem to be the only mechanism available for generating action. If there is no universal plan, then nothing can be done.

STRIPS triangle tables (Nilsson, 1984) are a representation for plans that specify reactions to situations. Triangle tables can give reactions to any one of a set of situations, provided that the situations occur in a pre-specified total order. That is, triangle tables determine a total order on actions to be executed, and cover action failure (where actions must be repeated) and serendipitous goal achievement (where actions can be skipped). Our work can be viewed as an extension of the triangle tables idea. Triangle tables cannot represent disjunctive behaviours, while SCRs can. Of course, universal plans can represent disjunction; indeed, this is one of their primary strengths. However, universal plans are used to drive the entire behaviour of the execution component that uses them. Our view is similar to Agre and Chapman's (1988), where plans simply inform action, and do not completely determine it.

Recent work by Nilsson (1988) considers the production of behaviour through action networks. These action networks are more expressive than plan nets. But to date, techniques for action network synthesis have not been presented. Our work seems closer to automatic synthesis, but we are not completely there yet. We have considered various algorithms for synthesizing SCR preconditions. It is not hard to imagine how to do it; the problem is (as ever) doing it efficiently. This is one of our major effort areas.

Rosenschein and Kaelbling's (1987) work on the theory of situated automata is in part an attempt to provide a theory of high-level symbolic control. They view a machine as situated in an environment, and provide a formal characterization of the machine's knowledge in terms of objective correlations between states of the machine and states of that environment. The GAPPS system (Kaelbling, 1988) is part of this work. GAPPS is a rule compiler: it accepts symbolic goal-reduction rules and produces a “circuit” which realizes the behaviour inherent in the rules. Each individual rule is expressed in terms of a locally defined leads-to operator. The problem of goal interactions between rules must be sorted out by the programmer. GAPPS assumes that the rules' recommendation for action can be composed conjunctively by taking the individual components of each recommendation, and “or"ing
them together. It is the business of classical planning to sort out such interactions. Our work deals with goal interactions by doing temporal projection: futures that do not satisfy all goals are prevented from occurring through the creation of appropriate SCRs.

Minton's (1988) work on the Prodigy system is similar in many respects to ours. Prodigy is a general-purpose planner that can improve its performance over time. It improves by learning knowledge regarding correct and incorrect decisions made during the planning process. This knowledge is expressed in control rules. Control rules are used during subsequent planning to improve the decisions made regarding goal ordering, variable binding, and operator introduction. Although Prodigy's control rules are learned and used across different problem instances, they are similar to our SCRs. Search control rules and situated control rules both express local choice information in terms of global objectives. However, Prodigy defines is search space differently. We use a chronologically organized space, where decisions in the search correspond with actions to perform in the domain. Prodigy defines its search to be through a space of partial plans. Thus, choices in its search correspond to goal orderings, variable bindings, and operator introductions. Future work will examine the relationship with Prodigy's search control rules more closely.

Acknowledgements

Dave Thompson and Bill Gevarter provided useful comments on this paper. Peter Jackson suggested that SCRs were more useful than I had initially thought. Ken Currie and I developed many of these ideas together when I worked at the Artificial Intelligence Applications Institute at the University of Edinburgh. Monte Zweben has provided many interesting discussions while I've been at NASA Ames. Steve Minton has helped me understand the connections between this work and Prodigy's approach to learning search control rules.

References


Truth Maintenance-Based Planning with Error Recovery

Paul Morris
IntelliCorp

Abstract

We describe work in progress that involves building a nonlinear planner using an underlying truth maintenance system (TMS) as an implementation tool. In contrast to the top-down approach of other work dealing with the integration of planning and execution, this is a bottom-up approach which seeks to explore possible uses of a promising new technology in this area. Thus, it does not so far address all the issues of concern here, but it provides a fresh perspective on some of those issues.

The expected benefits of truth maintenance in this area include the ability to make revokable assumptions about the external world and the exploitation of dependency-directed backtracking after a planning or execution failure. The TMS is used to record dependencies and incompatibilities among the decisions that occur in the planning process. This information can be used to help identify the decisions responsible for a particular failure, and to make minimal cleanups when those decisions are altered. The result is a form of plan repair that is well integrated with the search process.

We summarize here some of the features in a design of an early prototype system, focusing on the plan structure representation and the planner/TMS interface.

The plan structure is similar in outline to that considered in classical planning systems. In particular, it contains actions, goals, facts, and links as basic elements. (A link is a connection between a goal element and a fact element, indicating that the fact is earmarked for satisfying that particular goal.) In addition, there are associated constraints, including variable bindings and temporal ordering restrictions.

In general, we associate TMS nodes with elements of the plan structure that arise from decisions in the planning process. Thus, actions and links are directly associated with TMS nodes. Other elements of the plan structure point to these TMS nodes as appropriate. For example, goals and facts that arise from the introduction of an action will point to the TMS node of that action. Variable bindings will point to the TMS node of the link from which they arose.

The TMS node of a planning decision is given a justification that reflects its dependence on earlier decisions that created the context in which it arose. For example, the decision to introduce an action to achieve a subgoal is dependent upon the decision to introduce the action that gave rise to the subgoal. The justification also reflects the interdependence of a decision and its mutually exclusive alternatives.

As execution proceeds, more information will be acquired about the world and some of the assumptions may be invalidated. This will cause label updating in the TMS network. When a TMS node goes OUT as a result of label updating, the associated elements are removed from the plan structure. For example, if the TMS node associated with an action goes OUT, then the action and its associated goals and facts are removed from the plan structure. If a node that was OUT should come IN again in the course of label updating, the associated plan elements are restored to the plan structure.
During the planning process, impasses involving unsatisfiable goals may also cause earlier decisions to be revoked, resulting in a similar update process. This may give rise to complex dependency structures for substitute actions and links. For example, an action may be ruled out because one of its subgoals cannot be satisfied. That failure may become part of the dependency structure for a substitute action, so that if conditions change with respect to the subgoal, the original action may be reconsidered.
Action Networks

Nils J. Nilsson
Computer Science Department
Stanford University

Abstract

We describe new techniques for organizing actions in dynamic environments in which the effects of actions and the state of the environment can not always be reliably predicted. The ideas make use of state-transition graphs at various levels of descriptive detail to create hierarchical “action networks” of combinational circuits. These networks can be thought of as embodying and executing plans with a high degree of inherent conditionality. The entire network and each of its hierarchical components act like homeostatic servo systems attempting to maintain the environment in specified states. We view an agent as having a set of propositional beliefs and a set of propositional goals. At the lowest level, the goals energize primitive actions. Sensory inputs continuously modify the agent’s beliefs. The beliefs and goals combine in the action network to “release” actions appropriate to both. We also propose a computer language, called ASTRAL (Agent-Structuring, Teleo-Reactive Action Language). Execution of ASTRAL programs would build and modify the agent’s action network while the network functions concurrently to produce actions. Work on these topics is in a preliminary state, but the author will send draft notes to requestors. Editors’ note: Dr. Nilsson has provided us with a copy of his draft notes, which we included in this proceedings starting on the following page.
We describe new techniques for organizing actions in dynamic environments in which the effects of actions and the state of the environment can not always be reliably predicted. The ideas make use of state-transition graphs at various levels of descriptive detail to create hierarchical "action networks" of combinational circuits. These networks can be thought of as embodying and executing plans with a high degree of inherent conditionality. The entire network and each of its hierarchical components act like homeostatic servo systems attempting to maintain the environment in specified states. We view an agent as having a set of propositional beliefs and a set of propositional goals. At the lowest level, the goals energize primitive actions. Sensory inputs continuously modify the agent's beliefs. The beliefs and goals combine in the action network to "release" actions appropriate to both. We also propose a computer language, called ASTRAL (Agent-Structuring, Teleo-Reactive Action Language). Execution of ASTRAL programs builds and modifies the agent's action network while the network functions concurrently to produce actions.

I. Introduction and Rationale

Some Artificial Intelligence (AI) programs must work in environments in which either the effects of the program's actions on the environment are not entirely predictable or in which unpredictable changes might spontaneously occur in the environment. Such programs must be prepared to sample the states of their environments more often and more closely than would programs that can accurately predict environmental state. They must always be prepared to reorganize their behavior to take into account constantly changing environmental conditions. But, behavior must also be guided by the program's goals. Taking both goals and changing inputs into account, we might call such programs teleo-reactive.

Programs for controlling robots are good examples. A robot's actions might not always reliably achieve their intended effects. A robot hand might accidentally drop an object it is carrying; an unexpected obstacle might block a robot's path; and people (and other robots) might alter the world in which the robot operates.

Experience in designing robot control programs that generate plans or sequences of actions to achieve desired goals has highlighted the difficulty of smoothly combining the planning process with the plan execution and execution monitoring processes. Most of the AI planning systems, such as STRIPS (Fikes 1971), NOAH (Sacerdoti 1977), and SIPE (Wilkins 1988), produce plans with insufficient built-in conditionality; their execution in unpredictable worlds all too often and too quickly encounters environmental conditions for which the plans are
inappropriate---making replanning necessary. The use of "triangle tables" and "Markov tables" in the SRI robot SHAKEY (Fikes 1972; Nilsson 1984) were early attempts to provide more robust planning and plan execution systems.

One of the contributions of the research described here is a proposed system for controlling the actions of a robot (or of any other program that must function in unpredictable environments) that has a high degree of inherent conditionality---much higher than that possessed by the mainly linear code produced by most AI planning systems.

Another requirement of robot control programs (and other programs that must function in changing environments) is the need for real-time response. By real-time, we mean more than simply fast---fast is relative to the pace of change in the world in which the robot functions. We also mean bounded. We want to guarantee that appropriate actions happen within some preset time after certain environmental changes occur. A robot must stop within one-half second, say, of detecting a solid object in front of it. Yet, notwithstanding the need for real-time response, we want our programs also to be able to plan or reason about the actions that they take.

One can approach the problem of building intelligent robot control programs either from the point of view of control theory, in which case one needs to add symbolic reasoning abilities to the inherent real-time abilities of control systems; or from the point of view of AI, in which case one needs to find a way to achieve real-time performance in systems that are able to plan and reason.

Certainly, the need for real-time response means that we cannot depend on those kinds of planning systems requiring unbounded time to calculate actions. Unfortunately, most AI planning systems (like all reasoning programs) cannot be guaranteed to return answers in bounded time. Stopping the robot in time to avoid its bumping into a wall is not an appropriate matter for a theorem-prover.

Thus, another challenge for us is to combine reasoning with acting in such a way that action does not have to wait for reasoning even though it may often be guided by it. We are persuaded that the bounded computation of appropriate action must be an ongoing activity organized so that it can function without planning but also organized so that it can be affected by planning in those cases in which planning can have its effect in time. Another contribution of this research is to propose a system architecture with such a property.

Our goals, then, are to produce programs---let us call them agents---capable of reason-guided, real-time behavior in somewhat unpredictable environments. We assume that environmental changes may be both abrupt and gradual and that our agents will be able to sense some but not all aspects of their environments. Although we may have implied that our agents are robots functioning in a physical world, we are also interested in agents that exist in more symbolic worlds, for example data base agents.

Agents are given sets of "tasks" to perform. The tasks may be as simple as maintaining certain environmental conditions within specified limits or as complex as building complicated structures. The user may specify criteria (such as cost functions) for the agent to optimize in task performance. The user may give the agents a suite of tasks---leaving it up to the agent to decide which to work on at any given time. The user will not infrequently change the tasks and/or the conditions under which the tasks are to be performed. A very important ground rule is that it be possible for the user to specify tasks and changes in conditions declaratively---say in a language like the predicate calculus (if not, ultimately, English). Adhering to this rule would seem to require that our agents be capable of reasoning with declarative representations even if
timely actions must sometimes be performed without benefit of unbounded reasoning processes.

Several authors have proposed architectures for coupling sensing to acting in ways that do not depend on run-time use of declarative representations and reasoning systems. Thus, their work departs from traditional AI approaches in various ways. Among these are Brooks (Brooks 1985) who proposes assemblages of finite-state machines; Schoppers (Schoppers 1987) who advocates action plans with high inherent conditionality; Rosenschein and Kaelbling (Rosenschein, 1986; Kaelbling, 1988) who compile situated automata from specifications of what these automata are to know; and Agre and Chapman (Agre 1987) who stress the higher importance of indexical over representational knowledge. Many of the connectionist architectures (Rumelhart 1986) are also relevant to the problems we consider in this paper.

Just as a general-purpose reasoning system is inappropriate for real-time behavior, a hierarchical system of programs and subprograms with conventional control primitives is too unwieldy for dealing with unpredictable environments. With conventional control, when one program calls another, it surrenders control until control is explicitly returned by the called program. Surrendering control causes two problems. First, some sensed environmental changes will be of a character that can be dealt with by the high-level programs but not by the low-level ones. A called low-level program might then continue to function under environmental conditions that are inappropriate for it but that go unnoticed by suspended higher level programs.

A second problem is concerned with the proliferation of error messages resulting from unintended effects of the low level programs. As we observed in early work with Shakey, the SRI robot:

Errors are made most frequently at low levels by routines that are too primitive to cope with them. An error message may have to be passed up through several levels of routines before reaching one possessing sufficient knowledge of both the world and the goal to take corrective action. If any routine can fail in several ways, this presents the highest-level routine with a bewildering variety of error messages to analyze, and requires explicit coding for a large number of contingencies. (Nilsson 1984).

Yet, ease of design requires that the action-generating system be hierarchically organized. Clearly, control in this hierarchy has to be such that higher level routines enable lower level ones but do not surrender control.

With all of these issues in mind, we propose a new scheme for organizing behavior. It is based on networks of combinational circuits and on a programming language for generating these circuits. We will reveal the story in stages, beginning in the next section with the networks that actually implement an agent's actions. Many important aspects of the new scheme remain to be worked out; we hope the ideas presented in these notes will help stimulate discussion of these problems.

II. Networks of Action Units

One way to convert sensory signals into actions is by electronic circuitry. Suppose that the sensory signals can be represented in binary form (0,1). Suppose also that the actions that can be performed by an agent are binary---either the action is being performed or it is not. (The control of "on-off" action is called "bang-bang" control in control theory.) Converting sensory signals into actions can then be accomplished by logical switching functions, and these can be
implemented by networks of logical gates. For example, the AND gate below will execute action "A" if and only if all of its sensory inputs, S1, ..., S4, have value 1:

```
S1 ----> AND
S2 ----> A
S3 ----> A
S4 ----> A
```

If the sensory inputs are represented as binary-valued propositions, this AND gate can also be represented by the production rule:

```
S1 \land S2 \land S3 \land S4 \implies \text{action } A
```

(All of the networks that we discuss in these notes might, equivalently, be thought of as systems of production rules. We prefer the network metaphor, however, because it helps us remember that all parts of the system---higher levels and lower levels---are constantly sampling their inputs. There are no programs that are suspended during calls to subprograms.)

Networks of AND, OR, and NOT gates can implement any desired functions of the inputs to energize actions.

We will find it useful to define a special kind of logical gate called an action unit. We show a typical action unit below:

```
P
```

Our action units have three kinds of inputs and one type of output. The inputs coming into the left side of the unit (B1 and B2 in our example) are called precondition inputs. They originate in the agent's belief structure (to be discussed presently). The input coming into the top of the action unit is called the purpose input; it also originates in the agent's belief structure. The input coming into the bottom of the unit is called the calling input or the goal input. It originates in the agent's goal structure (also to be discussed later).

The outputs of an action unit (action A in our example) are on (have value 1) if and only if the purpose input is off (has value 0), the goal input is on, and some function of the precondition inputs (we usually presume the AND function) has value 1. (When referring to the inputs, outputs or status of action units, we treat the terms "on," "true," "active," and "1"
synonymously and the terms "off," "false," and "0" synonymously.) If the precondition function is AND, then our example action unit is equivalent to the following AND gate:

\[ -P \quad B1 \quad B2 \quad G \quad \text{AND} \quad \text{action A} \]

An agent's belief structure is a collection of binary-valued propositions. Many of them can be presumed to be connected to the agent's sensory apparatus. (In fact, that apparatus itself may be implemented in combinational circuits of some kind.) For example, B1 in our diagram above might be the proposition, "the temperature is above 20° C". To correlate with what is actually true in the world, such a proposition must be "wired" in some appropriate way to a thermometer. Other propositions may preserve history or state information of the agent. We call all these propositions beliefs of the agent.

The purpose input of an action unit is a belief which, when true, prevents the unit's output from being on. We take the view that the unit exists in the network in order to energize actions whose result (through the chain environment->sensors->beliefs) is to make the purpose true. When the purpose is true, the unit is off; when it is false (assuming the other inputs are on), the unit initiates action to "achieve its purpose." Thus, the purpose input plays a role analogous to a "set point" in a feedback control system designed to maintain some "plant" variable at the set point.

We assume the agent has a set of goals. These are stored as propositions in a goal structure. A goal-structure proposition, G, is typically connected as a goal input to action units whose purpose is also G. Thus, if a goal-structure proposition has value true (that is, the agent does in fact have the goal G), then the goal input to those action units that are supposed to be able to achieve G will be on, and (presuming that G is not already true in the agent's beliefs and that the relevant precondition inputs are on) the agent will take action(s) to achieve G. Since we sometimes also think of the goal input as the "calling" input, we say that units whose goal input is on have been "called." We might also sometimes say they are "enabled."

The outputs of an action unit are either calling inputs to other units or calls to execute some primitive action. In either case, we define the outputs as being part of an agent's goal structure. When an output has value 1, the corresponding goal has value true. Goals in the goal structure are connected to the appropriate action units through the calling inputs of these units. In the diagram below, we illustrate, abstractly, a network of three action units and how they are connected to an agent's belief and goal structures.
The diagram above can be thought of as a simple agent that executes the primitive actions A1 and A2 under appropriate circumstances. Describing how this agent works will illustrate the architecture we are proposing. This agent is given the overall goal G by setting to 1 the value of the proposition G in its goal structure. Unspecified sensory inputs (the arrows coming in to the left of the belief structure) maintain the agent's beliefs (B1, B2, B3, B4, G, G1) at values of either 1 or 0. If the agent "believes" G (that is, if G in the belief structure has value 1), then the agent does nothing. However if belief G has value 0 and if belief B3 has value 1, then (since goal G is on) action unit 1 in the above diagram is turned on giving the agent the subgoal of G1. This subgoal in turn "calls" action units 2 and 3. Action unit 2 activates primitive action A1 if belief G1 is not true and if the agent believes B1 and B2. The intended effect of action A1 in the world is to create conditions which, when sensed, cause the agent to believe G1 which terminates action A1. Action unit 2 activates action A1 by causing the agent to have the goal "Doing A1." That goal is primitive; its having value 1 in the goal structure directly energizes a primitive effector. A similar account can be given for action A2. In this example, we assume that G1 implies G so that achieving G1 also achieves G.

This agent can be thought of as a homeostat or servo control system that is at rest only when the environment is such that G is true (that is, when the environment is such that the agent's sensors indicate that G is true). When the environment changes so that G becomes untrue, the system becomes unstable and takes actions. The effects of the actions are to push the world in the direction of G being true. In fact, for any action unit in any of our networks the actions controlled by that action unit are intended to change the world in such a way as to make the unit's purpose true and thus turn the action unit off. Action units "want" to be off; they become active when the world differs from their purposes.

Note that the way in which actions are controlled in the agent illustrated above is such that even if G becomes true for reasons quite independent of the effects of the agent (say, for example, that some other agent makes G true), the system becomes immediately stable anyway and ceases action. All of this requires, of course, that sensory inputs must be continuously monitored and inputs to actions units continuously evaluated.
Although we think of the outputs of action units as goals (and in our diagram above we have run these outputs through the agent's goal structure to emphasize that point), we will conventionally connect the outputs of higher level action units directly to the lower level units that they enable and not show the goal structure explicitly.

It will be important to elaborate this simple model of an agent in various ways. First, we do not intend that every one of the agent's beliefs is the result of continuous sensing of the environment. Some of the agent's actions will be to activate sensors, and it will be important for the agent to "remember" some of these sensed values until they are sensed again. (The agent may also update these sensed values by "dead-reckoning" calculations.) Therefore, some of the agent's beliefs may be stored propositions. We may also want to have remembered goals (in addition to remembered beliefs). A remembered goal is one that can be set to have value true (by the output of an action unit) and then stays true until it is explicitly turned off (by some other action unit).

We introduce special units, called latches, for setting goals and beliefs to values that will be maintained until they are reset by other latches. A latch has a single input and a single output. We need ON latches and OFF latches. The output of an ON latch is set to 1 whenever its input becomes 1; its output stays at 1 until it is reset by an OFF latch. The output of an OFF latch is set to 0 whenever its input becomes 1; its output stays at 0 until it is reset by an ON latch.

Latches that are able to reset each other's outputs will have their outputs connected, and the combined output will be 1 if the ON latch was set more recently and will be 0 otherwise.

In network diagrams, we can represent latches as follows:

- An ON latch

- An OFF latch

Another elaboration is that in addition to the output of an action unit being the setting of a goal in the agent's goal structure, action units may also set beliefs. That is, the output of an action unit may be used as one of the (non-calling) inputs of other action units.

In the belief structure we can have terms as well as propositions. Terms have values, and these values must be ground. Term values can be supplied by sensors, for example. We might have a term, Temperature, with value 20, say. Term values can be tied dynamically to sensors. Propositions in the belief network can be composed from terms using built-in predicates and functions. The values of these propositions dynamically depend on the values of the terms; as the values of terms change, so might the values of the propositions. For example:
In the above, "Greater-than" is a built-in predicate, and "Plus" is a built-in function. "Temp-1", "Temp-2", and "Temp-3" are terms whose (ground) values might be dynamically changing. The value of the purpose condition on the unit is continuously maintained as the term values change.

The output of an action unit may set the value of a term. The setting can either be latched, in which case the term stays at that value until reset; or it can be set dynamically, in which case the term value tracks any values that it depends upon. For example, during the time the action unit below is on, the value of the term Theta dynamically tracks the value of the function DIR. (DIR computes the angular direction from Position to the coordinate pair (10, 10). The value of Position is a coordinate pair.) When the action unit is off, the value of Theta is undetermined. We use the symbol <- to denote that a term value is being set dynamically.

Below we show an example of latching a term value.
The value of PP is latched to the value of Position at the time that the unit becomes active. It retains that value until it is latched to some other value. We use the symbol $\leq$ for latching term values.

We make use of some of these ideas in the network below. We assume that the term "Position" tracks the current value of a robot's location coordinates. The goal, given by the proposition $\text{EQUAL(Position, (10,10))}$ in the robot's goal structure, is to have the robot at the location (10,10). Action unit 1 dynamically sets the value of Theta to the angular direction from the value of Position to (10,10). Action unit 2 rotates the robot until it is facing in that angular direction. (FACING is a built-in predicate.) Action unit 3 moves the robot and is on only when motion is in the direction calculated to be toward the goal. The entire system is quiescent only when the robot is at location (10,10). Note that unit 3 has F (false) as its purpose input.
Finally, the reader might have already noticed that our binary (0,1) logic is not able to distinguish among an agent’s believing a proposition $P$, believing the negation of $P$, and not knowing which of these may be true. One way to correct this deficiency is to introduce a three-valued logic with values -1 (for believing the negation of a proposition), +1 (for believing the proposition), and 0 (for having no opinion about the proposition). We must elaborate some of our action-unit conventions in the three-valued case. First, we probably want all of the precondition inputs to have the value +1 (in the case of an AND function for these inputs) in order to turn a unit on. We probably want the purpose input to have either the value -1 or 0 in order to turn a unit on. (Unless the agent believes that the purpose is already true, it should take action to make it true. In many cases it may be appropriate for such an action to include as a first component a test to see if the purpose is already true). We probably want the calling or goal input to have value +1 in order to turn a unit on. Our latching action units now have to be elaborated to include latch on, latch off, and simply reset to 0. For simplicity, we will assume-binary valued propositions in these notes.

In our examples so far, we have suggested that all of the precondition inputs had to be on in order to turn a unit on. But in general, switching functions (other than AND) of the input values might be appropriate. An interesting class of switching functions are the linearly
separable ones---implementable by a threshold logic unit (TLU). (See Nilsson, 1965.) A TLU implementation of a typical action unit is shown below:

In a TLU, the inputs are weighted, summed, and thresholded to produce a binary output. Depending on the weight values, TLUs can implement AND gates, OR gates, or any other of the linearly separable switching functions.

In order to respect the intended effect of the purpose input, P, its weight value is such that when P is 1, the unit is off. (That is, P’s weight is highly negative.) To respect the intended effect of the goal input, G, its weight and the weights on the precondition inputs must be such that the precondition inputs alone cannot turn the unit on.

We are interested in TLU realizations because they help us relate our action networks to connectionist or neural networks and because, by introducing explicit adjustable weights, they invite schemes for learning (to be discussed later). We also think that it is interesting to speculate about generalizing our binary- (or perhaps ternary-) valued goals and beliefs to ones having a greater range of values. Goals with very high values could take precedence over ones with smaller values; beliefs with high values might compensate for ones with smaller values. TLUs could have output values proportional to the amount by which their weighted sums exceeded their thresholds. In this manner, some subgoals would have higher values than others, and these values might be used to help adjudicate between conflicting actions.

Action networks can also be implemented efficiently by computer simulations with appropriate indexing, which confines the need for polling inputs to the times when inputs change. (An action network implementation is currently being developed by Phil Stubblefield of Rockwell Palo Alto Laboratories.) The techniques used in the implementation of AI truth maintenance systems (deKleer 1986) may be useful in implementing action networks.

III. Action Hierarchies

In our examples of action networks, we have suggested that actions are organized hierarchically---with higher level actions calling or enabling lower level actions. The question of how action hierarchies are organized will require some digression.

We think of an agent’s actions as changing the state of the world that the agent inhabits. World states can be described at different levels of detail. Transitions between states described at the
most primitive level are achieved by the primitive level actions. Transitions between states having higher level descriptions are achieved by higher level actions. Since the higher level actions are composed of lower level ones, the higher level state descriptions must be defined in terms of lower level ones.

We illustrate these ideas by a standard blocks-world example. Consider three blocks A, B, and C, on a table, T. The primitive actions for moving blocks are instantiations of Move(x,y) where x can be any of A, B, or C, and y can be any of A, B, C, or T (but x \neq y). We assume that the precondition for successful execution of Move(x,y) is Clear(x) \land Clear(y) with Clear(T) always true. At the most primitive level we describe each world state by a formula that gives the location of all of the blocks. The way in which states are linked by the primitive actions is given by a graph whose nodes are labeled by state descriptions and whose edges are labeled by actions. The graph in the figure below captures the possibilities for this problem.

Suppose the agent's goal is to have block A on block B and B on C. We want an action net that will be able to achieve this goal regardless of the initial condition of the blocks. One such net is shown in the figure below.
In this net, we have one high level action unit with no precondition and with a purpose formula representing the overall goal of the net. We also have an action unit for every (non-goal) node in the graph. The precondition of each action unit is the formula describing the corresponding node, and the purpose is the formulas describing the next node on a path toward the goal. Each unit enables a primitive block transferring action. The net can be regarded as a universal plan (Schoppers 1987) that achieves its goal regardless of initial conditions.

Basing universal plans and their associated action networks on the most primitive level of description will be impractical for large problems. Instead, we propose to aggregate subsets of primitive states into superstates to give us a way of defining action hierarchies. There are several ways in which states can be gathered into superstates. We describe one possibility here as an example.

Let N and M be disjoint subsets of nodes in a state transition graph such that there is at most one edge linking nodes in N with nodes in M. That is, there is at most one pair consisting of n in N and m in M having an edge between them. When there is exactly one pair, we will say that the subsets are narrowly connected. If there are two or more edges connecting the subsets we say they are non-narrowly connected. Let P be a partition of the nodes of a graph (into exhaustive and non-overlapping subsets of nodes) such that no two subsets of P are non-narrowly connected. We will say that such a partition is hierarchical. We define a supergraph G' of a hierarchically partitioned graph G as follows: There is a node in G' corresponding to every subset in the hierarchical partition of G. Nodes n and m in G' are connected by an edge if and only if the corresponding subsets in G are narrowly connected. A graph always has a
hierarchical partition and thus a supergraph. Typically, there are many ways to partition a graph hierarchically, and thus a graph will have several different supergraphs.

Returning to our blocks-world example, the primitive graph for the blocks world has, among others, the following hierarchical partition:

The subsets are shown shaded, and the narrowly connected subsets are connected by heavy lines in the figure, from which it is a simple matter to see what the corresponding supergraph would be.

We propose to use supergraphs, defined in this way, as (super)state transition graphs. To do so, we label each node in the supergraph by a formula describing (at this higher level) one of the (super)states of the world, and we label each edge by an action (described at this higher level) that takes us from state to state. We will usually define a new relation and its corresponding predicate to describe the superstates. This new relation can always be defined in terms of the disjunction of those formulas labelling the nodes in the corresponding subset. For example, we might define the formula

\[
\text{Dis-assembleable}(x,y,z) \iff \text{On}(y,z) \land [\text{On}(x,y) \lor \text{On}(x,T)]
\]

for \(x,y,z\) ranging over \(A,B,C\)

and

\[
\text{In-line}(A,B,C) \iff \text{On}(A,T) \land \text{On}(B,T) \land \text{On}(C,T)
\]

We define the goal node of the supergraph to be that node corresponding to the subset containing the goal node in the original graph. A solution tree of the supergraph is a tree with goal node as root and with a path to the root from every node in the graph. For example, the tree below is a solution tree for the supergraph of our blocks-world example:
We have labeled the edges of this graph by higher level actions that we will shortly define in terms of the primitive actions. The goal node is indicated by an enclosing circle. (The action "Stack(A,B,C)" labelling the one circular arc attached to the goal node is special in that rather than connecting one super state to another, it describes a collection of primitive actions totally within one superstate. Its expansion into primitive actions will be handled in a special way.)

Even before expanding the super actions into collections of primitive ones, we can define an action unit that represents a universal plan for solving our blocks-world problem (at this level of detail). This action unit will enable lower level units that implement the actions of the supergraph. This top-level unit is the following:
In order to expand a high level action (labelling an edge in the supergraph) we create an action graph by replacing the super nodes at the source and target of the edge by the primitive subgraphs that the supernodes represent. That (unique) primitive node in the target subgraph that has an edge to one of the nodes in the source subgraph is labeled as the goal node of the action graph. The high level action is then defined in terms of a solution tree (to the goal node) in the action graph. For example, the solution tree for Disassemble(C,A,B) is shown below:

\[
\text{Disassemble}(C,A,B) \\
\quad \bullet \quad \text{On}(C,A) \land \text{On}(A,B) \land \text{On}(B,T) \\
\quad \text{Move}(C,T) \\
\quad \quad \bullet \quad \text{On}(A,B) \land \text{On}(B,T) \land \text{On}(C,T) \\
\quad \text{Move}(A,T) \\
\quad \quad \bullet \quad \text{In-line}(A,B,C)
\]

Using this solution tree, we define the action unit implementing \text{Disassemble}(C,A,B) as follows:
The precondition of the action unit is the formula labeling the source node of Disassemble(C,A,B) in the supergraph, and the purpose is the formula labeling the target node. The outputs of the action unit enable action units that implement the actions in the solution tree of the action graph of Disassemble(C,A,B). The calling input originates in the top level action unit shown previously. The action unit implementing Disassemble(C,A,B) can be thought of as a "conditional universal plan" for achieving In-line(A,B,C). That is, it is a universal plan for any of the states in the subset of states satisfying Dis-assembleable(C,A,B). All of the other Disassemble actions are implemented in like fashion.

Similarly, we have the following pairs of solution trees and action units for the other high level actions in the supergraph:
Transfer(C, B, A)

\[ \text{On}(A, T) \land \text{On}(B, C) \land \text{On}(C, T) \]

- Move(B, C)
- Move(C, T)

Dis-assembleable(A, B, C)

Dis-assembleable(C, B, A)

Stack(A, B, C)

\[ \text{On}(A, T) \land \text{On}(B, C) \land \text{On}(C, T) \]

- Move(A, B)
- On(A, B) \land \text{On}(B, C) \land \text{On}(C, T)
Now we can implement the primitive actions Move(x, y) by action units having the following form:

Connecting the outputs and calling inputs of these units appropriately then produces the action network that will achieve the block-stacking goal, regardless of initial conditions. The reader should note that the block-stacking routine, defined as an action net, has the high degree of conditionality and robustness in the face of block-stacking errors and other environmental changes that we desire for programs controlling the actions of real-world agents.

Although this particular block-stacking problem is really too simple to require a hierarchical solution, action hierarchies will doubtless be very important in real domains. The hierarchies can be organized in any number of levels simply by partitioning supergraphs into super-supergraphs and so on recursively. We also expect that the partitioning does not have to be based strictly on narrowly connected subsets; some relaxation of that condition may be possible. Of particular interest will be learning methods which will allow agents to construct and appropriately partition graphs based on their experiences with the effects of their actions.

The hierarchical action network itself can be regarded as a decision tree that can be followed through various conditional tests (on sensory inputs and/or belief propositions) to determine which primitive action should be executed at any moment. In our example, the structure of the decision tree is:
Schoppers's universal plans (Schoppers 1987) also implement decision trees, but their structure is somewhat different because they are not based on the same hierarchical notion of actions.

To gain familiarity with our approach toward defining action hierarchies the reader may want to apply these ideas to the Tower of Hanoi problem whose (3-disk) version has the graph structure shown below. Working with this puzzle might give insight into how to exploit recursion in action networks.
Action networks solve some of the agent control problems we posed at the beginning of this paper. Specifically, they react appropriately in real time to changing environmental conditions. They can be regarded as a rather unusual but effective formalism for expressing plans of actions to accomplish a variety of goals. We next focus on synthesis procedures for these networks. Given a goal or a collection of goals, can we develop techniques that will automatically generate action networks that will robustly achieve these goals?

In order to generate or modify action networks, we need techniques for describing the effects of actions and the conditions under which actions can be executed. This section has already proposed hierarchies of state transition graphs as descriptions of the effects and preconditions of actions. Furthermore, these graphs can be used, as we have illustrated, to synthesize hierarchical action networks directly. Further investigations are needed of how hierarchies of such graphs can be learned by the agent (or otherwise constructed by it) and how incremental changes to the graphs can effect incremental changes to action networks.

We have also done some preliminary work on another strategy for generating action networks. This method is based on a language for describing actions. In the next section, we show how action networks can be generated by executing programs in this language.

IV. The ASTRAL Language

We stated that one of our objectives was to build agents that could both act in real time and let planning and reasoning affect their actions. In this section we propose a programming language in which a designer can describe (at several levels of detail) the actions of an agent. We call the language ASTRAL, an acronym for Agent-Structuring Teleo-Reactive Action Language. Executing a program in ASTRAL constructs an action network. It will turn out that some of what is accomplished by automatic planning systems can be achieved by executing ASTRAL programs.

We can combine acting and planning by executing ASTRAL programs in the context of an already existing and concurrently functioning action network. Many of the goals of agents, as well as the means for achieving these goals, survive for the life of the agent. Thus, we imagine that an agent possesses a large, core action network that is constantly being modified through the execution of ASTRAL programs. But since the agent always has an action network, it is always capable of real-time action.

ASTRAL programs are action descriptions. An action description consists of four parts: the invoking condition, the purpose, the precondition, and the body. These parts involve logical formulas in ordinary predicate calculus. The invoking condition is a literal (that is, an atomic formula or the negation of an atomic formula). The purpose is also a literal. The precondition is a set of literals. The body specifies changes that are to be made to an agent's belief and goal structures. The parts of an action description may have free variables. These are regarded as schema variables. We assume (for now) that all of the free variables in an action description
occur in the invoking condition. An ASTRAL program is *invoked* when one of the goals in an agent's goal structure is an instance of the program's invoking condition. We assume that an agent's goals are ground literals, therefore invocation binds all of the variables in an action description.

Once a program is invoked, it starts building an action unit and connecting it up to the agent's belief and goal structures. The goal input of the unit is connected to the invoking goal in the goal structure. The instantiated purpose of the action description is connected to the corresponding proposition in the belief structure. Each of the instantiated literals in the precondition is connected to the corresponding propositions in the belief structure. Often the purpose and the invoking condition of an ASTRAL-program will be the same literal.

Note that we assume that there are propositions existing in the agent's belief structure corresponding to the instantiations of literals in the purpose and preconditions of the action description. (When we say there is a proposition in the structure, we are saying that there is a place to attach a truth value. The truth value itself is determined either by sensors or by other action units—or both.)

The body of the action description specifies what happens to the output of the action unit we have just constructed. The detailed syntax and semantics of the body depend on whether we are using binary or ternary logic. We assume a binary-valued system here. We allow any combination of the following four statement types to appear in the body:

1) \((LGOAL f)\)
2) \((LBEL f)\)
3) \((DGOAL f)\)
4) \((DBEL f)\)

\(LGOAL\) and \(LBEL\) *latch* the values of propositions in the goal and belief structures, respectively. That is, if the action unit being created is ever turned on, \((LGOAL f)\) latches the value of instantiated proposition \(f\) in the goal structure to True, which value it retains until explicitly reset by some action unit. Similarly for \((LBEL f)\).

\(DGOAL\) and \(DBEL\) are used to set *dynamic* truth values of goal and belief propositions. That is, if the action unit is turned on, the outputs specified by \(DGOAL\) and \(DBEL\) set truth values that retain their settings only so long as the unit is on. When the unit is turned off, these truth values become undefined. Neither \(DGOAL\) nor \(DBEL\) can alter the values of propositions that are latched.

There are two regimes under which ASTRAL programs might be invoked: incremental and full. Incremental invocation of ASTRAL programs is a dynamic process that interacts with the action network. The process begins when a goal in the goal structure is made true. Any matching ASTRAL programs are invoked creating new action units. When these action units are turned on, goals might be made true, invoking other ASTRAL programs. (Concurrently with the invocation of ASTRAL programs, goals may also call existing action units. We assume a mechanism that prevents repetitive invocation of the same programs with the same bindings.)

Under full invocation, propositions in the goal structure, regardless of their truth values, invoke matching ASTRAL programs. These, in turn, invoke others as their outputs create new goal propositions. In either the case of incremental or full invocation, the processes of building or modifying action networks function concurrently with the already constructed action network.
Here is an ASTRAL program that, when invoked, creates one of the action units we used previously in our blocks-world example:

**Invoking Condition:** Disassemble(x,y,z)
**Purpose:** In-line(x,y,z)
**Precondition:** Dis-assembleable(x,y,z)
**Body:**
- (DGOAL Move(x,T))
- (DGOAL Move(y,T))

To make this notation consistent with that used in the blocks-world example, we interpret the names of the lower level actions such as Disassemble(x,y,z) and Move(x,T) as formulas denoting propositions (such as "x, y, and z are dis-assembled." Thus, to enable the routine Disassemble(x,y,z) is the same as giving the expression Disassemble(x,y,z) in the goal structure the value true.

ASTRAL programs bear a close resemblance to THCONSE theorems in MICROPLANNER (Sussman 1970). They are invoked by matching to goal conditions, and they can post subgoals and make and erase assertions in the agent’s database of beliefs. Unlike THCONSE theorems, ASTRAL programs can post subgoals and assertions that endure only so long as the units created by the programs are on. This ability leads to a much richer and more reactive type of control structure. The construction of action networks by ASTRAL programs is also reminiscent of the construction of "procedural networks by "SOUP code" (Sacerdoti 1977).

To build action networks that can set the values of terms, we introduce two new primitives into ASTRAL, namely LSETQ and DSETQ. The construct (LSETQ term value) fixes the value of the term named term to value. The construct (DSETQ term value) ties the value of the term named term to value—dynamically tracking changes in value. We assume that value is any expression that can be evaluated in LISP within the time bounds that we are imposing on the action network.

We will allow LSETQ or DSETQ expressions anywhere in an ASTRAL action description. If such an expression occurs outside of the body of the program, it is evaluated at the time the program is invoked—producing its (latched or dynamic) effects on terms directly. If an LSETQ or DSETQ expression occurs in the body of an ASTRAL program, it is evaluated when (and each time) the action unit built by that invocation of the program becomes active. LSETQ and DSETQ expressions are used to bind any variables in the program that are not bound by the matching operation performed during invocation.

As an illustration of these constructs, consider the following ASTRAL programs describing the actions needed for a robot to move to a point in the plane described by a pair of coordinates (assuming no obstructions):

**Invoking Condition:**
EQUAL(Position, (x,y)) ;Position is the robot's current location.
**Purpose:** EQUAL(Position, (x,y))
**Precondition:** T
**Body:**
- (DSETQ THETA DIR((x,y),Position)) ;DIR((x,y), Position) calculates ;the angular direction from Position to (x,y).
- (DGOAL Facing(T)) ;sets up the goal of turning the robot to ;face in direction T.
- (DGOAL Moving(T)) ;sets up the goal of moving the ;robot in direction T.
Invoking Condition: Facing(u)
Purpose: Facing(u)
Precondition: T
Body: (DGOAL Rotating)

Invoking Condition: Moving(u)
Purpose: F
Precondition: Facing(u)
Body: (DGOAL Moving)

Let's assume that a stationary robot believes it is at location (0,0) and heading east (angular heading of 0) and that we make true the proposition EQUAL(Position, (10,10)) in this robot's goal structure. Invocation of the above ASTRAL programs will then produce the action network that we used as an example on page 10. The first action unit constructed calculates the direction in which the robot must move in order to travel to the goal location. This calculation is performed using the robot's current position (which is assumed to be stored in a belief term) and the goal location. It then invokes a pair of programs with the desired heading as a parameter. The invoked programs create action units that turn the robot in the desired direction and move the robot along a path in the desired direction.

Programs in ASTRAL are capable of constructing action networks, but we have no guarantee that these networks will achieve the goal(s) that originally invoked the programs because the preconditions of critical units may not be satisfied. Automatic planning programs attack the problem of unsatisfied preconditions by turning them into subgoals and then finding programs of actions that achieve the subgoals. Various difficulties arise when it is important to achieve the elements of a set of preconditions in a certain order—an order that avoids the destruction by later actions of conditions already achieved by early actions (Waldinger, 1977; Nilsson, 1980, Chapter 8; Chapman, 1985; Sussman, 1975). Ordering the achievement of preconditions presents more difficulties for us than for some classical planning programs because we do not explicitly prevent concurrent actions and thus can never be sure in which order conditions are actually achieved in the environment. The question of how to generate plans to achieve conjunctive goals is still an important research topic—one that we ignore here by isolating certain simpler aspects of the problem of organizing actions from the more general problem of automatic planning.

We distinguish two reasons for composing actions (rather than using a single action) for achieving a goal. One reason is that the goal might consist of a conjunction of other goals, and different actions are needed to achieve the different conjuncts. The other reason is that an action may require that a precondition be achieved first before the action can be executed.

We can finesse the problem of achieving conjunctive preconditions by adopting the following doctrine: Assume that each action has a single precondition—an atomic one. This doctrine can be enforced by defining new atomic formulas to be equivalent to any non-atomic ones that we might otherwise have used. Coupled with this assumption, we provide at least one action description for every achievable atomic formula. Ordinarily, the actions that achieve non-primitive atomic formulas will be non-primitive actions, which call more primitive ones. In this way, we bury the problem of imposing appropriate partial orders on the achievement of subgoals by specifying constraints on possible orders in the definition of the non-primitive actions. Another way of saying this is that we solve the problem of achieving conjunctive preconditions of actions by specifying in advance a hierarchy of actions that call sequences of more primitive actions. This doctrine is compatible with our approach for specifying action hierarchies.
Of course, it is still possible that the overall, main goal is non-atomic. That problem is what we properly regard as planning, and it is a problem that we are not prepared to tackle here. We set for ourselves only the problem of achieving atomic (but not necessarily primitive) preconditions of actions.

Let us use a simple example to describe how we might approach the problem of achieving (atomic) preconditions of actions. Consider the following ASTRAL program and the action unit it builds:

Invoking Condition: G1
Purpose: G1
Precondition: P
Body: (DGOAL G)

To anticipate the possibility that the agent might not think that P is true, we can include among the ASTRAL programs the following auxiliary program for setting up the subgoal of achieving P:

Invoking Condition: G1
Purpose: P
Precondition: T
Body: (DGOAL P)

The auxiliary program has the same invoking condition, G1, as the main program and thus is invoked at the same time. (The agent doesn't necessarily want to achieve P except in so far as it is a precondition for achieving G1). Note that the precondition list is essentially empty (we regard T as always having value true.) The auxiliary program and the main program together produce the combined network below:
Notice that the purpose and the invoking condition of the auxiliary program (and of the action unit) are not the same. Also, we don't actually have to have the auxiliary program occurring explicitly; the routines that construct action units when ASTRAL programs are invoked can create the auxiliary action unit at the same time.

If $P$ is already true, then the auxiliary action unit does not become active and the subgoal $P$ is not called. Otherwise, the subgoal $P$ is given value 1, which invokes program(s) that create action units able to achieve $P$. If $R$ is a precondition for achieving $P$, then an auxiliary unit for achieving $R$ will be created along with the unit for achieving $P$ by invocation of their corresponding ASTRAL programs. And so on.

It may happen that more than one ASTRAL program is invoked simultaneously. It is in the spirit of our architecture that action units can be added concurrently by simultaneous invocations. Thus the invocation of programs to achieve a goal, $G_1$, might set up a tree of alternative chains of preconditions required to execute the action that achieves $G_1$ before one of the paths in this tree actually results in subgoal-achieving actions. This situation can be diagrammed as follows:
Suppose the heavy edges represent a chain of achievements that can actually happen (at planning time) in the world the agent inhabits—because S2 happens to be true (and believed true) in that world. Under these circumstances, an ordinary AI planning system possibly would have discovered the same plan and explored the same alternative branches. The conventional AI planning system would save for execution, however only the plan represented by the path indicated by the heavy edges—discarding the alternative paths. All of the alternative paths are saved in the action network and are available for execution should unexpected events alter the world in such a way that they can be followed.

(A somewhat more complex, alternative method of avoiding the problem of achieving conjunctive preconditions involves specifying precisely one atomic formula in each action unit that can be "back-chained" on. This formula may be a formula defined in terms of other more primitive ones. This method would allow us to regain the possibility of a non-atomic precondition formula without requiring that we have a way to achieve conjunctive preconditions—important for imposing ordering constraints among actions.)

V. Learning

In addition to the problems of synthesizing or radically modifying networks to organize behavior and to achieve complex goals, we are also interested in the problem of incremental modifications to existing networks through automatic learning strategies.

The following considerations seem to be important in inventing learning strategies for action networks:

1) Precondition inputs "similar" to those that made a unit active should themselves be able to make it active in the future—providing that the prior activity resulted in the unit achieving its purpose. In this case, the precondition should be weakened.
2) Precondition inputs similar to those that made a unit active should not be able to make it active in the future if the unit’s activity ceased (while still called) before achieving its purpose. In this case, the precondition should be strengthened.

3) Action subsequences that result in achieving the purpose of their calling unit should be reinforced. (That is, under similar conditions this same subsequence should become active in the same order.)

4) Sequences will most likely be built up starting from the last unit in the sequence and working backwards. This sets up what Hampson (1983) calls a goal gradient.

5) Units "lower" in the network (that is, units called by other units) ought to change more slowly than units higher in the network. The lower units control actions used by several higher units. Changing them will have radical effects. (Adults learn new words easily but tend to retain the pronunciation of the phonemic building blocks they learned as children.)

When units are implemented using threshold logic units, they can be changed by adjusting the values of their weights. A component of a weight-adjusting strategy that deals with considerations 1) and 2) is the following:

When a unit becomes active begin computing the average of its precondition input vectors. Call this average input vector \( V \). \( V \) is possibly a time-weighted average—favoring more recent inputs. Continue to compute \( V \) as long as the unit remains active. If the unit becomes inactive because its purpose is achieved, add \( V \) into the unit's precondition input weight vector—thus making it more likely that inputs near this average will turn the unit on. If the unit becomes inactive without achieving its purpose but while still called, subtract \( V \) from the unit's precondition input weight vector—thus making it less likely that inputs near this average will turn the unit on. In both cases make sure that the weights associated with the purpose and calling inputs are such that these inputs continue to function as intended.

We might also investigate schemes in which the calling input weights and purpose input weights are adjusted. This allows automatic reconfiguring of networks. In such schemes, we should make it more likely that units that helped a superordinate unit achieve its purpose are called again (under similar circumstances) and less likely that units are called that played no part in achieving the purpose of a superordinate unit.

There may even be weight adjusting schemes in which all the inputs are treated alike but with a bias toward one of them having a high negative weight (this one will become the purpose input) and one of them being necessarily on before the unit can turn on (this one will become the calling input).

Perhaps keeping in mind the homeostatic nature of such networks will aid in devising training strategies. A network that has adapted to its environment will tend to emit actions that turn the higher units off (because it will have achieved the purposes of these higher units).

Weight adjustment schemes, such as those described above, help synthesize (or at least modify) networks through direct contact with the environment. It is to be expected that such schemes will operate relatively slowly (although not as slowly as would evolutionary schemes (Edelman 1987) in which networks participate in a process of random mutation and selective survival). Perhaps such synthesis and modification could proceed more quickly if the anticipated environmental effects of actions were simulated in a model of the environment. Perhaps fast learning schemes in which the environmental effects of actions are simulated instead of experienced will provide us with some interesting new alternatives to planning. (Learning is fast evolution; planning is fast learning.)
VI. Conclusions

We have proposed here an architecture for agents that must act in real time in unpredictable environments. We think that the architecture will prove to be suitable for systems that must reason and plan at run time using declaratively represented information. The work is in a preliminary stage, and much remains to be done. Although we have presented some examples of how action networks might work on small problems, we don't yet know how such networks will scale to larger, more realistic problems. Although ASTRAL programs perform a simple kind of planning, we have yet to investigate how the connection between a general-purpose reasoning system and the acting system might be achieved. (One idea is to provide action units whose outputs invoke an inference process that operates on explicitly represented beliefs to produce deduced beliefs.)

The proposed architecture must also be compared with other approaches to “real-time AI.” In addition to those that were cited in Section I of these notes, some representative systems are those of Hayes-Roth (Hayes-Roth 1987), Barbera, et al. (Barbera 1984), Georgeff and Lansky (Georgeff 1987), Firby (Firby 1987), and Donner and Jameson (Donner 1986). It would also be useful to compare agent architectures being advanced by AI researchers with the already well established and proven methods for real-time, reactive response used in computer operating systems. And, of course, control system engineers deal explicitly with real-time response (Hale, 1973; Bollinger 1988).

However it is connected to the action component, it will be necessary for the reasoning system to be able to reason explicitly about what the action component (its own or that of another agent) will do under various circumstances. For this purpose, we may want to introduce the modal operators BEL and GOAL for composing sentences stating that a proposition is one of the agent’s beliefs or goals, respectively. The operation of the action network itself can then be described by axioms using BEL and GOAL operators together with appropriate notation for describing change. In analogy with Konolige’s sentential semantics of belief (Konolige 1984), we can base the semantics of BEL and GOAL on the sentences in the belief and goal structures of our proposed agent architecture.

There are a number of other important issues that must be investigated. One of these is how to capitalize on the ability of action networks to perform actions concurrently. Our networks based on state graphs and spanning trees are guaranteed to produce only one action at a time (because the conditions labeling nodes in the spanning tree are mutually exclusive and exhaustive). One might imagine that state graphs could be “factored” in some way (Subramanian 1986) into multiple state graphs---each resulting in a concurrently operating action network.

We must also deal with time explicitly---both in the process of reasoning about the activity of action networks and in the networks themselves. The individual action units will have inherent time delays, and we may also want to introduce variable time delays into the circuitry. The finite state machines of Rosenschein and Kaelbling (Rosenschein 1986) include time-delay elements.

If action networks are to be based on hierarchies of state graphs, we must develop a theory of such hierarchies to give us insight about how useful hierarchies can be composed. Some of the AI work on learning, such as that of Anderson and Farley (Anderson 1988), may be relevant here. Detailed exploration of some of the learning strategies that we briefly mentioned in Section V may also suggest techniques for the automatic acquisition of hierarchies.

Finally, we would like to invite attention to the possible use of action networks for developing theories of behavior mechanisms in animals. Ethologists, who study animal behavior, have occasionally speculated about structural models capable of producing goal-seeking behavior...
(Tinbergen 1951). The problem has also interested psychologists (Tolman 1932; Miller 1960), neurophysiologists (Deutsch 1960), and computer scientists (Friedman 1967; Becker 1970).

VII. Acknowledgements

Several people have made helpful suggestions including John S. Anderson, Michael Buckley, Tom Dean, Ken Fertig, Joseph Jacobs, Leslie Kaelbling, Phil Stubblefield, and Monte Zweben. This work has been partially supported by DARPA and NASA under NASA Grant NCC2-494. I am also grateful for support from the Palo Alto Laboratory of Rockwell International.

VIII. References

Agre 1987

Anderson 1988

Barbera 1984

Becker 1970

Bollinger 1988

Brooks 1985

Chapman 1985

deKleer 1986

Deutsch 1960

Donner 1986
Edelman 1987

Fikes 1971

Fikes 1972

Firby 1987

Friedman 1967

Georgeff 1987

Hale 1973

Hampson 1983

Hayes-Roth 1987

Kaelbling 1988

Konolige 1984

Miller 1960

Nilsson 1965
Nilsson 1980

Nilsson 1984

Rosenschein 1986

Rumelhart 1986

Sacerdoti 1977

Schoppers 1987

Subramanian 1986

Sussman 1970

Sussman 1975

Tinbergen 1951

Tolman 1932

Waldinger 1977

Wilkins 1988
Planning and Execution

Stanley J. Rosenschein
Teleos Research

Abstract

This talk examines several basic assumptions of traditional Artificial Intelligence research on planning and execution, arguing that these assumptions have hampered progress and that a more computationally-oriented analysis of information and control in embedded agents is needed.

The fundamental question of agent design is this: How can we design computational systems that transform, in real time and limited space, a stream of data carrying information about a dynamic environment into a stream of outputs that control the environment in desired ways? The challenge is to make the systems as general and flexible as possible while preserving their computational tractability. The classical AI approach to this problem is based on a view of agents as reasoning, self-programming homunculi. The agent is seen as structured into several subsystems that produce and manipulate symbolic expressions. The perceptual system extracts symbolic descriptions of the world and of conditions desired, a planning system constructs a symbolic description of a strategy (plan) for achieving these desired conditions in the world, and an execution system attempts to carry out the strategy by transforming this symbolic plan into overt behaviors.

While this model of behavior generation is attractive in that it provides an intuitive framework grounded in commonly held pre-theoretic beliefs about cognitive structure and function in humans, it is also pernicious because it relegates computational questions to secondary status, often to be addressed by poorly understood heuristic methods. In the long run any strategy based on describing a system in terms of semantically interpreted data objects like “plans” will only work if the intended semantics of these objects and the operations on them are compatible with computational facts of life.

Is this the case for symbolic planning and plan execution? Sadly, the answer is no. As much as we would like to shift the hard job of programming onto the agent by having it accept task descriptions and program itself at run-time, it is clear that the computational complexity of search will defeat this tactic in the general case. The semantics of formal languages for representing general beliefs, goals, and plans simply do not provide enough constraint on the data structures and algorithms to be of much use in guiding an efficient implementation. In fact, they imply a kind of over-generality that in principle cannot be computationally realized that and diverts attention from the real computational issues.

If the metaphor of an agent programming itself by doing general symbolic inference is not computationally realistic, are there any alternative frameworks that might be tried? One approach would be to ask what kinds of computation would be feasible what it is about specific task domains that makes real-time perception and control possible. The second part of this talk describes work based on this point of view, starting with a class of provably real-time computations and showing how those computations can be mathematically modeled as having the kinds of semantic properties we seek. This work has given rise to a programming methodology for agents in which the burden of figuring out what the system should do
is shared more fairly between human designers applying raw intuition, tractable but not real-time automatic compilation procedures, and provably real-time run-time processes.
Panel:

Micro vs. Macro Planning

Chair:
Mark Fox, Carnegie Mellon University

Members:
Amy Lansky, SRI International
N. S. Sridharan, FMC Corporation
Planning in the Large

Mark S. Fox
Carnegie Mellon University

Abstract

Historically, planning research has tended focus on tasks of small to moderate size. For example, early work at MIT focused on blocks world domain, while at CMU it focused on missionary and cannibals domain. By choosing a small domain, one can focus on the problems of interest. Experience has shown that the size of the task raises issues in of itself; that techniques may not “scale up”. The issue I will address is whether planning tasks that are orders of magnitude more complex raise problems (and give rise to solutions) that are outside of the “regular” planning literature.

The problem of scale has arisen in scheduling research. Scheduling is a type of planning problem where there is a large number of agents, activities, resources and constraints. In scheduling, the objective is not only to develop a good plan for each agent, but to optimize as much as possible the aggregate of the plans of all agents. In order to optimize the aggregate it may be necessary to suboptimize an individual’s plan.

The complex interaction of activities, resources and constraints makes local planning decisions myopic. That is, local decisions lead to poor global behaviors. The first insight drawn from planning in the large is that myopic planning behavior can be reduced by solving simple reformulations of the problem that focus on the first order effects and use the solutions to these reformulations to guide problem solving for the more complex case. Methods for reformulation include:

- omission of resources, activities and constraints
- aggregation or resources and time
- ordering variables (e.g., make decisions about shifts before actually scheduling)

Problem reformulation is not an optimization technique. In fact, for some problems, the complexity is so great that there may not exist a feasible solution, given a set of constraints, or the feasible solutions may be difficult to find. In order to construct a solution it is necessary to redefine the problem. This leads to the second insight, that for large problems relaxation of constraints is required. Relaxation requires the ability to differentiate among constraints in order to identify the most likely candidates to relax. Consequently, the goal statement of the problem is more like an objective function in OR.

The third insight is that the degree of contention for resources requires different problem solving behaviors. In cases where resources are bottlenecks, it becomes necessary to optimize the utilization of the resource. Consequently, focus must shift opportunistically from scheduling the activities of an agent to scheduling the activities associated with a resource (i.e., agent vs. resource centered planning).

In order to select the appropriate planning method, it is necessary to identify certain characteristics of the problem. For example, resource contention is determined, prior to
planning, by calculating the expected demand by each agent. This demand is not based on actual plans but on some "expectation" of what the plans will be. Obviously there is a degree of uncertainty in this calculation. This leads to the fourth insight, that planning in the large relies upon probabilistic measures of agent behaviors.

It is the fourth insight that truly differentiates planning in the large from planning in the small. Reasoning about the expected behavior of large groups of agents is a more macro view of planning where earlier research has taken a more micro view, focusing on individual, discrete decisions. Planning at the macro level, by necessity has to be more strategic oriented with a probabilistic/statistical bent, while planning at the micro level is more tactical and discrete.
In my opinion, the issue of “micro vs. macro” does not pertain to any inherent research goal, but rather, is a byproduct of the approach taken to multiagent planning and scheduling. Of course, the ultimate goal of planning should be to work on big (macro) problems. What would be the point otherwise? Humans are just great at working on small problems -- it's the big ones (especially in multiagent applications such as scheduling and coordination) that pose a problem.

Thus, where differences do arise is in approach or research philosophy. For example, if one tries to work directly on large problems to begin with, the inherent difficulty and intractibility of planning leads to mostly heuristic and ad-hoc approaches. If one approaches the problem from a more rigorous or formal point of view, the concerns are primarily with the correctness of problem solving, issues of representation, and the like. This also tends to force one into looking at smaller problems, since such problems are already quite difficult. After certain techniques are established, then the goal is to scale up.

Of course, the middle road is probably the best to pursue; a combination of formal and heuristic techniques should be used. Indeed, this is the way I am trying to approach my own work on the GEMPLAN multiagent planner. GEMPLAN is based on the GEM model -- a formal event-based model for multiagent domains. Domains are partitioned into regions and regional constraints are expressed in an event-based temporal logic. The GEMPLAN planner plans via constraint satisfaction: its task is to construct a network of interrelated events that satisfies all applicable regional constraints and also achieves some stated goal.

A key focus of my work has been on the use of localized techniques for domain representation and reasoning. Such techniques partition domain descriptions and the planning process itself according to the regional structure of a domain. This use of locality helps to alleviate tractibility problems by confining the application of constraint satisfaction algorithms to relatively small regions of activity rather than to large global spaces; the GEMPLAN constraint satisfaction search space is subdivided into regional planning search spaces. A domain's structure can also be used to help pinpoint exactly where regional interactions occur. For these reasons, I believe the use of locality will help to conquer the problem of "scaling up." Locality can also be used to alleviate aspects of the frame problem for multiagent domains.

My presentation in the workshop focused on GEMPLAN's approach to planning and its use of locality, as well as several more basic issues in planning including: multiagent domain representation, reactive versus traditional preplanning, how "planning" might be viewed in more general terms.

As far as representation goes, I stressed the importance of event-based representations for multiagent domains. There has been a growing trend towards the use of events, especially because of the difficulty in representing synchronization properties (for example, properties
dealing with event simultaneity) in traditional STRIPS-like state-based formalisms. This trend can be seen in the work of Kowalski (the event calculus), Drummond, and the interval logics of Allen, McDermott, and others. However, there has been a reluctance to leave the state-calculus behind, resulting in formalisms that blend states and events in often confusing ways. GEM's formalism is purely event-based (although state-based properties can be described in terms of events). GEM is also the only formalism I know of that makes explicit use of domain structure (i.e., locality) to describe domain properties.

As far as the notion of what planning “is,” I feel that planning should really be viewed as a combination of *constraint satisfaction* and *search* and that these two components are orthogonal to one another. In traditional planners, only one or two basic constraint forms are tackled and a limited set of search techniques are employed. Constraint forms are usually limited to the satisfaction of STRIPS-like preconditions (namely, making a state condition true at a particular point within the plan via the addition and manipulation of events or actions) and the decomposition of nonatomic actions into action schemas. Search is typically rigid, involving an interleaving of the use of action decomposition, forward and backward chaining to achieve state conditions, and global interaction analysis among action pre- and post-conditions. Often, notions of hierarchy in search and the inherent hierarchy of nonatomic events are confused.

In GEMPLAN, constraint-satisfaction is broadened to include any form of constraint that the planner is programmed to handle. The planner can also be given user-supplied code for satisfying constraints. The current GEMPLAN system can handle STRIPS-like constraints as well as event prerequisites, nonatomic event decomposition, and constraints described as regular-expression event-patterns that must hold true of a plan. Search is flexible, allowing for the application of constraints and satisfaction methods in a way that can be heuristically tuned by the user.

As far as preplanning versus reactive planning, my opinions are rooted in my experience with both SRI's Procedural Reasoning System – a reactive “planner,” and GEMPLAN. While the reactive application of procedures can achieve interesting and useful behaviors, especially in single-agent environments, I feel that more complicated mechanisms are needed to coordinate multiple agents. This dichotomy is reflected in everyday life as well. People can usually get by with their day-to-day activities by reactively applying known methods for doing things; it’s when coordination among people is involved that some advance thought is required. In domains such as factories, simply reacting would lead to chaos — indeed, human schedulers are usually required to get jobs through the factory in a timely and safe manner.

What is obviously desirable, of course, is to achieve a blend of preplanning and reactive behavior. When the coordination requirements of multiple agents are flexible, more reactive, communication-based coordination methods can be used; in essence, agents can “discuss” what they are doing and coordinate as they go. As things become more tightly constrained, synchronizers can be used that coordinate things in real time. Examples of such synchronizers are traffic lights, queues in stores, and operating systems mechanisms for computers. Finally, when coordination needs are more tightly constrained and correctness is critical, preplanned coordination is necessary. Ideally, a planning environment should be able to flow between these extremes. I believe that the use of locality can make such a flow easier, because the boundaries of interaction between plan components is better defined, making dynamic plan modifications easier to handle.
Panel:

Planning and Plan Recognition

Chair:
Henry Kautz, AT&T Bell Laboratories

Members:
Eugene Charniak, Brown University
Brad Goodman, BBN Inc.
Ray Perrault, SRI International
Plan Recognition

Henry A. Kautz
AT&T Bell Laboratories

Abstract

Many areas of artificial intelligence deal with murky, ill-posed problems, but the area of plan recognition has particularly suffered from the lack of clear descriptions of the subject matter and methodology. In this short session, we ask: what is plan recognition? How can plan recognition problems be formalized? What relation, if any, does plan recognition have to the subject of this workshop, planning?

Henry Kautz and Eugene Charniak will talk on different approaches to plan recognition, with commentaries by Brad Goodman and Ray Perrault.
Probability, Language, and Plan Recognition

Eugene Charniak
Department of Computer Science
Brown University

Abstract

Wimp2, a program developed by myself and Robert Goldman, understands simple (2 or 3 line) stories. It particularly concentrates on plan-recognition and how it interacts with more language-dependent phenomena, such as word-sense ambiguity, noun-phrase reference, case-determination, and semantic guidance of syntax. The first part of my talk will briefly review Wimp2. Wimp2 works by forward chaining from facts found by its syntactic parser, to conclusions about meanings of words, character's plans etc. When Wimp is confronted by ambiguity it asserts all of the the possibilities (as a logical disjunction), and uses an ATMS to handle the fact that the database contains mutually exclusive possibilities. It also assigns probabilities to the disjuncts. The probabilities as used in two ways, first, to concentrate the search so that it explores the most likely possibilities (and thereby Wimp can completely ignore the less likely, unless the more likely ones get ruled out), and second, to make decisions in the typical case that several of the possibilities a logically consistent, but one of them is more probable than the rest.

The second part of the talk will concentrate on a new formulation of Wimp which I am currently thinking about (i.e., there is no program yet). This new formulation was suggested by the observation that Wimp's ATMS is not doing very much work for us, and that, in fact, essentially all of its functionality could be taken over by a suitable probability mechanism. This in turn has lead to a view of story comprehension (and thus plan recognition) in which the problem is characterized by a Bayesian network in which the probabilities of word-senses, possible referents, etc. are all calculated. Unfortunately the network is clearly infinite (there is no bound on the number of plans in the world, or people to carry them out). Thus Wimp can be seen as calculating finite approximations of this network. What are currently formulated as forward chaining logical rules in Wimp2 are to be reformulated as "network constructor rules." These rules express how finite fragments of networks are to be expanded. This reformulation has had several beneficial effects. To take one example, Wimp2 mishandles the example "Jack wanted to kill himself. There was a rope of on floor." There are two ways to think of this example. As read in a story it might be reasonable to jump to a conclusion about hanging. However, it would be incorrect for a real plan recognizer, going about in the world, to make this inference. We want Wimp to act like a plan recognizer since we believe that plan recognition in both stories and the real world are carried out by the same mechanism (in people), and that the real-world version is evolutionarily primary, with the story understanding version parasitic off of it. Unfortunately, Wimp2 would jump to hanging conclusion. (The problem is not confined to Wimp2. Hobb's Tacitus program has the same problem — or would have it if it were extended to handle plan recognition.) This problem can now be diagnosed as a place where Wimp2's forward chaining is misleading. Note that standard forward chaining combines facts about what should key the inference, and the direction of causality as typically, but
misleadingly, replaced by material implication. Separating these two things is the key to the problem.

Lastly we will look at the problem of control of inference as it appears in this new formulation. The basic idea is that as we read a story we are constructing finite parts of our Bayesian network, and that these sections can be disconnected in our finite approximation when in the “real” version they are connected. Furthermore it can be the case that the missing connection radically changes the probability distribution. One possibility we are now exploring is a return to our “marker passing” ideas of a few years ago. This is especially appealing when combined with the use of probabilities to guide search. Thus the lack of a connection is due to the false low probability currently assigned to the boundary node. Assuming that marker passing works, in this new framework it will be finding missing Bayesian network connections between these boundary nodes. Furthermore, the topology of the new portion of the network is independent of the variable bindings (which is what you loose in marker passing) and thus new probability distributions can be computed on the assumption that consistent bindings exist. Thus it would be possible to compute the new probabilities for the boundary statements, and see if the new probabilities are sufficiently high to warrant exploring them any further. If they are not, then there is no point in extending the network by actually computing the relevant bindings. Thus this technique works as a (hopefully cheap) filter on marker-passing paths. It can also be used to justify heuristic filters which have already been proposed. For example, the so-call “isa plateau” rule falls out as a consequence of this interpretation of marker passing.
Metrics for Explanation Simplicity

C. Raymond Perrault
SRI International

Abstract

Charniak and Goldman's "A logic for semantic interpretation" is one of several methods proposed for semantic interpretation or plan recognition based on finding a "simplest explanation" for a text or observed actions, other notable suggestions coming from Allen, Hobbs, Kautz, and Wilensky. Parallels with work on diagnostic, speech and image analysis have also been drawn. Although these proposals are similar in spirit, they differ substantially on what simplicity metrics should be, and thus on the relation between observations and explanations. We will survey some of these differences.
Formal Theories of Plan Recognition

Henry A. Kautz
AT&T Bell Laboratories

1 Introduction

There has been a great deal of work in artificial intelligence in plan recognition. This area has also been called "motivation analysis" or "story understanding", and has been stretched to include practically all problems an intelligent agent encounters in understanding and interacting with the social world. Yet it has not been clear what actually constitutes a plan recognition problem or its solution. This paper will discuss various ways to formally define plan recognition, and will go into detail about a method which views it as deduction under a strong set of assumptions about the possible causes for actions. Yet we remain unsatisfied with the "passive" nature of any of these models of plan recognition, and will sketch an approach which emphasizes the role of plan recognition in coordinating the recognizer's own actions with those of other agents.

2 The Informal Problem Statement

An informal description of plan recognition seems straightforward: the task of inferring an agent's goals given some of his or her actions. But it is not enough, in general, to simply infer the ultimate goal or end-state the agent desires; indeed, Pollack [7] studied cases of plan recognition where the ultimate goal is known, and the observer must infer how the agent believes its actions will lead to the attainment of that goal. For example, if someone says, "I want to speak to Mary. What's the phone number of the hospital?", one can infer that the speaker believes he or she can talk to Mary by calling her at the hospital.

Thus the solution to a plan recognition problem would include not only a description of the agent's goals, but also the future actions of the agent, the causal links between the actions and the goals, and the auxiliary assumptions about the world which must hold if the actions are to be effective. In the example above, the assumptions include the fact that Mary is at the hospital.

3 Representing Plans

Many difficult problems arise in representing actions, beliefs, assumptions, and goals, but we will not concentrate on those issues here. A plan will simply be taken to be a partial state of affairs. A proposition is associated with each plan, such that the proposition holds if the plan actually occurs. A proposition is also associated with the occurrence of each action. The "plan library" axiomatizes the relationships between plans and actions. "Decomposition" axioms describe the substeps which must occur when a plan occurs. For example, the axiom

\[ \text{RobBank} \supset \text{GetGun} \land \text{EnterBank} \]
Formal Theories of Plan Recognition

asserts that if the \textit{RobBank} plan occurs, then \textit{GetGun} and \textit{EnterBank} must also occur. The latter two propositions could stand for either actions or plans. "Abstraction" axioms relate a plan or action to a more abstract way of describing that state of affairs. For example, \textit{GetMoney} would be true in any state of affairs in which the agent does something to obtain money. The \textit{RobBank} plan describes a particular way of obtaining money. Thus the library would contain the abstraction axiom:

\[ \text{RobBank} \supset \text{GetMoney} \]

For simplicity of exposition, this paper uses a simple propositional symbol such as \textit{RobBank} to stand for each plan or action. In general, however, one needs a representational system which allows one to distinguish different instances of a common plan type. The plan in which Bonnie robs the First National Bank yesterday is different from the one in which Clyde robs United Savings and Loan tomorrow, yet both are instances of robbing a bank. This greater expressive power can be obtained by introducing predicates to stand for plan or action types, and terms to stand for instances. For example, let us take \textit{RobBankP} to be a predicate, and \textit{Rob99} to a term which refers to yesterday's robbery by Bonnie. Applying the predicate to the term creates the proposition

\[ \text{RobBankP}(\text{Rob99}) \]

which is true just in case \textit{Rob99} really did occur. Functions applied to a plan or action instance return characteristic properties of the instance, such as its agent or time. Thus yesterday's robbery could be more fully described by the sentence:

\[ \text{RobBankP}(\text{Rob99}) \land \text{Agent}(\text{Rob99}) = \text{Bonnie} \land \text{TimeOf}(\text{Rob99}) = 12/09/88 \]

The plan library now contains universally quantified sentences, or schemas, which relate the plan types. The decomposition axiom above would become:

\[ \forall x. \text{RobBankP}(x) \supset \exists y, z. \text{GetGun}(y) \land \text{EnterBank}(z) \land \text{Before}(\text{TimeOf}(y), \text{TimeOf}(z)) \]

Note that sequencing information can be expressed by predicates such as \textit{Before} on the times of times of pairs of plan or action instances. More details of this representation appear in [3].

4 Formalizing Plan Recognition

How can one formally represent a plan recognition problem? While one can argue that any attempt to provide a simple formal model of plan recognition will necessarily abstract away interesting and vital aspects of the problem, the same objections can be made to practically any formal work in AI. The classic formalization of planning in terms of the situation calculus [5] surely neglects many important aspects of planning—such as goal choice, control, parallelism, and execution—but none the less serves as a foundation for understanding planning in general. Thus we seek a similarly idealized formalization of "classic" plan recognition, to serve as a basis for further understanding of this area.
Standard accounts of plan recognition [8] usually involve a passive observer who receives reports of an agent's actions. The reports are incomplete and fragmentary. The recognizer is then quizzed as to how the actions were related, what unobserved events must have occurred, and what will happen next. There have been some exceptions to this paradigm—notably, Allen's account of a plan recognizer [1] which is tightly coupled to a planner/debugger—but for now, we will remain within the world of non-interactive, "keyhole" plan recognition.

A natural way to answer such questions about the agent's intentions and beliefs is to reconstruct his or her plan, and use this single model as the basis for response. The reconstructed plan provides an explanation for the agent's actions. Therefore we'll examine some formal models of explanation in the context of plan recognition.

5 Plan Recognition as Explanation

A simple model of explanation [6] takes one to consist of three parts: a set of observations to be explained O, a set of general laws L, and a set of auxiliary facts and assumptions about the world A. (The set A is often simply called a set of facts, which can misleading, because the point of the inference is to determine A—the explainer need not know that A already holds.) In this "deductive nomological" (DN) model, the general laws together with the assumptions (facts) should entail the observations:

\[ L \cup A \models O \]

In plan recognition, the set of laws is the plan library. For example, let the observation be that the agent gets a gun:

\[ O = \{ \text{GetGun} \} \]

As described above, L might include an axiom that states that robbing a bank involves getting a gun:

\[ \text{RobBank} \supset \text{GetGun} \in L \]

The assumption set asserts that certain high-level actions occur. A possible candidate is of course RobBank, so the explanation becomes

\[ \{ \text{RobBank} \supset \text{GetGun} \} \cup \{ \text{RobBank} \} \models \text{GetGun} \]

There are several problems with this approach.

1. Goals don't entail actions. There are in general many ways that an agent can achieve a goal. For example, cashing a check and robbing a bank are both ways of getting money. Formally,

\[ \text{CashCheck} \supset \text{GetMoney} \]
\[ \text{RobBank} \supset \text{GetMoney} \]

Intuitively it seems reasonable to explain why someone robs a bank by saying that he or she wants money. But this kind of explanation does not fit the previous pattern. That is,

\[ \{ \text{RobBank} \supset \text{GetMoney} \} \cup \{ \text{GetMoney} \} \n \text{RobBank} \]
In brief, whenever there is a choice of ways of doing something, that choice must be included as one of the assumptions in $A$. This trivializes the explanation in this case; in fact, $L$ can be ignored, and there seems to be no reason to include $\text{GetMoney}$ in $A$.

$$\emptyset \cup \{\text{RobBank}\} \vdash \text{RobBank}$$

Thus it becomes hard to characterize what is allowed to appear as an auxiliary assumption; if anything can appear (such as $0$ itself) then the framework collapses.

(2) Explanations aren't unique. Another problem arises from the need to find a particular explanation for the observation. There may be several explanations available, and it may be neither necessary nor desirable to choose between them. To continue with our example, the plans to cash a check and rob a bank both involve going to a bank—

$$\text{CashCheck} \supset \text{EnterBank} \land \cdots$$

$$\text{RobBank} \supset \text{EnterBank} \land \cdots$$

A “safe” explanation for the observation $\text{EnterBank}$ would be $\text{GetMoney}$. But as we've seen, this explanation would not entail the observations. One would have to choose between

$$A_1 = \{\text{CashCheck}, \text{GetMoney}\}$$

and

$$A_2 = \{\text{RobBank}, \text{GetMoney}\}$$

A partial solution to this problem is to assign likelihoods to various candidate assumptions, and select the set with highest probability. While this is reasonable with the current example (certainly cashing a check is much more likely than robbery), cases will eventually arise where the alternatives are equally likely. Another (non-exclusive) solution is to introduce disjunction into the assumption set, e.g.

$$A_3 = \{\text{RobBank} \lor \text{CashCheck}, \text{GetMoney}\}$$

This greatly expands the number of possible solutions to a plan recognition problem (the number of $A$'s which satisfy the DN schema), making the plan recognition problem—already ill-posed—even more so.

## 6 Plan Recognition as Closed-World Reasoning

Concern with these problems led the author to develop a different formalization of plan recognition [3]. While the approach remains primarily deductive, the relation of the solution to the observations is reversed. Instead of finding plans which entail the observations, one is justified in concluding whatever follows from the conjunction of the observations, the plan library, and a number of closed-world and simplicity assumptions. Thus one is not required to construct a single, complete model of the agent's plans. In theory at least, one can match the amount of inference to be performed off the observations, library, and assumptions with the importance of the specific questions one has about the agent's intentions.

This reversal of the DN pattern in fact has an important relation to explanation. Consider the set of all possible explanations for the observations—that is, the set of plans
which, according to the decomposition axioms in the plan library, contain the observations as substeps. Then the recognizer can conclude any statement which follows from any of the explanations. (In the first-order case, one considers the set of plan types, any of whose instances entail the occurrence of an action of the same type as the observation. The observer can conclude any statement which follows from the existence of an instance of any of the explanatory plan types.)

Let the relation

\[ O \triangleright_L C \]

hold if \( C \) is such a conclusion from observations \( O \), relative to a plan library \( L \). Before formally defining \( \triangleright_L \), we'll employ it in our example where

\[
L = \left\{ \begin{array}{c}
\text{CashCheck} \triangleright \text{GetMoney} \\
\text{RobBank} \triangleright \text{GetMoney} \\
\text{CashCheck} \triangleright \text{EnterBank} \land \cdots \\
\text{RobBank} \triangleright \text{GetGun} \land \text{EnterBank} \land \cdots \\
\text{GoHunting} \triangleright \text{GetGun} \land \cdots
\end{array} \right\}
\]

Then

\[ \{\text{EnterBank}\} \triangleright_L \text{GetMoney} \]

holds. Note that \( \text{GetMoney} \) does contain \( \text{EnterBank} \) in its decomposition axiom (indeed, it has no decomposition axiom), but \( \text{EnterBank} \) does appear in the decomposition both \( \text{CashCheck} \) and \( \text{RobBank} \). The latter two plans explain the observation, and since \( \text{GetMoney} \) follows from both, it is a valid conclusion. Similarly it is the case that

\[ \{\text{EnterBank}\} \triangleright_L \text{RobBank} \lor \text{CashCheck} \]

But it is not the case that

\[ \{\text{EnterBank}\} \triangleright_L \text{RobBank} \]

A potential problem is the great number of possible explanations for a set of observations, which could lead to very weak conclusions. Two assumptions limit this: first, as noted, one only considers plans which are straightforward expansions of known plans—one does not synthesize original explanations by piecing together new sets of actions. Second, in the case of observations of multiple actions by the agent, the recognizer considers only explanations which contain as few unrelated plans as possible. For example, let the observation set be

\[ O = \{\text{GetGun}, \text{EnterBank}\} \]

This could be explained by supposing that two different plans were occurring, where the first is a \( \text{RobBank} \) or \( \text{GoHunting} \), and the second is either a \( \text{RobBank} \) or \( \text{CashCheck} \). If we assume that all the observed acts are part of the same plan, then we can conclude that the \( \text{RobBank} \) plan must be in progress. That is,

\[ \{\text{GetGun}, \text{EnterBank}\} \triangleright_L \text{RobBank} \]

The pattern of inference in this style of plan recognition often involves reasoning by cases. The first observation leads to a disjunctive conclusion (e.g. \( \text{RobBank} \lor \text{CashCheck} \)
above) and succeeding observations rule out various alternatives. (Alternatives can also be ruled out by disproving constraints on the plans, as described below.)

There is a simple definition for the $\triangleright_L$ operator in terms of ordinary deduction. The axioms in $L$ are conjoined with the following kinds of assumptions:

1. The most important assumption is that an action entails the disjunction of all higher-level actions which contain it as a substep. In the simple propositional case we're describing here, such an assumption is

   $$ EnterBank \triangleright RobBank \lor CashCheck $$

The first-order case is more complicated, because one must also consider cases where the explanation involves a more or less abstract redescription of the observed action. For example, every instance of entering a bank is also an instance of entering a public building. Further suppose that there is a plan to avoid getting wet in a rainstorm by ducking into a public building. This might be represented as follows, where the fact that it must be raining is constraint on the applicability of this plan. The variable $x$ ranges over action instances.

   $$ \forall x. StayDryP(x) \triangleright \\
   \exists y. EnterPublicBuildingP(y) \land \\
   Raining(TimeOf(x)) \land \cdots $$

Then a possible reason for entering a bank is to execute the StayDry plan. Thus the proper assumption for uses of EnterBank becomes

   $$ \forall x. EnterBankP(x) \triangleright \exists y. \\
   RobBankP(y) \lor \\
   CashCheckP(y) \lor \\
   StayDryP(y) $$

Naturally if the “raining” constraint is known to be false, then this extra alternative can be discarded.

2. Another set of assumptions states that all the ways of performing a high level action (e.g., the relation between GetMoney and CashCheck) appear in the plan library, and that all action types are disjoint unless known otherwise. [3] describes how these assumptions are specified and used.

3. Finally, simplicity assumptions are needed so that multiple observations decrease, rather than increase, the ambiguity of the conclusions. In the propositional case, one asserts that at most one highest-level action—that is, an action which doesn’t appear as a step of another action in the plan library—occurs. In this example, this assumption would be

   $$ (\neg RobBank \lor \neg CashCheck) \land \\
   (\neg RobBank \lor \neg StayDry) \land \\
   (\neg CashCheck \lor \neg StayDry) $$

The first-order case is handled by introducing a special action type for “End” (non-explainable) actions, and then minimizing the number of End actions. Again, details appear in [3].
The form of the inference, then, is

\[ \text{Observations} \cup \text{Assumptions} \cup \text{Library} \vdash \text{Conclusions} \]

corresponding to the relation

\[ O \triangleright_L C \]

described above.

7 Problems with the Paradigm

This formalization repairs the shortcomings of the simple DN framework described earlier, while including some weaknesses of its own. First, in many domains it is simply unrealistic to consider all plans which could include an action as a substep; the most common alternatives should be considered first. An implementation is free, of course, to employ various strategies to minimize this problem. For example, if the initial observation leads to highly ambiguous conclusions, the recognizer may choose to ignore it until more useful observations are input. The earlier observations could then be used to rule out some of the alternatives generated by the later observations, rather than the other way around. In addition, nothing in the framework is "anti" probabilistic. Indeed, the completeness assumptions are exactly the kind of assumptions that one implicitly makes in employing a Bayesian analysis. That is, one assumes that the set of hypotheses (the plans) is exhaustive and disjoint.

These questions of ambiguity and control aside, a troubling issue remains: is there something wrong with the whole notion of keyhole plan recognition? Who is this passive observer, and why should he or she bother to recognize plans? How should the observer gauge the effort to put into recognition? Why does there appear to be no significant relation between the process of planning and that of plan recognition?

8 Active Plan Recognition

People rarely try to recognize other agent's plans in a vacuum. There is usually some overriding purpose: to coordinate actions, to avoid conflicts, to achieve some mutual goal. Work on plan recognition has been misled, I suspect, by the emphasis on "story understanding", where the recognizer is detached from the actions of the characters. Understanding literature—not two or three sentence exercises—does involve the reader directly, calling into play his or her goals and intentions, and can often bring the reader to reason about the goals and intentions of the writer. One reads a narrative of a robbery, for example, differently in Oliver Twist than in a newspaper article; and if the newspaper article is written, say, by William F. Buckley, one is incessantly aware of the the writer's intention to communicate some other agenda through the description of the crime.

This suggests that the passive observer/active agent model should be replaced by one in which plan recognition naturally arises as part of the goal-seeking behavior on the part

\[ ^1 \text{The following speculations were partly inspired by comments Richard Korf made at the 1988 Distributed AI Workshop.} \]
Formal Theories of Plan Recognition

of multiple agents. The result may be a theory which includes the phenomenon of plan recognition without invoking plan recognition as an explicit concept.

The work in AI on game playing provides an example of such a theory. Under conditions of mutual hostility and perfect knowledge of goals and past actions, two agents try to select their own actions. The minimax algorithm performs this computation, finding actions which coordinate appropriately with the opponent’s long-term strategies (i.e., plans) without explicitly invoking the notion of a strategy. That is, it does not try to “explain” the opponent’s actions.

How can the game-playing approach be modified to handle the kind of plans that appear in plan recognition? Replace competition with coordination—e.g., replace minimax with minimin [4]. Relax the requirements of perfect information, but then use the notion of a plan library to limit the possible plans the other agent may be performing. That is, don’t consider all possible actions by the other “players”, but only actions which are grouped as part of common plan schemas.

These ideas are still sketchy, but here is a possible scenario, a kind of “blocks world” problem for active plan recognition.

1) You want to meet Joe when he arrives in town. He could be flying or driving. You know that it is a short plane trip but a hard drive, so you figure that he will prefer to fly. Accordingly you plan to meet him at the airport. Note that your own goal— to achieve coordination with Joe led you to reason about his plans and intentions.

2) You learn that Joe has rented a car. You explain this by supposing that he is going on a long trip—and further suppose this is the trip to visit you. So you tentatively conclude that Joe is going to drive, and don’t go to the airport.

3) Instead, you call Joe on the phone. He states that he is flying in. You now believe he is flying (not actually a trivial inference) and so you plan again to meet him at the airport. You forget about trying to explain why Joe rented a car—it now appears to be irrelevant to your own plans.

9 Conclusions

The theory of plan recognition as closed-world reasoning has been implemented and applied to a number of domains, including plans about cooking [3] and plans about the design of chemical processes [13]. Perhaps the most persuasive argument for the theory is that it provides a correctness criteria for a complete plan recognition system. The implementations do not compute all consequences justified by the theory, but what they do output is a correct consequence of the $\triangleright_L$ operator. There is no need to appeal to an external criteria to select between different conflicting conclusions, as one must in the deductive nomological model.

The closed-world theory is successful only in domains which have the characteristics described earlier, namely, that the recognizer is essentially passive and the plan library is static. These limitations became problematic when the recognizer was incorporated in the user interface of a real chemical design system. People, as opposed to characters in stereotyped stories, are always somewhat unpredictable, always performing some novel plans, even in seeming rigid and mundane tasks. In addition, people expect an “intelligent”
system to be an active partner who seeks to help accomplish the task at hand, not merely a passive observer who bookkeeps and warns of possible errors.

Work on simple multi-agent scenarios of the kind described above, as well as plan-based approaches to modeling natural language conversation, should make progress toward a theory of active plan recognition. Active recognition could provide a better basis for plan recognition in "intelligence assistance" programs, such as the design interface. The long-term result of development of the active model, may well include a better understanding of problems which were formerly thought to be keyhole recognition tasks, such as story understanding.

References


Towards Useable Plan Recognition

Bradley A. Goodman
BBN Systems and Technologies Corporation

Plan recognition in artificial intelligence has most often been studied as a way to improve the habitability of complex application systems or to explain certain phenomena in natural language understanding. Recent research in the area, however, has focused on formal studies of plan recognition and neglected practical implications. The formal studies do not consider the kinds of tasks where plan recognition is useful and the influence of those tasks on the plan recognition problem itself. They separate the specification of the plan recognition problem from the algorithms to solve it. In this paper we reflect on these issues and introduce a more constructive view of plan recognition that we feel can help bring back the influence of the task itself.

1 Exploring Plan Recognition for Plan Recognition's Sake

The use of plan recognition to enhance both natural language and other interface systems has been widely documented in the artificial intelligence literature [1, 4, 5, 7, 8, 10, 16, 21, 23, 24, 26, 25, 28]. Plan recognition has been used in natural language systems to provide helpful responses [1, 21], and for reference resolution to resolve elliptical fragments [5]. It has also been employed in more standard interfaces to provide helpful support for the novice user (e.g., help systems [10] and operating systems [29, 16]). Such research led to numerous strategies to recognize and track a user's plan [1, 26]. More recently, however, plan recognition research has focussed on formal aspects of the problem [9, 19]. Such efforts rigorously define plan recognition without regard to its use to solve particular problems. In the section that follows, we discuss the results of one of the formal analyses of plan recognition, the one by Kautz, and point to problems applying it to real tasks. We follow that with a push back to early plan recognition research to motivate other approaches. Finally, we describe our recent efforts at studying plan recognition in a real environment, an intelligent interface, where we must contend with realistic assumptions about our task domain.

2 Learning from Formal Plan Recognition

Kautz [19] was one of the first to formally define what actually constitutes a plan recognition problem and solution, independently of any algorithms. He formalized the semantics and clarified the limitations of a broad class of recent plan recognition theories, in particular those governed by a closed world assumption on the plan library [16, 26], and shows the limits of such approaches. Kautz's formal analysis of plan recognition provides a solid foundation for future research in plan recognition to build on, setting the standard for rigor.

---

1 Partial support for this research was provided by the National Science Foundation under grant IRI-8701874. The research described in this paper was done in collaboration with Diane J. Litman of AT&T Bell Laboratories.
and clarity. It accounts for numerous problems ignored by previous researchers in the areas of ambiguity, abstraction, and complex temporal interactions. Kautz insists that a theory of plan recognition conclude only what is absolutely justified on the basis of observations, one's knowledge of actions, and other explicitly defined closed world and simplicity assumptions. It is the formalizing of the explicitly defined assumptions that provides the potency in his analysis. Kautz provides a formal theory of recognition and presents us with a language for describing different recognition algorithms.

One problem, though, in Kautz's attempt to provide an all encompassing theory is that the algorithms associated with it are extremely expensive computationally. His formulation views plan recognition as identifying all possibilities but that is not always what one wants. In real life people often overreact or underreact to an evolving situation. For example, a teller in a bank does not press the alarm button every time someone walks into the bank just because they might be there to rob it. A person would hopefully, however, jump out of the way of an on-coming car even though it might veer away at the last instant. Plan recognition, hence, should also consider the likely probability of particular instances [11] or use heuristics to order them [7, 21]. Such considerations can limit the search space making searching it more tractable. Actual experience with the Kautz plan recognizer has shown immediate computational explosion as the knowledge base expands beyond toy examples. One likely reason for this is that he considers plan recognition in the abstract devoid of any particular application while many of the previous plan recognition algorithms were designed with particular applications in mind forcing their creators to contend with the computational issues.

Even with the robustness provided by Kautz's formulation and leaving implementation details to the system builder, his formulation still hinges on some of the same limiting assumptions made by many previous researchers in plan recognition. He assumes that a complete library of plans (which he calls the "set of general laws" - a set of plan schemas that decompose high-level actions into individual steps) is available to draw upon. Kautz's formulation, hence, can only consider direct expansions of known plans in the library. That eliminates any possibility of novel plans, defined here to be novel sequences of actions achieving known or novel goals. He also assumes that there is a single agent eliminating collaborative interactions amongst agents. Finally, he assumes the correctness of plans. Research by Pollack [23] addresses how to survive with erroneous plans. Research by Grosz and Sidner [14] describes collaborative interaction amongst agents to complete a plan. Some recent work by Goodman and Litman addresses the issue of novel plans and living without the completeness assumption. I will briefly describe the latter work in this paper.

We describe in the sections that follow a new kind of plan recognition that will allow us to tackle more robust applications. We start first with some motivation for our new plan

---

2For example, expensive searches through an interleaved abstraction/decomposition hierarchy is required in Kautz's algorithms. If he had considered the search costs as part of a particular application, he might have structured the abstraction/decomposition hierarchy more effectively. Of course, Kautz's formalization was intended to be just that and he left details of particular implementations to the system builder. We define in more detailed implications of Kautz's and other plan recognition algorithms in Goodman and Litman [13].

3In Kautz's paper in this volume, he steps back from this assumption and defines what he calls "active plan recognition." Active plan recognition is defined to arise as part of the goal-seeking behavior of multiple agents. It, too, requires the notion of a complete plan library to limit the possible plans the agent might be performing.
Towards Useable Plan Recognition

A recognition technique based on previous research in plan recognition. A more expanded version of the ideas in those sections can be found in Goodman and Litman [13].

3 Returning to the Roots of Plan Recognition

We propose looking further back to the early pioneering work in plan recognition [25, 10, 1], that is to the work which was not governed by the completeness assumption on the plan library. The analysis by Schmidt et al. [25] is heavily motivated by psychological modelling and uses causality instead of (implicitly) prestored “legal” action sequences (as in the “syntactic” approaches [16, 26] to plan recognition) to tie together observed actions. For example, preconditions of a newly observed action are compared to the effects of previous actions to determine the new action’s suitability, e.g., to determine if the new action is enabled by the previous action. After each observation, a small number of alternative plan hypotheses are generated. The plan recognizer uses these plans and knowledge about the domain to generate expectations concerning what actions to expect next. However, if the expectations are not satisfied, a revision process is employed to reformulate the current hypothesis. (With the completeness assumption, this situation would typically signal an error). Allen [1] developed a plan inference system that also chains together actions via decompositions and effects. His algorithm incorporates a set of plan recognition rules and a heuristic control strategy. The rules determine which inferences are possible. They define a valid chain of inferences from observed actions to plausible goals, while the control strategy considers the likelihood of these inferences in order to commit to a particular plan immediately. Allen's algorithm, as formulated, is intended to handle only single observations. Finally, Genesereth's plan recognition strategy [10] draws heavily on a prestored model of the typical user to detect mistakes and misconceptions by the user. The procedure reconstructs the plan based on underlying beliefs in the user model. It suggests partial parsings of the user's actions and uses the user model to filter those suggestions. Thus, it too differs from the syntactic approaches by using knowledge about the user that is outside the plan library. As in these early approaches, we propose to construct valid plans from observed action sequences using first principles. We briefly describe our constructive view of plan recognition (CPR) in the sections that follow.

4 Surviving with Constructive Plan Recognition

Many of the implementations of plan recognition algorithms have never been embedded inside a complete working application. As a result, crucial issues of robustness, reliability and inherent limitations remain. In particular, most current algorithms make the incorrect assumption that valid and complete plan knowledge is specified and shared by all agents, and they do not consider the fact that users often make mistakes or get sidetracked. Our research attempts to rectify these deficiencies.

First, we wish to remove the assumption that the system's knowledge of the user's plan is complete. This is especially necessary in contexts where the user is often creating new plans. If we remove this assumption, however, our plan recognition algorithm cannot be limited to matching into a predefined set of expected plans. It must be more constructive in nature and attempt to fill in potential knowledge gaps when presented with a novel plan.
It must dynamically construct from the incomplete knowledge new ways of performing high level actions, if such actions can be seen as purposeful. One goal of our work is, thus, to identify the relationship purposeful so that our plan recognizer can efficiently conclude the proper inferences.

Many existing plan recognizers define a purposeful action sequence “syntactically” [16, 26, 19]. They perform a process similar to parsing by attempting to fit observed user actions into an expected user plan as (implicitly) defined by a complete plan library, not unlike a language that defines all possible sentences. CPR attempts to extend this parsing analogy by developing a “cascaded” parsing algorithm where syntax and semantics work hand in hand [30]. However, while a traditional cascaded parser uses semantics to verify or eliminate existing syntactic choices, we are using semantics to construct additions to an incomplete syntactic language. That is, if CPR cannot parse an observed action sequence, it attempts to determine whether or not the sequence could be part of the plan language (i.e., is “purposeful”) by applying semantic information to piece together actions. Purposeful actions must be able to be seen as being part of an action sequence on the way towards achieving some goal. In the case of novel sequences, however, that goal is not necessarily known.

Similarly to the early work mentioned in the last section, CPR can use plan generation techniques to determine whether novel sequences of actions are potentially purposeful. For example, CPR incrementally examines (using local backward and forward chaining) the effects and preconditions of actions in an action sequence and the propagation of those effects to determine whether or not they fit together well. In other words, an action sequence is (potentially) purposeful if it could have been generated by a planner from first principles. When effects of earlier actions neither violate nor achieve preconditions of later actions, CPR can say nothing definitive about the causal structure of the plan. Instead, the desired interactions are viewed as expectations that need to be satisfied before the plan specification is through in order for the action sequence to remain valid. Because CPR uses bottom up techniques to incrementally verify the coherence of action sequences, the search explosion involved in earlier bottom up approaches to plan recognition (where search spaces were generated to propose action sequences) can be controlled.

Besides plan recognition by plan generation techniques, we are also investigating the recognition of plans by plan modification. What we learn from failed plan parses from the incomplete library, along with techniques of case-based and adaptive reasoning [2, 3, 15, 18, 20] can be used to determine if novel actions and goals form reasonable plans. Finally, we only detect an errorful plan in the case where our semantic plan recognition techniques fail. We can then employ relaxation techniques similar to those used in natural language [6, 12] to provide diagnostic support to repair the plan.

---

4 The plan parser, thus, fills in knowledge gaps in our plan library.
5 If an effect of one action violates a precondition of a later action in the sequence, then a plan generator determines if another action (other than directly undoing the original action) could be hypothesized to restore the desired precondition. If so, the input is considered potentially purposeful.
6 In the early semantic-based recognition algorithms the chaining together of actions was either controlled by expectations once the goal was known, or if totally bottom-up, it exploded.
5 Applying Constructive Plan Recognition

As part of our effort to determine how tasks constrain plan recognition algorithms, we are performing our research in the context of a real application. We are attempting to utilize constructive plan recognition in the context of a complete working application system for performing plan-based process design. Design appears to be an excellent forum for applying plan recognition. We believe that a plan-based system can add a further layer of sophistication to design interfaces, making them easier to use, while making them more powerful. For example, they could recognize known designs, determine when new designs are reasonable, and provide diagnostic support when they are not. Furthermore, design appears to be especially suited for the recognition and acquisition of novel plans: since the user is in general creating new designs, the system's design library is inherently incomplete. We can perform constructive plan recognition by virtue of the compositional kinds of actions and effects that make up process design operations. A more detailed treatment of our design interface can be found in Goodman and Litman [13].

6 Conclusion

Our empirical study of plan recognition for interfaces demonstrated that, contrary to recent trends, plan recognition should not be developed in isolation. Such a dose of reality requires changing the assumptions underlying the Kautz formalization of plan recognition. We propose that a more constructive view of plan recognition is needed to provide robustness to plan-based systems. In particular, plan recognition must contend with a fact about the world: agents have incomplete knowledge of each other and of the world. However, if we drop the completeness assumption, most previous plan recognition technology no longer suffices. We have proposed a framework, constructive plan recognition, that is designed to survive without the completeness assumption.

Constructive plan recognition is a form of problem solving that induces on the fly semantic generalizations on the plan library hierarchy. Instead of expending a lot of effort ahead of time to encode a very precise and complete hierarchy with lots of abstract entities represented, we form (and thus "learn") the generalizations on the basis of semantics. The burden, thus, is moved from the library to extensive descriptions of individual actions and their effects from which plans in the library are composed. Such actions, however, are much easier to isolate given a particular application domain.

Acknowledgements

Thanks to Kimberly Claffy of AT&T Bell Laboratories for her experimentation with Kautz's plan recognition implementation.

References


Towards Useable Plan Recognition


Panel:

Reactive vs. Strategic Planning

Chair:
Drew McDermott, Yale University

Members:
Dave Chapman, MIT AI Lab
Tom Dean, Brown University
Mike Georgeff, SRI International
From Planning to Instruction Use

David Chapman
MIT Artificial Intelligence Laboratory

Abstract

I argue that understanding plan use is a prerequisite to understanding plan making, and that understanding instruction use is a prerequisite to understanding plan use. I describe a system under construction, Sonja, which can make flexible, sensible use of instructions if it is given them in the course of activity.

Plans and plans

I and others have previously argued that Planning, the construction of programs controlling future action, is impractical and unnecessary [1, 2, 3, 4]. More recently, Phil Agre and I have argued that it is important that people can and do make and use plans [3]. By “plans”—with a lowercase “p”—we mean whatever a layman would use the word to mean. We don’t know whether robots need use plans in the same way people do, but as long as alternatives are in short supply, empirical studies of human plan use will be useful. Unfortunately there has been very careful little study of human planning. This paper argues that before studying planning, we need to understand plan use, and that before understanding plan use, we should understand the use of instructions given in the course of on-going activity. It describes an implementation in progress of a system which makes flexible use of such instructions.

Traditional Plans are used by Executing them as programs. Agre and I have argued [3] that real plans are used in much more complex, flexible ways. The plan’s user is as smart as the plan’s maker. This means that the plan’s maker can depend on the user to make sensible interpretations of the plan in context. This makes plan making much easier. We don’t as yet fully understand how plans are used; and since plan making depends on the nature of plan use, and since plan use can be studied in isolation, we believe that it is appropriate to begin there.

Many real plans are physical objects such as wall charts, instruction sheets, blueprints, and bound business plans. Others are spoken utterances, as might be produced in response to the question “So, what’s your plan for the afternoon?” The nature and use of these plans are relatively easy to study. People also make and use internally-represented plans. However, I believe that these internal plans are similar to external ones in important respects. In any case, external plans seem like a good place to start.

Reproduced here is a plan for making a “glider,” a sort of swinging sofa. This plan came as part of a glider kit. Candace Sidner videotaped two people constructing the glider from...
this kit. That there are two people collaborating makes it relatively easy to see how they use the plan.

Traditionally Plans have made extensive use of constant symbols, as in puton(a, b), to designate objects. This paper overs the hard problem of how we find b in the world. Finding objects referred to by a plan can take an arbitrary amount of work. For example, an early instruction mentions "31/4" bolts." The two plan users spend more than a minute figuring out which these objects are among the pile of several dozen miscellaneous fasteners on the floor. This requires a lot of visual work, motor work (poking about among the pieces), and social coordinative work (their first take on which sort of bolts these are is incorrect and they have to negotiate a disagreement and subsequent agreement).

I believe that understanding situated instruction use—the use of instructions given in the course of on-going activity—is a prerequisite to understanding plan use. Plan use is analogous to instruction use, except that rather than being given instructions by an external agent at appropriate times, a plan user must keep track of where it is in the plan.

I am now constructing a system called Sonja2 which, like Pengi [2, 1], can engage in complex activity without use of Plans (or plans), but which can also use instructions when given them.

Sonja is based in part upon an empirical study of video tapes of human video game players who are given advice by a kibitzer, or by another player when two are playing cooperatively. As an illustration of the complexity of instruction use, consider one such as "Turn left!", which occurs repeatedly in my tapes. Most often, the player does not immediately turn left. Yet this is not an error, nor is the advice erroneous, nor does the speaker consider that she has been disobeyed. In fact, a viewer will generally agree that the instruction was carried out. Activity other than immediately turning left can count as fulfilling the instruction in many domain-specific ways.

- In some cases, the doorway through which it will be possible to turn has not yet been reached, so that turning left would run you into a wall. In these cases, turning left is deferred.
- When the point at which a turn is possible is reached, there may also be a doorway on the right, and there may be a monster hiding behind the door. If the monster will shoot her in the back when she turns left, the player will actually turn right, kill the monster, and then turn full around to proceed.
- In one case, the player passes the turn to pick up a valuable energy pod and then returns to comply with the instruction.
- Again, it may be that there is no left turn available, but there is an obviously correct right turn; in this case, the player may well figure that her interlocutor has simply said "left" for "right" in the heat of the moment, and turn right without comment.

The player is only likely to say "huh?" when she can make no sense at all of the instruction.

The complexity and flexibility of instruction use derives from the fact that people understand the instructions they are given, rather than blindly executing them. Sonja has a

2Sonja is pronounced with an English j, "Sahn-djuh," not a Continental one, "Sewn-ya," because it is named after a comicbook character.
variety of mechanisms for instruction understanding, some of which I will sketch here. [5] explains these mechanisms much more concretely.

Instructions are understood in terms of the user's existing ability to act autonomously. An instruction's user already knows how to engage in complex routines that accomplish various objectives in the domain, and instructions in effect refer to these routines. For instance, if I tell Sonja to "Get that amulet," it'll go ahead and do so, already knowing how to navigate around obstacles to get to it and how to kill pesky monsters that pop up along the way. My instruction may be useful because it tells Sonja that getting the amulet, rather than, for example, clearing out a nearby nest of monsters, is the right thing to be doing at the moment. Instructions play only a management role. I can no more tell Sonja to perform the primitive actions available to it than I can tell you to send a particular sequence of neural impulses to your arm muscles.

The instructions in my data are highly indexical (context-dependent).

- "Grab that one." This is said with no conversational context. It refers to some object ("that one") on the screen. Both participants know which object must be meant; only one makes sense in context.
- "Forty percent energy." In context, this is understood as "You have forty percent of your energy left." In a slightly different context, it would mean "I have forty percent of my energy left."
- "Are we gonna go through the time warp again?" "We" is traditionally an indexical; it's trivially understandable in this context, though. Definite noun phrases are traditionally analyzed as non-indexical, but in fact here determining the referent of "the time warp" is much more complicated than that of "we". Both players know which time warp is meant, though it is not visible and though it is a different time warp than the one they went through last time ("again"). It's the time warp that it might make sense to go through this time.

Understanding indexical instructions involves complex perceptual processing, which is carried out using Sonja's simulated visual system. (This visual system is similar to, but more sophisticated than, that of Pengi.) This processing often results in a new take on the situation. For example, if you hear "Knife, knife, knife!" and you haven't yet noticed any knives in the scene, you can run a visual routine for finding knives: look in all the likely places, do whatever it takes to recognize something as a knife. You might also remember where relevant knives are, or ask, or engage in an open-ended collection of other ways of finding one. Once you can see the knife, you will notice properties of it, for example that it is easily accessible, and act on them.

Sonja, unlike a planner, does not construct models of hypothetical future worlds. It does, however, take probable future circumstances into account; I call this projection. Projection is implemented by visual routines. Sonja looks to see what might happen next. It knows that a particular configuration of walls and monsters, which can be found by a pattern of visual operations, makes it probable that a particular course of activity will kill the monsters. These operations move visual markers about, color regions, and so forth. Thus, a sort of visual imagery is involved, one that is done with open eyes and superimposed on the actual visual image.
Projection is central to understanding many instructions. An instruction like “No”, “Turn left”, or “Don’t go below that line” will typically invoke a projection of the specified course of events, which will lead to a new understanding of the situation. If I say “Don’t throw the barrel that way—throw it this way,” you will project what would happen if you throw it “this way,” which you might not have otherwise done, and you’ll see (if I’m right) that that makes better sense than throwing it “that way.”

[5] describes further mechanisms for using instructions that involve a variety of linguistic constructs. I’ll omit explanations of these mechanisms, because they require a detailed understanding of the workings of Sonja’s architecture.

- **Indexical reference to objects.** “Get the knife” presumably refers to the one by Sonja’s feet, not the one held by the oncoming thug.
- **Indexical reference to activities.** “Take it out” means very different things when “it” refers to treasure in a chest than when “it” refers to a monster.
- Negative suggestions such as “Don’t go up there” and “Don’t pick the knife up.”
- **Informings.** “Dynamite’s on the ground!” doesn’t directly suggest motor actions. It does result in visual activity that typically results a new understanding of the situation (that there’s a stick of dynamite that’s about to explode) that in turn leads activity. Inference, in this case, is a visual process, not a linguistic or symbol-manipulating one.
- **Temporal expressions** such as “Turn it now” “When you come back, make sure you take the first right,” “Keep going, straight, then take the right at the end of the hall,” and “That dude should be coming right about now.”
- **Spatial expressions** such as “Go around that amulet,” “Take the right channel,” and “In there!”

[3] explores the ways in which an understanding of flexible instruction use may be extended to an understanding of plan use and construction. It argues that the prevailing view of Planning is untenable because of the computational complexity of plan construction, the difficulty of prediction in a world of uncertainty and change, the necessity of accommodating the stupidity of executives by specifying plans in impractical detail, and the largely unaddressed issue of relating plan texts to concrete situations in the world. It suggests an alternate way to view plan use, in which plans are in a number of respects similar to natural language instructions. We discuss some examples of human plan use we’ve studied which support this view.

**Acknowledgements**

Portions of this research were conducted at Xerox PARC; I’d like to thank Johan deKleer, Jeff Shrager, Brian Smith, Lucy Suchman, and others there for making my visit possible and for collaborative work while there.

**References**


An Embedded Reasoning and Planning System

Michael P. Georgeff
Australian Artificial Intelligence Centre
Melbourne, Australia

Abstract

In Artificial Intelligence, relatively little has been done in the design of embedded systems that can reason and plan in real time. The aim of this paper is to describe one approach to this problem.

1 Introduction

There are an enormous number of choices to be made in designing embedded reasoning systems that are capable of operating in real time. Despite an increasing amount of work in this and related areas (e.g., [1, 5, 6, 9, 10]), very little is known about the best choices to make for any given application domain.

One approach is to take a conventional expert system or theorem prover and allow its beliefs and goals to change dynamically in response to changes in the environment. Deduced beliefs that were no longer supported could be withdrawn, as could the subgoals of any goals that had either been achieved or, for some reason or other, withdrawn. But what guarantees would we have that such a system would ever reach any conclusions or realize any of its goals? Under what conditions would goals be "dropped?"

The primary task of any embedded system is simply to decide what is an appropriate action to execute next. But how can we design a system capable of doing this? In some cases, it might be possible to construct a system that would, given a particular pattern of input parameters, be able to make this decision simply on the basis of a table lookup. But in more complex domains it may be necessary to perform deductions of unbounded length, or to analyze possible plans of activity to determine the consequences of any chosen course of action.

If one does take plans seriously, how do we propose to use them? Do we fully elaborate the plans to the level of primitive actions prior to execution, or do we somehow interleave planning and execution? If we do construct a plan and take the first step in that plan, what do we do next? Do we completely reassess the situation and, if our beliefs have changed, replan from scratch? How committed, if at all, should we be to a plan once we have begun its execution? What if there is a possibility that there exist other, perhaps preferable, options — how much time do we or can we expend in exploring these?

We also need to decide how to perform the planning. Should we reason from first principles or should we incorporate a library of plans as part of our knowledge about the

---

1 This research has been made possible by a gift from the System Development Foundation and by the National Aeronautics and Space Administration, Ames Research Center, under Contract No. NAS2-12521.

2 The author is also a member of the Artificial Intelligence Center and the Center for the Study of Language and Information, SRI International, Menlo Park, California.
world? Should we allow plans to be executed in parallel? And, in doing all this, how do we guarantee a response time that is adequate for the operational environment in which the system is embedded?

I don't propose in this paper to investigate systematically these issues. Instead, I shall describe an embedded system that is capable of reasoning and planning in real time, and the principles upon which it is based.

2 Procedural Reasoning Systems

As the sample problem domain, we chose the task of malfunction handling for the Reaction Control System (RCS) of NASA's space shuttle. The shuttle contains three such systems, one forward and two aft. Each is a relatively complex propulsion system that is used to control the attitude of the shuttle. A part of one of the malfunction procedures from NASA's malfunction handling manuals is shown in Figure 1. These procedures can be viewed as unelaborated plans of action, and are designed to be executed in a complex and changing environment (e.g., see Step 6 of Block 1.4).

The reasoning and planning system that we applied to this problem is called a Procedural Reasoning System (PRS) [3, 2, 4]. PRS consists of a database containing current beliefs or facts about the world; a set of current goals to be realized; a set of plans (which, for historical reasons, are called Knowledge Areas or KAs) describing how certain sequences of actions and tests may be performed to achieve given goals or to react to particular situations; and an intention structure containing all currently active (executing) KAs. An interpreter (or inference mechanism) manipulates these components, selecting appropriate plans based on the system's beliefs and goals, placing those selected on the intention structure, and executing them.

The basic structure of PRS is shown in Figure 2. The system interacts with its environment (including other systems) through its database (which acquires new beliefs in response to changes in the environment) and through the actions that it performs as it executes its intentions.

2.1 The System Database

The contents of the PRS database may be viewed as representing the current beliefs of the system. Some of these beliefs are provided initially by the system user. Typically, these will include facts about static properties of the application domain, such as the structure of some subsystem or the physical laws that must be obeyed by certain mechanical components. Other beliefs are derived by PRS itself as it executes its KAs. These will typically be current observations about the world or conclusions derived by the system from these observations, and these may change over time. For example, at some times PRS may believe that the pressure of an oxidizer tank is within acceptable operating limits, at other times not. Updates to the database therefore necessitate the use of consistency maintenance techniques.

The database itself consists of a set of state descriptions describing what is believed to be true at the current instant. We use first-order predicate calculus for the state description language. State descriptions can contain variables (implicitly assumed to be univer-
Figure 1: Portion of an RCS Malfunction Procedure
sally quantified)\(^3\) and the usual logical connectives (\(\land, \lor, \text{ and } \lnot\), representing respectively conjunction, disjunction, and negation). A sample set of database beliefs for the RCS application is given below.

\[
\begin{align*}
\text{(type valve frcs-fu-tk-isol-12-valve)} \\
\text{(connects frcs-ox-tk-isol-12-valve frcs-ox-tk frcs-ox-tk-12-leg)} \\
\text{(value frcs-ox-manf-1-p-xdcr 245)}
\end{align*}
\]

The first fact represents type information: it states that the object named frcs-fu-tk-isol-12-valve is a valve. The next fact represents structural information, and the third fact describes a dynamic property that changes over time. In the application explored here, over 650 such facts are utilized for the forward RCS alone.

State descriptions that describe internal system states are called metalevel expressions. The basic metalevel predicates and functions are predefined by the system. For example, the metalevel expression \((\ast\text{goal } g)\) is true if \(g\) is a current goal of the system.\(^4\)

---

\(^3\)Logical variables are represented in PRS by names with a \$ prefix.

\(^4\)We adopt the convention that metalevel predicates that are prefixed with an asterisk denote attitudes (e.g., goals, beliefs, or intentions) of the system itself; attitudes of other agents are represented by predicates without the asterisk and take an additional argument (the name of the agent). For historical reasons, we also use the predicate \(*\text{fact}\) synonymously with \(*\text{belief}\)
2.2 Goals

Unlike most AI planning and reasoning systems, PRS goals represent desired behaviors of the system, rather than static world states that are to be eventually achieved. Hence goals are expressed as conditions over some interval of time (i.e., over some sequence of world states). A given action (or sequence of actions) is said to succeed in achieving a given goal if its execution results in a behavior that satisfies the goal description.

Goal behaviors can be described by applying a temporal operator to a state description. Three temporal operators are currently being used. The expression (!p), where p is some state description (possibly involving logical connectives), is true of a sequence of states if p is true of the last state in the sequence. That is, it denotes those behaviors that achieve p. Thus we might use the behavior description (!position frcs-fu-tk-isol-12-valve c1) to describe the goal to close the valve frcs-fu-tk-isol-12-valve.

The expression (?p) is true of a sequence of states if p is true of the first state in the sequence. That is, it can be considered to denote those behaviors that result from a successful test for p. Let's examine this notion more carefully. We call an action a test for a condition p if successful completion of the action guarantees that p were true just prior to beginning the action. For example, imagine an action a such that, if p were true just prior to executing a, a certain condition, q say, would be observed at its completion (e.g., a might be the action of dipping litmus paper in a solution, q the condition that the paper be red, and p the condition that the solution be acidic). Clearly, if q is observed, the sequence of states involved in performing a must have been such that p is true of the first of these states. On the other hand, if q is not observed, p may not be true of the first state in the sequence. Thus, a will count as a test for p if it signals success when q is observed and signals failure otherwise.

Finally, (#p) if true if p is preserved (is maintained invariant) throughout the sequence. Usually, when one establishes a goal of maintenance, it is intended that that goal be maintained until some condition becomes true. Thus, goals of maintenance usually appear in the form (#p \& !q), meaning to preserve p until q is achieved.

Behavior descriptions can be combined by means of the logical operators \& and \lor, representing, respectively, the conjunction and disjunction of the component expressions. Existentially quantified variables are represented by symbols prefixed with a $ sign.

As with state descriptions, behavior descriptions are not restricted to describing the external environment, but can also characterize the internal behavior of the system. Such behavior specifications are called metalevel behavior specifications. One important metalevel behavior is described by an expression of the form (=> p). This specifies a behavior that places the state description p in the system database. Another way of describing this behavior is (!(*belief p)).

2.3 Knowledge Areas

Knowledge about how to accomplish given goals or react to certain situations is represented in PRS by declarative procedure specifications called Knowledge Areas (KAs). Each KA consists of a body, which describes the steps of the procedure, and an invocation condition, which specifies under what situations the KA is useful. Together, the invocation condition
and body of a KA express a declarative fact about the results and utility of performing certain sequences of actions under certain conditions [2].

The body of a KA is represented as a graphic network and can be viewed as a plan or plan schema. Each arc of the network is labeled with a goal to be achieved by the system. As the KA is executed, the system is able to reason about and choose the most effective means for accomplishing those goals in the given circumstances. This allows PRS to react appropriately in a great range of situations and in the presence of degraded or uncertain information.

The invocation condition has two components: a triggering part and a context part. Both must be satisfied for the KA to be invoked. The triggering part is a logical expression describing the events that must occur for the KA to be executed. Usually, these consist of some change in system goals (in which case, the KA is invoked in a goal-directed fashion) or system beliefs (resulting in data-directed or reactive invocation), and may involve both. The context part is a logical expression specifying those conditions that must be true of the current state for the KA to be executed.

There are some properties of KAs that are crucial for the correct functioning of the system. For example, if a KA is invoked by the establishment of some new goal, it is important to know whether or not successful execution of the KA (i.e., reaching an end-node) realizes that goal. Other KA properties may be required by application-specific metalevel KAs, such as information on the likelihood of success of the KA or its average execution time. These KA properties can be stored in the system database along with the other facts that are pertinent to the application domain or, equivalently, in predefined slots in the KA structure itself.

A typical example of part of a KA is given in figure 3. It describes a procedure to isolate a leak in an RCS. The invocation part describes under what conditions this KA is useful. In this case, the KA is considered useful whenever the system acquires the goal to isolate a leak in an RCS ($p-sys$), provided the various type and structural facts given in the context part are true. (In determining the truth value of the invocation part, some of the variables appearing in the invocation part will be bound to specific identifiers. In this case, all the variables will be so bound.)

The KA body describes what to do if the KA is chosen for execution. Execution begins at the start node in the network, and proceeds by following arcs through the network. Execution completes if execution reaches a finish node (a node with no exiting arcs). If more than one arc emanates from a given node, any one of the arcs emanating from that node may be traversed. To traverse an arc, the system must either (1) determine from the database that the goal has already been achieved or (2) find a KA (procedure) that achieves the goal labelling that arc. For example, to traverse the arc emanating from the start node requires either that the system be already secured or that some KA for securing the RCS be found and successfully executed. If the system fails to traverse an arc emanating from some node, other arcs emanating from that node may be tried. If, however, the system fails to achieve any of the goals on arcs emanating from the node, the KA as a whole will fail. For example, since only one arc emanates from the start node in Figure 3, if all attempts to secure the RCS fail, this procedure for isolating a leak in the system will also fail. The full KA for this procedure consists of over 45 nodes and is the largest in the system.

Some of the important properties of the KA are represented in the slots on the left side
Figure 3: Portion of a KA for Leak Isolation
of the KA structure. For example, the goal achiever slot is set to T (true), representing the fact that, upon successfully completing this KA, the goal that triggered execution will have been achieved.

Some KAs have no bodies. These are the primitive KAs of the system and have associated with them some primitive action that is directly performable by the system. Clearly, execution of any KA must eventually reduce to the execution of sequences of primitive KAs.

The set of KAs in a PRS application system not only consists of procedural knowledge about a specific domain, but also includes metalevel KAs — that is, information about the manipulation of the beliefs, desires, and intentions of PRS itself. For example, typical metalevel KAs encode various methods for choosing among multiple applicable KAs, determining how to achieve a conjunction or disjunction of goals, and computing the amount of additional reasoning that can be undertaken, given the real-time constraints of the problem domain. In achieving this, these metalevel KAs make use of information about KAs that is contained in the system database or in the property slots of the KA structures.

In the application considered herein, about 70 object-level KAs were used together with about 15 metalevel KAs.

2.4 The Intention Structure

The intention structure contains all those tasks that the system has adopted (chosen) for execution, either immediately or at some later time. These adopted tasks are called intentions. A single intention consists of some top-level KA or goal, together with all the various [sub-] KAs that are currently being used as means to fulfilling the requirements of the top-level KA. However, at any given moment, the intention structure may contain a number of such intentions, some of which may be suspended or deferred, some of which may be waiting for certain conditions to hold prior to activation, and some of which may be metalevel intentions for deciding among various alternative courses of action.

For example, in handling a malfunction in the RCS the system might have, at some instant, three tasks (intentions) on the intention structure: one suspended while waiting for, say, the fuel-tank pressure to decrease below some designated threshold; another suspended after having just posted some goal that is to be accomplished (such as the securing of the RCS); and the third (a metalevel procedure) being executed to decide which way to accomplish that goal.

The order in which these intentions are executed is determined by metalevel KAs, which themselves must be adopted as intentions to become effective. This metalevel control allows reasoning in arbitrarily complex ways about the scheduling of these tasks, while retaining the ability to respond quickly and appropriately to new goals and beliefs.

2.5 The System Interpreter

The PRS interpreter runs the entire system. From a conceptual standpoint, it operates in a relatively simple way. At any particular time, certain goals are active in the system and certain beliefs are held in the system database. Given these extant goals and beliefs, a subset of KAs in the system will be applicable (i.e., relevant). One or more of these applicable KAs will then be chosen for execution by placing them on the intention structure.
In determining KA applicability, the interpreter will not automatically perform any deduction. Both beliefs and goals are matched by using unification only. This allows appropriate KAs to be selected very quickly and guarantees a certain degree of reactivity. If we allowed arbitrary deductions to be made, we could no longer furnish such a guarantee. However, PRS is always able to perform any deductions it chooses by invoking appropriate metalevel KAs. These metalevel KAs are themselves interruptible, so that the reactivity of the system is retained.

In the course of executing the chosen KA, new subgoals will be posted and new beliefs derived. Changes in the environment may also modify the existing beliefs of the system. When new goals are established, the interpreter checks to see if any new KAs are relevant, chooses one or more, places them on the intention structure, and begins executing it. Likewise, whenever a new belief is added to the database, the interpreter will perform appropriate consistency maintenance procedures and possibly activate other relevant KAs. During this process, various metalevel KAs may also be called upon to make choices among alternative paths of execution, choose among multiple applicable KAs, decide what intentions to execute next, decompose composite goals into achievable components, and make other decisions.

Unless some new belief or goal activates some new KA, PRS will try to fulfill any intentions it has previously decided upon. But if some important new fact or goal does become known, PRS will reassess its current intentions, and perhaps choose to work on something else. Thus, not all options that are considered by PRS arise as a result of means-end reasoning. Changes in the environment may lead to changes in the system's beliefs, which in turn may result in the consideration of new plans that are not means to any already intended end. PRS is therefore able to change its focus completely and pursue new goals when the situation warrants it. In many space operations, this happens quite frequently as emergencies of various degrees of severity occur in the process of handling other, less critical tasks. PRS can even alter its intentions regarding its own reasoning processes — for example, it may decide that, given the current situation, it has no time for further reasoning and so must act immediately.

2.6 Multiple Asynchronous Systems

In some applications, it is necessary to monitor and process many sources of information at the same time. Because of this, PRS was designed to allow several instantiations of the basic system to run in parallel. Each PRS instantiation has its own database, goals, and KA library, and operates asynchronously relative to other PRS instantiations. Communication among the various PRS instantiations is achieved by message passing. The messages are written into the database of the receiving PRS, which must then decide what to do, if anything, with the new information. As a rule, this decision is made by a fact-invoked KA in the receiving PRS, which responds upon receipt of the external message. In accordance with such factors as the reliability of the sender, the type of message, and its own beliefs, goals, and current intentions, the receiver determines what to do about the message — for example, to acquire a new belief, establish a new goal, or modify its intentions.

In this particular application, two instances of PRS were set up. One, called INTERFACE, handles most of the low-level sensor readings, controls effectors, and checks for faults in these components. The other, called misleadingly RCS, contains most of the high-level
malfunction procedures, much as they appear in the malfunction handling manuals for the shuttle.

As an example of the communication between these two systems, consider the case in which INTERFACE wishes to advise RCS that the valve `frcs-fu-tk-isol-12-valve` is closed. To do so, it would send RCS the message

\[(\text{asserted INTERFACE (position frcs-fu-tk-isol-12-valve cl))}\]

RCS could then choose what to do with this message, given appropriate KAs for responding to it. Note that the belief that the valve is closed is not directly inserted into the database of the recipient. In complex domains in which processes or agents may be unreliable, it is preferable to store the fact that some agent (INTERFACE in this case) has asserted something, without committing to believing that assertion. The recipient then has the opportunity (using appropriate KAs) to accept the asserted fact and add it to its database, to reject it as unreliable, or to combine it with other evidence in some other way.

For similar reasons, when some PRS agent, \(A\), wants another PRS agent, \(B\), to adopt some particular goal, the only way this can be effected is by passing \(B\) a message that requests that \(B\) establish the given goal. That is, \(A\) cannot directly establish a goal for \(B\); the best \(A\) can do is to get \(B\) to believe that \(A\) desires that \(B\) adopt the given goal. For example, if RCS wished the INTERFACE to close a valve `frcs-fu-tk-isol-12-valve`, RCS would send INTERFACE the message

\[(\text{requested RCS (!(position frcs-fu-tk-isol-12-valve cl)))}\]

### 3 Critical Features of the System

In this section, we review in greater depth some of the features of PRS that are critical to its successful operation as an embedded real-time reasoning system.

#### 3.1 Invocation of KAs

The applicability of a KA is specified by means of its associated invocation condition. The invocation condition consists of two parts:

1. A logical expression specifying some pattern of *initiating events*
2. A logical expression specifying the *context* of invocation.

An initiating event is the acquisition of a new belief or the establishment of a new goal or intention. For example, if the initiating condition of a given KA were \((\text{*fact (position frcs-fu-tk-isol-12-valve cl)})\), the KA would be invoked whenever the system acquired the belief that the `frcs-fu-tk-isol-12-valve` was closed. Thus, it is the change in the system's beliefs that triggers the KA. Similarly, if the initiating condition were \((\text{*goal (!(secured frcs)})\), the KA would be invoked upon the system acquiring that goal.

The context specifies additional conditions that must be true for the KA to be invoked, once it has been triggered by some initiating event. Thus, if the context of a KA were
An Embedded Reasoning and Planning System

(*fact (position frcs-fu-he-tk-A-valve cl)), the KA could only be invoked if the system believed that the frcs-fu-he-tk-A-valve were closed.

Let's say that $I$ is the initiating condition for a given KA and $C$ is its invocation context. Furthermore, let us assume that at moment $t$ the system’s state (i.e., its beliefs, goals, and intentions) is $S$ and at the next moment $t'$ the state is $S'$. Then the KA will be invoked if and only if

$$(S' - S) \vdash I \text{ and } S' \vdash C$$

where $x \vdash y$ is true if the truth of expression $y$ can be directly deduced upon unifying each of $y$'s components with the components of $x$.

It is important to note that the only way a KA can be invoked is by the occurrence of some initiating event or change in the system's beliefs, goals, or intentions. Should such an event trigger a KA, and that KA not be immediately adopted as an intention, then that event cannot re-trigger the KA on subsequent cycles. (Of course, some subsequent occurrence of the same type of event could trigger the KA afresh.) What are the consequences of this?

Imagine that two different alarms, alarm-1 and alarm-2, are sounded at once, each triggering a different KA (say, KA-1 and KA-2, respectively). As the system has more than one applicable KA from which to choose, one or more meta-KAs will be invoked to determine what to do. If, as a result of the metalevel processing, one of these KAs is chosen for adoption (say KA-1), the other invoked KA will simply be discarded. In this case, although the system would “know” that alarm-2 had sounded (it would be in the system database), it would take no action with respect to that alarm. This need not mean that it would never take any action in response to that alarm, although usually this would be so. For example, there may be some KA that is invoked every now and then to check on things that have been left unattended. Such a KA could notice that alarm-2 was on, that nothing had been done about it, and then, indirectly, invoke KA-2 to respond to it.

The other possibility, dependent on the metalevel processing, is that both KAs are adopted as intentions. In this case, both alarms will be attended to. The order in which KA-1 and KA-2 are evaluated will depend on the ordering of their corresponding intentions on the intention structure; for example, both could be pursued in parallel, or one could be deferred until the other was finished. We say more about the intention structure below.

In summary, the system must respond to events — and form intentions appropriate to those events — as they occur. As changes to elements of the system’s state are, in most cases, rare in comparison to the total number of state elements, system efficiency (and response time) is thereby substantially enhanced. On the other hand, as at any time the current state of the system is determined entirely by its history of changes, we loose no deductive capabilities by this more restricted form of KA activation.

---

5This assumes an empty initial state. But if the initial state, $i$, say, is not empty, we can transform the problem into an equivalent one that has an empty initial state and that begins with an event that directly brings about $i$. 
3.2 Goals and Intentions

Goals are of two kinds: intrinsic goals and operational goals. Intrinsic goals are those that the system acquires directly from some specific sources. The only source of intrinsic goals for the shuttle implementation is the user (astronaut or mission controller), who has the power to impose arbitrary goals on the system. In other implementations, one could envisage other sources capable of imposing such goals. For example, an autonomous system being controlled by PRS might allow special sensors to generate intrinsic goals, such as to recharge batteries when battery power decreases below a certain threshold, or to escape in the presence of an overwhelming foe.

Operational goals are those that the system acquires in attempting to fulfill some intention. That is, operational goals are those that are established during execution of a KA that has been previously adopted as an intention. Thus, operational goals are always means to some end, although that end may not always be explicit. For example, when a KA is invoked by the occurrence of a new system goal, the goals in the body KA will be means toward achieving that goal. On the other hand, when a KA is invoked by the occurrence of a new system belief, the goals appearing in the KA are means of responding to the acquisition of that belief, but the end result the system is aiming to achieve is left unspecified. For example, upon the activation of some alarm, a KA that diagnoses the fault and corrects it might be invoked. The reason for invoking this KA (presumably, to maintain the integrity of the spacecraft) is not specified anywhere, yet each of the goals occurring in this KA are means to that (unspecified) end.

A KA invoked by the acquisition of some intrinsic goal or by some change in system beliefs can give rise, if adopted by the system, to a new system intention. In this case, the initiating event is called the purpose of the intention and the KA so invoked is called the head of the intention. As the head KA is executed, it will give rise to various operational goals. The KAs that respond to these operational goals will form part of the originating intention, together with any KAs that these KAs in turn invoke, and so on. Thus, a single intention consists of a head KA (invoked either by an intrinsic goal or new belief), together with the various other KAs that are utilized in attempting to execute the head KA.

3.3 The Establishment and Removal of Goals

Intrinsic goals are established by specific external sources and, as with beliefs, must be responded to immediately if at all. As with beliefs, they will be remembered, even if no intention is formed to accomplish them. They will be removed if explicitly requested (by the external source that established them) or if the system comes to believe that they are accomplished (either through its own efforts or those of some other agent).

Operational goals are established by the attempted execution of some KA that is part of some intention of the system. Should the system attempt to achieve an operational goal and fail, that goal will be reestablished, and another attempt made to achieve it. This will continue until the system comes to believe either that the goal is accomplished (through its own efforts or those of some other agent) or that the goal cannot be readily accomplished. Once this state is reached, the goal will be dropped.

---

*Some philosophers would call these desires.*
This raises two related issues: what other attempts are made to achieve the goal and how does the system come to believe a goal cannot be readily accomplished? We shall answer the latter question first. There are two ways the system can come to believe that a goal cannot be accomplished. One way is to deduce it, but this will depend on the provision of appropriate metalevel KAs for performing the deduction and upon their being invoked. The other way is simply to fail in all attempts at achieving it.

This brings us to the former question — what attempts does the system make to achieve a given goal? The system currently tries, exactly once, every possible KA instance that can possibly achieve the goal. It does not ask that previously achieved goals be re-achieved (in some other way), nor does it try the same KA instance more than once. In this sense, it is equivalent to a “fast-backtrack” parser. There is good reason for at least the first of these choices.

Unlike planning some course of action or parsing sentences, once some goal has been achieved there is no point in trying other ways to achieve it, even if these may benefit attempts to accomplish subsequent goals. Or more accurately, there is no reason to constrain oneself to look only at other ways to achieve this goal. For example, consider that we wish to accomplish some goal \( f \) and then some goal \( g \). Let’s say that, in attaining \( f \), some condition \( p \) is made true (perhaps as a side effect of the actions taken to achieve \( f \)). Unfortunately, with \( p \) true, it proves impossible to accomplish \( g \) with the KAs at our disposal (because, let’s imagine, \( \neg p \) is part of the context of the invocation part of all KAs that accomplish \( g \)). Now, a planner or full-backtrack parser might backtrack and reattempt to achieve \( f \) some other way, in the hope that it would succeed without incurring the troublesome side-effect. But PRS has actually achieved \( f \) — and also \( p \) — so that there is no use trying other methods to re-achieve that goal! Instead, one should now try to achieve \( \neg p \), preferably while maintaining \( f \).

The remaining issue concerns the number of tries we make of a single KA instance to achieve a given goal. It is, of course, quite possible that, where the first try does not succeed, the next will, even if we carry out exactly the same actions as we did the first time. However, to determine if retrying could succeed would, in most practical cases, require knowledge of the state of the world that goes well beyond that available to the system. Assuming the lack of such knowledge, we took the reasonable course of allowing at most one try for each KA instance. This clearly affects the capabilities of the system — in some cases, it may be that trying twice would succeed where trying once does not.

Once all attempts at achieving a given goal have in this way been exhausted, it is still possible for some metalevel KA to respond to this failure and invoke yet other means to achieve the goal. For example, a meta-KA could invoke certain deductive machinery, or could decide to retry some KA instances that, although having been tried once, appear (for some reason known to the meta-KA) to be worth trying again.

### 3.4 Conditional Intentions

Sometimes it is desirable to form the intention that, when some condition occurs, some action should be performed. We call these conditional intentions. Typically, we may get to a certain point in a procedure (KA) where we want to suspend execution until some condition is found to be satisfied.
For example, after having switched from a faulty regulator to a properly functioning one, we may want to switch control of the regulator valve over to the on-board computers (GPC). However, this cannot be done until the pressure in the system drops below 312 psi, as otherwise the computers will assume a failure in the new regulator and shut it off. Thus, one would like to be able to suspend further execution of the procedure until the pressure dropped below 312 psi, after which execution can proceed.

In PRS, conditional intentions are created by means of certain metalevel KAs. Their effect is to suspend execution of some given intention, which then remains in a dormant state until the appropriate activation condition (in the above case, a pressure below 312 psi) is satisfied (see Section 3.5).

Conditional intentions arise in many other situations; indeed, it would be hard to imagine an effective real-time reasoning system that did not require them. For example, conditional intentions are necessary in the following cases:

- When certain events may be indicative of a problem but allowance has to be made for transient effects; one often has to wait until any transients die away.
- When sampling parameters over time, such as when a rate of change is required.
- When waiting for a reply from a request to another system or a user; it is often desirable to suspend the process making the request until an answer is received, and if not received within a reasonable time to adopt some other course of action.

Of course, it is essential that, while waiting for an activation condition to become true, the system continues to monitor the environment and continues to execute other intentions as required. Thus, for example, while waiting for the pressure to drop below 312 psi, the RCS system keeps monitoring the status of the shuttle, responds to any observed changes in the situation (such as a failure in some other component), and performs any other tasks demanded of it.

3.5 Intention Types

One of the most interesting features of PRS is the manipulation of intentions within the intention structure. The purpose of this and the following two sections is to explain the mechanisms for manipulating the intention structure.

The intention structure contains those intentions that have been adopted by the system. The system commits to these intentions; that is, it will “try its best” to achieve them and plan its other activities in accord with them. The set of intentions comprising the intention structure form a partial ordering such as shown in Figure 4. Those that are roots of this graph (i.e., have no predecessors) are candidates for becoming the current intention. The current intention is the one that is currently being executed, and is surrounded by two small arrows in the figure. The directional arcs shown in the figure represent precedence constraints on the intentions. That is, the intention earlier in the ordering (as defined by the arcs in the partial order) must be finished (and thus disappear from the intention structure) before intentions appearing later in the ordering. This precedence relationship between intentions enables the system to establish priorities and other relationships between intentions.
An intention can be in one of three possible states:

**Normal**: This is the most common state. If such an intention is a root of the graph then it can be activated. Otherwise, it must wait until all its predecessors are finished.

**Sleeping**: A sleeping (or conditional) intention is one whose execution is suspended, awaiting some condition to be satisfied. To enter this state, an appropriate metalevel KA must be utilized. This metalevel KA can be invoked by the intention that is to be suspended or by some other intention outside it. The sleeping state implies the presence of an activation condition for the intention. This activation condition is a logical expression evaluable in the environment of the intention. As long as it is false, the intention is kept sleeping. As soon as it becomes true, the intention is put into a woken state.

**Woken**: As its name implies, an intention is in a woken state if has been “awakened” after having been in a sleeping state. The woken state is exactly the same as the normal state, except that if there is more than one possible current intention, the system will prefer the most recently woken one. As a result, the system will tend to activate intentions which have just been awakened. As soon as such an intention has been activated, the woken state is exited and the intention returns to the normal state.

3.6 Establishing Intentions — The System Interpreter

Intentions are established in two ways: (1) by the system interpreter; and (2) by particular metalevel KAs. The operation of the interpreter in establishing intentions is worth examining. The main problem to be solved is that, on any cycle, a number of KAs may be applicable. It is thus necessary to decide what to do with these applicable KAs — in particular, how many (if any) to establish as intentions and how to insert those so chosen into the intention structure. The notion of metalevel KAs was introduced to provide maximum flexibility in making these decisions.
But we have to find a mechanism for bringing these metalevel KAs to bear at the appropriate time. The way we chose to do this was to include in the invocation part of the metalevel KA some condition on the number or kind of object-level KAs that are applicable at each cycle. For example, a particular metalevel KA might be invoked on the basis of there simply being more than one applicable object-level KA at the current moment.

To enable this scheme to work, we first have to determine which object-level KAs are applicable on each cycle. This information becomes a new system belief. In particular, on each cycle, the system acquires the belief \((\text{soak } x)\), where \(x\) is the list of object-level KAs that are currently applicable. We then determine whether or not the acquisition of this new belief \((\text{soak } x)\) triggers any new metalevel KAs. If it does, the system acquires a new belief about the applicability of these metalevel KAs. In fact, it does so simply by updating the belief \((\text{soak } x)\) so that the list \(x\) now contains exactly those metalevel KAs that are now applicable. (The previous belief about applicable object-level KAs is removed from the database and so, in a sense, is forgotten. However, it is captured in the variable bindings of the invoked metalevel KAs.)

As PRS places no restrictions upon the invocation conditions of metalevel KAs, it is quite possible that more than one metalevel KA will be invoked at this stage. If this happens, we will now be left with the problem of deciding which of these metalevel KAs to invoke. There are a number of possible solutions to this problem. One would be simply to select one of the metalevel KAs at random, on the assumption that all are equally good at making the decision about which object-level KAs should be invoked. Another alternative would be to preassign priorities to the metalevel KAs and to invoke the one with the highest priority. However, in keeping with our aim of providing maximum flexibility, the solution we chose is to allow further metalevel KAs to operate on these lower-level metaKAs in the same way that the lower-level metaKAs operated on the object-level KAs.

The process of invoking metalevel KAs is thus continued until no further KAs are triggered. At that point, there may still be a set of applicable KAs from which to choose. It is then, and only then (i.e., only after failing to find any more applicable metalevel KAs), that we select one of these KAs at random.

Thus it is seen that, when more than one KA is applicable, and in the absence of any information about what is best to do, the system interpreter defaults to selecting one of these KAs at random. With no metalevel KAs, the system would thus randomly select one of the applicable object-level KAs. However, one usually provides metalevel KAs to help make an informed choice about the object level KAs. The applicable metalevel KAs themselves are subject to the same default action (i.e., one will be randomly selected) unless there are yet other metalevel KAs available to make a choice among them. In the end, at some level in the meta-hierarchy, the default action will be taken (of course, there may, at that level, be only one KA to choose).

Once selected, the chosen KAs must be inserted into the intention structure. If a selected KA arose due to an external goal or a fact, it will be inserted into the intention structure as a new intention at the root of the structure. For example, this will be the case for

---

7 Currently, it is left to the user to ensure that the triggering of higher and higher levels of metaKAs terminates in a bounded time. This is not difficult for the user to achieve. For example, one could assign each metalevel KA to one of a finite set of fixed levels (types), and enforce the condition that KAs only operate on others at the level below them in this finite hierarchy.
any metalevel KA that is invoked to decide among some set of applicable lower-level KAs. Otherwise, the KA instance must have arisen as a result of some subgoal of some existing intention, and will be “grown” (i.e., attached) as a subKA of that intention.

### 3.7 Manipulating Intentions — Metalevel KAs

The only other way to establish intentions is by invoking special metalevel KAs. These provide a variety of options for inserting KAs into the intention structure.

There are four different ways in which an intention can be inserted into the intention structure:

1. As a subKA of the current intention.
2. As a new intention at the root of the intention structure; i.e., prior to every other intention, including the current intention if it has not finished execution.
3. As a new intention that has precedence over some set of existing intentions.
4. As a new intention that will be initiated only after some set of existing intentions have been executed.

For example, consider the metalevel KA selector-2 in Figure 5. The goals (![(intend ...)] and (![(intend-all-safety-before ...)]) are satisfied by metalevel
KAs that eventually establish intentions, each in its own way. In particular, the KA that responds to the first goal inserts the KA instance directly after the current intention, whereas the KA that responds to the second goal gives those object-level KAs that have the property of being "safety handlers" priority over those that are not.

The initiating events for some of the more important metalevel KAs are listed below. Some of these KAs are used to create new intentions, others to modify the intention structure itself, and yet others to modify the status of existing intentions.

Intend the KA instance $x$:
Initiating Event: (*goal (! (intend $x$)))

Intend all KA instances in the set $x$, but give priority to "safety handlers":
Initiating Event: (*goal (! (intend-all-safety-before $x$)))

Make the current intention sleep for $t$ seconds:
Initiating Event: (*goal (! (sleep $t$)))

Awake a sleeping intention:
Initiating Event: (and (*fact (wake-up $x$)) (*fact (> (length $x$) 0)))

Wait until the condition $c$ is true (the current intention is automatically put to sleep):
Initiating Event: (*goal (! (wait-until $c$)))

3.8 Commitment to Intentions

PRS commits to its intentions. That is, unless some particular metalevel KA intervenes, PRS will perform its means-ends reasoning and other planning in the context of its existing intentions. For example, consider that PRS has adopted the intention of achieving a goal $g$ by accomplishing the subgoals $g_1$, $g_2$, and $g_3$, in that order. In the process of determining how to accomplish these subgoals, the system will not reconsider other means of achieving $g$. That is, it is committed to achieving $g$ by doing $g_1$, $g_2$, and $g_3$, even if circumstances have so changed that there is now a better way to achieve $g$ than the one chosen. The gain here is in reducing decision time — in highly dynamic domains it is not possible to continually reassess one's plans of action. What makes the approach workable is that the basis upon which one chooses a particular plan of action is more often correct than not.

Of course, the system is not committed to its intentions forever. For example, as discussed in Section 3.3, if PRS determines that it cannot achieve $g$ by doing $g_1$, $g_2$, and $g_3$, it will drop that plan and look for some other means of achieving $g$. Alternatively, it may remember the basis for choosing one plan over another, and utilize appropriate metalevel KAs to modify its intentions if support for that decision is subsequently found to be lacking.

It is not only in means-ends reasoning that PRS's commitment to its existing intentions is important. For example, in tackling some new task, it is often desirable that the means or time chosen for accomplishing that task take account of one's existing intentions towards the fulfillment of other tasks. In the RCS application, this happens, for example, when the PRS instance INTERFACE receives a request for a pressure reading when it is in the process of evaluating the status of a suspected faulty transducer. In this case, INTERFACE will either defer or suspend attention to that request (possibly advising the requester) until it has completed its evaluation of the transducer.
3.9 Supporting Goals and Beliefs

Some embedded reasoning systems are designed so that any given course of action is terminated whenever the beliefs or goals that brought about that action cease to be true. In this section, we shall consider this approach and how it compares to the one adopted in PRS.

Suppose that a certain alarm sounds, which in turn invokes some procedure to rectify the fault. In some cases, the alarm will only be turned off when the problem is rectified. Thus, it appears sensible that, should ever the alarm cease to sound, any procedure aimed toward that end be terminated (but more about this below). On the other hand, there are many cases where the ceasing of an alarm does not mean that the corresponding fault has been fixed: the alarm may only sound for a fixed interval of time, or it may be deliberately turned off to relieve the operator of the annoying noise. Thus, it is clearly bad policy to always terminate a procedure simply on the grounds that the beliefs that caused its activation no longer hold. Of course, if the initiating belief is that the alarm sounded at a particular time, and this belief is found to be false, it is usually sensible to consider termination of the procedure.

Should the goals that constitute the purpose of some course of action cease to exist, the rationale for continuing with the action disappears also. However, this need not mean that one should simply cease doing what one was doing and get on with other activities. Indeed, in most cases one would need to “clean up” after the action, and in some cases one may even be desire to see things through to the end (as, for example, when one has fired the jets to force reentry of the shuttle).

Thus, depending on circumstances, one has to reason carefully about the early termination of procedures, even when there is due cause for termination. PRS offers no special techniques for handling these problems, but instead provides the hooks by which such reasoning can be performed at the appropriate time.

PRS will always recognize goal failures. Without any meta-KAs to respond to such failures, the system will simply try some other procedure to achieve the failed goal, eventually terminating once all avenues of attack have been explored. On the other hand, should one wish some special action to occur on goal failure, it is straightforward to construct appropriate metalevel KAs to perform the necessary corrective actions.

PRS currently has no notion of supporting beliefs for given courses of action. Thus, if it is desired that a certain procedure be terminated should some belief be no longer held, a special KA must be set up to look for that condition during the execution of the procedure. While this is not difficult to do, it would be useful to provide generic mechanisms to assist in this task. We plan to address this and related issues in our future work on PRS.

3.10 Guaranteed Reactivity

Definitions of real-time systems revolve around the notion of response time. For example, Marsh and Greenwood [7] define a real-time system as one that is “predictably fast enough for use by the process being serviced” [my italics], and O’Reilly and Cromarty [8] require that “there is a strict time limit by which the system must have produced a response, regardless of the algorithm employed.” Response time is the time the system takes to recognize and respond to an external event. This measure is the most important in real-time applications; if events are not handled in a timely fashion, the operation can go out
of control. Amazingly, almost none of the existing “real-time” systems are guaranteed to respond within a bounded interval of time [6].

In contrast, PRS has been designed to operate under a well-defined measure of reactivity. Because the interpreter continuously attempts to match KAs with any newly acquired beliefs or goals, the system is able to notice newly applicable KAs after every primitive action it takes.

To estimate the bound on reaction time, let $p$ be an upper bound on the execution times of the primitive actions that the system is capable of performing. Let’s also assume that $n$ is an upper bound on the number of events that can occur in unit time, and that the PRS interpreter takes at most time $t$ to select the set of KAs applicable to each event occurrence.\(^8\) Then it is not difficult to show that PRS has a guaranteed reactivity delay of at most $\Delta_R = p + (np/(1-nt))t$, where we assume that $t < 1/n$.

This means that, provided the number of events that occur in unit time is less than $1/t$, PRS will notice every event that occurs [that is capable of triggering some KA] and is guaranteed to do so within a time interval $\Delta_R$. In the current implementation, the values of $p$ and $t$ are less than 0.1 seconds, giving a reactivity delay of at most 1 second for an event rate of 5 events per second.

Because metalevel procedures are treated just like any other, they too are subject to being interrupted after every primitive metalevel action they take. Thus, reactivity is guaranteed even when the system is choosing between alternative courses of action or performing deductions of arbitrary complexity.

Having reacted to some event, it is necessary for the system to respond to this event by performing some appropriate action. As the system can be performing other tasks at the time the critical event is observed, a choice has to be made concerning the possible termination or suspension of those tasks while the critical event is handled. Provided the information regarding event importance and required response time is represented explicitly in the system database (and thus does not require deductions of arbitrary length), it is not difficult to construct a metalevel KA that will preempt all other tasks and begin execution of an appropriate handler for the critical event.

Let’s assume that the user provides only one metalevel KA for giving priority to critical event handlers, and that this metalevel KA itself has top priority. Assume also that the upper bound on the execution time for such a metalevel KA is $m$ and the upper bound for the actual event handler is $e$. Then the maximum upper bound for noticing and responding to a critical event is equal to the $\Delta_R + m + e$. The parameter $m$ can be made reasonably small; in the current implementation, $m$ is less than half a second. The parameter $e$ is clearly dependent on the application and is fixed irrespective of the software system employed. Of course, if it is necessary to respond to more than one critical event at a time, we have to include the time taken to execute each of the corresponding event handlers.

### 3.11 Planning or Not?

There has always been some confusion in the literature about the notion of planning, especially with respect to the kind of practical reasoning that PRS performs.

\(^8\)As selection of KAs does not involve any general deduction beyond unification and evaluation of a boolean expression, an upper bound does indeed exist.
In the AI literature, planning is viewed as the generation of a sequence of actions to achieve some given goal. The classical approach to this problem is to simulate the effects of performing the actions so as to ensure that their execution does indeed achieve the required goal. All this planning is done, in most cases, prior to performing any physical action in the actual world.

It is quite straightforward to run PRS in this way — the primitive actions performed by the system are decoupled from the actual world, and the KAs simply become the "operators" of classical planning systems. Thus, the system simulates execution of the KAs, and its database reflects beliefs about the state of the world as it would be had those KAs actually been executed. As the system explores all possible sequences of activity that could possibly lead to the goal condition being achieved, it will find a plan if one exists. In this sense, PRS is capable of planning in the classical tradition, albeit not very efficiently.

The kind of planning discussed above all takes place prior to performing any actions in the actual world. However, it is also possible to form plans during the course of performing some task. Assume one has some goal to achieve, say, and a variety of ways to achieve that goal. Let's say that there are two options: achieve $g_1$ followed by $g_2$; or achieve $f_1$ followed by $f_2$. Now one could choose arbitrarily between these options, or one could engage in some level of planning to determine which was the best course of action in the given circumstances. This kind of planning may involve simulating the possible outcomes of each approach by fully elaborating these options as done in classical planning. However, one could alternatively select from a great variety of other techniques. For example, the choice could be based on the expected time to complete the actions, or the likelihood of success of the plans as gained through experience. In any case, simply making the choice as to which course of action to pursue, no matter how one does it, constitutes forming a plan to achieve the goal $g$. Having chosen one of these courses of action (or, indeed, none or both), one repeats the process. For example, if the course involving $g_1$ and $g_2$ were chosen, and one had various ways of achieving $g_1$, then it would be necessary to plan how best to achieve that subgoal, and so on.

This is exactly the way PRS operates. The method of choosing between alternative courses of action is embedded in the metalevel KAs of the system and thus, in essence, the particular approach to forming plans is not hard-wired into the system. To the extent that the choice is made arbitrarily, one may wish to avoid calling this process "planning." But where it is based on any information at all, no matter how meagre, the determination of an appropriate course of action is indeed planning.

In the RCS example, the system decides between different courses of action depending on how the KA was invoked and what sort of priority it has. This is clearly quite a weak form of planning, and more complex meta-KAs — taking time availability, costs, and benefits into account — would improve system reliability.

Of course, it is important to determine exactly what algorithms (metalevel KAs) are needed for effective planning. The RCS problem, as we said above, uses a very simple form of planning, which, in itself, is probably not of much interest. However, what is of interest

---

9In fact, this depends on how the metalevel KAs are written. In particular, one has to ensure that all possible interleavings of any conjunctive goals are explored.

10Such as when both approaches appear unlikely to achieve the goal.

11Such as courting both Jane and Mary, in the hope that one of them will eventually marry you.
is just how weak the planning component can be when we have a wealth of experience (a rich set of object-level KAs) to assist us.

4 Conclusions

The system described above was implemented on a Symbolics 3600 LISP machine and has been used to detect and recover from most of the possible malfunctions of the RCS, including sensor faults, leaking components, and regulator and jet failures. This was accomplished by using multiple communicating instantiations of PRS and a simulator for providing real-time input to the system. The experiment provided a severe and positive test of the system’s ability to coordinate various plans of action, modify intentions appropriately, and shift its focus of attention. In addition, PRS met every criterion outlined by Laffey et al. [6] for evaluating real-time reasoning systems: high performance, guaranteed response, temporal reasoning capabilities, support for asynchronous inputs, interrupt handling, continuous operation, handling of noisy (possibly inaccurate) data, and shift of focus of attention.

The features of PRS that, we believe, contributed most to its success at this task were (1) its partial planning strategy, (2) its reactivity, (3) its use of procedural knowledge, and (4) its metalevel (reflective) capabilities. At any time, the plans the system is intending to execute (i.e., the selected KAs) are both partial and hierarchical — that is, while certain general goals have been decided upon, the specific means for achieving these ends have been left open for future deliberation. By finding and executing relevant procedures only when needed and only when sufficient information is available for making prudent decisions, the system stands a better chance of achieving its goals under real-time constraints.

The wealth of procedural knowledge possessed by the system is also critical in allowing the system to operate effectively in real-time and to perform a variety of very complex tasks. In particular, the powerful control constructs that can be represented by KAs (such as conditionals and loops) were essential to realizing the malfunction handling procedures used for the shuttle, and allowed low-level system functions to be implemented in the same formalism (i.e., without resorting to LISP or some other programming language).

PRS also makes it possible to have a large number of diverse KAs available for achieving a goal. Each may vary in its ability to accomplish a goal, as well as in its applicability in particular situations. Thus, if there is insufficient information about a given situation to allow one KA to be used, another (perhaps less reliable) might be available instead. Parallelism and reactivity also help in providing robustness. For example, if one PRS instantiation were busy diagnosing some system fault, other instantiations could remain active, monitoring environmental changes, keeping the spacecraft in a stable and safe configuration.

The metalevel reasoning capabilities of PRS are particularly important in managing the application of the various KAs in different situations. Such capabilities can be critical in deciding how best to meet the real-time constraints of a domain. In particular, the combination of a rich intention structure — supporting multiple active, suspended, and conditional intentions — and the metalevel scheduling capabilities appear to be essential components of complex real-time applications.

There are many issues that need further study. In particular, it is important to re-examine the semantics of KAs to allow for external events to occur during their execution and for other KAs to interrupt processing. The truth maintenance techniques used for
handling the database need further study and extension, and goals of maintenance must be considered. It is necessary to examine more complex scheduling schemes and, finally, to expand the application domain to handle fully and properly the malfunction procedures of the RCS.

Acknowledgments

Amy Lansky and Felix Ingrand assisted greatly in the design of PRS and in working out the theoretical foundations of the system. Pierre Bessiere, Lorna Shinkle, Joshua Singer, and Mabry Tyson helped in the development of PRS and extended the implementation as needed. Dave Wilkins built the simulator and provided valuable advice. The project has also benefited greatly from our participation in and interactions with members of the Rational Agency Group (RATAG) at the Center for the Study of Language and Information, especially Michael Bratman, Phil Cohen, David Israel, Martha Pollack, and others mentioned above.

References


On the Value of Goals

Thomas Dean 1
Department of Computer Science
Brown University

Michael P. Wellman
AFWAL/TXI

Abstract

Planning systems make decisions; they routinely choose from among sets of possible courses of action. This paper examines the question "What does it mean for a machine to make a good decision?" We begin by considering an agent with unlimited computational resources. The use of goals in planning research is contrasted with the use of value functions in the decision sciences. We then consider the question "What would it mean for an embedded planning system with finite computational resources to make a good decision?" In exploring some of the problems involved in providing a normative basis for resource-bounded decision making, we consider the difference between knowing and deciding, and examine what sorts of things are worth knowing. Finally, we consider a new model for analyzing planning systems that allows us to speak about the internal states of a machine embedded in a complex environment.

5 Introduction

Planning systems are designed to make decisions about how to act in a given set of circumstances. Decisions correspond to allocations of resources controlled by the planner. Presumably, we are interested in building systems that make good decisions, but researchers in planning have been somewhat evasive about what constitutes a good decision.

If one accepts the axioms of probability and utility, then decision theory provides a normative basis for decision making (i.e., it supplies criteria for consistency among beliefs and preferences that any "rational" decision maker would agree with). There is one problem with decision theory, however: it abstracts from the fact that all decision makers have finite computational resources at their disposal. The act of deciding itself constitutes an irrevocable allocation of these resources.

Researchers designing planning systems have, for the most part, ignored decision theory, concentrating instead on a particular sort of problem solving in which the planner takes a specification of some desirable property, a goal, and tries to generate a sequence of actions guaranteed to achieve a state of affairs satisfying the goal. As long as the resulting state satisfies the goal, the planner's efforts are judged a success.

As the field has matured, research in planning has begun to deal with problems requiring complex tradeoffs (e.g., problems in robotics and factory automation)—problems of the sort that the decision sciences have been concerned with all along, at least from a theoretical

---

1This work was supported in part by the National Science Foundation under grant IRI-8612644 and by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Air Force Office of Scientific Research under Contract No. F49620-88-C-0132.
standpoint. The decision sciences, on the other hand, in attempting to provide computational solutions to such problems, have been forced to deal with the disparity between the time and other resources available for decision making and those required to determine the decision dictated by the normative theory. In the following contrived dialogue, a researcher in the decision sciences (DS) and a researcher in artificial intelligence (AI) discuss how their respective disciplines approach the problems inherent in real-world decision making.

DS: I am interested in how planning researchers deal with the issue of evaluating and comparing alternative solutions to planning problems. In the decision sciences, we employ what is referred to as a value function. Value functions are used to determine the desirability of a given outcome. A decision maker chooses the available course of action with greatest value. Is there anything corresponding to the value function in planning theory?

AI: Some planning researchers employ value functions, but a much more prevalent practice is to employ the notion of a goal. A goal corresponds to a property of the environment which, if true, has some value to the agent. Perhaps you would like to interpret this as meaning that states where the goal is satisfied are preferred to states where it is not.

DS: Could an agent have conflicting or even contradictory goals? For example, could it be said that an agent has a goal to stay in bed sleeping and a goal to be at the office working?

AI: An agent could certainly have a conjunctive goal such that achieving some of the conjuncts makes others more difficult to achieve. Indeed, a great deal of research effort has been expended on strategies for resolving interactions among conflicting plans. Nevertheless, if the conjunctive goal is logically inconsistent, the planner will surely fail. For the most part, planning techniques tacitly assume that, if an agent is given a set of goals, then there exists a plan to achieve them.

DS: This assumption seems to me neither realistic nor sufficient. First, how can one ensure that only satisfiable goals are assigned? Second, when goals are achievable, there will generally be many satisfying plans, some of which will be better than others. However, the agent has no basis for deciding among them, since its preferences are represented exclusively by the goals.

AI: Some planners resort to “meta-goals” for choice among a collection of plans or for relaxing goals that are unachievable. Others apply evaluation functions to account for features of the plan (cost, for example) not included in the goal specification. I must confess, however, that at present these approaches tend to be rather ad hoc.

DS: From my perspective, all of these mechanisms look like proxies for the value function, which describes the agent’s true preferences for states of affairs. For example, you dismissed the possibility of an agent simultaneously having the goals of being in bed and at the office, but what about the goals of getting enough rest and pursuing a career? Even if one were willing to define what counts as enough rest and sufficient career pursuit, this would not really capture the idea that the agent wants to achieve both objectives to the greatest extent possible. Given that goals are rarely absolute, wouldn’t value functions be more suitable as a basic representation?

AI: In principle, I can agree that a value function is a truer, more comprehensive representation of preferences. However, I think that you overlook the practical difficulties of specifying a value function over every possible state of affairs, as well as the computational advantages of goal predicates. The state spaces employed in typical applications of decision
theory are relatively impoverished in comparison to those generable by our planners. It seems unreasonable to expect the agent to consider every complete state in advance, or similarly to require the designer to specify a complete, universally applicable evaluation function. On the other hand, planning for a particular goal condition allows the agent to focus on a narrow slice of the complete state. And unlike optimality, goal satisfaction is a local property, which makes it much easier for a computationally limited agent to verify.

DS: We in the decision sciences are also familiar with Simon’s arguments about satisficing and bounded rationality. While appreciating your practical concerns, I am still left wondering in what sense or under what conditions goal-seeking behavior can be considered rational, bounded or otherwise.

AI: Perhaps it is best to interpret goals in their heuristic role for finding good plans. Our assumption is that we can represent actions and goals such that our planning techniques approximate rational behavior. To account for interactions, our rules for generating plans explicitly note exceptions that would render them inappropriate. In this way, we can approximate the sort of context sensitivity that you seem to achieve by evaluating complete states of affairs. For instance, one way of achieving the goal to get enough rest is to sleep late in the morning. An exception to this rule might be indicated by another rule stating that it is generally a bad idea to sleep late on a work day if you are interested in your career. There also may be exceptions to this exception; for example, if you stayed up half of the night preparing a report, it might be a good idea to get some extra sleep in the morning so you can effectively present the report in the afternoon. Using such rules, we believe a planner can generate good plans without referring to any complete state of affairs, and do so in a reasonable amount of time. Unfortunately, we have thus far done little to validate this belief, and currently lack the theoretical apparatus to critically examine the relation of the approximation to the ideal. For now it is an empirical matter, to be decided by comparing the performance of competing approaches.

DS: You may be able to build efficient planners by employing such rules, but I believe that ultimately you will require value functions as a normative standard to justify your rules. It is not necessary that a planner actually reason about complete states of affairs; it is only necessary that it act as though it has. At the same time, it might be useful for the agent to have some representation of its more fundamental objectives, even if it relies primarily on goal predicates for efficient planning.

It would seem that those of us working in the decision sciences have labored under the assumption that we would somehow be able to quantify over all states of affairs, while you in artificial intelligence have avoided speaking about complete states of affairs altogether. If we adopt the decision-theoretic view, the notion of goal makes little sense as a representation of preferences. If we adopt the artificial intelligence view, the notion of a value function defined over complete states of affairs plays no computational role. Perhaps we can find a reasonable compromise between these extremes.

AI: The problem as I see it is that the decision sciences do not want to abandon their normative basis for rational decision making despite the fact that it assumes an ideal agent with unlimited computational resources, and artificial intelligence does not want to abandon its commitment to building resource-bounded agents making decisions about an impossibly complex world despite the absence of a satisfactory alternative to the standard for rational decision making espoused by decision theory. It appears that we both have much to gain from the development of a normative basis for resource-bounded decision making.
6 Bounded Rationality

Over the last thirty years, there has been a great deal of research performed under the heading "bounded rationality." The work, strongly shaped by the path-breaking efforts of Simon (1982), has been concerned primarily with modeling the performance of agents in real decision-making environments. Although this literature has received little explicit attention in AI, its influence is reflected in the goal-seeking planning paradigm that dominates the field.

Limits in both informational and computational resources bound the rationality of decision-making agents. The two resource limitations are related, as computation constrains the body of information an agent is able to acquire and process. Indeed, it is possible to argue (though we do not) that agents are limited informationally only because they are limited computationally. Nevertheless, it is useful to separate the two types of resource because their deficiencies pose different sorts of challenges to AI.

Limited information (or knowledge) about the world is the source—if not the definition—of uncertainty. Accounting for uncertainty in the effects of actions requires significant modifications to the AI planning paradigm, since goal achievement can rarely be guaranteed. To a point, we can study the computational aspects of uncertain reasoning strategies without explicit mechanisms for managing computational resources. After all, an ideal, computationally unbounded agent would still be required to make decisions under uncertainty.

It seems logical that a better computational understanding of uncertain reasoning is a prerequisite for the development of good techniques for allocating computational resources to the various components of the task. In the rest of this paper, we attempt to contribute to this understanding by examining criteria for evaluating decision-making agents, both with and without computational constraints. The first step in the examination is to characterize knowledge and uncertainty in the agents of interest.

7 Knowing and Deciding

Before considering what it means for a machine to make good decisions, it is useful to ask what it could mean for a machine to know something about its environment. One proposed answer to that question (Rosenschein 1987, 1988) relies on possible-worlds semantics and a modal operator interpreted in terms of knowledge.

The machine (or robot) is represented as a deterministic automaton that takes as input a signal \( \sigma \in \Sigma \) and outputs some action \( \alpha \in \mathcal{A} \). Depending on its current state and its input signal, the robot makes a transition to a new state and generates an output action. The environment can be viewed as another automaton that takes as input the robot's output and generates a signal to serve as the robot's input. The two automata are coupled as shown in Figure 1, trading blows in a cycle of continual interaction. At any instant, the environment can be in any of a large number of world states \( \omega \in \Omega \).

In Rosenschein’s framework, possible worlds correspond to world states or instantaneous states of affairs, and the accessibility relation on possible worlds is stated in terms of objective correlations between states of the machine and states of environment. We denote the state of the machine in world state \( \omega \) by \( \psi(\omega) \). One possible world \( \omega' \) is knowledge
The advantage of this characterization is that it does not rely on any outside observer to interpret the internal states of the robot.

For the most part, the decision-theoretic language (e.g., “the robot’s expectation is that . . .”) seems more appropriate than the epistemic language (e.g., “the robot knows that . . .”) in describing the robot’s belief state. We are concerned not only with what an agent knows with certainty, but rather with its entire basis for action, including probabilities over possible events. Fortunately, the epistemic approach is perfectly compatible with decision theory, as probabilities are nothing more than statements about an agent’s state of uncertain knowledge (at least according to the Bayesian view). Taken in a broad sense—encompassing the revision of uncertain as well as certain beliefs—knowledge of its environment plays a central role in robotic decision making. In the next section, we look at the idea of gaining knowledge in the context of resource-bounded agents.

8 Evaluating Bounded Agents

In developing a theory of bounded rationality, we need some way of determining when one robot control program is better than another. In making a decision, a robot irrevocably allocates and consumes computational resources. Effort spent reasoning about one matter could have been devoted to another. Suppose we fix the hardware for our robot; we specify a particular machine, including memory, microprocessors, clocks, sensors, actuators, and the rest. We are interested in choosing between programs that run on this hardware, and, in particular, in choosing programs that maximize utility. It is not necessary to ascribe to the robot an explicit utility function; the preferences of concern belong to us, the designers.

Once we specify a program, the states of the robot are completely determined as a function of the environment in which it is embedded. We can assume that the robot has some finite lifetime that depends upon the program it is running and its environment. The utility of a lifetime is defined by the utility function applied to the stream of states it passes through. If the environment is deterministic (from our perspective, i.e., we can specify an equivalent deterministic automaton), we choose the program that results in the highest
lifetime utility. If the environment is uncertain, there are several possible lifetimes and we choose the program that maximizes expected lifetime utility, where expected lifetime utility is the sum over all possible lifetimes of their utilities, weighted by their respective probabilities.

Expected utility provides a fully general criterion for asserting that one embedded robot program is better than another. This generality has led to its incorporation in methodologies such as decision analysis (Raiffa 1968) and time-optimal control (Athans 1966). As in those applications, our use of the criterion for robot design assumes that we can specify a complete probabilistic model of the environmental response to alternative choices. This premise, unfortunately, is untenable; supplying a complete model describing the probability distribution of results for all plans in all situations is not generally feasible. For this reason, researchers in AI prefer programs able to generate reasoned explanations for their actions. Such programs are more likely to be robust and general because they indicate the ability to revise their conclusions in a principled way on the basis of changes in the situation.

To enforce this preference, we require a mechanism for evaluating the reasons behind a robot’s decisions. Suppose, for example, that the designer knows that at some point the robot must decide whether to turn right or left at a fork in the road. If the robot turns right, it is destroyed; if it turns left, it wins the lottery. We don’t want to penalize the robot for bad luck or reward it for good. Ideally, we would somehow reward the robot for only those decisions that it had “good reasons” for making.

The following paragraphs relate one attempt to provide a basis for preferring programs that provide good reasons for their actions. The attempt fails—the quest for knowledge in the absence of a theory of what is worth knowing is difficult to justify—but the account nevertheless should prove instructive.

Suppose a robot must decide whether to turn right or left on a street which happens to be one-way. If the robot turns left it will be fined $50; if it turns right, there will be no change in its assets. The robot is on its way to a meeting at which it will be asked whether or not it wants to invest $50 in a company $C$ that is destined to succeed. If it says yes, it will earn $500; otherwise, its assets remain unchanged. We are interested in the decisions concerning turns and investments, of course, but also that they be made for the right reasons.

Consider the following four programs. Program 1 is built so that it always turns right and invests in anything offered. Fortunately for Program 1, turning right will always be a good thing for it to do and no one will ever offer it a bad investment. Program 2 is built so that it always carefully attends to its driving, so in the above scenario, it takes note of the one-way-street sign and turns right. Program 2 is also constituted to invest in anything offered. Program 3 is built in such a way that it can easily be distracted from the task at hand. In the above scenario, it is distracted thinking about $C$’s prospects for success, fails to notice the one-way-street sign, and turns left. However, its preoccupation enabled it to infer that $C$ must succeed, therefore, it invests when the opportunity arises. Program 4 is also built to carefully attend to its driving, and turns right. In addition, however, Program 4 convinces itself at some earlier and less critical point in its driving that $C$ will succeed and that it should invest in the company.

We assume that all four programs are exposed to exactly the same stimuli; the states of the robot corresponding to its sensors register exactly the same information up to and
including the present moment. Only internal state distinguishes the different robots (i.e., different programs running on identical hardware). To help in ranking robot control programs, we introduce the notion of a possible time line, and a value function on the set of possible worlds. The set of possible time lines is notated $\Omega$ and an element of that set $\omega$. There are twelve possible time lines associated with our scenario. We assume that every program knows it cannot profit from an investment unless it actually invests; otherwise, there would be sixteen. The possible time lines are listed in Table 1 along with the payoffs realized in each. For simplicity, let us assume that the value function $u$ is the identity function on dollar payoffs.

Rather than judge a program solely on the basis of what actually happens (i.e., the true time line), we also consider what could happen with respect to what the robot knows (i.e., the time lines consistent with its knowledge, using the criterion of Section 7). The table has a column associated with each of the four programs, with a check indicating those time lines consistent with what the robot running each program knows. We could rate a decision made by a program on the basis of utility averaged over the time lines consistent with its knowledge the time the decision is made. If $\Omega_\omega$ is the set of all possible time lines consistent with what the robot knows in world state $\omega$, then we are interested in programs that maximize the decision rating function:

$$\bar{u}(\omega) = \frac{\sum_{\omega \in \Omega_\omega} u(\omega)}{|\Omega_\omega|}.$$ 

Table 2 compares the four programs using $\bar{u}$ applied to the true time line. Our rating method has some bugs, however. In the scenario we constructed, it so happens that knowledge of the future always served to eliminate less desirable outcomes. Consider the case where there are two possible worlds, one in which the robot loses $500 and an alternative

---

2The use of averaging here is arbitrary. Other combination functions would be equally reasonable.
one in which it wins $500. The robot that knows it will lose $500 is penalized using the decision rating function above because the winning possibility does not get averaged in. If there had been something that the robot could do to avoid the loss, the penalty would have been appropriate. But mere knowledge of unpleasant circumstances, no matter how useless, is not fair grounds for penalizing a program.

While it would seem to be difficult to reward a robot for knowing something, it may be possible to penalize a robot for ignorance. Suppose that, instead of including only the time lines consistent with what a robot knows, we include also the time lines consistent with what a robot could know. For instance, suppose that we consider the set of time lines consistent with the input seen by the robot so far. This set of time lines is the same for all programs in the scenario above. In this approach, the robot that always turns right would be penalized when the possibility of a fine is consistent with the input, even though fines on right turns in the real world may be exceedingly rare. While this approach may satisfy some researcher’s intuitions, it makes no sense from a decision-theoretic point of view; knowledge about regularities of the environment should be exploited to make robotic decision-making as efficient as possible. The mere possibility that a decision is incorrect is insufficient reason to refrain from adopting a corresponding policy in decision-making programs.

The intuition that knowledge is good may arise out of our interest in building truly “general-purpose” control programs. Even if the designer of a robot control program knows the exact automaton governing the environment in which a robot will be embedded, there is a desire to design programs robust enough to deal with slightly different environments. The right way to view the designer’s objective is as an attempt to maximize expected performance over a set of possible automata. But, again, it matters only that the robot act in an appropriate way; knowledge of the actual automaton governing the environment need not improve the robot’s performance, and may even distract it in such a way as to degrade performance.

The real issue concerns the value of knowledge, and it would appear that knowledge has no intrinsic value in the schemes for comparing programs that we have looked at so far. It is not what a robot knows that counts; it is what a robot does. It is not even necessary for a robot to know that what it does has value, it is only necessary for a robot to act as though it knows.

<table>
<thead>
<tr>
<th>Program</th>
<th>Possible Time Lines</th>
<th>Decision Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{\omega_7, \omega_8, \omega_{10}, \omega_{11}}</td>
<td>175</td>
</tr>
<tr>
<td>2</td>
<td>{\omega_{10}, \omega_{11}}</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>{\omega_1, \omega_4}</td>
<td>425</td>
</tr>
<tr>
<td>4</td>
<td>{\omega_{10}}</td>
<td>450</td>
</tr>
</tbody>
</table>

Table 2: Ratings for the four control programs
9 Analyzing Embedded Systems

In coming up with a theory of resource-bounded decision making, we need a model of the embedded system to talk about. One obvious candidate is the model used by Rosenschein in explicating situated automata theory, depicted in Figure 1. The advantage of this approach is that it allows us to fix the environment and vary the machine, ascribing epistemic properties to machines based on correlations between their states and states of the environment. The disadvantage is that it abstracts away from the embedding of the machine in its environment; the environment just sits around while the machine computes a response. Without this notion, we cannot say anything interesting about how conditions or internal states of the robot depend on the environment or are independent given the robot's input. This section outlines an alternative model in which the states of the machine are just part of the states of the environment.

We begin by assuming that the environment can be accurately modeled as a Markov process. The state space consists of a set of states $\Omega$. We assume that time is discrete and that $\Omega$ is finite, so the process corresponds to a Markov chain. The instantaneous state of the world at time $t$ can be described in terms of a vector of state variables, $\mathcal{P} = \{P_1, \ldots, P_n\}$. Some subset of $\mathcal{P}$ captures the state of the robot: the states of its sensors, actuators, and internal hardware elements. The rest of $\mathcal{P}$ captures the rest of the environment; factors that may or may not influence the state of the robot.

As in the situated automata model, the state of the robot at time $t$ is completely determined by the state of the world at time $t - \Delta$. We can describe exactly how the
states of the robot depend upon the states of the environment by employing a special form of Bayesian Network (Pearl 1988) that takes time into account (Dean & Kanazawa 1988). Figure 2 provides a graphical description of how the state of the robot changes as a function of time and the state of the environment. The black circles represent random variables corresponding to the values of the elements of $P$ at various instants of time. The squares correspond to functions that determine the conditional probability of the single variable leading in from the right given the set of variables leading in from the left. In Figure 2, $P_1$, $P_2$, and $P_3$ are environmental factors that determine the values at the robot's inputs $P_4$ and $P_5$. $P_6$ and $P_7$ are internal variables that are conditionally independent of the environment given $P_4$ and $P_5$ (i.e., $P_6$ and $P_7$ are completely determined by $P_4$ and $P_5$). The darkened squares illustrate how information flows from the environment through the robot and back to the environment as time passes.

The above model clearly indicates how the internal states of the robot depend on the states of the environment external to the robot. Using this model we can analyze behavior such as that resulting from lags in response time due to processing delays. For instance, suppose that at $t_1$ the input to a robot driving a vehicle provides evidence that a traffic light is green. At $t_2$ this evidence influences some internal state prompting the robot to press harder on the accelerator at $t_3$. Now suppose that at $t_3$ the input to the robot provides evidence that the traffic light is yellow; the robot will still be reacting to evidence from $t_1$ even though the new evidence would seem to counter this reaction.

10 Related Work

As noted above, there has been a great deal of work in economics and psychology on bounded rationality. This section provides pointers to a small number of recent papers that make connections to AI. Horvitz, Breese, and Henrion (1988) provide an interesting survey of the use and abuse of decision theory in artificial intelligence and medical expert systems. One recent effort is Wellman's (1988) decision-theoretic approach to automated planning. In this approach, the planner formulates tradeoffs by proving decision theoretically that certain classes of plans are dominated based on qualitative relations in the domain.

AI has always been interested in resource-bounded reasoning. Recently, increasing attention has been focused on uncertain reasoning and rationality. Horvitz (1988) considers the problem of choosing the best decision procedure among a set of candidates for solving a specified problem. The choice of which procedure to use relies on the decision maker having accurate expectations of the accuracy and running time of the available decision procedures. Dean and Boddy (1988) consider the allocation of processor time to several decision procedures in attempting to generate solutions to several independent problems under time pressure. It is assumed that, for any given problem, there exists a decision procedure that returns an answer whose expected utility is a known function of the time it is given to compute. Doyle (1985, 1988) explores a number of issues concerning the rationality of agents both real and idealized. Of particular interest is his discussion of preferences in artificial intelligence and the decision sciences; in the latter, an agent's preferences are taken as given, while, in the former, deliberation often involves selecting and even inventing preferences on the fly.
11 Conclusions

Decision theory provides an account of how an agent might make coherent use of its beliefs and preferences in the process of decision making. Preferences are described in terms of a value function. In defining what it would mean for a resource-bounded agent to make a good decision, we found it convenient to refer to a value function. It is not, however, clear that the agents themselves need to employ such a function; in fact, it would seem that one task an agent has is to continually reassess its preferences. It also appears that specifying a value function over complete state of affairs is an unrealistic requirement for the design of autonomous decision makers. As a concession to complexity, the designer of a robot control program may build in a value function that applies to partial states of affairs. It then becomes necessary for the program to be able to evaluate states of affairs that relate to multiple partial states. To the extent that goals serve as good representations for partial states of affairs, the techniques AI has developed for manipulating them will serve a useful function in the design of rational control programs.

Acknowledgements

We thank Jack Breese, Jon Doyle, Eric Horvitz, Ron Loui, Stan Rosenschein, and Yoav Shoham for discussions about the issues addressed in this paper.

References


1 Introduction

It seems to be getting harder and harder to define what planning is, but one thing is becoming clear, and that is that there can be no theory of planning without a theory of behaving. That’s because planning is the redesign of an agent’s intentions in order to make them more effective. Until we figure out what we mean by “intention” and “effective,” there’s nothing else to say. It’s entirely possible that once we do figure these things out, we’ll find that there’s no general theory of either acting or planning.

I have slipped a presupposition into this apparently tautologous introduction, the presupposition that it is possible to act without any planning at all. I assume that nowadays everyone would agree with this idea. When you play a video game, you may have constructed a long-range plan to play for a while and then go get lunch, but during the play such plan construction does not occur. Instead, one simply acts. This is in fact the norm. When you drive to work, your behavior can be quite complex, but no planning is going on.

Granted that behavior can proceed for long periods without planning, the question remains whether your behavior is nonetheless under the control of a plan during such periods. In my opinion, this isn’t much of a question. I think it arises because many people don’t like the idea that the agent is “under the control” of something; it doesn’t feel as if I am under the control of anything. Let’s assume that this is not an issue, so that there is some kind of program-like object governing an agent’s behavior at all times. And now let’s go one step further and call these things plans, dispensing with a more complicated two-level scheme. With this terminology, we can see that the problem is to construct a plan interpreter that is able to control an agent’s behavior for a long time without additional planning. In some cases that’s all we need. But in general it must be possible to create and optimize plans, and it must be possible to learn new plan schemas.

In what follows I will completely ignore the problem of learning new plan schemas.¹ I will make the usual assumption that there is plan library full of carefully hand-coded plan schemas, and that the planner can match its current tasks to schemas in a straightforward way.²

2 Plan Notations

Let’s look at what we want in a plan notation.³ We will need a program counter, so that an agent can intend to do action $B$, but only after doing action $A$. Variables can be bound

---

¹Kris Hammond’s dissertation looks at this problem.
²Most planners match each task to one schema. Simmons’s planner is an exception.
³Much of this section reflects work done by Jim Firby in his forthcoming Ph.D. dissertation.
by one action and substituted into another. An agent can intend to find an object $X$ in a box and take it out. Loops and conditionals must be supported.

Plans are executed in the service of tasks, where a task is an intention to carry out an action. Plans give rise to new tasks. However, a plan does not exactly consist of a collection of tasks. If an action contains variables that will need to be filled in before it is executed, then it cannot be considered a task until they are known. In fact, we may as well avoid using the term task for any intention that is not presently active. With this terminology, we say that executing a loop will generate an indefinite sequence of tasks, not the same task over and over. A plan is best thought of as a special sort of program, which only indirectly describes the tasks it will give rise to. Part of a planner’s job is extracting useful descriptions of future tasks from active and proposed plans.

Actions are designated by terms like $(\text{MOVE-HAND-TO } z)$, where $z$ is a term denoting an object in the vicinity of the agent. Right away we run into an important issue. In a realistic agent, a term like $z$ must ultimately be passed to an effector, some piece of hardware or software that can translate it into “coordinates,” or, more generally, what we might call an effective designator, a term that can mean something to a sensor-effector system. Agre and Chapman are fond of effective designators like “the coffee cup in front of me.” It is not too implausible to assume that such a term could be passed to an effector, which would do what is required with that cup (or fail because the cup could not be seen). But if someone wants back “the diskette I gave you this morning,” that is not an effective designator. The plan “Pick up (the diskette Fred gave me this morning); Give it to Fred” cannot in general be carried out.

We cannot rule out the use of ineffective designators. We just must make sure that they never get handed to the sensor-effector system. Whenever an operation is required on an object known only by an ineffective designator, sensory steps must be executed to find an effective designator that can be inferred to denote it. If the agent knows that Fred’s diskette is the only one that has “FRED’S SPACE WAR” written on it, then it can look for that feature, and produce an effective term of the form “Diskette in front of me that has FRED’S SPACE WAR written on it.”

Plans are inevitably failure-prone, because sensors and effectors don’t always work, because the world contains interfering forces, and because the agent can interrupt itself. In fact, it is difficult to say what makes a plan or plan schema correct. There ought to be at least a theoretical statement on this issue, but I don’t know of one. For example, a plan schema might have the property that it screws itself up one out of every hundred times it is executed, and it still might be better than any alternative. To adapt an old example from Savage, consider a plan for obtaining $n$ beaten eggs. One plan is to get $n$ eggs, crack each one, drop it into the bowl, and beat the result. If the $n$th egg is rotten, this plan will waste a lot of eggs. Nonetheless, almost no one adopts a more complex plan to avoid this problem, because it’s so rare.

Putting aside the issue of plan correctness (or, more generally, goodness), it is clear that plans will have to involve a lot of sensory feedback. If while working with eggs you detect a rotten-egg smell, the plan must explicitly encode what to do. For almost any task, there must be a check whether the plan adopted for carrying it out actually succeeded.

---

4It might not. Instead, there might be a more general monitor for the smell of rotten eggs which escapes to a diagnostic reasoner.
There is a design decision here, whether to have such checks be active or passive. An active success checker would make "check success" an explicit task, typically requiring some interaction with the world. A passive success checker would just look in its current world model to see if the success conditions are met. While the former seems more sophisticated, the latter seems more practical. A passive checker relies on plans to include steps that do enough sensing to update success or failure conditions in the world model. Most plans will therefore contain as a last step "Look at the work area." The resulting sensor scan will update the world model so that the success check will work correctly. In cases where a local sensor scan is inadequate, the plan itself can specify whether to go somewhere to do a check or to assume success based on the success of substeps. For example, in doing wiring, one must usually go somewhere to disable a fuse or circuit breaker, thus turning off the electricity to the area being rewired. The purpose of the step is to turn off the electricity. Because of the consequences of failure, the plan for disabling the fuse will say to go to the area being rewired and make sure the power is off. The plan does not finish until this is done, and then a passive check will verify that the plan succeeded.

We therefore make it the responsibility of the plan interpreter to check (passively) for the success of every plan. There are other places where such checks need to be inserted. Consider the classic "protection violation," in which a prerequisite for a plan step ceases to be true. The classical preoccupation here was how to anticipate and forestall violations. Thinking about realistic plans changes the problem a bit. Just as for plan-success checks, we make it the responsibility of the plan interpreter to check passively that protected facts are still true when required. However, in the real world it is entirely possible that the check can fail. In general, the correct response depends on exactly what kind of protection is involved (see below). But the default response of the plan interpreter to any kind of failure can only be to try again. In unhurried situations this approach will often work. Either a different plan will be selected in the new circumstances, or the old plan will just work this time. While the retry is going on, a planner can perform a more global reanalysis of the failure. In time-pressured situations, failures will usually result in various tasks just going away, because they had deadlines that were not met. In such situations, plans just must work most of the time.

Plans must include interrupts, implemented by monitors. A monitor is a plan that checks for a condition and suspends until the condition is true. When the condition becomes true, the plan can resume, and, if its priority is high enough, it can interrupt another plan. This condition check is passive, too. If the condition is not sensed in the normal course of things, another task must be active to check for it periodically.

Monitor plans can be used to implement protections. Whenever a protected fact becomes false, the monitor can take action. There are several different kinds of protection, besides "hard" protections that cause plan failures when detected. The agent could be given the task of reestablishing a fact within $x$ minutes whenever it becomes false; with minimizing the time during which it is false; or with keeping the fact as "nearly true" as possible (appropriate for quantifiable facts like "number of unpackaged grenades is 0"). Each of these tasks would require a different kind of plan.

The final issue is how plan schemas are selected. We assume that every action type is associated with a set of schemas, or methods, for that action type. Each method has a context filter that evaluates to TRUE when it is appropriate. Given a task, the interpreter randomly selects a method with a true filter, and expands it to generate a bunch of subtasks.
Jim Firby has implemented an execution system with the properties described, which he calls the Reactive Action Package (RAP) interpreter. It runs a robot delivery truck in a simulated world. We have done experiments in which the truck is given several jobs to carry out while an evil demon steps in periodically to add or remove objects in its vicinity. The system copes with such interferences, requeuing failed tasks, until the interference rate gets to be too high. Even then, it succeeds on a subset of the tasks that were given to it. The system exhibits the kind of graceful degradation one would want in an execution module. The main criticism it is open to is that its sensors operate at an unrealistically high level.

3 Planning

The RAP plan notation, and similar models under development elsewhere (described below), are an order-of-magnitude more complex than anything that planners have worked with before. This raises two possibilities: that planning might be unnecessary, and that it might be impossible.

Let me define planning more precisely: Planning is the contemplation of future courses of events in order to choose among them. Our behavior model does almost no planning. All method choices are based on local considerations, tests of the current database. In spite of this, the system manages to muddle through many planning scenarios, because of its doggish persistence. If one attempt doesn't succeed, it tries again.

This persistence makes some of classical planning unnecessary. There is little point in trying to carefully avoid having to move a block twice, when the system will probably have to move blocks more than once anyway just to compensate for effector errors and outside interference.

Some have hinted that competent behavior makes all of planning unnecessary, but it is hard to take such hints seriously. There are lots of circumstances where there is time for planning and the payoff is large. The prime example is in scheduling such things as mobile-robot trips or machine-tool operations. Doing things in the wrong order can cost a lot of money or cause deadlines to be missed. It seems clear that here is where the focus of planning research should be. The only disappointment from this conclusion is that there is not likely to be a general-purpose algorithm for doing scheduling. Each scheduling problem requires a complex search through a space all its own, using whatever techniques fit. "Hierarchy," "least commitment," and "prerequisite-clobbers-brother-goal" will not do much for us here.

The role for planning research is to find architectures that allow us to plug in this kind of domain-specific expertise where it is needed. It seems to me that the proper framework is the "generate-test-debug" (GTD) strategy studied by Sussman, Hammond, Simmons, Linden, and others. The planner must generate a plan using its plan library. The planner then projects the effects of the plan, notes potential problems, and applies transformations (again drawn from a library) to fix those problems. This technique has been applied successfully in the domains of blocks, Chinese cooking, geology, and autonomous land vehicles.

The GTD technique has been applied mainly to the problem of detecting and correcting fatal bugs, which will keep the plan from being executed at all. As I suggested earlier,

---

5The only exception is a little bit of route planning.
this is probably misguided, and betrays the computer scientist's fascination with intricate puzzle solving. We would hope that in the usual case the plans in the plan library can cope with most planning bugs. Unfixable crashes should be rare. The main purpose of the transformation phase is to improve the efficiency of a plan that, if need be, can just be executed as is. This is the focus of our current work. We are looking at transformations like these:

- If the plan contains tasks at remote locations that haven't been scheduled yet, transform by scheduling them (using a black-box heuristic scheduler).
- If a task's location is unknown, and there are several possible locations for it (e.g., "Fill up with gasoline" can be done at any fuel depot), assign the location nearest to the scene of other tasks' locations.
- If a task is done at one point in space-time, and it could be done at a different point closer to the current planned route, undo the schedule and reassign the task to that point.

It seems clear that having a few scheduling transformations will be a big win. What remains to be seen is whether having tons of transformations will cause the system to blow up.

The Panelists

I encouraged my panelists to let a hundred flowers bloom, and tell me what's been on their minds lately regarding planning systems. The panelists are David Chapman, Tom Dean, and Mike Georgeff. Chapman wrote a description of the system he is implementing to take advice while playing a video game. Dean wrote an essay on the value of goals and knowledge in planning. Georgeff wrote a description of the latest version of PRS, which is used to carry out diagnostic and repair plans on a simulated space shuttle. I will comment briefly on each of these.

Chapman: A System Illustrating Situated Instruction Use

Chapman's work with Phil Agre on "situated activity" is probably well known to everyone. I think it fits in nicely with the approach to planning and execution described above, although I'm not sure they agree. They tend to emphasize the following ideas:

- Planning is too expensive to be engaged in very often, and its results are unreliable.
- Usually an agent knows what to do in a situation like the current one, without much regard to how it got into the situation.
- Terms denoting objects are indexical and functional. Most of the objects agents deal with are referred to with effective designators like "the coffee cup in front of me," which could refer to different entities at different times, as opposed to "my spouse," which always refers to the same entity.
The perceptual system bears the responsibility of carving the world up into pieces with useful functional properties. When indexical-functional terms get bound to actual perceived objects, behaviors involving those objects get proposed, voted on, and selected. There are a fixed collection of possible behaviors, namely, those appropriate to objects of the types that can be perceived. Hence the behavior-selection can be done using simple combinational hardware.

Video games have been the domain for study of these ideas, an appropriate choice given the real-time constraints and disutility of planning. Chapman proposes to study language understanding in this context, of the sort required when one video player hears advice shouted by his partner. He proposes (based on analysis of videotapes of people playing such games) that most utterances are designed to focus the hearer's attention on possibilities he is already at least potentially aware of. For example, "Turn left" means "Turns left the next time it makes sense to turn left." Utterances get latched in buffers, and hang around long enough to influence behaviors, in this case, until a turn presents itself. At that point the turn-left behavior gets activated (because the perceptual system sees a gap to the left), and the advice to turn left serves as an extra vote in favor of doing so.

I think this is interesting work. The model of language certainly seems plausible (or rather the standard competitor seems rather implausible here, there being so little time to engage in parsing, semantic interpretation, speech-act analysis, etc.). The overall framework, of competent behaviors that wait to be triggered and just do the right thing, fits with the kind of behavior model I espoused above.

Chapman is open to an objection that perhaps is a problem for this whole area: If we want a program to play a video game, we can just write one, and isn't that what he is doing? Similarly, if Firby and I want to deliver some rocks with a certain simulated vehicle, why don't we just write a program to do it? Why not just use Lisp instead of devising a fancy notation that's executed much slower? I could make the same objection to the PRS system, described below.

For Firby's RAP system and PRS, I think the answer is, perspicuity. The notation is designed to be mechanically manipulated as well as executed. We go to the trouble of keeping track of discrete tasks and plans because we want to be able to rearrange them before executing them. For Pengi and Sonja, Chapman's video game players, the same answer doesn't work. I think there are other answers implicit in what they are doing, but they should be made explicit. Here are two possibilities:

1. Behaviors get proposed, voted on, and chosen or overridden. For this to be possible, there has to be explicit, uniform representation of a behavior. A wire will suffice.

2. Behaviors are learned. Hence although little or no planning goes on during the play of a video game, there is construction and debugging of plan schemas (i.e., behavior networks) over a longer time scale.

**Georgeff: An Embedded Reasoning and Planning System**

Georgeff describes PRS, the Procedural Reasoning System, which is being used to carry out diagnostic and repair procedures on a space-shuttle Reaction Control System (RCS). PRS
Planning and Behaving

satisfies many of my desiderata for a behavior controller. Plans are triggered by sensory events. If more than one is appropriate, there are standard ways of selecting among them. Several plans can be active at once, some waiting for events to occur or facts to become true. When they wake up, they can pre-empt other activities, giving the system the ability to respond quickly to changing circumstances.

PRS is one of those systems with metalevels. Behavior occurs at level 0, and thinking about what to do occurs at higher levels. In particular, the choice between plans is made by executing a metalevel plan that arrives at a decision. The metalevel plans are better thought of as performing inferences than as doing things.

There are some things missing from PRS. The responsibility for checking goal satisfaction rests with plans for carrying the goal out. But this isn't too different from the sort of passive checking that I espoused above. One can think of it as “ultrapassive” checking, in which the interpreter assumes a plan worked unless the plan itself looks and realizes it did not.

It is not clear what position is embodied in PRS on the question of effective designators. The system has been applied to situations in which the issue doesn't come up. In the RCS, terms designate valves and sensors, for which effective designators are naturally available. In low-level robot control (the original application of PRS), there were two kinds of terms: those that referred to immediate sensory data (“the clearance in front of the robot”), which are clearly effective, and those that referred to geographical locations (“the west wing of the building”). It was assumed that the robot knew where it was, so the latter class was effective, too. (Firby and I make similar assumptions.) The problem of dealing with ineffective terms comes up only when a robot can interact with an object, lose immediate track of it, and then need that particular object again. It is not clear how this issue will be addressed in PRS.

Georgeff's paper contains a discussion of planning that I do not agree with. He proposes that “simply making the choice as to which course of action to pursue, no matter how one does it, constitutes forming a plan to achieve the goal g.” He goes on to point out that, in PRS,

The method of choosing between alternative courses of action is embedded in the metalevel KAs of the system and thus ... the particular approach to forming plans is not hard-wired into the system. To the extent that the choice is made arbitrarily, one may wish to avoid calling this process “planning.” But where it is based on any information at all — no matter how meager — the determination of an appropriate course of action is indeed planning.

I would urge that we reject this broadmindedness, in favor of the definition I proposed above, that planning is reasoning about alternative possible courses of future events. I think this makes a pretty clean separation between models in which the selection among behaviors is done based on past and present data, and models in which it is based on expected outcomes. Philosophers can argue about exactly when a machine is thinking “about” the future (or about anything else), but it's pretty clear in most cases. For example, Chapman's video game player decides to go left or right around an obstacle by drawing a line from the current location to the desired location and seeing whether it passes to the left or the right of the centroid of the convex hull of the obstacle. This is about as borderline as a case
could be, but it seems to me to be definitely classifiable as planning. But most of the time the systems of Chapman, Georgeff, and Firby decide what to do on the basis of checks for what’s true now, because such checks can be done quickly and usually work as well as anything else. For instance, in deciding whether to try to pick up an object with an unaided arm or special tool, the “pick-up” RAP asks whether the object is known to be too heavy. If not, the unaided arm is used. No calculation is performed of the probability that this will work and the consequences of failure. If it doesn’t work, the task will fail, and when re-tried the object will be known to be too heavy.

Dean: *On the Value of Goals*

Dean’s paper is somewhat different from the others. It is about the relation between AI planning and decision theory. It makes some useful observations:

- The notion of goals that must be satisfied is quite odd from a decision-theoretic point of view, on which utility must be maximized.

- The value of knowledge is hard to quantify, because an ignorant robot that does the right thing is as good as a knowledgeable robot that does the right thing, even if the latter intuitively has better reasons.

The paper concludes by saying that a goal can be considered a partial utility function, which tells a robot something about how well it is doing.

I have trouble following the thread of Dean’s paper. There are certain things I just disagree with. He proposes that the utility of a robot’s lifetime is the sum over the states it traverses of the values of its utility function applied to each state. This picture is appealing, at least intellectually, for the case of a human being, where one might try to add up all the “pleasure” experienced by the human over his lifetime. (And then try to select the lifetime with the greatest total pleasure.) For a robot, it makes no sense at all. We might well want the robot to sacrifice itself for some future gain that would occur well after its lifetime had ended. Leaving such kamikaze scenarios aside, it is in general the case that a robot must assign probabilities to plans that (presumably) cover only a small part of its life. It must, for instance, assign a utility to one plan over another based on the amount of resources left over after each. The utility of resources depends on future demands for them, not just by this robot but by all its allies. Whether these demands fall beyond the end of the robot’s life span seems irrelevant.

Still, we should take Dean’s arguments for decision theory seriously. Georgeff notes that tasks are predicates on time intervals: “Achieve $p$” or “Maintain $q$,” for instance. I think we need to note that these predicates fall into two categories: those that refer only to occurrences within the intervals, and those that refer to alternative time lines. An example of the former is, “Make no noise.” An example of the latter is, “Make as little noise as possible.” A logical analysis of this kind of task would have to quantify over alternative time lines in which more or less noise was made. We can call tasks of the first sort *constraints*, tasks of the second sort *preferences*.

In general there can be only one preference in effect at a time. The need to maintain silence must be balanced against a need to proceed quickly, to avoid running down the
batteries, to consume little gasoline, to move during the night whenever possible, and so forth. These preferences cannot all be met simultaneously, so they must be combined in some overall utility function, and what we really ask is that its value be maximized. As Dean points out, there has been almost no work on automating planning to maximize expected utility, and one reason is that the general form of utility functions is a mystery.