IRCS Technical Reports Series

Institute for Research in Cognitive Science

12-1-1994

SodaJack: An Architecture For Agents That Search For And Manipulate Objects

Christopher W. Geib
University of Pennsylvania

Libby Levison
University of Pennsylvania

Michael B. Moore
University of Pennsylvania

University of Pennsylvania Institute for Research in Cognitive Science Technical Report No. IRCS-94-31.

SodaJack: An Architecture For Agents That Search For And Manipulate Objects

by

Christopher Geib
Libby Levison
Michael B. Moore

University of Pennsylvania
3401 Walnut Street, Suite 400C
Philadelphia, PA 19104-6228

December 1994

IRCS Report 94-31

SodaJack: an architecture for agents that search for and manipulate objects [*]

Christopher Geib Libby Levison Michael B. Moore
University of Pennsylvania
Department of Computer and Information Science
200 S. 33rd Street, Philadelphia, PA 19104
Email: (geib,libby,mmoore)@linc.cis.upenn.edu

January 22, 1994

Content Areas: planning, AI architectures

[*]This research is partially supported by ARO DAAL03-89-C-0031 including U.S. Army Research Laboratory (Aberdeen); U.S. Air Force DEPTH through Hughes Missile Systems F33615-91-C-0001; Naval Training Systems Center N61339-93-M-0843; DMSO through the University of Iowa; NSF IRI91-17110, CISE CDA88-22719, and Instrumentation and Laboratory Improvement Program #USE-9152503.

SodaJack: an architecture for agents that search for and manipulate objects

Content Areas: planning, AI architectures

Abstract

This paper presents an architecture for agents that search for and manipulate objects. It is demonstrated in the SodaJack system, a system that animates a human working at a soda fountain. The system is constructed as a set of three interacting planners. Two of these planners are special-purpose modules which contribute context-specific plans for the tasks of searching for and manipulating objects. The search planner is used to convert knowledge acquisition goals into goals of searching locations. An object specific reasoner is used to build object sensitive plans for manipulating specific objects. Finally, an incremental hierarchical planner is used to integrate these two special purpose planners into a complete system which interleaves planning and acting.

1 Introduction

Suppose that you had a personal robot assistant. Many of the commands you might give would require it go find an X and do Y with it. For example, the command `Get a scoop and scoop the ice cream` implicitly means, go find a scoop and use it to scoop the icecream. This suggests that if we are going to design robotic agents to assist us, they must be able to locate objects and manipulate them.

For many planning systems, the issue of searching for objects never arises. For example, [1, 5, 12, 17] work under the simplifying assumption that the agent knows all of the objects in the world and their locations, and every object in a plan uniquely refers to an object in the world. Thus when the robot is given the command `Get a scoop`, the command actually refers to a unique scoop; the assistant knows this, and will go and get it[1].

[1]This is not to suggest only one scoop is available, but rather that the agent does not consider getting some other scoop.

Much of the work in planning [2, 5, 10, 12, 13, 18] has also abstracted away crucial details of how to carry out low-level object manipulation. Questions such as where to grasp objects, how many hands to use, and what to do if the surface is too slippery for the agent to grasp firmly remain unanswered by these systems.

This work confronts these issues by addressing two major problems: (1) locating objects to manipulate and (2) performing the low-level reasoning required to actually manipulate them. This paper will present the SodaJack system that embodies an architecture for autonomous agents that are able to search for and manipulate objects. The following section provides some background on SodaJack followed by two sections that present an overview of the system and a trace of an example. Sections 5 and 6 describe in more detail the search planning algorithm and the reasoner used to plan object manipulation. We will close with a discussion of related work.

2 Background

The SodaJack system and architecture are an attempt to develop realistic animations of human figures carrying out tasks specified by high-level goals. These animations are created using the *Jack* animation system [3]. We have named the system SodaJack after its first domain, the counter of a soda fountain (Figure 1).

SodaJack accepts as input an ordered set of goals to be achieved. Its task is to develop and monitor the execution of a plan for the agent to achieve these goals. *Jack* accepts as input low-level *motion directives* [4] providing a simple interface to the animated agent. Such motion directives are the output of SodaJack.

It is important to recognize that since we are working with an animated agent, many of the problems associated with perception in real agents are not significant for us. While all of the modules of SodaJack use sensing operations to acquire information about the state of the world, all of the needed information is contained within, or derivable from, the databases used in the *Jack* system. Thus, our current system is not confronted with the problems of scene or object recognition that it might otherwise face.

However, this does not mean that we have ignored all of the problems of vision. For example, our task of finding objects would be trivial if our model

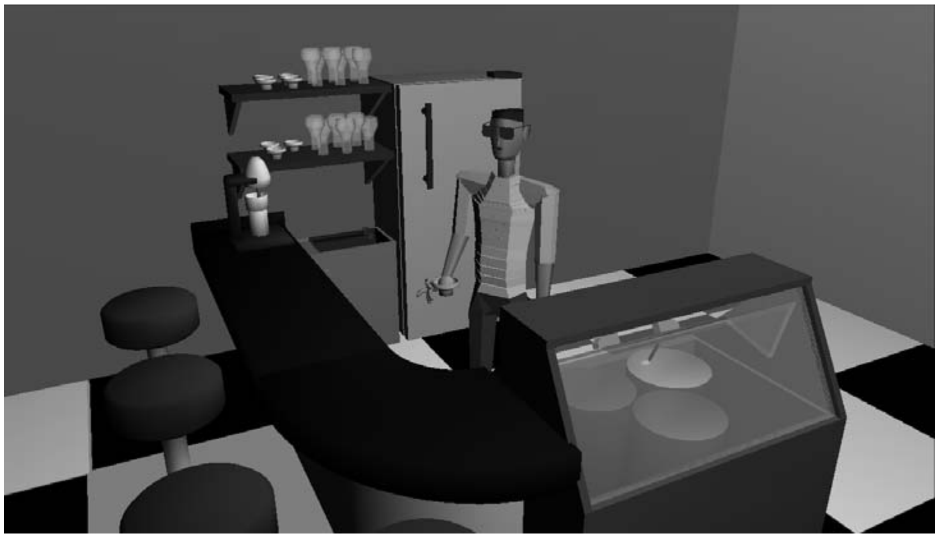

Figure 1: The Soda Fountain

did not allow for containers that the agent cannot see into. To this end, we have limited the agent's perception to only those objects in containers that are open. Thus the agent will be forced to open containers to find objects.

3 SodaJack Architecture

At its core, SodaJack is a set of three interacting planners. A hierarchical planner (ItPlanS) uses two special purpose planners, a search planner and an object specific reasoner, as experts for planning search and object manipulation. We have adopted a hierarchical planning architecture for SodaJack, since it directly supports the different levels of abstraction used for search planning and object manipulation. Figure 2 shows a system diagram for SodaJack. The communication between its components will be described in Section 4.

ItPlanS [6] uses a simple goal expansion method to perform its hierarchical planning. It selects an expansion for an unsatisfied goal from a library of possible expansions, using the world state as a guide. This process is repeated down to the level of primitives actions. ItPlanS then calls on a series of experts to verify that the plan will in fact achieve its goals. How-

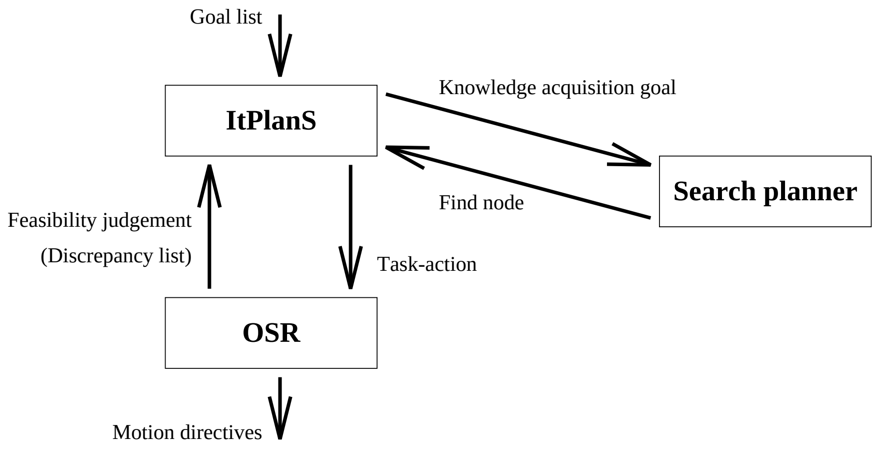

Figure 2: System Diagram

ever, in planning for locating objects and for reasoning below the level of primitive actions, we have found it more efficient to create new plans than to use ItPlanS' standard planning method. Therefore, a search planning module and an object specific reasoner contribute new, context-sensitive and object-sensitive plans. In short, the ItPlanS chooses from among known plans in a context sensitive manner, but relies on the search planner and the OSR to generate new plans for known goals in specific situations.

4 SodaJack Implementation

Having given a brief overview of the system, we now illustrate the workings of SodaJack with an example. Consider the goal `Get a scoop`. This would be specified as **get(X)** with the added constraint that **type(X) = scoop**. ItPlanS expands this goal into the subgoals: **goto(X)** and **pickup(X)** (Figure 3).

ItPlanS considers the action **goto(X)** to be primitive but underspecified since the variable **X** is not bound to a particular object. In order to bind the variable, the search planner must be called to generate a plan for locating a scoop. To this end, ItPlanS adds to the plan a *find node* and calls the search planner to instantiate a search plan (Figure 4).

The search planner reasons from a knowledge acquisition goal, in this case locating a scoop, to the goal of exploring locations where a scoop might be.

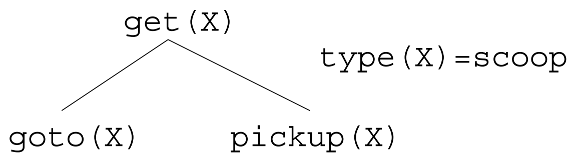

Figure 3: Simple decomposition of **get(X)**

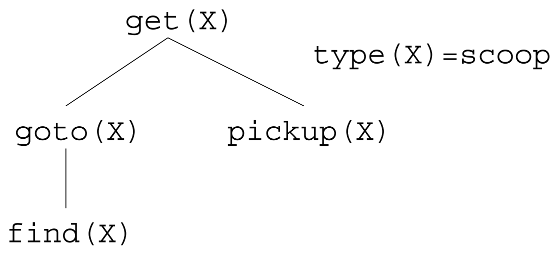

Figure 4: Addition of the find node

Satisfying this goal requires physically searching through possible locations.

ItPlanS asks the search planner to expand the find node. Each time a find node is expanded, the search planner first examines the *Jack* environment to determine if the object is visible to the agent. If no object is found, the search planner selects a location to explore next, generates a goal to explore that location, and adds it to the plan (Figure 5). This goal is then further expanded by ItPlanS (Figure 6.)

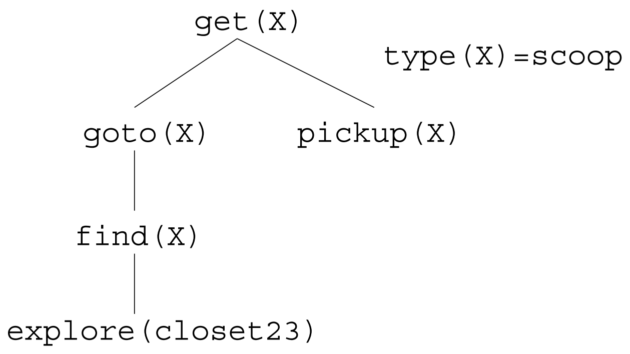

Figure 5: Establishing the first site

In our example, having noticed that the scoop is not visible in the envi-

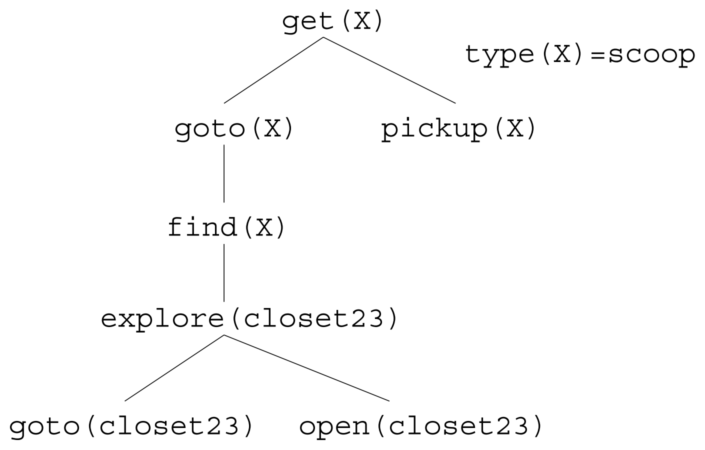

Figure 6: Searching the first site

ronment, the find node is expanded down to the primitive action of going to the closet. Since all of the arguments in the action are bound to specific objects, the OSR can now be consulted to verify that the action can be carried out. If the action were not feasible, it would fall to ItPlanS to find another way to explore the closet.

In this case, the OSR reports that the action is feasible, and ItPlanS commits to its performance. ItPlanS then invokes the OSR to generate an appropriate series of motion directives, situated in context, to be given to *Jack* for animation. This process is repeated for the task of opening the closet door. Since committing to an action results in the motion directives being given to *Jack* for animation, planning and execution are interleaved in our system.

After the closet is explored, ItPlanS considers whether to expand the find node again. Using the search planner to evaluate the progress of the search, the world is examined for objects having the property of being a scoop. If one is located, the search is considered successful. If not, a new location is selected for exploration and the searching process repeats until there are no more locations to explore. If this occurs, the search is considered unsuccessful, and SodaJack must give up.

Suppose the search terminates successfully, finding a scoop in the closet. The system still has not achieved the goal of getting the scoop. However, having bound the variable **X** to a specific scoop, ItPlanS *is* in a position to

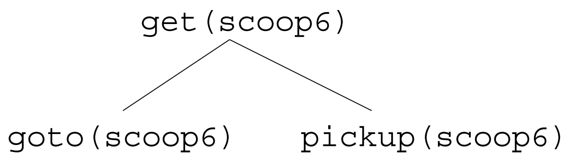

Figure 7: Plan after search

call the OSR to consider the action of going to the scoop, and subsequently picking it up (Figure 7). We now turn our attention to how the search planner and OSR work in more detail.

5 How Search Planning Works

In planning to find objects, a planner is confronted with a specific case of a very general problem, planning to act to acquire knowledge. That is, general knowledge acquisition goals must be converted to actions, so that an agent can act to acquire the desired information. Our approach is to isolate this reasoning in a specialized module. In the case of our specific problem, the search planner translates information acquisition goals to high-level physical goals for the exploration of the environment.

There are a number of desiderata that must be satisfied by planning systems that want to build search plans. First, as Haas points out [9], any plan for acquiring information must rest on what the agent *knows* about the environment. That is, an agent must know the locations that an object could be at, in order to search for it. In our architecture, the search planner selects a single location from among the places the agent is considering for future exploration.

Second, planning for searches requires conditional selection of actions to decide whether to continue exploring locations. This requires building conditional plans, however conditional planners such as [16, 19] do not distinguish between what the agent *knows* about the environment from what is *true* in it. This makes them unacceptable choices for planning searches.

This distinction between what the agent knows and what is true in the world is important for limiting the conditions that are placed in search plans. These conditions should be restricted so that an agent only tests its *knowledge* of the environment when deciding which branch of conditional to execute.

If the agent tests the environment directly, it could be involved in arbitrary amounts of acting and reasoning. This constraint amounts to requiring the system to plan to acquire information before a conditional branch is reached.

As mentioned above, in SodaJack, perception is blocked by obstacles such as walls and closed containers. The search planner uses this fact to infer a goal of opening a closed container from a goal of acquiring knowledge about an object. Extending this inference is discussed in Section 6.3.

Searches are planned by first identifying currently known locations where an object may be located and systematically exploring this space. A plan is developed for exploring each location in turn. After such an exploration plan is executed, the environment is observed to determine whether the agent can see an object with the desired properties. During this observation phase, new potential locations may be seen by the agent. These new locations are considered for future exploration as needed.

Searches terminate successfully when a referent object is seen in the environment. They terminate unsuccessfully when there are no more locations to explore or if the environment changes in a way that obviates the search. For example, if the goal that the search is a part of is dropped the search will be terminated. In the case where a search terminates successfully in an environment containing many objects which satisfy the description, an ordered set of these candidate objects is made available.

Sequential planners can be used to generate search behavior. (A *sequential planner* is any planner in which the plans produced are simply sequences of actions.) Our approach to search planning relies on producing and executing a sequential plan as a subroutine in a heuristic search algorithm [14]. The heuristic search has as its goal finding the container which contains the desired object. The heuristic search currently uses only distance from the agent to order locations for exploration. Two lists of locations are maintained by the search algorithm, an *open* list of locations yet to be explored and a *closed* list of locations which have been explored.

In one iteration of the search, the closest open location is selected to be explored. ItPlanS is used to generate a sequential plan for exploring the selected location. After that plan is executed, the resulting world is observed to determine if the desired object has been located. New locations observed during the action are added to the open list at this time.

6 How the OSR Works

The Object Specific Reasoner (OSR) expands existing goals generated by ItPlanS and passes these plans to *Jack* for animation. Basic actions concerning object manipulation, or *task-actions*, are the primary input to the OSR. Given this input, the OSR can perform two functions: (1) it can determine if a primitive action is *feasible*, i.e., could be performed by the agent in the given context, (2) it can construct a set of *motion directives* to execute the task-action. The following two subsections will consider these functions separately.

6.1 Checking Action Feasibility

Checking the feasibility of a task-action is performed in four steps: selecting an *action outline*, conditionally expanding all steps of the outline, using details of the object of the current task-action to refine the outline, and verifying that the agent can perform this specific action on this specific object. If an outline can be found and tailored to the details of the agent and the object, the task-action is judged to be feasible.

In the first step, the task-action and the type of the object are used to select an action outline from a library of outlines. This library is indexed by both the task-action and a taxonomy of object types. For each task-action, there may be separate action outlines for each type of object that can appear as an argument. An action outline is defined as a set of conditional steps, with the conditions being used to determine if the step is necessary for the given object.

The second step expands all the conditionals in the action outline. Each conditional step is either another action outline or a *motion directive*. Steps are selected for inclusion if their associated condition is true. This process continues until all action outlines have been replaced by motion directives. The original outline may specify a partial order on the resulting set of motion directives.

The third step binds parameters of the motion directives based on information about the specific object. Each motion is defined in terms of the agent resources and the object attributes on which it depends. Adding the values which describe the specific object refines the motion directive to the exact context in which it is being used.

The fourth step involves checking dependencies between the agent resources and attributes. Each motion directive includes a predicate which specifies those pairs of resources and attributes to be checked. For example, the OSR might check whether the agent's hand is large enough to grip the handle of a scoop. If all the dependencies for all the motions in this outline are within tolerance, the OSR reports that the task-action is feasible.

If the agent and object attributes fail the tolerance test, then control is returned to ItPlanS along with a record, called a *discrepancy list*, of those resource/attribute pairs that are out of tolerance. How this list can be used is discussed in Section 6.3.

6.2 Action Execution

The feasibility check described above produces a set of motion directives. When called upon to output the set to *Jack*, the OSR must first provide a start time and an approximate duration for each motion. Durations are calculated from a temporal database which contains both rules to generate times for parameterized motions (such as a reach) and fixed values for other actions. When the temporal information is added, task-action refinement is finished, and the motion directives are sent to *Jack* for animation. Errors may occur during this animation, and the OSR relays these errors back to ItPlanS for replanning.

6.3 Extending the OSR

We noted that the OSR signals failure when the agent and object attributes for a motion directive are not in tolerance. The OSR places any attribute pairs which are out of tolerance on a *discrepancy list*, which is returned to ItPlanS. A failure of this type can be corrected by allowing the agent to employ a *tool* to bring the attributes within tolerance. The information on the discrepancy provides the attributes for the needed tool. We are working to allow ItPlanS to use the discrepancy list as input to an extended search planner in order to find a tool to mediate the task-action. If such a tool is found, ItPlanS would have the option of using it to accomplish its task.

The OSR's ability to discriminate task-actions by object type allows ItPlanS to use a single task-action without having to consider its particular object. In addition, the reasoning process that refines a motion directive to its specific contextual use, allows the OSR to make discriminations using a finite number of action outlines. Instantiating the current parameters provides a robust low-level planning system, without enumerating each possible plan variation.

7 Related Work

We are not aware of other systems that integrate the kind of search planning and manipulation of objects that we have outlined here. Thus, the comparisons that will be made here are at an abstract level or concern only one facet of our system.

Haas [9] has presented an architecture for a reactive system which engages in search. He argues that there is a conflict between classical planning and the strong representation languages required for reasoning about information acquisition, a claim which we dispute. Our approach is to isolate reasoning about knowledge in the search planner. This allows the complex reasoning about knowledge to take place in a rich descriptive language while allowing the incremental planning method of ItPlanS to effectively control the expansion of the goals produced for the search.

Some of the work on searching graphs and trees is also relevant to our problems in planning searches: specifically any of the work based on searching partially known graphs and trees. For example, Korf [11] has examined application of heuristic search when the entire search tree is not known before a node must be selected. Pemberton and Korf [15] present algorithms for heuristic search on graph spaces, and where only a portion of the graph is available before the agent must commit to an action.

The most obvious work to compare the SodaJack system to might be Georgeff's work on PRS [7, 8]. SodaJack and PRS both interleave planning and action. However while PRS provides a rich formalism for the design of agents, it does not provide solutions for specific problems. In this work we have identified where various problems occur in attempting to build a system to work in our domain, and given concrete solutions for these problems. Thus, while our work could be built on top of the PRS framework, doing so would not change the structure of the system or the general solution methods for the problems we have looked at.

A more interesting parallel is that of the work in case-based planning [10].

While we have presented this work as a hierarchical planner it could be viewed from the case-based perspective as well. As described here, ItPlanS is, in effect, doing nothing more than selecting previously learned plans from a library. While there is no mechanism to repair these plans, the OSR and search planner do provide a method for building plans that are not in the plan library (within their limited context). Thus, while case-based planners have a plan library and a plan repair system, SodaJack has a plan library and experts for building context-sensitive plans.

8 Conclusions

This paper has presented an architecture for agents that need to engage in search and manipulation in their environments. Any system that wishes to perform this task will have to face three significant problems: (1) planning for action, (2) developing goals for the acquisition of knowledge, and (3) tailoring general low-level actions to specific objects.

In SodaJack this has been achieved by augmenting a hierarchical planner with two special purpose planners. The hierarchical planner appeals to the special purpose planners to generate situation and object-specific plans upon demand. We believe that breaking these tasks into separate components allows these problems to be solved in an efficient manner, and the integration of these modules allows the system as a whole to solve problems that any one of the systems working alone would be unable to achieve.

References

[1] Philip Agre. The dynamic structure of everyday life. Technical Report 1085, MIT Artificial Intelligence Laboratory, 1988.

[2] Philip Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of AAAI*, 1987.

[3] Norman Badler, Bonnie Webber, Jeff Esakov, and Jugal Kalita. Animation from instructions. In Norman Badler, Brian A. Barsky, and David Zeltzer, editors, *Making them Move: Mechanics, Control, and Animation of Articulated Figures*. Morgan-Kaufmann, 1991.

[4] Welton Becket and Norman I. Badler. Integrated behavioral agent architecture. 1993 Conference on Computer Generated Forces and Behavior Representation, 1993.

[5] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[6] Christopher Geib. Intentions in means-end planning. Technical Report MS-CIS-92-73, Department of Computer and Information Science, University of Pennsylvania, 1992.

[7] Michael Georgeff and Francois Felix Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of IJCAI*, 1989.

[8] Michael Georgeff and Amy Lansky. Reactive reasoning and planning. In *Proceedings of AAAI*, 1987.

[9] Andrew Haas. Natural language and robot planning. Technical Report 9318, Department of Computer Science, SUNY Albany, 1993.

[10] Kristian Hammond. *Case-Based Planning: Viewing planning as a memory task*. Academic Press, 1989.

[11] Richard E. Korf. Real-time heuristic search. *AIJ*, 43(2-3):197–221, 1990.

[12] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of AAAI*, 1991.

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

[14] Michael B. Moore. Search plans. Technical Report MS-CIS-93-55/LINC LAB 250, Department of Computer and Information Science, University of Pennsylvania, 1993.

[15] Joseph C. Pemberton and Richard E. Korf. Incremental path planning on graphs with cycles. In *Proceedings of AIPS 92*, pages 179–188, 1992.

[16] Mark A. Peot and David E. Smith. Conditional nonlinear planning. In *Proceedings of AIPS*, 1992.

[17] Earl Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[18] Marcel Schoppers. Universal plans of reactive robots in unpredictable environments. In *Proceedings of IJCAI*, 1987.

[19] D. Warren. Generating conditional plans and programs. In *Proceedings of the Summer Conference on AI and the Simulation of Behavior*, Edinburgh, 1976.