CHAPTER 19

Dynamic Abstraction Problem Solving in Soar

A. Unruh, Stanford University, P. S. Rosenbloom, Stanford University, and J. E. Laird, University of Michigan

Abstract

A technique is described for abstracting dynamically during problem solving. Abstraction occurs when the problem solver does not know how to proceed on a task. By searching in abstract spaces, the problem solver gathers information which allows it to proceed on the task. Problem solving in the abstract space is much more efficient than in the original space because many details are ignored. The abstract spaces are themselves dynamically generated during problem solving. This technique has been implemented in the Soar system, which includes a learning mechanism called chunking that creates new rules during the search of the abstract space. Abstraction and learning work together to improve the efficiency of the problem solving. A demonstration is provided in the context of a computer configuration task.

1 Introduction

Abstraction is a powerful tool for controlling the combinatorics of a problem-solving search [7]. Abstract problem solving was first investigated in Planning GPS [12]. Symbolic logic problems were solved in an abstract problem space in which some of the less important aspects of the problem were ignored. The solution path in the abstract space was then used as a sequence of subgoals in the full symbolic-logic problem space. By ignoring the effects of the operators on certain details, a solution in the abstract problem space was much easier to find; yet the plan created from this solution proved effective in controlling the search in the full task space. After Planning GPS, an abstraction-based version of the Strips problem solver [2], called ABStrips, was created that employed multiple levels of abstractions [15]. Problem solving at each level yielded a plan for the problem solving at the next less abstract level. Since that time, abstraction has been used as a technique for generating search evaluations [5,6,13,19], and has been employed by many problem solvers in AI [1,3,4,16,17,18,21].

*This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract N00039-86-C-0133 and by the Sloan Foundation. Computer facilities were partially provided by NIH grant RR-00785 to Sumex-Aim. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the US Government, the Sloan Foundation, or the National Institutes of Health.

In this paper we describe a system which dynamically performs abstract problem solving in order to gather information for decision making. The dynamic aspects of the abstraction occur in two different ways. Firstly, given a problem to be solved, the system does not precommit to solving an abstract version of the problem. Instead, it attempts to directly solve the full problem and resorts to abstraction only in those circumstances in which it does not know how to proceed with the full problem. Specifically, if the system is uncertain about which of several operators to apply next, it performs look-ahead searches in abstract problem spaces in order to gather information about the likely consequences of the operators. Secondly, the abstract versions of the problem spaces are themselves dynamically constructed during the course of the problem solving. Although other problem solvers have dynamically constructed an abstract space by abstracting a restricted subset of the problem features [15], we present a method in which any independent aspect of a task may be dynamically abstracted.

The process of abstracting some part of problem solving can be viewed as the process of dropping constraints about what must be true of the problem solving. The constraints that can be dropped include constraints about what must be true in the goal state, constraints on the operators that can be selected for particular states, etc. Dropping a constraint may in itself make the problem easier to solve. For example, ABStrips abstracted by dropping operator preconditions. This allowed operators to apply when they otherwise wouldn't, reducing the amount of problem solving necessary to establish the preconditions of important operators. However, the process of dropping a constraint by itself may not lead to a simpler problem to solve. For example, if an aspect of the goal test is dropped, the problem solver may still go through the same sequence of problem solving steps to reach a goal state. Under these circumstances, a gain can only be achieved if the problem space is also abstracted. Given the abstracted goal test, some operators may no longer be relevant, or only partially relevant, so they can be dropped from the problem space or themselves abstracted. Likewise, the state may be abstracted by dropping information in it that is not needed for the abstracted goal. Problem solving in the resulting abstracted space may be much simpler. Planning GPS proceeded in exactly this fashion, dropping aspects of the goal test (i.e., differences), which allowed it then to simplify the states and operators, and to eliminate any operators that were now redundant or functionless.

In this paper we concentrate on abstractions in which a portion of the goal test is dropped and the problem space is simplified in some corresponding way. One of the key issues that we address is how the abstract problem space can be constructed dynamically from the full problem space. To do this we employ an approach called *procedural abstraction*, in which abstraction of procedurally encoded operators occurs by partial operator execution. Once the abstraction is specified, only parts of the procedural implementation of the operator get carried out. Neither separate abstract implementations nor declarative representations of the operators are necessary. The key to this behavior is that the original problem space is factored so that whatever can be done will be done, while what can't be done because of an abstraction is simply ignored.

This approach to dynamic abstraction has been implemented in Soar, an architecture for general intelligence [8,9]. One of our research goals is to make this technique available as a general "weak method" in Soar, which could be used to aid problem solving in any task when more domain-intensive information is not available. In the remainder of this paper we give a review of Soar, describe how dynamic abstraction works in Soar; present examples of dynamic abstraction in R1-Soar, a computer configuration system implemented in Soar [14]; and discuss future plans.

2 Review of Soar

Soar is organized around the *Problem Space Hypothesis* [11], which states that all goal-oriented behavior is based on search in problem spaces. The problem space determines the set of legal states and operators that can be used during the processing to attain the goal. The states represent situations. There is an initial state representing the initial situation, and a set of desired states that represent the goal. An operator, when applied to a state in the problem space, yields another state in the problem space. The goal is achieved when one of the desired states is reached as the result of a string of operator applications starting from the initial state.

Goals, problem spaces, states, and operators exist as data structures in Soar's working memory — a short-term declarative memory. Each goal defines a problem solving context — or context for short — which is a data structure in the working memory that contains, in addition to a goal, roles for a problem space, a state, and an operator. Problem solving is driven by the acts of selecting problem spaces, states, and operators for the appropriate roles in the context. Given a goal, a problem space is to be selected. The selection of the problem space is followed by the selection of an initial state, and then of an operator to be applied to the initial state. When a problem space, state, and operator are all selected, the operator is applied to the state to yield a result state. Problem solving within the problem space can proceed with the selection of the result state, and the selection of an operator to apply to the result state. The goal is achieved when a desired state is selected.

Each of the deliberate acts of the Soar architecture — a selection of a problem space, a state or an operator — is accomplished via a two-phase decision cycle. First, during the elaboration phase, the description of the current situation (that is, the contents of working memory) is elaborated with relevant information from Soar's long-term production memory. One important type of knowledge that may be added during the elaboration phase is knowledge of preferences. There is a fixed language of preferences which is used to describe the acceptability and desirability of the alternatives being considered for selection. In the second phase of the decision cycle, the preferences in working memory are interpreted by a fixed decision procedure. If the preferences uniquely specify an object to be selected for a role in a context, then a decision can be made, and the specified object becomes the current value of the role. If the preferences for a decision are either incomplete or inconsistent, then the system does not know how to proceed on the goal, and an impasse occurs in problem solving. When an impasse occurs, a subgoal, with an associated problem solving context, is automatically generated for the task of resolving the impasse. The impasses, and thus their subgoals, vary from problems of selection (of problem spaces, states, and operators) to problems of generation (e.g., operator application).

Given a subgoal, Soar can bring its full problem solving capability and knowledge to bear on resolving the impasse that caused the subgoal. For impasses in operator application, the usual approach is to decompose the problem of applying the operator into a set of simpler operator applications in a sub-problem-space. This leads to a hierarchical structure in which higher-level operators are implemented by combinations of lower-level operators. For impasses in selection, the usual response is to perform a search to determine which of the alternatives is best. A problem space (called the selection space) is used which has an operator (called evaluate-object) which evaluates an alternative. If productions exist to directly implement this operator, preferences can be generated directly from the resulting evaluations. If not, an impasse occurs in operator application and a second-level subgoal is generated within the existing selection subgoal. In this subgoal, the problem of evaluating an alternative — say, an operator — is accomplished in three steps: (1) establishing the evaluation context (selecting the original problem space and state), (2) selecting and applying the operator to be evaluated, and (3) evaluating the resulting

state. If the state can be evaluated, the evaluate-object operator has completed, and another of the alternatives can be evaluated. Otherwise the search continues with the attempt to select an operator for the result state, leading to further subgoals. The resulting look-ahead search terminates when an evaluation for the original operator is generated.

Soar learns by a process of chunking that automatically acquires new productions that summarize the processing in a subgoal. The actions of the new productions are based on the results of the subgoal. The conditions are based on those aspects of the pre-goal situation that were relevant to the determination of those results. In relevantly similar situations in the future, the new production will fire during the elaboration phase, directly producing the required information. No impasse will occur, and problem solving proceeds smoothly. Chunking is thus a form of goal-based caching which avoids redundant future effort by directly producing a result that once required problem solving to determine.

3 Abstraction in Soar

As mentioned in the introduction, the approach we take is to abstract dynamically when operator selection is problematic. At such a time, an abstraction is specified, the resulting abstract problem is solved, and the results of the search are used as the basis for selecting an operator. The ability to exhibit this type of behavior is based on the ability to dynamically create abstract goal tests and problem spaces. This ability is in turn dependent on the structure imposed on the goal tests and problem spaces when they were created. The types of abstractions that can be performed are determined by the way the goal test and the various components of the problem space were *factored* into independent sub-components. Abstractions are specified at problem solving time by deliberately ignoring some sub-components of the goal test and/or the problem space. For example, an abstraction can be specified by removing one of the conjuncts of a conjunctive goal test and one of the sub-components of the problem space's initial state. Each alternative factoring allows certain abstractions to be performed while disallowing others. Flexibility is maximized if the representation is factored as finely as possible.

Once an initial abstraction has been specified, it propagates via procedural abstraction. If a portion of the initial state has been removed, any operator that depends on that information will not be able to fully apply. However, if the operator has been factored into independent sub-components, those sub-components that do not depend on the removed information can still apply. The way the problem space is factored thus also determines how the abstraction propagates. Appropriate propagation of the abstraction is maximized if all of the relevant problem space components are factored: initial state creation, operator precondition testing, operator implementation, state evaluation, and goal testing.

In Soar, the problem space components can all be implemented by either productions or problem solving in a subgoal. Factoring a problem-solving-in-subgoal implementation is straightforward. It is done by factoring the goal test and problem space used in the subgoal. A production implementation can be factored by having an independent production for each of the sub-components. For components that create new states — initial state creation and operator implementation — a production exists which creates a bare symbol representing the new state. Then, whichever sub-component productions can fire, do so in parallel, creating as complete a new state as possible. Each operator precondition is implemented by a production which rejects the operator for selection if the precondition is not met. These productions are completely independent. The remaining two components — goal tests and evaluation functions — have independent productions for each of their subcomponents, but they also require an additional production that can integrate together the resulting information. For a conjunctive goal test, there is a production that tests each conjunct, as well as one which tests whether all of the conjuncts have been successfully tested.

The remainder of this section discusses the five basic steps in the abstraction process: (1) determining when abstraction should be performed, (2) determining what should be abstracted, (3) specifying the abstraction, (4) performing the abstract search, and (5) using the abstract results in the full problem.

3.1 Determining When to Abstract

Systems that have focused on abstraction planning always begin by solving an abstract version of the problem. The approach we have taken in Soar is to begin by trying to solve the full problem, and to only abstract when it is unclear how to proceed with the full problem. That is, abstraction is performed dynamically, when an impasse occurs. In this paper we consider only the case where an operator-tie impasse (a selection impasse between two or more operators) occurs.

3.2 Determining What to Abstract

What *can* be abstracted is a function of the factorization of the goal test and problem space. What *should* be abstracted is a function of task-dependent knowledge. At the moment, we finesse this issue by selecting the initial goal test and problem space abstractions by hand. Following the initial specification of the abstraction, it propagates through the problem space via procedural abstraction.

3.3 Specifying the Abstraction

The goal-test abstraction is specified by creating one or more productions which always assert that the abstracted subcomponents are achieved. The problem-space abstraction is specified in one of three ways. The first way is to drop one of the operators from the problem space. This is accomplished by having a production that rejects the operator for selection during an abstract search. The second way is to drop a portion of the initial state. In the current system, the initial-state creation productions test that an abstraction is not in force before creating an initial state augmentation that should be missing in such an abstraction. (A proposed new version of Soar would implement this process more cleanly by allowing state augmentations as well as operators to be rejected.) The third way is to abstract one of the operators in the problem space. This is currently done only for operators that are implemented by problem solving in subgoals, by abstracting the goal test and problem space in the subgoal.

3.4 Performing the Abstract Search

As described in Section 2, Soar's normal response to an operator-tie impasse is to evaluate the alternative operators by performing a look-ahead search. In the system described here, rather than performing a normal look-ahead search, an abstract look-ahead search is performed. Procedural abstraction allows the abstract search to be pushed through to a successful conclusion in the absence of normally available problem-space information. If the initial abstraction involves ignoring a portion of the state, later operators will be procedurally, and dynamically, abstracted. If the initial abstraction involves ignoring or abstracting an operator, the resulting state will be abstract, and the same type of behavior will appear. If one of the operators is implemented via

problem solving in a subgoal and if the subgoal's initial state can only be partially constructed, the abstraction gets propagated to the subgoal. The abstract problem solving is complete when the abstract goal test succeeds.

3.5 Using the Abstract Information

The results of the abstract search are used to generate evaluations for the operators involved in the operator-tie impasse. These evaluations are then used to generate preferences that will allow one of the operators to be selected. If the abstraction is a good one, the selection will be as good as what would have been done without the abstraction. If it was a bad abstraction, the selection may lead the problem solver astray. As long as there are no sudden-death situations, the system will eventually recover (by search) and reach its ultimate goal.

In addition to using the results of the abstract problem solving directly, chunks are learned to reflect what was done during the abstract search. Some of the chunks generate abstract evaluation information for operators, allowing the system to avoid having to redo the abstract look-ahead searches. Other chunks directly generate preferences about the operators competing for selection. These chunks act as abstract search-control rules that can aid in picking future operators. They can be viewed as encoding an abstract plan that will suggest the same sequence of operators in similar (but not necessarily abstract) situations. The chunks learned during abstraction may be more general than those learned normally because they are not dependent on the details that were abstracted away. The generality and appropriateness of the chunks is a direct reflection of the degree and appropriateness of the abstraction.

Chunks are also learned for the implementation of complex task operators. When operator-implementation chunks are learned during abstraction, they can reduce the amount of work required to perform search in the abstract space. However, if these chunks are applicable in the task space, they may produce an abstract, or incomplete result. It is possible to recover from this incompleteness, but it requires additional work. One approach is to post-process the results of the chunk to add the missing aspects. This results in the learning of a chunk that can do the post-processing in the future. Another approach is to reject the results of the chunk, redo the operator from scratch, and learn a new correct implementation rule. Which approach is more efficient for any particular occurrence of the problem is a task-dependent question.

4 Abstraction in R1-Soar

R1-Soar [14,20] is a re-implementation in Soar of 25% of the functionality of R1 [10], an expert system for configuring computers. This task is particularly interesting for an investigation into abstraction because it is a complex real-world domain, and R1 occasionally used predefined abstract look-ahead searches that were reflected in the original R1-Soar as full-detail searches.

In this work we focus on a portion of R1-Soar's task that involves the configuration of a list of modules into a set of backplanes which are themselves to be configured into boxes. The modules are the functional units that determine the capabilities of the computer, such as CPUs, disk controllers, memory, etc. The backplanes hold the modules, and the boxes contain the backplanes. Figure 1 shows a partial configuration of eight modules, two backplanes, and one box. Each module uses up some amount of several limited resources, including space (in a backplane), power (provided by a box), and others. Modules consist of one or more boards, which have a number of distinguishing characteristics, including pin type and width (either 4 or 6). Each board must be put into a slot of the backplane that is at

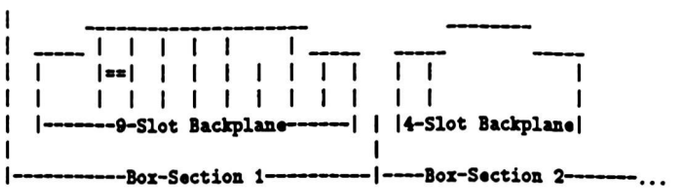

Figure 1: A partial configuration in R1-Soar. The two-board modules have a connecting == in them.

least as wide as the board. A backplane consists of a sequence of slots. Most of the modules go into one of two types of backplane that have the same pin type but differ in the number of slots they have, either 4 or 9. Backplanes are placed in boxes, using up space and other resources.

The input to this task is a sequence of modules to be configured plus a set of backplanes and boxes that the customer has ordered from the manufacturer. More backplanes and boxes can be ordered if necessary, but it is expensive to do so. The configuration task is organized into a hierarchical set of problem spaces and operators. The hierarchy comes about as a result of subgoals created to implement complex high-level operators. The top operator performs the entire subtask of configuring the sequence of modules. A subgoal is created when the operator is selected, in which the operator is implemented by problem solving in a subordinate problem space. This problem space itself contains a general operator to configure a backplane of a particular type, the *configure-a-backplane* operator. In the problem space in which this operator is implemented, suboperators are used to order a backplane of the chosen type (if one is not available), get a backplane of the appropriate type from the order, place the backplane into a box, and fill the backplane with modules. Although control information is added when possible to reduce search, often the choice between operators can not be made without performing a look-ahead search in which operators are tried and the results compared.

A total of three abstractions have been tried so far in R1-Soar. Two of them involve ignoring operators in the problem space that implements the configure-a-backplane operator. The first abstraction is accomplished by ignoring the *put-backplane-in-box* operator. The backplane is never put in the box, with the rest of the configuration proceeding as if this were not necessary. The second abstraction ignores the *put-modules-in-backplane* operator. The third abstraction ignores the fact that the width of the module's boards is of importance when putting them into backplane slots. The three abstractions are not exclusive of each other; it is possible to perform the configuration task while under more than one abstraction.

In the remainder of this section we focus on the workings of one particular illustrative abstraction, the one that involves dropping the put-backplane-in-box operator. Figure 2 shows a simplified representation of the problem solving that occurs both with and without the put-backplane-in-box abstraction. The abstraction has two major effects on problem solving. The first effect is the obvious consequence that the resulting abstract configuration is missing the information linking the backplane to a particular location in a section of one of the boxes. The second effect is that information about the amount of power available for the modules is also missing (recall that power is provided by the box). The put-modules-in-backplane operator normally makes use of this available-power information when deciding which modules it can put

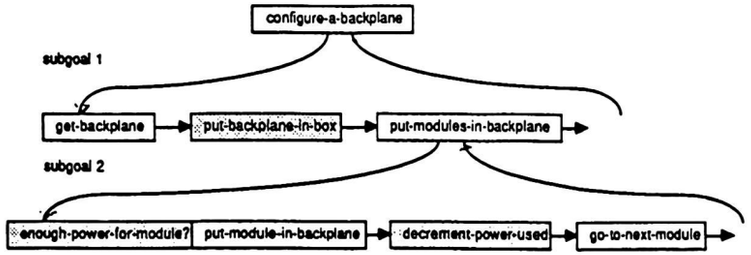

Figure 2: A simplified representation of problem solving with and without the put-backplane-in-box abstraction. Rectangles are operators: a split rectangle indicates an operator precondition and implementation. Shaded actions are not performed under the abstraction.

into the backplane. That is, even if the next module in the series could fit physically into the backplane, it must not be inserted if there is not enough power for it.

When the implementation subgoal for the put-modules-in-backplane operator is entered during the abstract search, an incomplete initial state is created in which the available-power information is missing. After the initial state has been created, the individual modules are inserted in the backplane. One of the preconditions for the put-module-in-backplane operator is that there is sufficient power available for the module. As there is now no knowledge about power, the production which implements the precondition never fires, and the operator is able to proceed without testing the precondition. Following the insertion of a module, the available power would normally be decremented. However, in the abstract case, the power is not available, so the decrement-power operator is not selected. Instead the next module is inserted and the problem solving continues until all modules are in place and the backplane is configured. Thus the problem-solving process becomes abstract; it changes dynamically in response to whatever information is available.

As a result of abstracting away the put-backplane-in-box sub-operator, information about available power and box space is missing in the configure-a-backplane problem space during evaluation and will not contribute to the evaluation function. Other results, such as whether one backplane is more readily available than others (that is, it does not have to be ordered), and the number of modules able to fit into a backplane, can cause one backplane to be favored over another. If the backplane picked turns out to have been incorrect, this will be discovered because all complete configurations are tested for correctness.

Optimality of the operator picked is more difficult to detect than correctness, because R1-Soar contains no explicit test for optimality. (In theory, a "complete" goal test could check for optimality.) In fact, this abstraction may lead to the generation of a configuration that is not quite as good as would have been generated without the abstraction. The configuration resulting from the abstraction can use more box space, since the information about the amount of box space used is not available when abstracting. Thus, there is a real trade-off, at least for this abstraction, between time to solve the problem and quality of the output.

This example demonstrates the fundamental nature of procedural abstraction. Once the put-backplane-in-box operator is rejected, no other explicit mention of an abstraction is necessary

Learning		Abstraction			
		None	Box	Module	Both
	None	879	800*	260	224
	Search Control	760	629*	260	224
	All Goals	353	337	222	198

Table 1: Number of decisions per task for multi-backplane order under varying abstractions and assumptions about what is learned. These figures are averaged; random selection of operators "indifferent" to each other produces varying results.

until the goal is tested. All of the problem solving is able to proceed with whatever information is available, be it abstract or not.

Table 4-1 shows the result of this abstraction, plus several others: the put-modules-in-backplane abstraction, both abstractions together, and no abstraction. The configuration task utilized in these runs is the one shown in Figure 4-1. It turns out that the put-backplane-in-box abstraction is only a mediocre abstraction, producing a savings of about 9%. The put-modules-in-backplane abstraction, though less interesting, provides a better tradeoff between computation time and quality of output because of the relationship between the task and the evaluation function used. It produces a savings of 70%. Both abstractions together generate a 75% savings.

Runs were also made with learning: both search control only, and learning in all subgoals. Learning by itself (with no abstraction) generates a savings of about 60%, while learning with both abstractions produces a savings of 77%. Clearly both abstraction and learning significantly decrease the amount of processing necessary to solve the problem.

One complexity that can arise when doing both learning and abstraction together is that the incomplete operator implementations may cause a tradeoff between not using the abstraction and being able to chunk all of the operators, and using the abstraction to decrease the look-ahead search, but most likely having to recover from overgeneral chunks. In the example of the put-backplane-in-box abstraction, an incomplete implementation is learned for the put-modules-in-backplane operator. Although power information is missing from the current state after the configuration operators have all been applied, there is still a good chance that the configuration is a valid one because usually there is enough power. Therefore, it is possible to use the chunk for part of the operator implementation and fix up those aspects that are missing. This is accomplished by having a goal test that detects that the chunk only did some of the operator implementation. Soar drops back down into the put-modules-in-backplane problem space, detects that the modules are in place, and decrements from the available power the amount used by each module. For this task it is able to use a subset of the operators normally used to configure the modules in this space. That is, it is doing a portion of the normal problem solving in the same manner it did in the absence of power information. What portions are done depends upon what information is in the initial state. If there is enough power, the configuration process proceeds as it would normally from this point. Otherwise the current backplane operator is rejected and another one is attempted in its place.

5 Conclusion

In this article we have described a technique by which problem solving can be dynamically abstracted. The relevant goal tests and problem spaces must be factored into semi-independent sub-components at creation time. Then, whenever a choice between a set of operators is problematic, abstract look-ahead searches are performed in which the alternatives are evaluated. Once the initial abstraction is specified, it propagates throughout the problem space and its subgoals via procedural abstraction. The information gained during this search can help in resolving the problematic decision. The information also forms the basis for the acquisition of new productions which evaluate operators, provide abstract search control knowledge, and partially implement task operators.

This abstraction technique has been implemented in Soar and applied to the complex task of computer configuration. This gets us much of the way towards our goal of turning abstraction into a general weak method that can be used for any problem implemented in Soar, but some problems remain. The most important remaining problem is how to augment the system so that it can automatically select appropriate abstractions. A variety of approaches are possible, including the use of criticality information to determine the relative importance of the problem solving subcomponents, and the use of impasses to signal when information is missing and thus should be ignored. An important related problem is how to ensure that the problem space and goal test are abstracted in compatible fashions. If a portion of the initial state is dropped, all aspects of the goal test which depend on that portion of the state must also be dropped. Otherwise, the goal test may never succeed during the abstract search. This problem can show up both when the system is selecting its own initial abstractions and when problem spaces in subgoals are dynamically abstracted. Another important problem is understanding the relationship between problem space invariants and goal tests. Often a goal test will not need to explicitly test some property of the state because the structure of the problem space guarantees that that property will be true in all states. However, abstracting the problem space may allow the invariant to be violated. When this happens, the goal test becomes underspecified, possibly accepting states that it should not.

We are currently investigating solutions to these problems, and applying dynamic abstraction techniques to different tasks. We are also attempting to demonstrate two of ABStrips's capabilities that have not yet been demonstrated in Soar. The first is the ability to specify abstractions by ignoring operator preconditions; preconditions are procedurally abstracted in the current work, but not yet abstracted directly. The second is the use of multiple levels of abstraction. We have so far demonstrated only the implementation of a single level of abstraction (though the abstraction can propagate through multiple levels of subgoals). The general approach we are taking to multi-level abstraction is to increase the level of abstraction as the level of subgoal increases.

References

[1] D. Chen. Shallow planning and recovery planning based on the vertical decomposition of the flight domain. In *Proceedings of IJCAI-85*, pages 1063–1065, 1985.

[2] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[3] P. Friedland. *Knowledge-Based Experiment Design in Molecular Genetics.* Technical Report STAN-CS-79-771, Stanford University, 1979.

[4] P. E. Friedland and Y. Iwasaki. The concept and implementation of skeletal plans. *Journal of Automated Reasoning*, 1:161–208, 1985.

[5] J. Gaschnig. A problem simililarity approach to devising heuristics. In *Proceedings of IJCAI-79*, pages 301–307, 1979.

[6] D. Kibler. *Generation of heuristics by transforming the problem representation.* Technical Report TR-85-20, ICS, 1985.

[7] R. E. Korf. An analysis of abstraction in problem solving. In *Proceedings of the A.C.M. Technical Symposium on Intelligent Systems*, Association for Computing Machinery, Gaithersburg, MD, 1985.

[8] J. E. Laird. *Soar User's Manual: Version 4.0.* Xerox Palo Alto Research Center, January 1986.

[9] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: an architecture for general intelligence. *Artificial Intelligence*, 33(3), 1987.

[10] J. McDermott. R1: a rule based configurer of computer systems. *Artificial Intelligence*, 19:39–88, 1982.

[11] A. Newell. Reasoning, problem solving and decision processes: the problem space as a fundamental category. In R. Nickerson, editor, *Attention and Performance VIII*, Erlbaum, Hillsdale, NJ, 1980.

[12] A. Newell and H. A. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, 1972.

[13] J. Pearl. On the discovery and generation of certain heuristics. *AI Magazine*, 23–33, 1983.

[14] P. S. Rosenbloom, J. E. Laird, J. McDermott, A. Newell, and E. Orciuch. R1-Soar: an experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(5):561–569, 1985.

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

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

[17] M. Stefik. Planning and meta-planning (molgen: part 2). *Artificial Intelligence*, 16:141–169, 1981.

[18] A. Tate. Generating project networks. In *Proceedings of IJCAI-77*, pages 888–893, 1977.

[19] M. Valtorta. *A Result on the Computational Complexity of Heuristic Estimates for the A* Algorithm*. Technical Report, University of North Carolina, 1981.

[20] A. van de Brug, P. S. Rosenbloom, and A. Newell. *Some Experiments with R1-Soar*. Technical Report, Computer Science Department, Carnegie-Mellon University, 1986. (in preparation).

[21] D. E. Wilkins. Domain-independent planning: representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.

The Soar Papers

1988