

Robo-Soar: An Integration of External Interaction, Planning, and Learning using Soar*

John E. Laird Eric S. Yager Michael Hucka
Christopher M. Tuck
Artificial Intelligence Laboratory
The University of Michigan
Ann Arbor, MI 48109-2110

Draft as of January 16, 1990

Abstract

This chapter reports progress in extending the Soar architecture to tasks that involve interaction with external environments. The tasks are performed using a Puma arm and a camera in a system called Robo-Soar. The tasks require the integration of a variety of capabilities including problem solving with incomplete knowledge, reactivity, planning, guidance from external advice, and learning to improve the efficiency and correctness of problem solving. All of these capabilities are achieved without the addition of special purpose modules or subsystems to Soar.

1 Introduction

An intelligent agent working on multiple goals in a dynamic environment must have many capabilities. It must react quickly to changes in the environment relevant to its goals, use knowledge appropriately, plan, and learn from experience. If it has only incomplete knowledge of the environment, it should be able to accept guidance from other agents when advice is available. Creating a viable agent requires an underlying architecture that can support the integration of these capabilities. Many different approaches to creating such an architecture may be possible, but to date, none has been completely successful. In this chapter we present initial results in an attempt to extend the Soar architecture to tasks that require an integration of these capabilities.

Soar is a general symbolic AI architecture with integrated problem solving and learning [11]. Soar's problem solving and learning have been demonstrated on a large number of tasks including puzzles, computer configuration [19], medical diagnosis[26], natural language understanding [13], production scheduling [9], and algorithm discovery [23]. All of these tasks are solved internally; they do not require interaction with an external environment. This parallels much of the traditional work in AI where the majority of the processing involves the manipulation of internal models and data. Interaction is restricted to the specification of the problem or the delivery of the solution. Interaction with an external environment requires seemingly new capabilities, such

*This research was sponsored by grant NCC2-517 from NASA Ames and ONR grant N00014-88-K-0554.

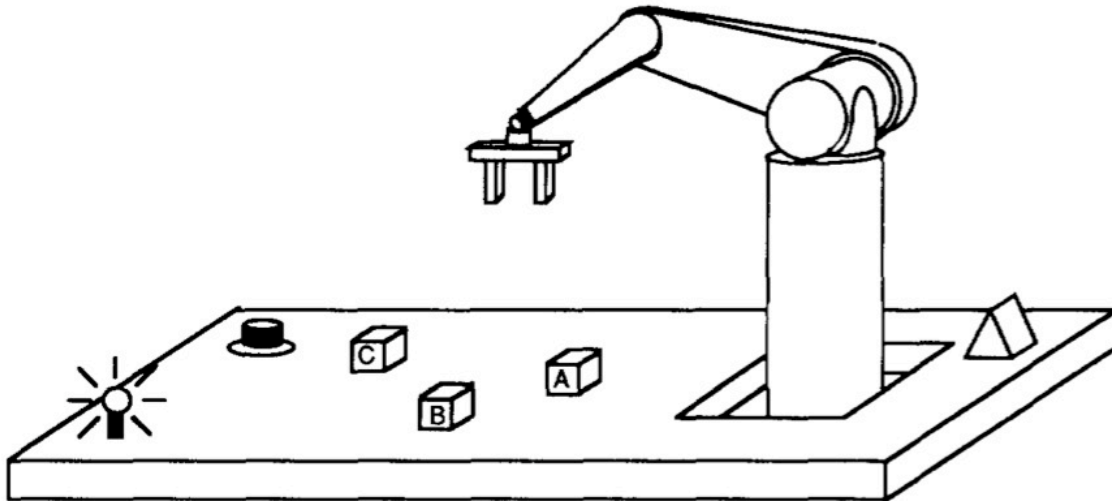


Figure 1: Example block alignment and button pushing task for Robo-Soar.

as problem solving with incomplete and time-dependent sensor data, reacting quickly to changes in the environment, and learning from interactions with the environment.

Robo-Soar is a system implemented in Soar that performs simple block manipulation and button-pushing tasks using a Puma robot arm and a camera-based vision system [12]. The tasks are quite simple, but they serve as a testbed for investigating important issues in external interaction. The setup for the task is shown in Figure 1. The camera is overhead of the blocks and the button. The goal of the robot is to line-up a set of small blocks that have been scattered over the work area while monitoring a light. If the light goes on, the button must be pushed as soon as possible to turn the light off. For the first task, all of the blocks are simple cubes that the gripper can pick up in two different orientations. In the second task, one of the blocks is a triangular prism. To pick up this block, the gripper must be oriented so that it closes over the vertical faces of the block. Robo-Soar starts with knowledge of the individual operators for the robot arm, such as opening and closing the gripper, but it has no control knowledge for deciding when these operators are appropriate, nor does it have knowledge about the special properties of prism blocks. It learns this knowledge from experience and outside guidance.

Soar is unlike previous architectures used for interacting with external environments because of its uniform representation of all long-term knowledge (production rules), its uniform representation of short-term knowledge (symbolic identifier-attribute-value triples), and its single learning mechanism (chunking) that applies to all tasks and subtasks [11]. In contrast, others architectures being used to support intelligent interaction either eschew symbolic representations and models (such as Brooks' "creatures" [3]) or consist of heterogeneous modules, where each capability is supported directly by a separate module [14, 22]. Soar represents an alternative approach, embracing not only the need for a highly reactive system and the need for symbolic representations to support planning, but also stressing the advantages of a uniform architecture and the need for learning.

In Section 2, we present the capabilities that we found necessary for Robo-Soar. Section 3 presents Soar, with an emphasis on how it supports these capabilities compared to other archi-

tures. Section 4 presents the basic knowledge encoded in Robo-Soar. The presentation of Robo-Soar is followed by demonstrations of the capabilities presented in Section 2. The final section is a discussion of our plans for future work.

2 Robo-Soar Capabilities

Below is a list of the capabilities in Robo-Soar. In some cases it would have been possible to engineer the domain to eliminate some of the problems that arose (such as adding force sensors to the gripper to eliminate any ambiguity as to whether a block had been grasped). We avoided these temptations and attacked the problems as they arose in Robo-Soar.

This list by no means exhausts the capabilities required in all cases of external interaction, but it includes all the capabilities that were required in Robo-Soar. Some of the capabilities that might be required in other applications include reasoning about time, reacting in bounded time, processing large amounts of sensor data, handling noisy sensor data, and learning internal models of the environment. Therefore, although Robo-Soar demonstrates the integration of some aspects of external interaction, planning, and learning, it is far from demonstrating all of the capabilities that are necessary for general external interaction.

2.1 Problem solving with incomplete perception

Because of limitations in a system's sensors, reasoning must be done with only limited knowledge of its environment. In Robo-Soar, the perception of the world is limited to that which is available from the camera mounted directly above the workspace; there are no sensors on the gripper to detect if it is holding a block. Further, the camera's view is obscured whenever the arm is used to pickup a block or push the button. Only by withdrawing the arm from the work area after attempting to pick up a block can Robo-Soar determine if it has actually grasped it.

2.2 Problem solving with delayed perception

When interacting with the real world, the results of actions are detected only after time delays. For a completely internal problem, all actions are instantaneous and there is never the need to wait for actions to complete or sensors to deliver feedback. Many symbolic AI systems are designed with this assumption. In Robo-Soar, some delay arises because it takes time to move the arm and the gripper. These delays are overshadowed by the vision system which takes approximately 5 seconds to process an image.

2.3 Planning

Many tasks involving external action do not require planning. With sufficient knowledge of the environment and the tasks to be performed, it is enough to simply react. This is preferred, but it is not always possible in novel situations where the ability to plan out a solution can avoid time-consuming and possibly dangerous explorations of the environment. A search of an internal model of the problem can often proceed orders of magnitude faster than performing the same search in the external environment, without the costs and risks.

Although aligning blocks is a seemingly simple task, the search space is rather large if the system does not have any pre-encoded control knowledge. Planning does require an internal

model of the environment in addition to the ability to execute the corresponding actions in the environment. To reduce the planning time, the internal model can be abstract, not including all of the details of the real problem [25]. For example, when issuing commands to the robot arm, actual real-valued coordinates must be used to specify new positions. While planning, the system only needs to consider symbolic relations, such as “right-of” or “left-of”, to create a plan.

Planning can be improved further through the use of a hierarchical decomposition of the task. In Robo-Soar, when more than two blocks are to be aligned, there is a straightforward hierarchical decomposition of the task into two levels. The top level consists of the tasks of aligning blocks, which can be decomposed into aligning pairs of blocks. The second level consists of the individual movements of the arm necessary to bring one block into alignment with another.

2.4 Learning from external guidance

When an intelligent agent has little or no knowledge about how to proceed on a task, it can try to obtain that knowledge either through search and exploration or through advice from another agent. The ability to obtain knowledge without expensive search can improve the agent’s knowledge much faster than if it were forced to solve the problem on its own. Conversely, if a system can work independently, it will not always be dependent on external guidance.

In Robo-Soar, even with hierarchical planning, the search required to discover a successful solution is quite large (for the simplest two block problem, the branching factor on the solution path varies between 2 and 8 and the depth is 8). To make this search tractable, Robo-Soar can accept advice from a human during its search. This requires the ability to identify when knowledge is needed as well as provide the necessary problem solving context for the human so that supplying the knowledge is easy. We must avoid requiring the human to provide some program or a plan for solving the complete problem. It must also be possible for the system to proceed without advice, either when a human is unwilling or unavailable. If guidance is available, Robo-Soar will utilize its own planning ability together with the advice to quickly plan a solution. Following the completion of planning, Robo-Soar executes the action that actually perform the task. As a side-effect of planning, Robo-Soar learns search control knowledge for selecting the correct operators on the path to the solution. This control knowledge guides Robo-Soar to the solution for the current problem. In future problems it eliminates the need for external guidance.

2.5 Interruption and reactivity

In an internally represented problem, the environment is completely under the control of the problem solver and there are no unexpected changes. Thus, once a plan is created for such a problem, its execution is not problematic. When working with external tasks, unexpected changes in the environment may invalidate existing plans or interrupt the current goal. Robo-Soar must be able to react quickly to changes, as well as interrupt the current task if a more important task arises. Robo-Soar’s first task is to arrange blocks, and the second task is to push the button whenever a light goes on. While working on the first task, the blocks can be moved at any time by some external agent, either to help or hinder Robo-Soar. Also during this task, the light may go on at anytime, requiring the system abandon the block task and immediately push the button.

2.6 Improve efficiency and reactivity

For sufficiently complex tasks, it is impossible to completely prepare a system for all possible problems it might have to solve, and therefore, we include planning and problem solving as key capabilities. However, once a system has planned a solution or solved a problem, learning can convert that experience to knowledge so that next time, planning and problem solving are not necessary. This allows the system to improve the efficiency with which it solves similar problems, possibly transferring partial solutions from previous problems. Learning can also make the system more reactive, that is, transform previously deliberate decisions into reflex behavior [2, 7]. This is related to improving efficiency, but differs in two ways. First, it is the elimination of deliberation, not just its reduction. Second, the results of learning must be able to interrupt ongoing processes as well as be interruptable themselves.

2.7 Improve correctness

An intelligent system must be able to correct and extend its knowledge when it encounters inconsistencies between its internal knowledge and the way the world works. In Robo-Soar, the system initially learns to manipulate cubic blocks. It is then presented with a prism that to the overhead camera is identical to the cubic blocks except for the edge line down the middle of block (see Figure 1). Its first attempt to pick up this block can fail because it will not correctly align the gripper. The problem is that it does not understand the significance of this feature. Once an error has been made, it must learn the implications of this feature for its future actions, specifically, that the gripper must be aligned so that it is orthogonal to the line. In Robo-Soar this is achieved through a combination of automatic analysis and guidance from a human.

3 System Architecture

The integration of these new capabilities was achieved using the basic structure of Soar without any additional modules designed specifically to aid interaction. Some changes to the architecture were required; however, these changes affect all tasks implemented in Soar and they maintain its uniform approaches to knowledge representation and learning [?].

Figure 2 shows the basic Soar architecture on which Robo-Soar is built. Input comes from a camera mounted directly above the work area. A separate computer processes the images, providing asynchronous input to the rest of the system. The vision processing system extracts the position and orientation of each block in the work area as well as distinctive features, such as an identifying block number. Soar accepts new visual information whenever it arrives.

In Soar, a task is solved by searching through a problem space of possible states, applying operators to transform an initial state to some desired state. For the block manipulation task, the states are different configurations of the blocks and gripper in the external environment. Some basic operators are shown in the trace of Robo-Soar solving a simple block manipulation problem in Figure 3. These operators correspond to commands sent to the robot controller. Robo-Soar solves a problem by selecting and applying operators until it achieves the goal. When an operator is selected, motor commands are sent to the Puma controller and executed. Two operators, **snap-in** and **snap-out**, move the arm in and out of the work area so that the camera has an unobstructed view. These operators are necessary, but for simplicity, they will not be included in any of the examples.

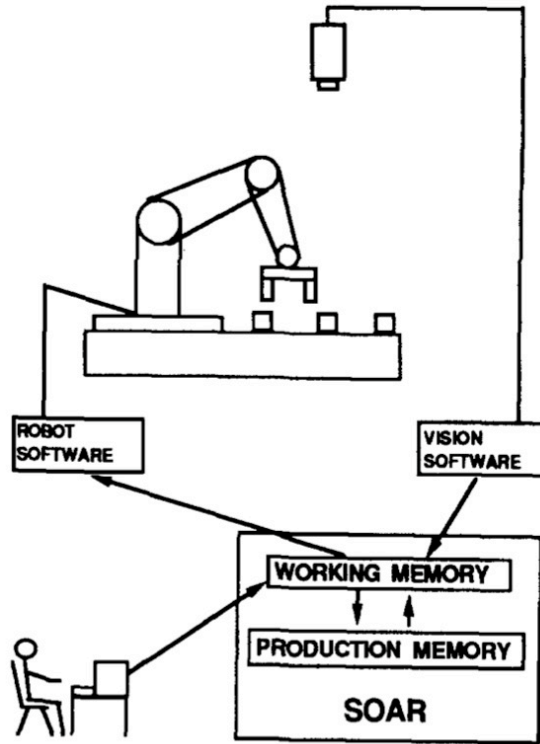


Figure 2: Robo-Soar system architecture.

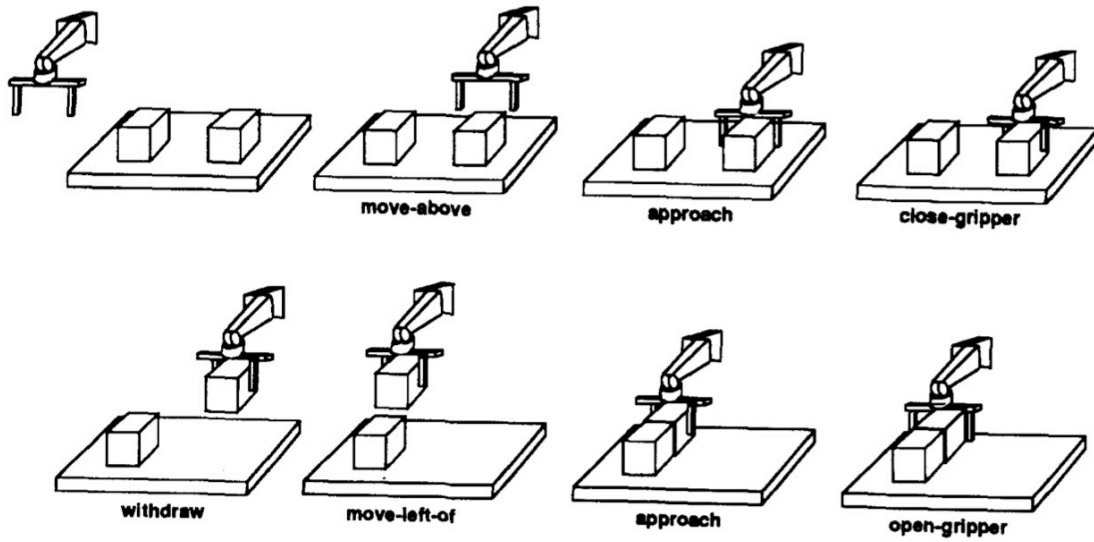


Figure 3: Moving a block using the primitive operators for block manipulation task.

This characterization of Robo-Soar does not distinguish it from any other robot controller. What is different is the way Soar makes the decisions to select an operator for a familiar task. Many AI or robotic systems create a plan of actions that the robot must execute. Instead of creating a plan, Soar makes each decision based on a consideration of its long-term knowledge, its perception of the environment, and its own internal goals. As shown in Figure 2, Soar’s long-term knowledge is represented as productions (condition-action rules) that are continually matched against *working memory*. Working memory contains data about the current situation, including all active goals, problem spaces, states, and operators as well as all input and output.

In contrast to traditional production systems such as OPS5, Soar fires all successfully matched productions in parallel, allowing them to elaborate the current situation, or create *preferences* for the next action to be taken. There is a fixed preference language that allows productions to assert that operators are acceptable, not acceptable, better than other operators, as good as others, and so on. Production firing continues until *quiescence* is reached (no additional productions match) so that chains of monotonic inference are possible. Following quiescence, Soar examines the preferences and selects the best operator for the given situation (possibly maintaining the current operator if its actions have not yet completed). Some productions act as bottom-up recognizers, parsing incoming data in parallel and building up symbolic descriptions. Based on these descriptions, other productions create acceptable preferences for the operators that are possible in the current situation. Other productions compare these descriptions to the goals of the system and create preferences to select from the set of available operators.

To encode a task in Soar, productions must be added to long-term memory that create preferences for appropriate problem spaces, states and operators for the goals the system is to achieve. Soar starts with a *base goal* that represents its basic drives. As will be described later, Soar can have a stack of goals, with new goals being created when progress cannot be made for the most recently created goal. For each of these goals, a problem space is selected, followed by an initial state from which problem solving proceeds through the application of operators.

In a familiar domain, Soar’s knowledge is adequate to pick and apply an appropriate operator without further problem solving. However, when Soar’s preferences do not determine a best choice or when it is unable to implement the selected operator directly, an *impasse* arises and Soar automatically generates a subgoal. In the subgoal, Soar casts the problem of resolving the impasse as a search through a problem space and uses its production memory to control the search when possible. The operators in the subgoal can modify or query the environment, or they may operate internally, possibly simulating external operators on internal representations. Soar’s production memory provides knowledge for selecting operators during the search as well as for implementing the actions of the internal operators. Impasses can arise while selecting or implementing internal operators so that a hierarchy of subgoals may be created dynamically. Within these subgoals, Soar can request advice from a human to help it determine the solution; if advice is not available, it will search for a solution on its own. It is Soar’s ability to automatically generate subgoals, and thereby plan, that distinguishes it from other robot controllers and reactive systems.

When Soar creates *results* in its subgoals, it learns productions, called *chunks*, that summarize the processing that occurred. The actions of a chunk are based on the results of the subgoal. The conditions are based on the working-memory elements that were tested in the subgoal in order to derive the results. This technique is similar to explanation-based learning [5, 15]. Although the technique is similar to EBL, EBL systems learn a plan or schema of actions. The plan is

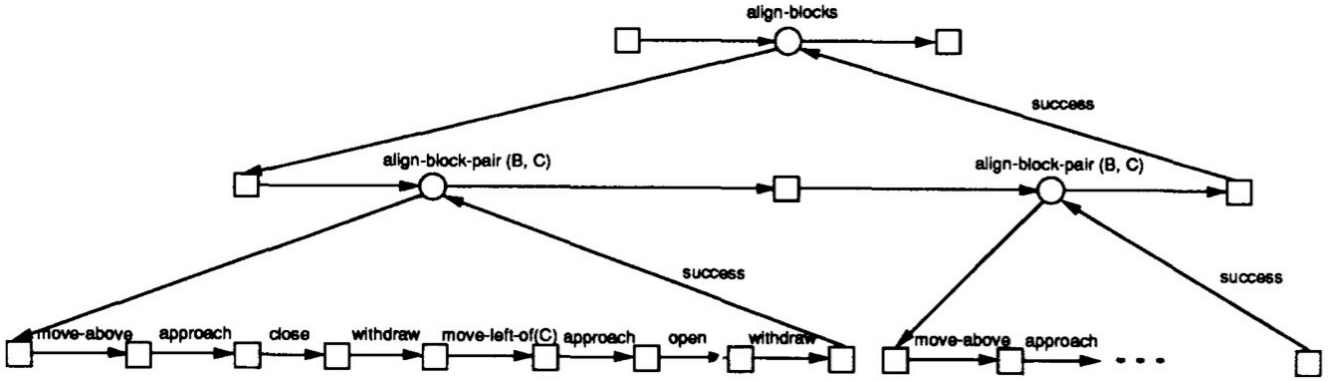


Figure 4: Trace of operator implementation goal hierarchy for aligning three blocks. Horizontal arcs represent operator applications, while squares represent states. A horizontal arc with a circle represents an operator that could not be applied directly, but resulted in an impasse. Downward pointing arcs are used to represent the creation of subgoals, and upward pointing arcs represent the termination of subgoals.

used to control problem solving in similar goals in the future. Soar’s approach is different. The production learned for a subgoal does not provide control information for future problem solving in that same subgoal. Instead, the chunk encapsulates the processing in the subgoal, eliminating the impasse that gave rise to the subgoal. Control knowledge for a goal is learned from subgoals created to select between competing operators.

4 Robo-Soar

To apply Soar to a task, some initial knowledge about the problem spaces and operators for the task must be encoded in Soar as productions. Robo-Soar’s initial problem space consists of two operators: **align-blocks**, and **turn-out-light**. In other AI systems, these might be encoded directly as goals; however, in Soar, deliberate goals such as these are represented as operators. As operators, they will lead to impasses because they are not implemented directly in productions, and instead require additional problem solving to be applied. As with primitive operators, such as opening or closing the gripper, the preference scheme provides a method for selecting between these more goal-like operators. For example, the **align-blocks** operator is created whenever blocks are out of alignment, and the **turn-out-light** operator is created whenever the light turns on. If both operators are available, a production creates a preference to prefer **turn-out-light** because of the urgency of its action. Therefore, Robo-Soar uses the preference scheme to control the selection of multiple “goals”, providing it with the ability to interrupt one goal whenever a more important goal arises.

We will now trace through the problem solving in Robo-Soar when the light is out and there are three blocks out of alignment as in Figure 1. As a first step, the **align-blocks** operator is created and selected. Figure 4 provides a trace of the problem solving to implement this operator.

The selection of the **align-blocks** operator leads immediately to an impasse and the creation of a subgoal because there are no productions that can implement the operator directly. Within this subgoal, the problem of aligning the three blocks can be considered. Instead of immediately

considering the robot arm commands necessary to move the gripper to pick up a block, the problem can be decomposed into aligning pairs of blocks, such as A and B, and then aligning block C with B. In Robo-Soar, the problem space for implementing the **align-blocks** operator consists of just these operators. In other systems, these would be normally considered conjunctive subgoals, but Soar represents them as operators called **align-block-pair**.

For a three block problem, there are six possible **align-block-pair** operators: align A with B, align B with A, align A with C, align C with A, align B with C, and align C with B. Productions can be added to the system to create preferences that will select between the competing **align-block-pair** operators so that the blocks will be properly aligned. If such preferences are not available, which would be the case if this is a novel task, a tie-impasse would arise and problem solving would be required to make the decision. We will return to this problem later, but for now, we will assume that there exists a production that can create preferences to select the appropriate **align-block-pair** operators, in this case **align-block-pair(B,C)** followed by **align-block-pair(A,B)**. Once an **align-block-pair** operator has been selected, another subgoal will be generated to implement it. Finally, in this subgoal, Robo-Soar uses a problem space, called **Puma-arm-commands**, with operators that issue commands to the robot arm.

The **Puma-arm-commands** problem space is where Robo-Soar must interact with the outside world. The states of this problem space must provide access to the perceptual data of the sensors. This is possible even in a subgoal because the state of the **Puma-arm-commands** problem space is the same as the state in the supergoal. Operators in the **Puma-arm-commands** problem space create commands for the motor system and receive input data. As these commands are issued and the perceptions change, the states in the higher problem space also change; however, their operators stay selected. New operators will be selected only when additional productions fire that signal that the current operator has terminated. For example, there is a production for the **align-blocks** operator that tests that all of the blocks being aligned, and then terminates the operator. Similarly, there is a production that tests if the appropriate pair of blocks are aligned and then terminates the **align-block-pair** operator.

To have a reactive system, the productions must be sensitive to the environment. Unfortunately, the data received from the sensors is often not complete enough to be the basis for proposing and selecting operators. The robot arm can obscure blocks, and feedback is not received when the robot uses its gripper. Therefore, the system must maintain an independent representation of the outside world. For the most part this is created by copying information directly from the sensors using productions; however, in those cases when the arm moves into the work area, this representation is maintained independently of the feedback from the sensors, that is, the productions do not remove blocks from the internal representation when they disappear while the arm is in the work area. In addition, if the system closes the gripper around a block, it creates an expectation that the block is being held. This expectation is later verified when the arm is removed from the work area by testing that the block is indeed gone.

A second problem with our sensor system is that it takes approximately 5 seconds to process an image. Once the arm has withdrawn from the work area, the system must wait for feedback before it knows whether it was successful in picking up a block. Waiting arises automatically in Robo-Soar by including a test for new data in the conditions of the operator that snaps the arm out of the work area. Once this operator is selected, the command to snap out is issued. However, since the **snap-out** operator is not finished, a subgoal arises to implement the operator. The only operator in the problem space is **wait**. The **wait** operator is continually selected until new input

arrives, leading to the termination of the snap-out operator and continued problem solving. By making the waiting explicit, the productions in Robo-Soar are always active so that the system can respond as soon as there is input available.

Although Robo-Soar can handle some of the problems that arise from using real sensors, it avoids many others. For example, the vision and robotic systems are accurate enough so that there is never any uncertainty in placing the gripper in the correct location to pick up a block. For a task in which more accuracy is required, difficult issues in problem solving and learning under uncertainty would have to be addressed (see Bennett (1989) for one approach to learning under uncertainty).

5 Planning and Learning using External Advice

With just the basic knowledge for proposing and implementing the operators described above, and no additional control knowledge, Robo-Soar will attempt the task of aligning the blocks, decomposing the problem into subproblems. However, lacking control knowledge, it will have to search for the solution. In this section we describe how Robo-Soar plans using outside guidance and predefined abstractions. The results of planning are saved by chunking so that future problems are solved without any external advice. This combination of planning, advice and learning has been demonstrated for other Soar systems [8]; our current implementation is a straightforward extension of those earlier techniques to an external environment.

We now return to the three block problem described earlier. Planning is necessary at two levels. First, Robo-Soar must sequence the alignment of the pairs of blocks. Second, once it has decided to align two blocks, it must issue the robot commands that actually move one block next to another. For most traditional AI systems, planning is the primary method for solving problems. The first step is to construct a plan, and then execute the plan in the environment. During execution, the plan and the environment are monitored to ensure that the plan is still appropriate. If unexpected changes occur in the environment, then the system will either replan, or attempt to find another plan that is relevant for the current situation.

In Soar, planning is used only when there is insufficient knowledge to make the next decision, which is signalled by a tie impasse. In our current example, a tie impasse will arise whenever an **align-block-pair** operator is to be selected, as shown in Figure 5. We will call this a tie between *task* operators, to distinguish these from operators used in supporting problem spaces. Robo-Soar will perform a limited look-ahead search using an internal model of the task to determine the correct sequence of operator selections. The knowledge to control this search is encoded in Soar as productions and is part of the default knowledge available for every task encoded in Soar.

Planning arises as a side-effect of Soar's default method for choosing between tied operators. In response to the tie impasse, Soar uses the **selection** problem space, which is used to evaluate and compare the tied operators, ultimately creating preferences that resolve the tie. The tied operators are evaluated by creating multiple instantiations of an operator, called **evaluate-object**, whose purpose is to evaluate the utility of a task operator. In this example, six **evaluate-object** operators are created, one for each task operator.

Robo-Soar's planning is a relatively straightforward look-ahead search. This is sufficient to demonstrate the integration of multiple capabilities in a single system. More sophisticated planning techniques would require more knowledge about the task, specifically for the **selection** problem space and the **evaluate-object** operators.

blocks. An alternative is for the system execute an abstract simulation of the **align-block-pair** operator. This simulation can be used in the look-ahead, when the details of moving the gripper can be ignored. This is easily implemented in Soar through a production that tests if the **align-block-pair** operator is selected, and the current state is a simulation, and whose action is to modify the abstract relations between the objects in the scene.

If the resulting internal state achieves the goal, the operator is evaluated as being “best.” If no evaluation is produced, the search continues in order to determine if another operator can lead Soar to the goal. This results in another impasse, and as should be evident, a depth-first search is performed recursively. In our example, only two **align-block-pair** operators are required to achieve the goal. Once the goal is achieved within the internal search, preferences are created to select the operators that were selected along the path to the solution.¹

Each of the preferences is a result of a subgoal, and productions are built that summarize the processing that led to their creation. That is, for ties between the task operators, productions were learned to select the appropriate operator for the current state. Therefore, these productions act as an implicit and distributed plan, applying whenever their conditions are satisfied.

Because of the creation of the preferences, the original tie-impasse is resolved and **align-block-pair(B,C)** is selected. This time, the operator is not applied to an internal copy, but rather to the state representing the current situation in the external world. The production that provided an abstract simulation of it does not fire, and a subgoal is created to implement the operator, as shown in Figure 6.

In the subgoal to implement **align-block-pair(B,C)**, there is once again a tie to decide which robot command to issue. As before, the **selection** and **advice** problem spaces are used, and a depth-first look-ahead search is performed. Now the search is at the level of the individual robot commands; however, it is done using abstract relations, such as that the gripper is to the right of a block. When a sequence of operators is found that implements **align-block-pair**, the tie subgoals terminate, and the first operator in the sequence is selected.

As the result of the look-ahead, chunks are learned for each decision on the path to implementing the **align-block-pair** operator. These apply one after the other, not because of any explicit sequencing, but because their conditions are sensitive to the appropriate relations and properties of the state. Figure 7 is an example of the production that is learned for the **approach** operator. Notice that it not only tests aspects of the current situation, but also aspects of the goal, in this case implementing the **align-block-pair** operator. The productions learned from this search are quite general and do not include any tests of the exact positions or names of the blocks. These features are not included because they were not tested in the subgoal. An important observation about chunking is that the generality of the learning is closely tied to the generality of the goals and the knowledge used to achieve the goals.

Following the look-ahead search and the accompanying learning, the system is able to directly solve the problem and similar problems with different initial block configurations. In fact, once it has learned the productions to control the selection of operators during implementation of the first **align-block-pair**, it directly implements the second without requiring search or advice. It has learned the appropriate operator to apply at each decision point in the search. However, this is not a blind application of a plan. Each of the productions that was learned will test aspects of the environment to insure that they are used only when appropriate. If the production for one step in the solution is not required, possibly because of an unexpected change in the environment,

¹If the search leads to illegal or unacceptable states, a preference is created to avoid the responsible operator.

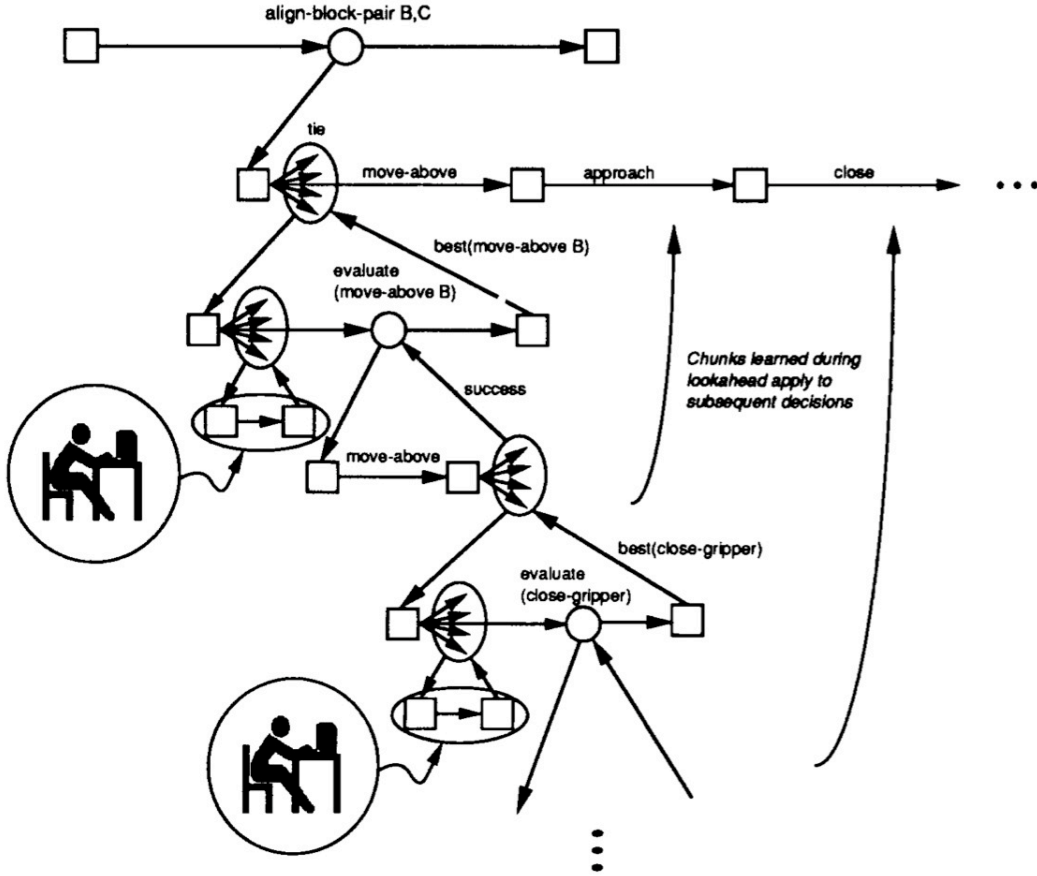


Figure 6: Trace of problem solving for applying `align-block-pair` operator.

If the `approach` operator is applicable, and
the gripper is holding nothing, in the safe plane above a block,
and that block must be moved to implement the goal-operator,
then create a best preference for the `approach` operator.

Figure 7: Example production learned by Robo-Soar.

the production for the following step will fire, skipping ahead. If an operator has an unexpected result, Robo-Soar will not continue with the learned sequence of operators, but will respond using a production from some previously learned plan or fall into an impasse in which planning can be used to compute the appropriate response.

In Soar, chunks can be learned for all subgoals, even those to implement operators such as `align-block-pair`. While applying `align-block-pair` in a subgoal, chunking converts the deliberate application of the external operators, such as `open-gripper`, into productions that issue robot commands directly when `align-block-pair` is selected in the future. Unfortunately, in some cases, the current implementation of chunking creates overgeneral productions, and thus, for these demonstrations, chunking was used to learn only control knowledge.

Robo-Soar’s learning is an extension of earlier robot programming systems where a human leads the system through a fixed set of commands to perform a task, and the exact command sequence is stored for later use. This earlier approach was quite limited because the command sequence included specific arm movements without any generalization or conditionality. More recently, “learning apprentices” have been developed that create generalized plans using a technique called explanation-based learning (EBL) [5, 16].

The ARMS system developed by Alberto Segre used EBL to learn generalized plans for simulated manipulator control [21]. The input to the system was the sequence of manipulator moves necessary to construct a specific example of a simple object, such as a revolute joint. Through analysis of the sequence and its own underlying theory of the domain, the system was able to learn general plans that would be independent of the specific example. Our approach uses the same general method, although it extends Segre’s work to an external task where the operators are organized hierarchically and where the external guidance is limited to local advice, as opposed to a complete solution. Robo-Soar has its own planning capabilities so that it has the potential for solving the problem if only limited advice is available. Our approach leads to more transfer between plans as well as increased reactivity because Robo-Soar learns individual control productions for each decision. Segre’s approach has the virtue that plan interpretation is extremely fast because the sequencing of actions is represented explicitly.

6 Interruption and Reactivity

During the block manipulation task, Robo-Soar must push a button to turn out the light as soon as the light goes on. It must do this if it is in the middle of planning the block manipulation task, deep in subgoals that involve look-ahead search. It must do this if it is the middle of executing the task, with the problem partially solved. It must do this if it is waiting for advice from the outside. Once it has pushed the button, it must be able to return to the original task with as little disruption as possible.

Achieving this interruptability is relatively easy in Robo-Soar because of the reactive nature of productions and Soar’s decision procedure. Robo-Soar has a production that continually monitors the vision input, testing if the light comes on. The action of this production is to create an operator, **turn-off-light**, and a “require” preference for it; that is, this operator must be selected. If the light goes on, the production fires and the **turn-off-light** operator is created. Following quiescence, the decision procedure selects the **turn-off-light** operator, which is then applied. In this case, the application is performed in a subgoal where the sequence of actions is executed. When the application is completed, the **turn-off-light** operator is removed from working memory and work on the original task can proceed.

Important details were left out of the above description. First, in which goal in the subgoal hierarchy is **turn-off-light** to be proposed? If it is proposed and selected at the very top, all impasses below it will be resolved (the unfinished operators will be terminated by the new selection) and will have to be rebuilt after the interruption is processed. On the other hand, if **turn-off-light** is proposed and selected at the most recent subgoal, it may be applied to only an internal model of the environment created for look-ahead. To avoid both of these problems, Robo-Soar proposes the **turn-off-light** operator in the most recent subgoal that has the state with the inputs from the environment as its current state. When the **turn-off-light** operator is then selected, it is applied externally. Its selection eliminates the existing tie impasses (and all

subgoals within these impasse) that arose because of uncertainty about which task operator to apply next. Therefore, the interruption will destroy any unresolved progress made in the look-ahead. This is appropriate because there is no way to judge how much of this look-ahead will be invalidated in the handling of the interruption.

When the interrupting operator is completed, productions will fire to decide on the next operator to apply. This may again cause a tie and the system will return to planning. If there is already sufficient knowledge to select a single task operator, performance of the original task will proceed immediately.

Although Soar directly supports interruptability, there are limits to its responsiveness. First, it is limited by the time it takes to match and fire productions. For the Robo-Soar task, where the vision system has delays of up to five seconds, Soar is more than fast enough. However, if Soar were applied to a task where reaction time was on the order of milli-seconds, faster implementations of the production system would be required. By exploiting parallelism, advanced production system architectures have proven to be sufficiently efficient to match and fire productions at these rates [24].

The second limitation is that it cannot interrupt the firing of productions. A new operator can only be selected at quiescence. As long as quiescence is achieved quickly, as it is in Robo-Soar, this is not a problem. However, if the environment is changing rapidly, in ways that are tested by productions, there could be no bound on the time it takes to react. Currently, Soar has a fixed upper bound on the time it will take to match productions before making a decision. If the upper bound is reached, a decision will be made without waiting for quiescence. This does not completely destroy the validity of the decision, it just forces the system to make a decision with incomplete information. This is a rather crude approach, but has been sufficient for the tasks currently implemented in Soar.

The third limitation to interruptability is that the current Soar architecture is implemented on a uniprocessor that must match productions, make decisions and build chunks. During the time the system is chunking, it is not matching productions, and will therefore be unresponsive to changes in the environment. In our current implementation, chunking comprises no more than 30 % of the execution time; however, it may be possible to eliminate this overhead by implementing chunking on a parallel co-processor.

7 Refining Incorrect Knowledge

A problem with the traditional learning apprentice approaches that have been used for teaching computers is that the learning is only as good as the underlying knowledge. The system uses existing knowledge to generalize from new experiences, but it never modifies the original knowledge. If there is an error in the original knowledge, the human has no avenue available for communicating corrections. This is a general problem with deductive learning techniques such as EBL. Although one could argue that errors in the original knowledge can be avoided through careful coding, the same problem can arise when an existing system encounters a problem outside its original specification.

We explored a simple case of this problem by attempting the same block alignment task with a block shaped as a triangular prism. If the original operators were implemented with only cubes in mind, all of the control knowledge and underlying simulation would not be sensitive to the fact that there is another feature in the input that must be attended to. To the Robo-Soar vision

system, the prisms look just like cubes, except for a line down the middle at the apex of the triangle. In order to pick up these blocks, the gripper must be aligned with the vertical faces of the block, not just any two sides. If it is not correctly aligned, the gripper will close, but upon withdrawing the gripper, the block will not be picked up.

There are many possible approaches to this problem. First, the system could have an underlying theory of inclined planes, grippers, friction, etc. that it uses to understand why the block was not picked up [6]. This approach merely pushes the problem down one level and requires the addition of knowledge that is often difficult to obtain. A second approach is to gather examples of failure and use inductive learning techniques to hypothesize which feature in the environment was responsible for the problem [17]. This may identify the feature, but it requires many failures and also gives no hint as to the appropriate action. A third approach is for the system to experiment with its available operators to see what actually works [4]. This approach can be quite effective, but it also can be quite time consuming and possibly dangerous. A fourth approach is to manually reprogram the robot. This requires skilled programmers and may be difficult if the robot is remotely located. A fifth approach involves increasing the interaction between the human and the robot so that the human can point out relevant features in the environment and associate them with the potential success or failure of a given operator or set of operators.

We have selected that last approach. This approach builds on previous work in Soar on recovery from incorrect knowledge [10]. In Soar, recovery is complicated by the fact that chunking is the only learning mechanism, and it only adds knowledge to long-term memory, never modifying existing productions. In our original work, we demonstrated that it was possible to learn new productions that created preferences to correct decisions.

The first step in this approach is to notice that an incorrect decision has been made. When Robo-Soar attempts the problem with the prism, the productions it learned on the first task leads it through the first four panels of Figure 8. It correctly moves the gripper above the block. At this point it then approaches the block, closes the gripper, withdraws the gripper and the moves the gripper out of the way of the camera. Robo-Soar creates an explicit expectation in working memory that the block will be moved when it receives a new image of the work area. If this expectation is violated, productions detect the the violation and create an operator, called **detect-error**, that marks that an error has occurred.

The error arose because Robo-Soar’s internal knowledge is incorrect; none of its operators, such as **approach** or **close**, is sensitive to the orientation of the prism. If it tried to recover by simply opening the gripper, it will once again try to approach the block, using its incorrect knowledge.

To avoid repeating the incorrect actions, domain-independent knowledge is included that attempts to recover from errors. The general strategy is to force the system to reconsider each decision and accept guidance from outside. As before, the guidance includes suggestions of appropriate actions, but this time it may include suggestions as to what are inappropriate actions as well as suggestions as to when to avoid an inappropriate action. The remaining paragraphs of this section go through the details of this approach to recovery.

Reconsideration is implemented by forcing impasses for every decision once an error has been detected. These impasses are forced by creating an operator called **deliberate-impasse**, together with preferences that make it both better and worse than the other operators. This is guaranteed to force an impasse and allows for the reconsideration of the decision in the resulting subgoal. In our example, the first action to take is to open the gripper. We skip over the process for

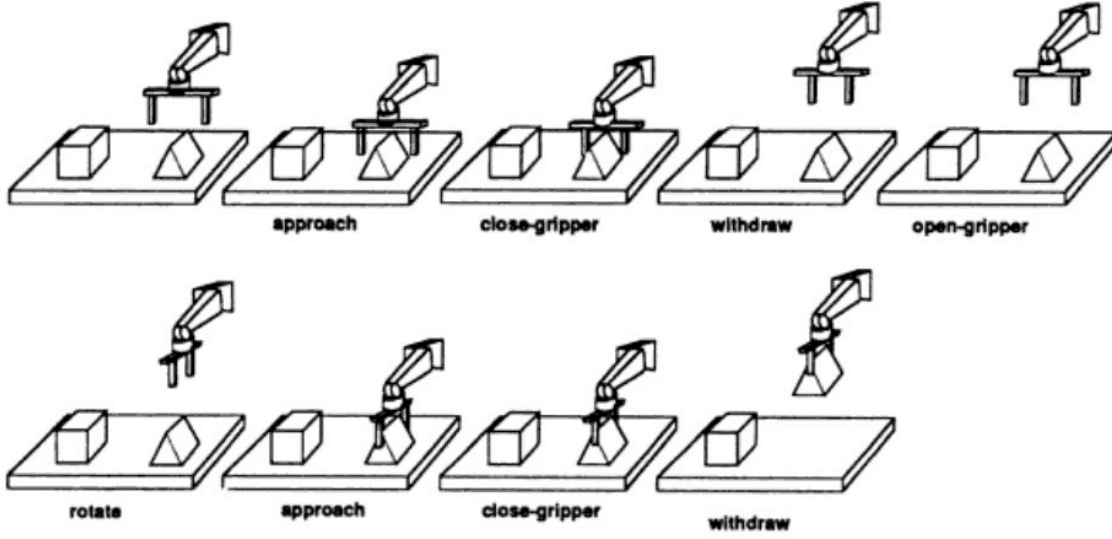


Figure 8: Trace of operator sequence using recovery.

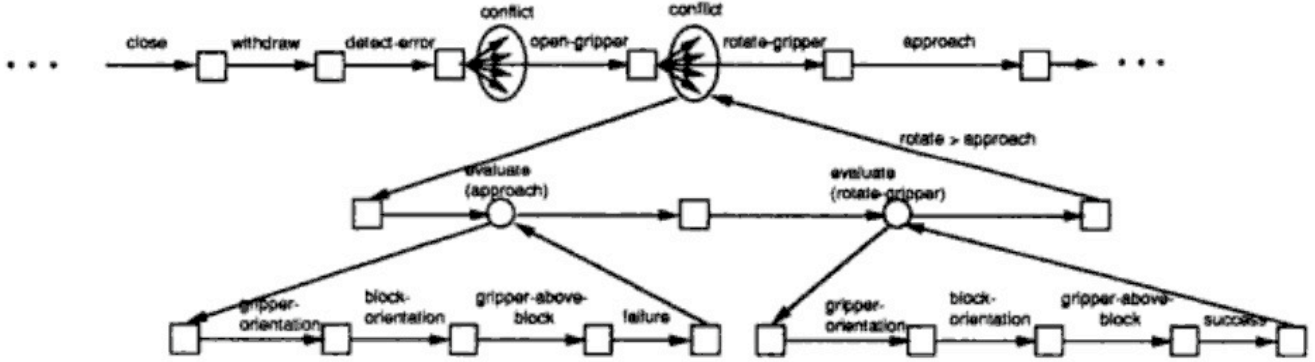


Figure 9: Trace of problem solving with recovery, omitting the advice problem space.

doing that and go to the situation after the gripper is open as shown in Figure 9. A chunk from the previous problem creates a “best” preference for **approach**. However, the conflict impasse prevents its selection and allows reconsideration.

Within the conflict subgoal, the **selection** space is chosen and internal operators are created to evaluate all of the external operators that were legal for the original state; this is the same as when an ordinary tie is encountered. At this point the human should direct the system to evaluate both the correct action (**rotate-gripper**) and the incorrect action (**approach**). The order in which they are evaluated does not matter, but both must be evaluated so that a preference can be created making **approach** “worse” than **rotate-gripper**. This preference will lead to the correct selection. For this example, we will assume that the human requested that **approach** be evaluated first.

Once an evaluation operator for **approach** is selected, any evaluation computed by a produc-

```

If the approach operator is applicable, and
  the gripper is above a block, and the gripper's orientation
  is different from a line in the middle of the block,
  and the rotate operator is available,
then create a preference that rotate is better than approach.

```

Figure 10: Example production learned by Robo-Soar.

tion is rejected because of a potential for error. Therefore, a subgoal will arise to compute the evaluation. In contrast to earlier look-ahead searches, this evaluation is going to be based on input from the human. The human will point out the features of the state that are relevant to determining the appropriateness of the operator being evaluated for the current situation, as well as provide the evaluation.

To compute this evaluation, a new problem space, called **examine-state** is selected. Its selection is predicated on the fact that an error has been encountered. The problem space consists of operators that examine and compare the features of the state, as well as operators that evaluate the state as being on the path to success or failure. Through interactions with a human, operators are chosen to select and test the appropriate features and evaluate the task operator. In this case, **approach** is evaluated as leading to failure following the examination of the orientation of the gripper and the block it is above. Following the evaluation of **approach**, **rotate-gripper** is evaluated, with the appropriate features of the state being tested before it is judged to be on the path to success. Following the evaluation of **rotate-gripper** the production in Figure 10 is learned that prevents selection of **approach** whenever the gripper is not aligned.

After **rotate-gripper** is selected, another conflict impasse arises, but in this case the human signals that the problem has been fixed, and the error signal is eliminated from working memory. From this point, the chunks from the original problem apply and take Robo-Soar to the solution. When Robo-Soar encounters future tasks involving prism blocks, it immediately aligns the gripper before approaching a prism, avoiding the error.

The examine-state problem space is somewhat of a brute-force technique to learn new features. It requires an outside agent to lead the system through a search of potentially relevant features. Although it may not be considered the most elegant machine learning technique, it allows the human to easily correct the system. In addition, this same approach can be used without an outside agent by having the system engage in experimentation. To experiment, the system can guess at relevant features. It will often pay attention to irrelevant features, and thus create overgeneral chunks. But after many interactions with its environment, it will learn to ignore irrelevant features. If this search was a completely blind one, it would be similar to the search through hypothesis space performed by empirical learning techniques. One way to view the examine-state problem space is as an empirical learning method implemented on top of a deductive learning system.

Of course, the search for relevant features does not have to be blind. Many powerful heuristics are available, such as concentrating on new, unknown features, as well as those features that are modified by the operators under consideration. For example, if the system has discovered that the rotate operator is necessary, it could concentrate its search for features relevant to avoiding **approach** to those modified by **rotate**. Carbonell and Gil [4] have proposed more deliberate experimentation techniques that would be applicable for this task. Rajamoney and DeJong [18]

have described a more elaborate approach to experimentation that is used to learn theories of physical devices.

8 Discussion

Though our actual task was quite simple, Robo-Soar's ability to learn to solve these problems demonstrates the integration of several capabilities in a single system. Our current goal is to extend Robo-Soar to more complex tasks, and to more types of knowledge acquisition. Currently, Robo-Soar is pre-programmed with a large amount of task specific knowledge including the decomposition of the task into problem spaces, the appropriate level for abstraction, and the knowledge to simulate internally its external actions. To expand the knowledge acquisition capabilities, we will consider both increased human interaction and increased automated learning. To increase human interaction, we plan to investigate increasing the modes of communication, so that the user can interrupt Soar at any time to give advice that can not only correct control knowledge but also add problem spaces and operators. One of the original inspirations of the Soar project has always been to create an Instructable Production System where the system is never programmed, only given high-level advice [20]. We are also studying experimentation techniques so that Robo-Soar will be able to learn much of the same information on its own, when human advice is unavailable.

9 Acknowledgments

We would like to thank Karen McMahon for implementing Soar 5, and Mark Wiesmeyer for developing and implementing the Soar input and output interfaces. Without these extensions to Soar, Robo-Soar would not have been possible. We also like to thank Mike Walker and Joe Dionese for help with the Puma robot control system and Terry Weymouth for help with the vision system. Finally, we thank Paul Rosenbloom and Allen Newell for ideas relating to this work.

References

- [1] S.W. Bennett. Learning approximate plans for use in the real world. In *Proceedings of the Sixth International Machine Learning Workshop*, pages 224–228, Cornell, NY, June 1989. Morgan Kaufmann Publishers.
- [2] J. Blythe and T. M. Mitchell. On becoming reactive. In *Proceedings of the Sixth International Machine Learning Workshop*, pages 255–259, Cornell, NY, June 1989. Morgan Kaufmann Publishers.
- [3] R. A. Brooks. Intelligence without representation. In *Proceedings of the Workshop on the Foundations of Artificial Intelligence*. MIT, June 1987.
- [4] J. C. Carbonell and Y. Gil. Learning by experimentation. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 256–266, 1987.

- [5] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.
- [6] R. Doyle. Constructing and refining causal explanations from an inconsistent domain theory. In *Proceedings of AAAI-86*. Morgan Kaufmann, 1986.
- [7] M.T. Gervasio and G.F. DeJong. Explanation-based learning of reactive operators. In *Proceedings of the Sixth International Machine Learning Workshop*, pages 252–254, Ithaca, NY, June 1989. Morgan Kaufmann Publishers.
- [8] A. Golding, P. S. Rosenbloom, and J. E. Laird. Learning general search control from outside guidance. In *Proceedings of IJCAI-87*, Milano, Italy, August 1987.
- [9] W. Hsu, M. Prietula, and D. Steier. Merl-Soar: Scheduling within a general architecture for intelligence. In *Proceedings of the Third International Conference on Expert Systems and the Leading Edge Production and Operations Management*, May 1989.
- [10] J. E. Laird. Recovery from incorrect knowledge in Soar. In *Proceedings of the AAAI-88*, August 1988.
- [11] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(3), 1987.
- [12] J. E. Laird, E. S. Yager, C. M. Tuck, and M. Hucka. Learning in tele-autonomous systems using Soar. In *Proceedings of the NASA Conference on Space Telerobotics*, pages 415–424, January 1989.
- [13] R.L. Lewis, A. Newell, and T.A. Polk. Toward a Soar theory of taking instructions for immediate reasoning tasks. In *Proceedings of the Annual Conference of the Cognitive Science Society*, pages 514–521, August 1989.
- [14] S. Minton, J. G. Carbonell, C.A. Knoblock, D.R. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40(1–3):163–118, 1989.
- [15] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1, 1986.
- [16] T. M. Mitchell, S. Mahadevan, and L. I. Steinberg. LEAP: A learning apprentice for VLSI design. In *Proceedings of IJCAI-85*, pages 616–623, Los Angeles, CA, August 1985.
- [17] M. Pazzani, M. Dyer, and M. Flowers. The role of prior causal theories in generalization. In *Proceedings of AAAI-86*, Philadelphia, PA, 1986. American Association for Artificial Intelligence, Morgan Kaufmann.
- [18] S. Rajamoney and G. DeJong. Active explanation reduction: An approach to the multiple explanations problem. In J. Laird, editor, *Proceedings of Fifth International Conference on Machine Learning*, pages 242–255, Ann Arbor, MI, 1988. Morgan Kaufmann.

- [19] 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.
- [20] M. D. Rychener. The instructable production system: A retrospective analysis. In *Machine Learning: An Artificial Intelligence Approach*. Tioga, Palo Alto, CA, 1983.
- [21] A. M. Segre. *Explanation-Based Learning of Generalized Robot Assembly Plans*. PhD thesis, University of Illinois at Urbana-Champaign, 1987.
- [22] P.F. Spelt, G. de Saussure, E. Lyness, F. G. Pin, and C. R. Weisbin. Learning by an autonomous robot at a process control panel. *IEEE Expert*, 4(4):8–16, 1989.
- [23] D. M. Steier. Cypress-Soar: A case study in search and learning in algorithm design. In *Proceedings of IJCAI-87*, Milano, Italy, August 1987. Morgan Kaufmann.
- [24] M. Tambe, D. Kalp, A. Gupta, C.L. Forgy, B.G. Milnes, and A. Newell. Soar/PSM-E: Investigating match parallelism in a learning production system. In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with applications, languages, and systems*, pages 146–160, July 1988.
- [25] A. Unruh, P. S. Rosenbloom, and J. E. Laird. Dynamic abstraction problem solving in Soar. In *Proceedings of the Third Annual Aerospace Applications of Artificial Intelligence Conference*, pages 245–256, Dayton, OH, 1987.
- [26] R. Washington and P. S. Rosenbloom. Applying problem solving and learning to diagnosis. Computer Science Department, Stanford University., 1989.