# Integrating Execution, Planning, and Learning in Soar for External Environments*

**John E. Laird**
Artificial Intelligence Laboratory
The University of Michigan
1101 Beal Ave.
Ann Arbor, MI 48109-2110
laird@caen.engin.umich.edu

**Paul S. Rosenbloom**
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
rosenbloom@isi.edu

## Abstract

Three key components of an autonomous intelligent system are planning, execution, and learning. This paper describes how the Soar architecture supports planning, execution, and learning in unpredictable and dynamic environments. The tight integration of these components provides reactive execution, hierarchical execution, interruption, on demand planning, and the conversion of deliberate planning to reaction. These capabilities are demonstrated on two robotic systems controlled by Soar, one using a Puma robot arm and an overhead camera, the second using a small mobile robot with an arm.

## Introduction

The architecture of an intelligent agent that interacts with an external environment has often been decomposed into a set of cooperating processes including planning, execution and learning. Few AI systems since STRIPS [Fikes *et al.*, 1972] have included all of these processes. Instead, the emphasis has often been on individual components, or pairs of components, such as planning and execution, or planning and learning. Recently, a few systems have been implemented that incorporate planning, execution, and learning [Blythe & Mitchell, 1989; Hammond, 1989; Langley *et al.*, 1989].

Soar [Laird *et al.*, 1987] is one such system. It tightly couples problem solving and learning in every task it attempts to execute. Problem solving is used to find a solution path, which the learning mechanism generalizes and stores as a plan in long-term memory. The generalized plan can then be retrieved and used during execution of the task (or on later problems). This basic approach has been demonstrated in Soar on a large number of tasks [Rosenbloom *et al.*, 1990]; however, all of these demonstrations are essentially internal — both planning and execution occur completely within the scope of the system. Thus they do not involve direct execution in a real external environment and they safely ignore many of the issues inherent to such environments.

Recently, Soar has been extended so that it can interact with external environments [Laird *et al.*, 1990b]. What may be surprising is that Soar's basic structure already supports many of the capabilities necessary to interact with external environments — reactive execution, hierarchical execution, interruption, on demand planning, and the conversion of deliberate planning to reaction.

In this paper, we present the integrated approach to planning, execution, and learning embodied by the Soar architecture. We focus on the aspects of Soar that support effective performance in unpredictable environments in which perception can be uncertain and incomplete. Soar's approach to interaction with external environments is distinguished by the following three characteristics:

1. Planning and execution share the same architecture and knowledge bases. This provides strong constraints on the design of the architecture — the reactive capabilities required by execution must also be adequate for planning — and eliminates the need to explicitly transfer knowledge between planning and execution.
2. External actions can be controlled at three levels, from high-speed reflexes, to deliberate selection, to unrestricted planning and problem solving.
3. Learning automatically converts planning activity into control knowledge and reflexes for reactive execution.

Throughout this presentation we demonstrate these capabilities using two systems. The first is called *Robo-Soar* [Laird *et al.*, 1989; Laird *et al.*, 1990a]. Robo-Soar controls a Puma robot arm using a camera vision system as shown in Figure 1. The vision system provides the position and orientation of blocks in the robot's work area, as well as the status of a trouble light. Robo-Soar's task is to align blocks in its work area, unless the light goes on, in which case it must immedi-

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

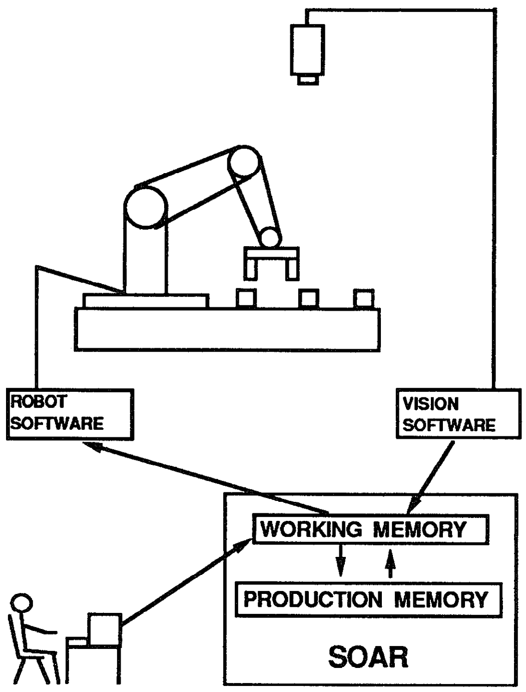

Figure 1: Robo-Soar system architecture.

ately push a button. The environment for Robo-Soar is unpredictable because the light can go on at any time, and an outside agent may intervene at any time by moving blocks in the work area, either helping or hindering Robo-Soar's efforts to align the blocks. In addition, Robo-Soar's perception of the environment is incomplete because the robot arm occludes the vision system while a block is being grasped. There is no feedback as to whether a block has been picked up until the arm is moved out of the work area.

The second system, called *Hero-Soar*, controls a Hero 2000 robot. The Hero 2000 is a mobile robot with an arm for picking up objects and sonar sensors for detecting objects in the environment. Hero-Soar's task is to pick up cups and deposit them in a waste basket. Our initial demonstrations of Soar will use Robo-Soar. At the end of the paper we will return to Hero-Soar and describe it more fully.

## Execution

In Soar, all deliberate activity takes place within the context of goals or subgoals. A goal (or subgoal) is attempted by selecting and applying operators to transform an initial state into intermediate states until a desired state of the goal is reached. For Robo-Soar, one goal that arises is to align the blocks in the work area. A subgoal is to align a pair of blocks. Within a goal, the first decision is the selection of a problem space. The problem space determines the set of operators that are available in a goal. In Robo-Soar, the problem space for manipulating the arm consists of operators such as `open-gripper` and `move-gripper`.

The second decision selects the initial state of the problem space. For goals requiring interaction with an external environment, the states include data from the system sensors, as well as internally computed elaborations of this data. In Robo-Soar, the states include the position and orientation of all visible blocks and the gripper, their relative positions, and hypotheses about the positions of occluded blocks. Once the initial state is selected, decisions are made to select operators, one after another, until the goal is achieved.

Every decision made by Soar, be it to select a problem space, initial state, or operator for a goal, is based on *preferences* retrieved from Soar's long-term production memory. A preference is an absolute or relative statement of the worth of a specific object for a specific decision. The simplest preference, called *acceptable,* means that an object should be considered for a decision. Other preferences help distinguish between the acceptable objects. For example, a preference in Robo-Soar might be that it is better to select operator `move-gripper` than operator `close-gripper`.

A preference is only considered for a decision if it has been retrieved from the long-term production memory. Productions are continually matched against a working memory — which contains the active goals and their associated problem spaces, states, and operators — and when matched, create preferences for specific decisions. For example, a production in Robo-Soar that proposes the `close-gripper` operator might be:

```
If the problem space is robot-arm and
   the gripper is open and surrounds a block
then create an acceptable preference
   for the close gripper operator.
```

Once an operator is proposed with an acceptable preference, it becomes a candidate for selection. The selection of operators is controlled by productions that create preferences for candidate operators. For example, the following production prefers opening the gripper over moving a block that is in place.

```
If the goal is to move block A next to block B and
   the problem space is robot-arm and
   block A is next to block B and
   the gripper is closed and surrounds block A
then create a preference that opening the gripper
   is better than withdrawing the gripper.
```

Arbitrary control knowledge can be encoding as productions so that Soar is not constrained to any fixed method. The exact method is a result of a synthesis of all available control knowledge [Laird *et al.*, 1986].

Soar's production memory is unusual in that it fires all matched production instantiations in parallel, and it *retracts* the actions of production instantiations that no longer match, as in a JTMS [Doyle, 1979].[1] Thus,

[1]Retraction in Soar was introduced in version 5. Earlier versions of Soar did not retract the actions of productions.

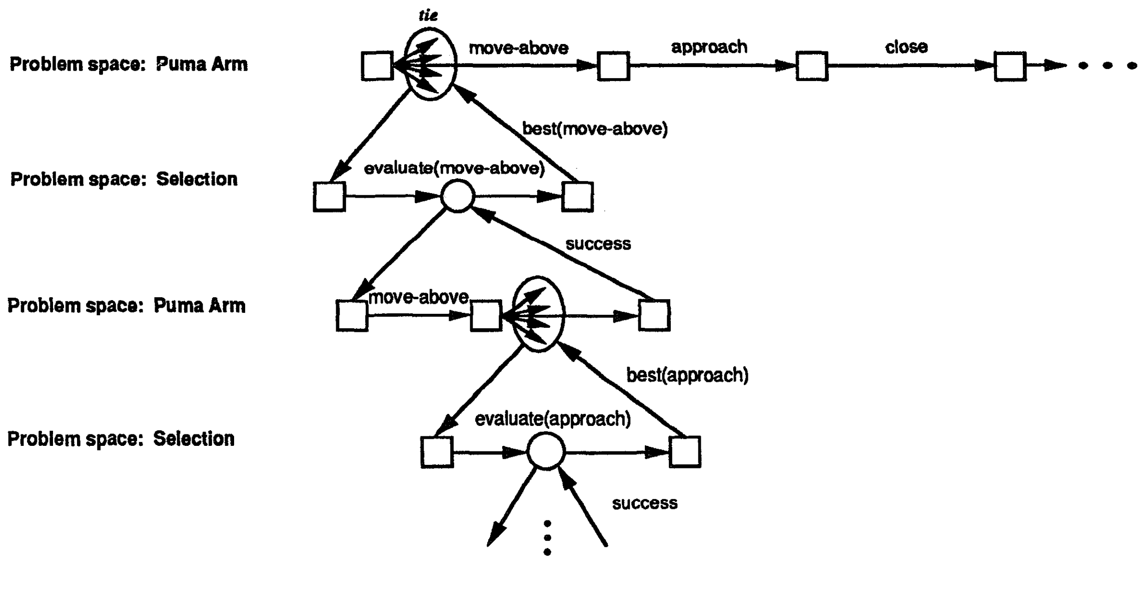

Figure 2: Example of planning in Robo-Soar to move a block. Squares represent states, while horizontal arcs represent operator applications. Downward pointing arcs are used to represent the creation of subgoals, and upward pointing arcs represent the termination of subgoals and the creation of results.

sufficient preferences have been created to allow the decision procedure to make a single choice, the subgoal is automatically terminated and the appropriate selection is made.

If there is more than a single point of indecision on the path to the goal, then it is necessary to create a longer term plan. If other decisions are underdetermined, then they will also lead to impasses and associated subgoals during the look-ahead search. The result is a recursive application of the planning strategy to each decision in the search where the current knowledge is insufficient.

Figure 2 shows a trace of the problem solving for Robo-Soar as it does look-ahead for moving a single block. At the left of the figure, the system is faced with an indecision as to which Puma command should used first. In the ensuing impasse, it performs a look-ahead search to find a sequence of Puma commands that pickup and move the block. Because of the size of the search space, Robo-Soar uses guidance from a human to determine which operators it should evaluate first [Laird *et al.*, 1989]. When a solution is found, preferences are created to make each of the decisions that required a subgoal, such as `best(approach)` and `best(move-above)` in the figure. Unfortunately, these preferences cannot directly serve as a plan because they are associated with specific planning subgoals that were created for the look-ahead search. These preferences are removed from working memory when their associated subgoals are terminated.

At this point, Soar's learning mechanism, called *chunking*, comes into play to preserve the control knowledge that was produced in the subgoals. Chunking is based on the observation that: (1) an impasse arises because of a lack of directly available knowledge, and (2) problem solving in the associated subgoal produces new information that is available to resolve the impasse. Chunking caches the processing of the subgoal by creating a production whose actions recreate the results of the subgoal. The conditions of the production are based on those working-memory elements in parent goals that were tested by productions in the subgoal and found necessary to produce the results. This is a process very similar to explanation-based learning [Rosenbloom & Laird, 1986].

When chunking is used in conjunction with the planning scheme described above, Robo-Soar learns new productions that create preferences for operators. Since the preferences were created by a search for a solution to the task, the new productions include all of the relevant tests of the current situation that are necessary to achieve the task. Chunking creates new productions not only for the original operator decision, but also for each decision that had an impasse in a subgoal. As a result, productions are learned that create sufficient preferences for making each decision along the path to the goal. Once the original impasse is resolved, the productions learned during planning will apply, creating sufficient preferences to select each operator on the path to the goal. This is shown in Figure 2 as the straight line of operator applications across the top of figure after the planning is complete.

In Robo-Soar, the productions learned for aligning blocks are very general. They ignore all of the details of the specific blocks because the planning was done using a abstract problem space. Similarly, the productions preferences and working memory elements exist only when they are relevant to the current situation as dictated by the conditions of the productions that created them. For example, there may be many productions that create preferences under different situations for a given operator.

Once the relevant preferences have been created by productions, a fixed decision procedure uses the preferences created by productions to select the current problem space, the initial state, and operators. The decision procedure is invoked when Soar's production memory reaches quiescence, that is, when there are no new changes to working memory.

Once an operator is selected, productions sensitive to that operator can fire to implement the operator's actions. Operator implementation productions do not retract their actions when they no longer match. By nature they make changes to the state that must persist until explicitly changed by other operators. For an internal operator, the productions modify the current state. For an operator involving interaction with an external environment, the productions augment the current state with appropriate motor commands. The Soar architecture detects these augmentations and sends them directly to the robot controller. For both internal and external operators, there is an additional production that tests that the operator was successfully applied and signals that the operator has terminated so that a new operator can be selected. The exact nature of the test is dependent on the operator and may involve testing both internal data structures and feedback from sensors.

At this point, the basic execution level of Soar has been defined. This differs from the execution level of most systems in that each control decision is made through the run-time integration of long-term knowledge. Most planning systems build a plan, and follow it step by step, never opening up the individual decisions to global long-term knowledge. Other "reactive" learning systems, such as Theo [Blythe & Mitchell, 1989; Mitchell *et al.*, 1990] and Schoppers' Universal plans [Schoppers, 1986] create stimulus-response rules that do not allow the integration at run-time of control knowledge. Soar extends this notion of run-time combination to its operator implementations as well, so that an operator is not defined declaratively as in STRIPS. This will be expanded later to include both more reflexive and more deliberate execution.

## Planning

In Soar, operator selection is the basic control act for which planning can provide additional knowledge. For situations in which Soar has sufficient knowledge, the preferences created for each operator decision will lead to the selection of a single operator. Once the operator is selected, productions will apply it by making appropriate changes to the state. However, for many situations, the knowledge encoded as productions will be incomplete or inconsistent. We call such an underdetermined decision an *impasse*. For example, an impasse will arise when the preferences for selecting operators do not suggest a unique best choice. The Soar architecture detects impasses and automatically creates subgoals to determine the best choice. Within a subgoal, Soar once again casts the problem within a problem space, but this time the goal is to determine which operator to select. Within the subgoal, additional impasses may arise, leading to a goal stack. The impasse is resolved, and the subgoal terminated, when sufficient preferences have been added to working memory so that a decision can be made.

To determine the best operator, any number of methods can be used in the subgoal, such as drawing analogies to previous problems, asking an outside agent, or various planning strategies. In Soar, the selection of a problem space for the goal determines which approach will be taken, so that depending on the available knowledge, many different approaches are possible. This distinguishes Soar from many other systems that use only a single planning technique to generate control knowledge.

Robo-Soar uses an abstract look-ahead planning strategy. Look-ahead planning requires additional domain knowledge, specifically, the ability to simulate the actions of external operators on the internal model of the world. As expected, this knowledge is encoded as productions that directly modify the internal state when an operator is selected to apply to it.

The internal simulations of operators do not replicate the behavior of the environment exactly, but are abstractions. In Robo-Soar, these abstractions are predetermined by the productions that implement the operators, although in other work in Soar abstractions have been generated automatically based on ignoring impasses that arise during the look-ahead search [Unruh & Rosenbloom, 1989]. For Robo-Soar, an abstract plan is created to align a set of blocks by moving one block at a time. This level completely ignores moving the gripper and grasping blocks. This plan is later refined to movements of the gripper by further planning once the first block movement has been determined. Even this level is abstract in that it does not simulate exact sensor values (such as block A is at location 3.4, 5.5) but only relative positions of blocks and the gripper (block A is to the right of block B).

Planning in Robo-Soar is performed by creating an internal model of the environment and then evaluating the result of applying alternative operators using available domain knowledge. The exact nature of the search is dependent on the available knowledge. For some tasks, it may be possible to evaluate the result of a single operator, but for other tasks, such as Robo-Soar, evaluation may be possible only after applying many operators until a desired (of failed) state is achieved. Planning knowledge converts the evaluations computed in the search into preferences. When learned for moving the gripper ignore the exact names and positions of the blocks, but are sensitive to the final relative positions of the blocks.

The ramifications of this approach to planning are as follows:

1. **Planning without monolithic plans.**
   In classical planning, the plan is a monolithic data structure that provides communication between the planner and the execution module. In Soar, a monolithic declarative plan is not created, but instead a set of control productions are learned that jointly direct execution. The plan consists of the preferences stored in these control rules, and the rule conditions which determine when the preferences are applicable.
2. **Expressive planning language.**
   The expressibility of Soar's plan language is a function of: (1) the fine-grained conditionality provided by embedding the control knowledge in a set of rules; and (2) the preference language. The first factor makes it easy to encode such control structures as conditionals, loops, and recursion. The second factor makes it easy to not only directly suggest the appropriate operator to select, but also to suggest that an operator be avoided, or that a partial order holds among a set of operators. This differs from systems that use stimulus-response rules in which the actions are commands to the motor system [Mitchell *et al.*, 1990; Schoppers, 1986]. In Soar, the actions of the productions are preferences that contribute to the decision as to which operator to select. Thus Soar has a wider vocabulary for expressing control knowledge than these other systems.
3. **On-demand planning.**
   Soar invokes planning whenever knowledge is insufficient for making a decision and it terminates planning as soon as sufficient knowledge is found. Because of this, planning is always in service of execution. Also because of this, planning and replanning are indistinguishable activities. Both are initiated because of indecision, and both provide knowledge that resolves the indecision.
4. **Learning improves future execution and planning.**
   Once a control production is learned, it can be used for future problems that match its conditions. These productions improve both execution and planning by eliminating indecision in both external and internal problem solving. The effect is not unlike the utilization of previous cases in case-based reasoning [Hammond, 1989]. This is in contrast to other planning systems that build "situated control rules" for providing reactive execution of the current plan, but do not generalize or store them for future goals [Drummond, 1989].
5. **Run-time combination of multiple plans.**
   When a new situation is encountered, all relevant productions will fire. It makes no difference in which previous problem the productions were learned. For a novel problem, it is possible to have productions from many different plans contribute to the selection of operators on the solution path (unlike case-based reasoning). For those aspects of the problem not covered by what has been learned from previous problems, on-demand planning is available to fill in the gaps.

It is this last observation that is probably most important for planning in uncertain and unpredictable environment. By not committing to a single plan, but instead allowing all cached planning knowledge to be combined at run-time, Soar can respond to unexpected changes in the environment, as long as it has previously encountered a similar situation. If it does not have sufficient knowledge for the current situation, it will plan, learn the appropriate knowledge, and in the future be able to respond directly without planning.

## Interruption

The emphasis in our prior description of planning was on acquiring knowledge that could be responsive to changes in the environment during execution. This ignores the issue of how the system responds to changes in its environment during planning. Consider two scenarios from Robo-Soar. In the first scenario, one of the blocks is removed from the table while Robo-Soar is planning how to align the blocks. In the second, a trouble light goes on while Robo-Soar is planning how to align the blocks. This light signals that Robo-Soar must push a button *as soon as possible*. The key to both of these scenarios is that Soar's productions are continually matched against all of working memory, including incoming sensor data, and all goals and subgoals. When a change is detected, planning can be revised or abandoned if necessary.

In the first example, the removal of the block does not eliminate the necessity to plan, it just changes the current state, the desired state (fewer blocks need to be aligned) and the set of available operators (fewer blocks can be moved). The change in the set of available operators modifies the impasse but does not eliminate it. Within the subgoal, operators and data that were specific to the removed block will be automatically retracted from working memory. The exact effect will depend on the state of the planning and its dependence on the eliminated block. In the case where an outside agent suddenly aligned all but one of the blocks, and Robo-Soar had sufficient knowledge for that specific case, the impasse would be eliminated and the appropriate operator selected.

In the second example, we assume that there exists a production that will direct Robo-Soar to push a button when a light is turned on. This production will test for the light and create a preference that the `push-button` operator *must* be selected. When the next operator decision is made, there is no longer a tie, and the **push-button** operator is selected. Interruption of planning can be predicated on a variety of stimuli. For example, productions can keep track of the time spent planning and abort the planning if it is taking too much time. Planning would be aborted by creating a preference for the best action given the currently available information. One disadvantage of this scheme is that any partial planning that has not been captured in chunks will be lost.

## Hierarchical Planning and Execution

In our previous Robo-Soar examples, the set of operators corresponded quite closely to the motor commands of the robot controller. However, Soar has no restriction that problem space operators must directly correspond to individual actions of the motor system. For many problems, planning is greatly simplified if it is performed with abstract operators far removed from the primitive actions of the hardware. For execution, the hierarchical decomposition provided by multiple levels of operators can provide important context for dealing with execution errors and unexpected changes in the environment.

Soar provides hierarchical decomposition by creating subgoals whenever there is insufficient knowledge encoded as productions to implement an operator directly. In the subgoal, the implementation of the abstract operator is carried out by selecting and applying less abstract operators, until the abstract operator is terminated.

To demonstrate Soar's capabilities in hierarchical planning and execution we will use our second system, Hero-Soar. Hero-Soar searches for cups using sonar sensors. The basic motor commands include positioning the various parts of the arm, opening and closing the gripper, orienting sonar sensors, and moving and turning the robot. A more useful set includes operators such as **search-for-object**, **center-object**, **pickup-cup**, and **drop-cup**. The execution of each of these operators involves a combination of more primitive operators that can only be determined at run-time. For example, **search-for-an-object** involves an exploration of the room until the sonar sensors detect an object.

In Hero-Soar, the problem space for the top-most goal consists of just these operators. Control knowledge selects the operators when they are appropriate. However, once one of these operators is selected, an impasse arises because there are no relevant implementation productions. For example, once the **search-for-object** operator is selected, a subgoal is generated and a problem space is selected that contains operators for moving the robot and analyzing sonar readings.

Operators such as **search-for-object** would be considered goals in most other systems. In contrast, goals in Soar arise only when knowledge is insufficient to make progress. One advantage of Soar's more uniform approach is that all the decision making and planning methods also apply to these "goals" (abstract operators like **search-for-object**). For example, if there is an abstract internal simulation of an operator such as **pickup-cup**, it can be used in planning for the top goal in the same way planning would be performed at more primitive levels.

A second advantage of treating incomplete operator applications as goals is that even seemingly primitive acts, such as **move-arm** can become goals, providing hierarchical execution. This is especially important when there is uncertainty as to whether a primitive action will complete successfully. Hero-Soar has exactly these characteristics because its sensors are imperfect and because it sometimes loses motor commands and sensor data when communicating with the Hero robot. Hero-Soar handles this uncertainty by selecting an operator, such as **move-arm**, and then waiting for feedback that the arm is in the correct position before terminating the operator. While the command is executing on the Hero hardware, a subgoal is created. In this subgoal, the **wait** operator is repeatedly applied, continually counting how long it is waiting. If appropriate feedback is received from the Hero, the **move-arm** operator terminates, a new operator is selected, and the subgoal is removed. However, if the motor command or feedback was lost, or there is some other problem, such as an obstruction preventing completion of the operator, the waiting continues. Productions sensitive to the selected operator and the current count detect when the operator has taken longer than expected. These productions propose operators that directly query the feedback sensors, retry the operator, or attempt some other recovery strategy. Because of the relative computational speed differences between the Hero and Soar on an Explorer II+, Hero-Soar spends approximately 30% of its time waiting for its external actions to complete.

Hierarchical execution is not unique to Soar. Georgeff and Lansky have used a similar approach in PRS for controlling a mobile robot [Georgeff & Lansky, 1987]. In PRS, declarative procedures, called *Knowledge Areas* (KAs) loosely correspond to abstract operators in Soar. Each KA has a body consisting of the steps of the procedure represented as a graphic network. Just as Soar can use additional abstract operators in the implementation of an operator, a KA can have goals as part of its procedure which lead to additional KAs being invoked. PRS maintains reactivity by continually comparing the conditions of its KAs against the current situation and goals, just as Soar is continually matching it productions. A significant difference between PRS and Soar is in the representation of control knowledge and operators. Within a KA, the control is a fixed declarative procedure. Soar's control knowledge is represented as preferences in productions that can be used for any relevant decision. Thus the knowledge is not constrained to a specific procedure, and will be used when the conditions of the production that generates the preference match the current situation. In addition, new productions can be added to Soar through learning, and the actions of these productions will be integrated with existing knowledge at run-time.

## Reactive Execution

Hierarchical execution provides important context for complex activities. Unfortunately it also exacts a cost in terms of run-time efficiency. In order to perform a primitive act, impasses must be detected, goals created, problem spaces selected, and so on, until the motor command is generated. Execution can be performed more efficiently by directly selecting and applying primitive operators. However, operator application has its own overheads. The actions of an operator will only be executed after the operator has been selected following quiescence, thus forcing a delay. The advantage of these two approaches is that they allow knowledge to be integrated at run-time, so that a decision is not based on an isolated production.

Soar also supports direct reflex actions where a production creates motor commands without testing the current operator. These productions act as reflexes for low level responses, such as stopping the wheel motors when an object is directly in front of the robot. Along with the increase responsiveness comes a loss of control; no other knowledge will contribute to the decision to stop the robot.

The ultimate limits on reactivity rest with Soar's ability to match productions and process preferences. Unfortunately, there are currently no fixed time bounds on Soar's responsiveness. Given Soar's learning, an even greater concern is that extended planning and learning will actually reduce responsiveness as more and more productions must be matched [Tambe & Newell, 1988]. Recent results suggest that these problems can be avoided by restricting the expressiveness of the production conditions [Tambe & Rosenbloom, 1989].

Although there are no time bounds, Soar is well matched for both Hero-Soar and Robo-Soar. In neither case does Soar's processing provide the main bottleneck. However, as we move into domains with more limited time constraints, further research on bounding Soar's execution time will be necessary.

## Discussion

Perhaps the key reason that Soar is able to exhibit effective execution, planning (extended, hierarchical, and reactive), and interruption, is that it has three distinct levels at which external actions can be controlled. These levels differ both in the speed with which they occur and the scope of knowledge that they can take into consideration in making a decision. At the lowest level, an external action can be selected directly by a production. This is the fastest level — Soar can fire 40 productions per second on a TI Explorer II+ while controlling the Hero using 300 productions — but the knowledge utilized is limited to what is expressed locally in a single production.[2] This level is appropriately described as reflexive behavior — it is fast, uncontrollable by other knowledge, and difficult to change.

At the middle level, an external action can be selected through selecting an operator. This is somewhat slower — in the comparable situation as above, only 10 decisions can be made per second — but it can take into account any knowledge about the current problem solving context that can be retrieved directly by firing productions (without changing the context). It allows for the consideration and comparisons of actions before a selection is made. This level is appropriately described as a dynamic mixture of top-down (plan-driven) and bottom-up (data-driven) behavior. It is based on previously-stored plan fragments (learned control rules) and the current situation, and can dynamically, at run-time, adjudicate among their various demands. This level can be changed simply by learning new plan fragments.

At the highest level, an external action can be selected as a result of extended problem solving in subgoals. This can be arbitrarily slow, but potentially allows any knowledge in the system — or outside of it, if external interaction is allowed — to be taken into consideration. This level is appropriately described as global planning behavior.

Soar's learning is closely tied into these three levels. Learning is invoked automatically whenever the knowledge available in the bottom two levels is insufficient. Learning moves knowledge from planning to the middle level of deliberate action and, also to the bottom level of reflexes. Without learning, one could attempt to combine the bottom and middle layers by precompiling their knowledge into a fixed decision network as in REX [Kaelbling, 1986; Rosenschein, 1985]. However, for an autonomous system that is continually learning new control knowledge and operators [Laird *et al.*, 1990a], the only chance to bring together all of the relevant knowledge for a decision is when the decision is to be made.

The integration of planning, execution, and learning in Soar is quite similar to that in Theo because of the mutual dependence upon impasse-driven planning and the caching of plans as productions or rules. Schoppers' Universal Plans also caches the results of planning; however, Schoppers' system plans during an initial design stage and exhaustively generates all possible plans through back-chaining. In contrast, Theo and Soar plan only when necessary, and do not generate all

[2]Hero-Soar is limited in absolute response time by delays in the communication link between the Hero and the Explorer, and the speed of the Hero central processor. The actual response time of Hero-Soar to a change in its environment is around .5 seconds.

possible plans; however, Theo as yet does not support interruption, nor can it maintain any history. All decisions must be based on its current sensors readings. Soar is further distinguished from Theo in that Soar supports not only reactive behavior and planning, but also deliberative execution in which multiple sources of knowledge are integrated at run-time. This middle level of deliberate execution is especially important in learning systems when planning knowledge is combined dynamically at run-time.

## Acknowledgments

The authors would like to thank Michael Hucka, Eric Yager, Chris Tuck, Arie Covrigaru and Clare Congdon for help in developing Robo-Soar and Hero-Soar.

## References

[Blythe & Mitchell, 1989] J. Blythe & T. M. Mitchell. On becoming reactive. In *Proceedings of the Sixth International Machine Learning Workshop*, pages 255–259, Cornell, NY, June 1989. Morgan Kaufmann.

[Doyle, 1979] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Drummond, 1989] M. Drummond. Situated control rules. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, May 1989. Morgan Kaufmann.

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

[Georgeff & Lansky, 1987] M. P. Georgeff & A. L. Lansky. Reactive reasoning and planning. *Proceedings of AAAI-87*, 1987.

[Hammond, 1989] K. J. Hammond. *Case-Based Planning: Viewing Planning as a Memory Task.* Academic Press, Inc., Boston, 1989.

[Kaelbling, 1986] L. P. Kaelbling. An architecture for intelligent reactive systems. In M. P. Georgeff & A. L. Lansky, editors, *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, 95 First Street, 1986. Morgan Kaufomann.

[Laird *et al.*, 1986] J. E. Laird, P. S. Rosenbloom, & A. Newell. *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies.* Kluwer Academic Publishers, Hingham, MA, 1986.

[Laird *et al.*, 1987] J. E. Laird, A. Newell, & P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(3), 1987.

[Laird *et al.*, 1989] J. E. Laird, E. S. Yager, C. M. Tuck, & M. Hucka. Learning in tele-autonomous systems using Soar. In *Proceedings of the 1989 NASA Conference on Space Telerobotics*, 1989.

[Laird *et al.*, 1990a] J. E. Laird, M. Hucka, E. S. Yager, & C. M. Tuck. Correcting and extending domain knowledge using outside guidance. In *Proceedings of the Seventh International Conference on Machine Learning*, June 1990.

[Laird *et al.*, 1990b] J. E. Laird, K. Swedlow, E. Altmann, & C. B. Congdon. *Soar 5 User's Manual.* University of Michigan, 1990. In preparation.

[Langley *et al.*, 1989] P. Langley, K. Thompson, W. Iba, J. H. Gennari, & J. A. Allen. An integrated cognitive architecture for autonomous agents. Technical Report 89-28, Department of Information & Computer Science, University of California, Irvine, September 1989.

[Mitchell *et al.*, 1990] T. M. Mitchell, J. Allen, P. Chalasani, J. Cheng, O. Etzionoi, M. Ringuette, & J. Schlimmer. Theo: A framework for self-improving systems. In K. VanLehn, editor, *Architectures for Intelligence.* Erlbaum, Hillsdale, NJ, 1990. In press.

[Rosenbloom & Laird, 1986] P. S. Rosenbloom & J. E. Laird. Mapping explanation-based generalization onto Soar. In *Proceedings of AAAI-86*, Philadelphia, PA, 1986. American Association for Artificial Intelligence.

[Rosenbloom *et al.*, 1990] P. S. Rosenbloom, J. E. Laird, A. Newell, & R. McCarl. A preliminary analysis of the foundations of the Soar architecture as a basis for general intelligence. In *Foundations of Artificial Intelligence.* MIT Press, Cambridge, MA, 1990. In press.

[Rosenschein, 1985] S. Rosenschein. Formal theories of knowledge in AI and robotics. *New Generation Computing*, 3:345–357, 1985.

[Schoppers, 1986] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In M. P. Georgeff & A. L. Lansky, editors, *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop.* Morgan Kaufmann, 1986.

[Tambe & Newell, 1988] M. Tambe & A. Newell. Some chunks are expensive. In *Proceedings of the Fifth International Conference on Machine Learning*, 1988.

[Tambe & Rosenbloom, 1989] M. Tambe & P. S. Rosenbloom. Eliminating expensive chunks by restricting expressiveness. In *Proceedings of IJCAI-89*, 1989.

[Unruh & Rosenbloom, 1989] A. Unruh & P. S. Rosenbloom. Abstraction in problem solving and learning. In *Proceedings of IJCAI-89*, 1989.