CHAPTER 30

Learning New Tasks in Soar

G. R. Yost and A. Newell, Carnegie Mellon University

Abstract

Currently, Soar acquires new tasks by loading a set of productions written by a programmer. Because Soar is intended to function autonomously, we must exhibit mechanisms by which Soar can acquire new tasks on its own through interaction with its environment. We describe TAQ, a Soar system that comprehends external task descriptions and learns productions implementing the new task. The comprehension process decodes task descriptions into declarative descriptions of problem spaces for performing the task. TAQ interpretively executes selected parts of these problem space descriptions to resolve difficulties arising during task performance. During this interpretive execution, Soar's learning mechanism, chunking, automatically builds new task-implementation productions. The new productions allow Soar to perform some part of the task directly in the future, without recourse to the interpreter or to the declarative task representation. Through TAQ, Soar learns *all* aspects of a task, from problem space selection to operator application and goal testing.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499, and monitored by the Air Force Avionics Laboratory. The research was also supported in part under a National Science Foundation Graduate Fellowship to Gregg Yost, and in part by Digital Equipment Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation, Digital Equipment Corporation, or the U.S. Government.

1. Introduction

An intelligent agent must have the ability to learn new tasks in response to the demands of its environment: it must not rely on the skills of a human programmer familiar with its internal structure. We examine this problem in the context of the Soar architecture (Laird, Newell & Rosenbloom, 1987; Laird, 1986). Our specific goal is to have Soar accept task descriptions expressed in some convenient language, and develop from them the problem spaces required to perform the task. Soar must learn the productions it needs to work in these problem spaces, and it must learn *all* aspects of the task. This includes problem space selection, initial state construction, operator proposal, selection, and application, and recognition of success or failure.

Before proceeding to our model of task acquisition, we briefly describe some relevant aspects of the Soar architecture.

2. Soar

Soar is an architecture that is to be capable of general intelligent behavior. All behavior is cast in terms of search in a problem space. A problem solving configuration is described by a *goal context*, which indicates the current goal, problem space, state, and operator. Soar progresses towards a goal by applying knowledge to make changes in the goal context until reaching a desired configuration. Normally Soar selects a problem space and initial state, and then repeatedly selects and applies operators in search of a desired state.

All knowledge is represented by productions. Productions fire to make additions to working memory or to express *preferences* for changes to the current context. A *decision procedure* examines the set of preferences in an effort to find a single context change that is preferred above all others. In some cases the knowledge the productions provide directly is insufficient to specify a unique context change. Rather than make an arbitrary choice, Soar signals an *impasse* and automatically creates a subgoal to resolve the impasse. A subgoal is simply a new goal context object, so that Soar can bring its full problem solving abilities to bear on resolving an impasse. An impasse is resolved when problem solving in a subgoal produces sufficient preferences to specify a unique change in a supercontext.

There are four types of impasses. A *no-change* impasse occurs when there are no preferences for context changes; a *tie* impasse occurs when several different context changes are proposed; a *rejection* impasse occurs when every proposed context change is rejected for some reason; and a

conflict impasse occurs when preferences conflict. Impasses are further specified by the most recently filled context slot. For example, an *operator no-change* impasse occurs when an operator has been selected but no productions fire to construct a result state and make preferences for it.

Chunking is Soar's learning mechanism (Laird, Rosenbloom & Newell, 1986). It summarizes the processing of a subgoal in a *chunk*. The chunk is a production whose conditions are the inputs of the subgoal, and whose actions are the final results of the subgoal. The subgoal inputs are those features of the pre-subgoal situation upon which the results depend, and are determined by backtracing through the productions which fired during problem solving in the subgoal. Once a chunk is learned for a subgoal, Soar can apply it in relevant situations in the future. This saves the effort of going into a subgoal again to rederive the results. Because features that the results do not depend on are omitted from a chunk, the chunk attains an appropriate degree of generality.

3. Task Acquisition in Soar

In Soar, all tasks must be expressed as productions implementing problem spaces. Furthermore, all new productions must arise through chunking. Thus task acquisition in Soar requires at least two functions, which we call COMPREHEND and INTERNALIZE. COMPREHEND decodes external task descriptions and casts them in terms of declarative problem space descriptions. INTERNALIZE uses chunking to build productions implementing these problem spaces. Because chunking is an experience-based learning mechanism, the productions must be built during the course of executing the new task. Therefore, INTERNALIZE interpretively executes the problem space descriptions COMPREHEND provides. During this interpretive execution, chunking automatically builds the productions Soar needs to execute the task directly in the future. Figure 3-1 shows this model of task acquisition, which we have realized in the Soar system TAQ. The box labeled FETCH in Figure 3-1 is responsible for retrieving appropriate problem space description components for use by EXECUTE. Section 4.3 discusses FETCH in more detail. FETCH and EXECUTE together implement the INTERNALIZE function.

The main part of the paper focuses on the role of learning in the implementation of INTERNALIZE. First, however, we will say a few more words about COMPREHEND. As noted earlier, COMPREHEND's function is to decode task knowledge from the environment into problem

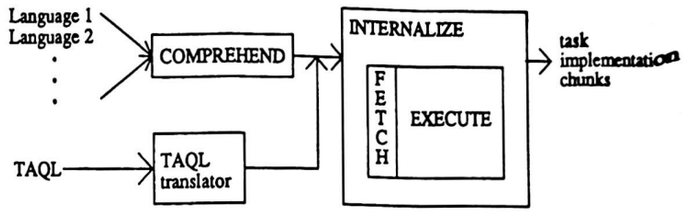

Figure 3-1: Soar model of task acquisition

space descriptions. Figure 3-1 indicates that COMPREHEND accepts input in a variety of languages, both natural and artificial. In fact, it should handle a great deal more variability than that. Task information from the environment can vary on a number of dimensions. It can vary in type, including instructions, examples, and advice. It can vary in degree of explicitness and detail. It can vary in form from language to pictures, sounds, and smells.

We are not prepared to address COMPREHEND in full generality. The current version of TAQ allows tasks to be expressed in simple English sentences, presented to Soar as text already segmented into words (earlier versions of TAQ also admitted formal notations). The main component of COMPREHEND is a problem space that applies *comprehension operators* to the input. Its operation is loosely based on Small's Word Expert Parser (Small, 1979). Each comprehension operator has knowledge about how to decode a particular word in a particular context; comprehension operators can apply to partial decodings as well as words.

3.1. The TAQL Task Description Language

As we indicated above, there is an indefinite set of tasks of extracting knowledge from the environment to help perform tasks. Because we want to focus in this paper on the INTERNALIZE component of task acquisition, we assume a situation where the environment directly provides the problem space descriptions, bypassing COMPREHEND.

To this end we have developed TAQL, a simple language for describing problem spaces. Figure 3-1 shows the relation of TAQL to TAQ. TAQL is able to bypass COMPREHEND, because the intended output of COMPREHEND, problem space descriptions, is exactly what TAQL provides. The only mediation required between TAQL and INTERNALIZE is a simple translation from the human-readable TAQL syntax to the attribute-value structures Soar requires. A simple Lisp program performs the translation. This program does nothing but a straightforward syntactic translation. Soar, through TAQ, does *all* of the semantic operations involved in task

acquisition.

Ignoring the subtleties of comprehension in our initial pass at task acquisition has an important advantage. By first refining INTERNALIZE, we discover exactly what COMPREHEND must produce to specify an executable task. Thus, TAQL corresponds directly to the abilities of EXECUTE.

There are five fundamental problem space operations: problem space selection, initial state construction, operator proposal, operator implementation, and goal testing. For each of these operations there are one or more TAQL constructs, as described below.

1. *Problem space selection.* There are two TAQL constructs for describing problem space selection operations. The *top-space* construct names a task and a space which can implement that task. The *operator-implementation-space* construct names an operator and a space which can implement that operator.
2. *Initial state construction.* The *initial-state* TAQL construct indicates how to build the initial state for a specified problem space. It also lets one specify additional conditions on the problem solving context restricting when that particular initial state may be built.
3. *Operator proposal.* The *op-instantiation* construct specifies how to build an operator object for a particular operator, and under what conditions the operator should be proposed.
4. *Operator implementation.* The *op-implementation* construct specifies how to apply a primitive operator. It indicates the name of the operator it describes, a set of tests against the goal context (often used to bind variables for later use), and add and drop lists. The add list specifies new features to be added to the result state. The drop list specifies which features of the current state should be excluded from the result state. Implicit in this is that all features of the current state not explicitly dropped are included in the result state. Operators that cannot easily be described in this simple form can specify that they should be implemented in a separate problem space, using the *operator-implementation-space* construct described above.
5. *Goal testing.* There are two TAQL constructs for describing goal testing information. The *desired-test* construct specifies conditions on the goal context for recognizing when the desired state has been reached. The *undesired-test* construct specifies conditions for recognizing that the desired state has not yet been reached.

In Soar, all aspects of a task are represented by problem spaces. Therefore TAQL does not include special constructs for specifying search control information. Rather, search control is specified by giving operator implementations for the evaluate-object operator defined in Soar's default selection space (Laird, Newell & Rosenbloom, 1987).

TAQL is a higher-level problem space description formalism than Soar productions, and thus

it is often easier to express a task in TAQL than in productions. Like most high-level languages, though, TAQL trades expressive power for ease of use. For example, TAQL represents all primitive operators using add and drop lists, whereas Soar productions permit a wider variety of implementation methods.

Figure 3-2 shows an example TAQL construct taken from the eight puzzle task description. It indicates how to implement the move-tile operator, which moves into the empty cell one of the numbered tiles occupying an adjacent cell[1].

```
(op-implementation :for move-tile
  :test (
    (goal <g> ^state <s> ^operator <o>)
    (operator <o> ^name move-tile ^move-from-cell <from>
                  ^move-to-cell <to>)
    (state <s> ^blank-cell <to> ^cell-contents <c1> <c2>)
    (cell-contents <c1> ^cell <to> ^tile <blank-tile>)
    (cell-contents <c2> ^cell <from> ^tile <from-tile>)
  )
  :add (
    (state ^cell-contents <c3> <c4> ^blank-cell <from>)
    (cell-contents <c3> ^cell <from> ^tile <blank-tile>)
    (cell-contents <c4> ^cell <to> ^tile <from-tile>)
  )
  :drop (
    (cell-contents <c1>)
    (cell-contents <c2>)
    (blank-cell <to>)
  ))
```

Figure 3-2: TAQL description of the move-tile operator

The example construct specifies that the contents of the two cells involved in the move should be swapped, and that the attribute indicating the current blank cell should be updated. The updating is done by adding the new values to the result state, dropping the old values from the prior state but copying all of the undropped attributes to the result state.

4. Internalizing Problem Space Descriptions

We now turn to the main topic of the paper: the implementation of INTERNALIZE. There are several constraints on the design.

C1. Soar's learning mechanism, chunking, is such that Soar can only learn productions implementing a task by doing the task (or an analogue that permits transfer).

C2. Task implementation chunks should operate directly at the level of the Soar

[1] A complete description of the eight puzzle task representation is in (Laird, Newell & Rosenbloom, 1987).

architecture. They should work within the model of computation provided by Soar, rather than attempting to work within some other fixed structure imposed on Soar.

C3. The task implementation chunks learned should not test the declarative task representation. A design in which Soar can only perform a task when the declarative representation is in working memory is unacceptable.

Constraint C1 implies that at least the first time Soar executes a task, it must do so by interpretively executing the declarative task description. It also implies that the task implementation chunks must be learned in the course of this interpretive execution. Given that we must use an interpreter, constraint C2 requires Soar to ultimately build task implementation chunks which do not require the services of the interpreter. That is, the task must break free of the interpreter and execute directly at the level of the Soar architecture. A good test of this is whether task implementation chunks look like the productions a person coding the task would write. Constraint C3 imposes a related requirement: task implementation chunks must ultimately break free of the declarative task representation.

4.1. The TAQ interpreter

As mentioned above, the internalized task must run without relying on the interpreter. Therefore, we use an interpreter that only takes control when Soar is unable to proceed with task execution. This situation is inherently easy to recognize in Soar. Soar is unable to proceed with task execution exactly in those situations when it gets an impasse and no problem space is proposed to resolve the impasse. Thus the TAQ interpreter is *impasse-driven*: it is invoked only when Soar gets an impasse it does not know how to resolve in any other way.

The interpreter's operation is outlined in Figure 4-1. Soar begins by attempting to run the new task as if it already knew how to implement it. If in fact it does not, it will get an impasse at the point where something new must be learned. TAQ recognizes the impasse as something it could potentially resolve, and so makes a preference for a TAQ space that knows how to use task description information to resolve the particular type of impasse that arose.

The selected TAQ interpreter space is called learn-more-about-task in Figure 4-1. The first operator selected in such a space is fetch-task-info. Its function is to fetch potentially useful task description components (recall that task descriptions available to the interpreter are declarative representations of problem spaces and their parts). See Section 4.3 for more details.

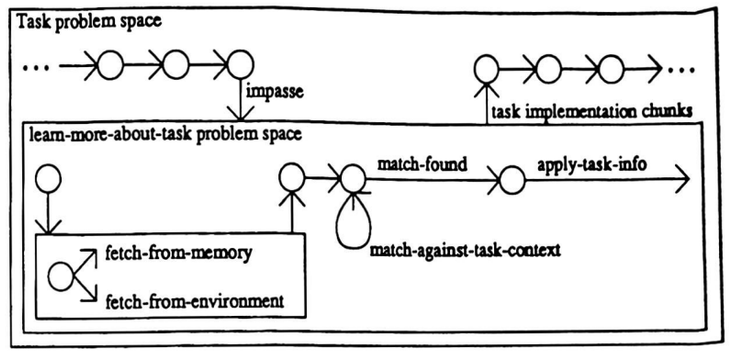

Figure 4-1: Generic structure of the impasse-driven interpreter

Next, TAQ applies the match-against-task-context operator to the retrieved task description components. This operator matches any preconditions in the components against the current task context. If the preconditions are satisfied, the interpreter applies the indicated problem space operation to the task context. If this resolves the task's impasse, Soar will build a chunk whose conditions are the tests performed on the task context during component fetch, precondition matching, and possibly component application. The actions of the chunk are the actions performed during the component application. Such a chunk directly implements the portion of the task just performed by the interpreter: the left hand side specifies when the operation should be performed and binds variables for use on the right hand side; the right hand side performs the problem space operation required for task execution to proceed.

Figure 4-1 shows the general form of a TAQ interpreter space. In actuality there is an interpreter space for each kind of problem space operation, such as operator proposal or goal testing. Each space knows how it can use its problem space component to resolve certain impasses. Table 4-1 indicates which components can be used to resolve each of Soar's impasses. TAQ is sensitive to *no-choice* impasses, that is, either a no-change impasse or a rejection impasse. To TAQ, it does not matter whether the task cannot proceed because Soar never did know what to do in that situation, or whether the task cannot proceed because something proposed earlier was later rejected. In either case Soar must learn new task information.

Impasse Type	Problem Space Operation
goal no-choice	problem space selection
problem space no-choice	initial state construction
state no-choice	goal testing, if the state is not classified as desired or undesired operator proposal, if the state is classified as undesired
operator no-choice	operator implementation
tie	search control (evaluate-object operator implementations in the selection space, currently)
conflict	not resolvable by acquisition of task

Table 4-1: Impasse types and the problem space operations which can resolve them

4.2. An example

Figure 4-2 shows an abstracted trace of TAQ acquiring the eight puzzle. It starts with the problem space descriptions in working memory (placed there by the TAQL translator), but does not have any task implementation productions. Space constraints prevent us from showing the entire eight puzzle TAQL description, but the move-tile operator implementation description in Figure 3-2 is representative. The trace shows the major problem solving steps, the impasses that eventually lead to the construction of new task implementation chunks, and where those chunks are built and used.

On line 1, as soon as Soar first tries to execute the task, it gets a goal no-change impasse because it does not know what problem space to use. The impasse is resolved on lines 2 and 3 by entering a TAQ space which knows how to retrieve and apply problem space selection knowledge. A chunk is built on line 4 which will propose the eight-puzzle space to implement the solve-eight-puzzle task in the future. On line 5 the first bit of progress towards solving the eight puzzle is made, when the eight-puzzle space is installed as the current problem space. Lines 6-9 show Soar learning to construct the initial state. After installing the state on line 10, it gets a state no-change impasse because it does not recognize the state as either desired or undesired. It then enters the test-desired TAQ space. After selecting and applying an appropriate piece of task description, it builds a chunk on line 14 that recognizes the current state as undesired. As we will see later, this chunk is sufficiently general to recognize *all* undesired states for this eight puzzle task, so that Soar will not have to search for this type of task


```
1  ==> goal no-change impasse
2      enter choose-problem-space problem space
3      fetch and apply relevant task description component (TDC)
4      Build chunk: select-eight-puzzle-space
5  eight-puzzle installed in problem space slot
6  ==> problem space no-change impasse
7      enter construct-initial-state problem space
8      fetch and apply relevant TDC
9      Build chunk: construct-initial-state-for-eight-puzzle-space
10 initial state installed
11 ==> state no-change impasse
12     enter test-desired problem space
13     fetch and apply relevant TDC
14     Build chunk: recognize-undesired-eight-puzzle-state
15     enter propose-operators problem space
16     fetch potentially relevant TDCs
17     match against eight-puzzle context to find applicable component
18     apply applicable component
19     Build chunk: propose-move-tile-operator
20 ==> operator tie impasse
21     enter selection problem space
22     propose evaluate-object operators to evaluate the tied move-tile operators
23     install one of the evaluate objects operators in operator slot
24     ==> operator no-change impasse
25         enter apply-operator problem space
26         fetch TDCs for implementing evaluate-object operator
27         match against context to find applicable component
28         apply applicable component
29         Build chunk: implement-evaluate-object-1
30     install a different evaluate-object operator
31     Fire chunk: implement-evaluate-object-1
32     Build chunk: make-relative-preference-for-move-tile-operator-1
33     install a different evaluate-object operator
34     ==> operator no-change impasse
35         enter apply-operator problem space
36         fetch TDCs, match, and apply
37         Build chunk: implement-evaluate-object-2
38     Build chunk: make-relative-preference-for-move-tile-operator-2
39 install selected instance of move-tile operator
40 ==> operator no-change impasse
41     enter apply-operator problem space
42     fetch TDCs for implementing move-tile operator
43     match, and apply applicable component
44     Build chunk: implement-move-tile-operator
45 install result state
46 Fire chunk: recognize-undesired-eight-puzzle-state
47 Fire chunk: propose-move-tile-operator (two instantiations fire)
48 Fire chunk: make-relative-preference-for-move-tile-operator-2
49 install selected move-tile operator
50 Fire chunk: implement-move-tile-operator
51 install result state
52 [Repeat as in lines 46-51 until desired state is installed ...]
53 ==> state no-change impasse
54     enter test-desired problem space
55     fetch TDCs for recognizing desired states in eight-puzzle space
56     match, and apply applicable component
57     Build chunk: recognize-desired-eight-puzzle-state
58 Halt, task completed successfully
```

Figure 4-2: Trace of TAQ acquiring the eight puzzle description component again.

Simply recognizing that the initial state is not the desired state is not enough to resolve the state no-change impasse from line 11. An operator must be proposed in the eight puzzle space to make further progress. So, on line 15 the test-desired TAQ space is replaced by the propose-operators TAQ space. On the next several lines, task description information is applied leading Soar on line 19 to build a chunk that will propose legal instances of the move-tile operator. Four tiles can be moved in the initial state, and since Soar does not know which of the four move-tile operators to apply it gets an operator tie impasse on line 20. The impasse is handled by Soar's default selection space, which tries to evaluate the tied operators by applying the evaluate-object operator to each of them. On line 23 Soar arbitrarily selects one of these evaluate-object operators to apply. It does not know how to evaluate a move-tile operator, though, so it gets an operator no-change impasse and TAQ comes into play again. Soar enters the apply-operator TAQ space and searches for problem space description information that will tell it how to evaluate the specified instance of move-tile. It finds such information, and applies it to produce an evaluation. On line 29 Soar builds a chunk that is able to apply evaluate-object to move-tile in some cases.

On line 30 another of the evaluate-object operators is selected to evaluate a different instance of move-tile. Here we see one of the chunks produced by TAQ firing to implement evaluate-object, obviating a repeat of the search through the problem space descriptions. A third evaluate-object operator is selected on line 33, but the implement-evaluate-object-1 chunk is not sufficiently general to perform the evaluation this time. So TAQ is invoked again and by line 37 builds another chunk allowing it to perform this evaluation as well. This evaluation finally gives Soar all the preferences it needs to resolve the tie impasse from line 20, and on line 39 it installs the selected instance of move-tile.

On lines 40-44 Soar learns how to implement move-tile. By line 44 it has all the task implementation chunks needed to continue searching for the desired state without further intervention by TAQ. However, it does not yet know how to recognize the desired state. So on line 53 Soar gets a state no-change impasse because it does not know whether the state it just installed is desired or undesired. The test-desired TAQ space resolves the impasse by finding parts of the problem space description indicating that the current state is in fact the desired state. Problem solving halts with success on line 58 after building a chunk which can recognize the desired state in the future.

As required by constraint C2, the task implementation chunks that Soar learns operate directly at the level of the Soar architecture, and do not require the services of the interpreter to apply. They are essentially identical to the hand-coded eight puzzle productions presented as an example in the Soar user's manual (Laird, 1986). Furthermore, as required by constraint C3, the chunks do not depend on the task description being present in working memory, so that Soar will still be able to solve the eight puzzle long after the declarative task description has been forgotten (if, indeed, it was ever learned).

4.3. Retrieving task description components

So far we have not said how TAQ retrieves potentially useful bits of problem space description when an impasse arises during task execution. There are two issues: first, how to locate the task description information, and second, how to quickly select a potentially relevant subset to examine further. The second issue is rather easy to resolve. Problem space description components are tagged with their type (e.g., operator implementation) and with the name of the object they help define. For example, an operator implementation component is tagged with the name of the operator it implements. When an impasse arises during task execution, the relevant component types are determined using the mapping given in Table 4-1. The relevant object names are available as attributes of the task context at the point the impasse arose. For example, for an operator no-change impasse we know that we want to retrieve operator implementation information for the operator that currently occupies the operator slot in the task context.

The first issue above is where to look for the task description information. There are three possibilities: working memory, long term memory, and the environment. In every case the information is retrieved by the fetch-task-info operator. Retrieval from working memory is the simplest option. Any working memory elements attached to a task-description attribute of the task goal context are treated as a source of task information. Task information may also be stored in long term memory by using so-called *data chunks* (Rosenbloom, Laird & Newell, 1987; Rosenbloom, Laird, & Newell, 1988). A data chunk is a production that tests some retrieval cue on its left hand side and adds some declarative structure to working memory on its right hand side. In the task acquisition domain, the cues are the type of information needed and the name of the object the information should help define. Chunks built in the course of implementing the fetch-task-info operator have this form.

The final potential source of task description information is the environment. TAQ can reinvoke COMPREHEND to obtain more information. Currently, TAQ only looks to the environment if it cannot find useful information in either working memory or long term memory.

We have glossed over an important issue. If task description information is retrieved from working memory and was present in working memory prior to the impasse which invoked TAQ, any task implementation chunks built will depend on the description being present in working memory. This violates constraint C3. Fortunately there is a way to avoid having the chunks test the presence of the task descriptions. The technique used is analogous to the one Rosenbloom describes for constructing data chunks (Rosenbloom, Laird & Newell, 1987), and we will not describe it further here. There is no such problem when task descriptions are retrieved from either long term memory or the environment. In these cases the information does not enter working memory until *after* the impasse that invoked TAQ. Because chunking includes as conditions only tests against working memory elements that existed before the impasse, the resulting task implementation chunks do not test for the presence of the task description.

5. Discussion

We have described TAQ, a domain-independent Soar system which allows Soar to learn new tasks given an external task description. TAQ has two major components, COMPREHEND and INTERNALIZE. COMPREHEND decodes task knowledge in the environment into declarative descriptions of problem spaces for performing the task. INTERNALIZE transforms these problem space descriptions into the productions Soar needs to execute the task. It does so by selecting and interpretively executing appropriate pieces of the problem space descriptions whenever Soar reaches an impasse during task execution and knows of no other way to resolve the impasse. Soar's learning mechanism, chunking, automatically builds productions during this interpretive execution. The resulting productions implement the task directly at the level of the Soar architecture, so that in the future Soar will not need the interpreter to perform the same operation. Furthermore, the task implementation chunks do not depend on the presence in working memory of the declarative problem space descriptions, so that Soar can still perform the task even if it loses access to these descriptions. To put the matter in other terms, Soar abstracts from the occasion of learning, in which the declarative information is available, and transfers the learning to new situations in which this information is not available. ACT* takes a similar approach to internalizing new skills (Anderson, 1983, chapter 6). It too uses its learning mechanism to automatically build task implementation productions in the course of interpretively applying declarative knowledge.

TAQ is an important demonstration for Soar. It shows that chunking, Soar's simple experience-based production building mechanism, is capable of building productions implementing entirely new tasks. It is not restricted to making existing tasks run faster, or to transferring learning within classes of similar tasks. Chunking is a basic learning mechanism that provides multiple ways of accomplishing higher-level functionality (Steier, et al., 1987). Thus TAQ is not the only way for Soar to take in external knowledge. By using chunking in different ways, Soar can also acquire long term declarative knowledge (Rosenbloom, Laird & Newell, 1987) and external advice (Golding, Rosenbloom & Laird, 1987).

TAQ addresses many of the same issues as various other AI systems, including knowledge acquisition tools and expert system shells, advice takers, learning apprentices, and many natural language understanding systems. Two things that set TAQ apart are things it does *not* do:

- TAQ does not require knowledge of Soar's internal task representation (productions). Rather, TAQ represents task knowledge in a convenient form (problem spaces) and executes the problem space descriptions interpretively to perform the task. Soar, through chunking, automatically learns task implementation productions. TAQ itself understands only problem spaces, and has no knowledge of productions.
- TAQ does not analyze task descriptions for adequacy. Rather, inadequacies are shown up by the occurrence of an impasse during task performance.

In other words, TAQ lets Soar acquire abilities by *using* knowledge, not by *analyzing* knowledge. Productions are built by the same simple automatic learning mechanism that applies to everything Soar does, instead of by deliberate construction on the part of TAQ. Also, task acquisition and task performance are not separate operations. Task acquisition is interspersed with task performance, with problem solving difficulties signaling the need for new task information. Furthermore, whereas most learning systems learn only operator selection rules, TAQ learns *all* aspects of a task.

So far, TAQ has been used to acquire several small tasks, and to acquire algorithm specifications in an automatic algorithm design system. TAQ is quite efficient. It takes only 225 decision cycles (goal context changes) to acquire the eight-puzzle space. Only 158 of these are actually spent acquiring new task information. The remaining 67 decision cycles are spent

performing the actual task, searching for the desired eight puzzle state. TAQ is implemented by 292 productions in fourteen problem spaces.

Future work on TAQ will be in two areas. First, we will test the generality of the TAQ interpreter and TAQL by using them to acquire a diverse selection of tasks. Second, we will expand the capabilities of COMPREHEND, focusing on how the representation of a task as a set of problem spaces arises from the comprehension of a natural language task description. Either of these may involve extending TAQ's fundamental abilities.

References

Anderson, J. R. (1983). *The Architecture of Cognition.* Harvard University Press.

Golding, A., Rosenbloom, P. S., and Laird, J. E. (August 1987). Learning general search control from outside guidance. *Proceedings of the Tenth International Conference on Artificial Intelligence.* Milano, Italy: Proceedings of IJCAI-87.

Laird, J. E. (January 1986). *Soar User's Manual* (Tech. Rep. ISL-15). Xerox Corporation.

Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence, 33*(1), 1-64.

Laird, J. E., Rosenbloom, P. S., and Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning,* Vol. *1*(1).

Rosenbloom, P. S., Laird, J. E., and Newell, A. (1987). Knowledge-level learning in Soar. *Proceedings of AAAI-87.* Seattle, WA: American Association for Artificial Intelligence.

Rosenbloom, P. S., Laird, J. E., & Newell, A. (1988). The chunking of skill and knowledge. In H. Bouma & B. A. G. Elsendoorn (Eds.), *Working Models of Human Perception.* London: Academic Press. In press.

Small, S. (August 1979). Word expert parsing. *Proceedings of the 17th Annual Meeting of the Association for Computational Linguistics.* La Jolla, CA: University of California at San Diego.

Steier, D. M., Laird, J. E., Newell, A., Rosenbloom, P.S., Flynn, R. A., Golding, A., Polk, T. A., Shivers, O. G., Unruh, A., and Yost, G. R. (June 1987). Varieties of learning in Soar: 1987. P. Langley (Ed.), *Proceedings of the Fourth International Workshop on Machine Learning.* Los Altos, CA: University of California, Irvine, Morgan Kaufmann Publishers, Inc.

Index

1023-choice reaction-time task, 295
  See also Seibel task

A*, 274, 465, 522

Abstraction, 549-550, 885, 959-972, 1181-1188
  Abstract plan, 786, 1110-1111
  Abstract problem spaces, 543, 550, 556
  Abstract rules, 959
  Abstract search, 553-554
  Abstract simulation, 1110
  Abstraction planning, 395, 542, 885, 1103, 1189
  Abstraction propagation, 964-967
  Abstractions in learning, 1188
  Dynamic Abstraction, 549-558
  Iterative abstraction, 1181-1188

ABStrips, 251, 542, 963, 1111, 1184

Acceptability preferences, 616, 900, 908
  See also Preferences

Acceptable preference, 616, 617
  See also Preferences

Accumulator models, 119-120

Acquisition, 1361
  Acquisition of cognitive skill, 1361
  Acquisition of conservation, 1361
  Acquisition of declarative knowledge, 1361
  Acquisition of episodic knowledge, 1361
  Acquisition of macro-operators, 372-382
  Acquisition of tasks, 541
  See also Learning

Across-context search, 881

Across-task transfer, 536, 546

Across-trial transfer, 513, 536

Act*, 352, 392, 464, 700, 701, 720, 770-781

Action performance, 865, 909-910

Activation, 771

Activation-based production system, 771

Advice, 460

Advice-taking, 1106

Affect-set size, 671
  See also Rete match algorithm

Agenda mechanism, 282

Agent, 155, 246, 1294-1305, 1343

Algebraic equations, 807

Algorithm design, 653-657, 942-956
  See also Cypress-Soar and Designer-Soar

Algorithm schemes, 653

All-goals chunking, 845

Alpha branch, 412, 414, 436-437, 438, 439,
  See also Rete match algorithm

Alpha memory, 413, 422-423, 425-426, 622, 623
  See also Rete match algorithm

Alpha nodes, 412, 422
  See also Rete match algorithm

Alpha-beta search, 413, 465, 472, 522

Alpine, 1111

Alto computer, 185-186

Always productions, 203

AM, 251, 353

Amygdala, 1406

Analogical transfer, 1106

Analogy, 801, 954

Analysis-by-synthesis, 466

Analytic learning methods, 1027, 1031

And-node, 421-422, 661

And/Or search, 300, 465, 1105

And/Or hierarchy, 1106

Annotated models, 628, 629, 720, 814, 1373

Antecedent derivation, 533

Applicability completeness, 1109

Applicability impasses, 1105, 1109

Application phase, 261

Approximation, 962
  See also Abstraction

Architecture, 639-650, 755, 897-931

Archplan, 1115, 1116

Arithmetical capability, 872

As-needed planning, 1111

Associationism, 707

Associative memories, 729

Assumption counting, 1181-1188

Assumption-based goal tests, 965

Asymptote of learning, 99