

# Designing an Intelligent Tutoring System Based on a Reactive Model of Skill Acquisition

Randall W. Hill, Jr.  
Jet Propulsion Laboratory  
California Institute of Technology  
4800 Oak Grove Drive, Pasadena, CA 91109, USA

W. Lewis Johnson  
Information Sciences Institute  
University of Southern California  
4676 Admiralty Way, Marina del Rey, CA 90292, USA

**Abstract:** We describe a detailed computational model of skilled behavior and learning in a reactive task domain. We model not only the problem solving behavior, but also the skill acquisition process: our model can improve its performance over time, acquire new knowledge, and recover from incorrect knowledge. This model allowed us to predict the effectiveness of various tutoring system and curriculum design choices, which has provided guidance for the development of an intelligent tutoring system.

## 1. Introduction

We have designed an intelligent tutoring system (ITS) to train operators of complex equipment. Our training domain requires reactive, goal-oriented task skills, which contrasts with the static task skills that have been the subject of numerous other ITS's. The problem solvers in our task domain interact with an unpredictable external environment, i.e., frequent and unexpected state changes occur. These changes force the problem solver to react to the new state of the external environment while simultaneously pursuing the task goals. In contrast, static tasks do not typically interact with an external environment, and if they do, the environment is predictable. Problem solving on static tasks can be done by internally simulating the effects of actions without worrying about the state changing unexpectedly.

The questions that must be addressed in designing an intelligent tutor are familiar ones: How should the tasks be structured for the student? How should the student's actions be tracked? When should the tutor intervene? Finally, how should the intervention be enacted?

One tutoring methodology that has seen some success in a number of different ITS implementations is *model tracing*, which has its roots in a cognitive theory of skill acquisition, namely ACT\* (Anderson, 1983) and its successor, PUPS (Anderson, 1989; Anderson, 1990). The model tracing paradigm was used to build tutors for Lisp programming (Reiser, 1985; Anderson, 1990), Geometry theorem-proving (Anderson, 1990), Algebra (Anderson, 1990), and Electric Fields (Ward, 1991). Note that all of the cognitive modeling work that inspired the model tracing approach in these ITS's has been aimed primarily at static tasks such as programming and mathematical problem solving. Likewise, VanLehn's SIERRA model (VanLehn, 1987) of skill acquisition was applied to a static task, namely subtraction. There has been relatively little work to date on modeling tasks that interact with an unpredictable external environment and using these models to motivate tutoring system design. Since existing model tracing approaches have been developed with static tasks in mind, they cannot be assumed to be appropriate for dynamic tasks. For example, some model tracing systems interrupt the student frequently, which may be too distracting if the students are busy attending to and responding to the dynamic environment.

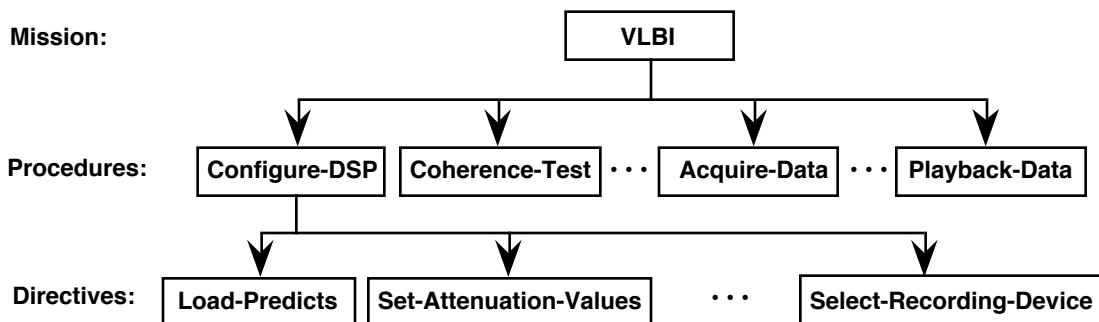
In this paper we describe a detailed computational model of skilled behavior and learning in our task domain. This model allowed us to predict the effectiveness of various tutoring system and curriculum design choices, by projecting how they would interact with our cognitive model. These predictions have provided guidance for the development of an intelligent tutoring system (Hill & Johnson, 1993). We plan to test these predictions and the resulting tutoring system both through evaluations with trainees and by using the cognitive model as an artificial student that the tutoring system must train. We were able to make use of the expert cognitive model in the tutoring system itself, e.g., as a generator of advice on how to solve problems. However, we did not build the

model in order to conform to a preconceived ITS design. Rather, we built the model in order to help us evaluate alternative design choices.

To achieve these purposes, we model not only the problem solving behavior, but also the skill acquisition process: our model can improve its performance over time, acquire new knowledge, and recover from incorrect knowledge. The model is able to respond to dynamically changing situations and revise its problem solving procedures accordingly. We also model how the problem solver behaves in response to tutorial assistance. We chose to build our cognitive model in Soar, an architecture that implements a theory of human cognition.

## 2. Task Domain Description

Our training domain is the operation of a communications link in NASA's Deep Space Network (DSN). The DSN is a worldwide system for navigating, tracking and communicating with all of NASA's unmanned interplanetary spacecraft. Tasks in this domain are organized into three levels: *mission*, *procedure*, and *directive*. The mission is a description of the overall task: it has a set of goals, it has a collection of devices assigned to a communications link, and it has a set of procedures, where some of the procedures may be common to other mission types. Procedures also have goals, usually with respect to the state of the devices that they affect. Each procedure has a sequence of directives that will, under ideal circumstances, cause the link devices to behave in a desired manner. A directive is a command that is issued by the link operator to control a device in the communications link, where the types of devices range from the hydraulic pumps used to move a 70-meter antenna to the software that controls the receivers, exciters and digital spectral processor (DSP). For each directive issued, a directive response is sent back indicating whether the device accepted or rejected the directive. If the directive is accepted, the operator watches for event notice messages and attends to subsystem displays for indications that the directive had its intended effect.



**Figure 1** Example of a task organized into three levels: Mission, Procedure, Directive. The cognitive model treats the task organization as a problem space hierarchy, where each box represents a problem space.

An example of a task in this domain is shown in Figure 1. The mission is VLBI (Very Long Baseline Interferometry), and it involves performing the procedures called Configure-DSP, Coherence-Test, and so on. The Configure-DSP procedure has directives to: load the mission-specific prediction data file (Load-Predicts), set the attenuation values on the Intermediate Frequency Video Down Converter (Set-Attenuation-Values), ..., and select a recording device to capture the mission's communication data (Select-Recording-Device). Each of the directive actions involves issuing a command (e.g. the Load-Predicts directive is *NLOAD predicts-file*).

Looking at Figure 1, it would appear that the tasks in this domain are straightforward and would require little or no training -- just follow the mission procedure manuals. We have found, however, that this is not the approach taken by domain experts. Through extensive interviews with expert operators and system engineers, we determined that the procedure manuals only provide a subset of the knowledge needed to successfully perform a mission task. What is generally lacking in the procedure manuals is a complete description of the required device state conditions before and after a directive is issued. Thus, the expert operator possesses a knowledge of the preconditions and postconditions for each directive and verifies these conditions are satisfied before and after each directive is sent. Operators who lack this knowledge may find it difficult to complete even simple procedures, since the directives may be rejected, or worse, put the device into an incorrect state for the procedure or mission. For example, one of the preconditions for the Load-Predicts directive is that the predicts-file being loaded must be present on the system. If the Load-Predicts directive is issued for the predicts-file named "JK" (i.e., *NLOAD JK*), it will be rejected if a file by that name is not present in the predicts file directory.

Devices may change state unexpectedly due to failures. Furthermore, commands may have unexpected effects when issued in the wrong situation. In either case, to become an expert operator requires learning to recognize the device state requirements (i.e., preconditions and postconditions) for each directive in every procedure. It also requires understanding the goals associated with each procedure, since in many instances it is necessary to work

around device state anomalies in order to complete the procedure. Thus, we characterize the task skills in this domain as being reactive and goal-oriented.

In contrast, static domains do not involve interacting with an unpredictable external environment. For instance, Lisp programs change only if the programmer edits them; they do not change on their own.<sup>1</sup> Furthermore, in static domains it is feasible to perform significant amounts of lookahead search and backtracking to solve problems since the effects of actions are predictable and can be mentally simulated. In an unpredictable domain it is not possible to do a lookahead search because the effects of actions cannot be confidently predicted, and once an action is taken it cannot be undone, which rules out the use of backtracking.

### 3. Overview of the Cognitive Model

We developed our cognitive model in Soar, an architecture that implements a theory of human cognition (Laird et al., 1987; Newell, 1990). This section briefly describes Soar and what it brings to the modeling effort. Using this foundation, we then describe the behaviors of novice and expert problem solvers as they are modeled in Soar.

#### 3.1 Soar Cognitive Architecture

Soar provides an integrated problem solving and learning architecture, which enables us to model a spectrum of problem solving behaviors as well as the skill acquisition process that transforms a novice into an expert. Tasks in Soar involve goal-oriented search through a hierarchy of problem spaces. For each goal, a problem space is selected or generated, where the problem space contains a set of operators and an initial state. Problem solving involves generating new states within a problem space by applying operators, until a desired state is achieved. When the problem solver reaches an impasse (e.g., it doesn't have sufficient knowledge to implement a particular operator, or no more operators can be applied to the current state), the Soar architecture supports the creation of a subgoal where other problem spaces can be searched for a solution. In this way, the problem solving activity involves setting and achieving goals in a hierarchical fashion.

Learning in Soar occurs when the results returned from a subgoal are converted into new productions that can be applied to achieve the same results under similar circumstances (Rosenbloom & Newell, 1986). These learned productions are called *chunks*, and they summarize both the goal and state context in which an operator applies. Chunks are added to the production memory as they are learned, and constitute the recognition knowledge in the architecture. Whereas objects can be added to or deleted from the working memory, productions are never deleted.

#### 3.2 Novice Problem Solving

Human novice operators perform tasks in this domain by rotely following the procedure manual. Since the novice has little or no knowledge of directive preconditions and postconditions, frequent problem solving impasses occur (e.g., in section 2 we described the case where the Load-Predicts directive was rejected because the predicts file was not present). Thus, an accurate model of a novice operator should produce the same kind of behavior, including errors, as we have observed in the protocol analysis.

We model novice problem solving behavior by implementing the task description shown in Figure 1 as a hierarchy of Soar problem spaces. In order to produce the rote behavior we expect of the novice, we force the cognitive model to execute the procedures and directives in a predetermined (i.e., procedure manual) order. The model's execution order is implemented through the use of Soar desirability preferences. Thus, our novice problem solver searches in the VLBI problem space for an operator and finds there are four to choose from: Configure-DSP > Coherence-Test > Acquire-Data > Playback-Data, where the ">" indicates a *better than* desirability preference. Since the Configure-DSP operator is the most desirable choice of the set, it is selected, a sub-goal is created, and a Configure-DSP problem space is created.

The problem solver searches the Configure-DSP problem space and finds that there are numerous applicable operators, along with another preference ordering: Load-Predicts > Set-Attenuation-Values > Select-Recording-Device. A subgoal is created for the Load-Predicts operator, and within the Load-Predicts problem space the NLOAD directive is applied to a device and the Load-Predicts goal terminates. This process is repeated for each of the directive operators in the Configure-DSP problem space. Once all the directives have been applied, the Configure-DSP subgoal terminates, and the next subgoal, Coherence-Test, is created, and so on.

The novice model does not consider preconditions in selecting procedures and directives, thus the device states are irrelevant to the problem solver unless it reaches a problem solving failure impasse (i.e., the directive

---

<sup>1</sup>If the programmer tests the program by executing it, the program can in fact behave in unexpected ways, and the problem solving task becomes a reactive one. However, programming tutors up to now have ignored the interaction between the programmer and the dynamic computing environment, and focus on the process of writing the program.

is rejected by the device, which sends a message to the operator indicating it is unable to perform the action; or the procedure's goals are not achieved, assuming, that is, that the problem solver understands what the goals are!). As a result, our novice model produces behaviors, including problem solving impasses, that we have observed and expect in human novice operators.

### 3.3 Expert Problem Solving

We have observed that expert operators behave differently than novice operators, who perform the procedures rote. Expert operators verify directive preconditions and postconditions, which often results in the directives being issued in a different order than what is specified by the procedure manual. Our expert cognitive model reflects this observation by building some additional problem solving knowledge into the novice model, and as a result the expert model performs the task correctly, avoiding many of the problem solving impasses encountered by the novice.

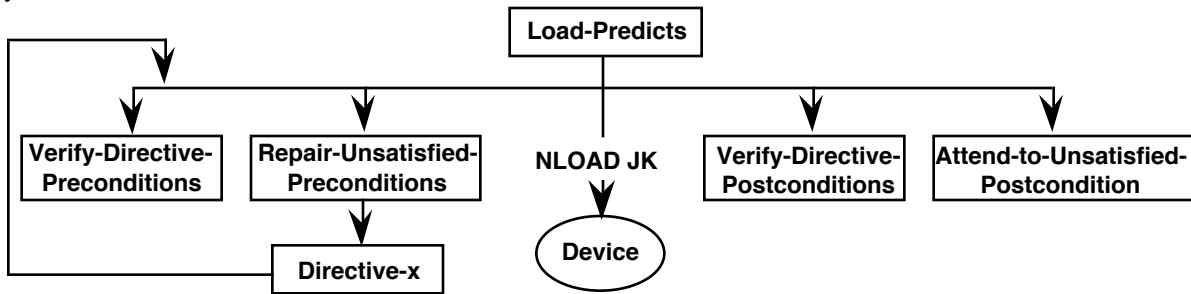


Figure 2 Verification and Recovery Problem Spaces Shown with Load-Predicts Problem Space

We implemented the expert cognitive model, first, by adding two problem spaces below each directive problem space: Verify-Directive-Preconditions and Verify-Directive-Postconditions.<sup>2</sup> These problem spaces are searched before and after applying a directive to a device. Figure 2 shows an example of the problem space hierarchy for the Load-Predicts operator. Whereas the novice would immediately apply the NLOAD directive to the device, the expert first verifies whether the Load-Predicts preconditions, shown in Figure 3, are satisfied. Preconditions and postconditions are modeled declaratively, thus they exist in working memory initially. If the preconditions are satisfied, it sends the NLOAD directive. Once the directive has been sent, it searches the Verify-Directive-Postconditions problem space to verify that the postcondition, shown in Figure 4, is satisfied. If it is satisfied, the Load-Predicts problem space terminates. The process repeats itself when the next directive operator is chosen. In addition to verifying the preconditions and postconditions, we also observe that human operators find ways to work around problem solving impasses. To account for this, we have given our expert model two additional problem spaces: Repair-Unsatisfied-Precondition and Attend-to-Unsatisfied-Postcondition. These problem spaces, which are explained in a later section, are crucial for recovering from failures and for learning.

Operator	Command	Precondition	Device	Attribute	Value
Load-Predicts	V NLOAD <id>	Load-Predicts-PC1	NCB-PROGRAM	MODE	IDLE
		Load-Predicts-PC2	VLBI-PREDICTS	RECEIVED?	YES
		Load-Predicts-PC3	VLBI-PREDICTS	QUALITY	OK

Figure 3 Preconditions for Load-Predicts Operator

Operator	Command	Postcondition	Device	Attribute	Value
Load-Predicts	V NLOAD <id>	Load-Predicts-PO1	VLBI-PREDICTS	LOADED?	YES

Figure 4 Postconditions for Load-Predicts Operator

## 4. Knowledge Compilation

Learning occurs continuously during problem solving in this model. Problem solvers, whether novice or expert, compile their knowledge during subgoaling, where the Soar chunking mechanism builds productions that summarize the subgoal results. Thus, given that the next task situation is similar to the current one, the problem solver will apply the learned chunks instead of searching for an operator in the problem space hierarchy.

<sup>2</sup> This approach was inspired by [Unruh, 1990].

Consequently, as knowledge is compiled the problem solver's procedural performance improves in terms of how rapidly it executes a task.<sup>3</sup> A novice's compiled knowledge differs from an expert's in that it lacks the knowledge of directive preconditions and postconditions as well as the steps involved with verifying that they are satisfied. For instance, Figure 5 shows a chunk that a novice learns while rotely performing the Configure-DSP procedure. This production applies the NLOAD directive immediately after starting the Configure-DSP procedure, without checking the state situation.

```

IF goal is VLBI
& operator is Configure-DSP
& state has name(object, DSP)
-->
command(DSP, NLOAD JK)

```

**Figure 5** Pseudo-code novice chunk that applies the NLOAD directive for the Load-Predicts Operator

This contrasts with the expert chunks shown in Figures 6 and 7. The chunk in Figure 6 verifies that the Load-Predicts-PC1 precondition (from Figure 3) is satisfied. Observe that this production summarizes the result of the Verify-Directive-Preconditions problem-space shown in Figure 2, and marks the evaluation-result with a SAT (for satisfied) if the conditions are met. The left-hand side of the production in Figure 6 also contains information about the goal and state context in which to take the action. A similar chunk is built for each of the preconditions that must be verified for the Load-Predicts operator. In fact, verification chunks will be built for all directive preconditions. Hence, for each precondition that the novice does not know, it will concomitantly lack a verification chunk.

```

IF goal is VLBI
& operator is Configure-DSP
& state has name(object, NCB-PROGRAM)
& state has mode(object, NIDLE)
-->
Load-Predicts-PC1(Load-Predicts, Satisfied)

```

**Figure 6** Pseudo-code expert chunk that verifies a precondition for the Load-Predicts Operator

```

IF goal is VLBI
& operator is Configure-DSP
& state has name(object, DSP)
& Load-Predicts-PC1(Load-Predicts, Satisfied)
& Load-Predicts-PC2(Load-Predicts, Satisfied)
& Load-Predicts-PC3(Load-Predicts, Satisfied)
-->
command(DSP, NLOAD JK)

```

**Figure 7** Pseudo-code expert chunk that applies the command for the Load-Predicts Operator

Figure 7 shows the expert's chunk that recognizes when to apply the NLOAD directive to the device. This chunk can be applied when the goal and state context match the current situation and the listed preconditions (e.g., Load-Predicts-PC1) are all satisfied. Once this chunk exists, it is not necessary to subgoal to the Load-Predicts problem space again unless the actual goal or state context differs somehow from the chunk, or else one of the preconditions is not satisfied. Given the same task in the future, the problem solver will apply its compiled knowledge without subgoaling.

The fact that our model builds two types of chunks for this action instead of one large chunk reflects what we see in expert behavior. Experts think through what state the device needs to be in prior to applying a directive, and this is reflected by the use of working memory elements indicating whether each precondition is satisfied.

## 5. Failure Recovery and Learning

There are two ways that problem solvers can acquire knowledge about preconditions and postconditions in our cognitive model. The first way is to just give the problem solver some or all of the directives' preconditions and

<sup>3</sup>This basically follows the account of knowledge compilation given by ACT\* [Anderson, 1983] and PUPS [Anderson, 1989; Anderson, 1990] where new productions are built to summarize the computations performed while solving a problem. Knowledge compilation in our model differs from PUPS, however, in that the Soar chunking mechanism immediately generates new productions after a subgoal terminates, which contrasts with the analogy-based compilation performed in PUPS. In PUPS, productions are built from the induced implications formed during the analogy process or by composing two or more of these implications [Anderson, 1989].

postconditions. It is helpful to us as modelers to be able to add this knowledge as declarative working memory elements because it enables us to test the behaviors of problem solvers with differing levels of skill, but it doesn't really tell us anything about the process of acquiring and proceduralizing this knowledge. The focus of this section is on a second way of doing this: learning while recovering from failures that are a result of missing or incorrect knowledge. In this section we illustrate how the model recovers from failure with an example, which also reveals how a tutor could intervene to facilitate the learning.

### 5.1 Incorrect Knowledge

Let us assume that the problem solver begins the VLBI task (Figure 1) with a misconception about one of the preconditions of the Load-Predicts operator, namely, Load-Predicts-PC1 (Figure 3). Instead of believing that the MODE attribute should have a value of IDLE, the problem solver expects the value to be RUN. Hence, when the task begins and the NCB-PROGRAM device is in the IDLE MODE, it means that though Load-Predicts-PC1 is actually satisfied, the problem solver erroneously concludes that it is unsatisfied. As a result, it subgoals into the Repair-Unsatisfied-Precondition problem space (Figure 2) and issues a directive to change the NCB-PROGRAM to the RUN MODE to repair the situation.

Once the NCB-PROGRAM device's MODE has changed to RUN, the problem solver believes that Load-Predicts-PC1 is satisfied. Given that the other preconditions are satisfied, it issues the NLOAD directive to the device. But because the device is in the RUN mode, it rejects the directive since it only accepts this directive when it is in the IDLE MODE.

When the problem solver tries to verify that the postcondition (shown in Figure 4) is satisfied and finds that it is not (i.e., it couldn't be since the directive was never accepted), it subgoals into the Attend-to-Unsatisfied-Postcondition problem space. Here it attends to the rejection message from the device and tries to determine a reason for the failure. It reads the message and determines that the NCB-PROGRAM was supposed to be in the IDLE mode. It also notes that the information in the message conflicts with one of its own preconditions, Load-Predicts-PC1. As a result, the problem solver makes a declarative modification of this precondition, changing it from RUN to IDLE.

The problem solver again verifies the Load-Predicts preconditions, this time using the modified version of Load-Predicts-PC1. It is unsatisfied, due to its earlier attempt to correct the situation that put the NCB-PROGRAM into the RUN MODE. Once again the problem solver subgoals into the Repair-Unsatisfied-Preconditions problem space, and this time it selects and issues a directive that puts the NCB-PROGRAM back into the IDLE MODE, which satisfies the corrected precondition and allows the NLOAD directive to be re-issued. This time the directive is accepted, the postconditions achieved, and the Load-Predicts goal is successfully terminated.

### 5.2 Learning while Recovering from Failure

This is a prime example of how our cognitive model acquires new knowledge about directive preconditions in this domain and integrates this knowledge with its existing skills through the knowledge compilation process.<sup>4</sup> Note that the new knowledge, which in this example corrected a misconception, was acquired in the context of a problem solving impasse. The misconception was corrected by providing a new declarative value for the MODE attribute. This new knowledge was applied to recover from the failure, and it was compiled into chunks, which were built each time subgoaling occurred.

Recovery from failures caused by missing knowledge (i.e., missing preconditions) has a similar result: the problem solver attends to the failure, acquires some declarative knowledge, applies it, and moves on to the next part of the task. In cases where the problem solver is missing a precondition but the device happens to be in the correct state, the problem solver's skill will end up being incomplete. The problem solver will only acquire the whole skill during subsequent problem solving episodes where the hidden precondition causes a failure impasse.

## 6. Results and Design Desiderata

We have described a cognitive model that gives an account of problem solving and skill acquisition in a task domain where the problem solver interacts with an unpredictable external environment. We have shown that the difference in skill between expert and novice problem solvers can be explained in terms of the knowledge of directive preconditions and postconditions, but this is only part of the story. ITS researchers have been modeling differences in problem solving behavior along these lines for years (e.g., Langley, 1984; Ohlsson, 1988; Burton, 1982). A smaller number have also modeled skill acquisition (Anderson, 1983,1989,1990; VanLehn, 1987).

---

<sup>4</sup> We have not yet modeled the acquisition of postconditions in this manner, but it will be similar.

Others have built, or are building, detailed cognitive models similar to the one described in this paper. Blake Ward developed a cognitive model in Soar of a student solving electrostatics problems (Ward, 1991). It carefully models many details of the problem solving process, such as perception and focus of attention; however, it does not learn. Ralph Morelli<sup>5</sup> is working to extend Ward's work, and his model does learn. However, it does not address questions of how misconceptions are acquired or recovered from, nor how tutorial intervention might influence the learning process. Conati and Lehman<sup>6</sup> are modeling problem solving and learning in a microworld learning environment called "Electric Field Hockey." It can recall failed problem solving attempts, in order to avoid previous mistakes. However, this domain is more limited than the domain that we are concerned with. It does not involve interaction with a complex environment during an extended problem solving episode. The learning issues that their work addresses are largely orthogonal to the issues addressed by this work; the problem solver in that domain learns sequences of operations to perform to achieve a goal, whereas in our domain the focus is on learning the conditions under which operations are appropriate to be performed and the effects that they should be expected to achieve.

Two things distinguish our cognitive modeling effort. First, it provides a detailed account of both problem solving and skill acquisition in task domains of this type (i.e. unpredictable and reactive), and second, it uses the model to design and evaluate an intelligent tutoring system for this domain. The following paragraphs summarize how the cognitive modeling results have motivated our tutoring system design:

(1) *Learning by doing.* Learning involves solving problems. In our cognitive model this is a direct result of the way that the Soar chunking mechanism works (Newell, 1990), but it also agrees with Anderson's account of proceduralization and skill acquisition (Anderson, 1983). It is not sufficient to acquire declarative knowledge about a task, rather, knowledge must be brought directly to bear on solving problems. Our cognitive model's task performance improves only as it compiles its knowledge about how to perform the domain task.

From these observations, we predict that students will acquire the desired skills most effectively by applying their knowledge to realistic problems. In so doing, novices will compile their knowledge by building chunks that summarize their problem solving experiences. We expect that novices who have compiled their knowledge through practice will perform domain tasks better than ones who have only acquired declarative knowledge about the task. Moreover, a system design that supports interactive problem solving and realistic situations will likely have the best effects.

(2) *Learning about Preconditions.* Novice problem solvers tend to follow a procedural script without paying attention to the device situation, but experts evaluate the situation before and after each action to ensure that action preconditions and postconditions are satisfied. Thus, the knowledge of preconditions and postconditions must be acquired in order to become an expert problem solver in our domain. By focusing on how to assist novice problem solvers to perform procedures in the context of a dynamically changing situation, we expect the novices' overall performance to improve in terms of making fewer errors in a wide variety of situations.

(3) *Rote Learning is Incomplete.* Rote learning of directive sequences is helpful, but, as we have seen, it does not provide complete knowledge of how to do a task. When our cognitive model simulates a novice (i.e., an operator who has no knowledge of preconditions), it solves problems by taking actions in the order specified in the procedure manual. When this knowledge is chunked the problem solver recognizes what step to take based only on the current goal, and it makes errors when the device properties are not correct. Of course, if the procedure manuals included information about the directive preconditions and postconditions then the operators could also rote learn this information, but they would still need to learn how to respond to an unsatisfied precondition or postcondition by reactively planning a course of action. We conclude that the skills required to recognize and react to undesirable situations should be acquired only partially through rote learning; the rest of the skill comes from performing the task and resolving impasses as they arise.

(4) *Learning Occurs in a Goal Context.* Problem solving is a goal-driven activity, and learning occurs in a goal context. This observation is embodied in Soar as a cognitive architecture, and it is also reflected in the way that we constructed the cognitive model for this domain. Though our domain demands reactive problem solving, goals drive the search for solutions. Recall that the chunks built during problem solving include goal context information, which enables the problem solver to recognize when to apply the chunk in future situations.

Because of the central role that the goal context plays in learning, we predict that tutorial intervention will be most effective when the student has the goal to resolve an impasse. The two problem spaces where this

---

<sup>5</sup> Trinity College, Hartford, CT, work in progress

<sup>6</sup> Carnegie Mellon University, Pittsburgh, PA, work in progress

occurs in our cognitive model are *Repair-Unsatisfied-Precondition* and *Attend-to-Unsatisfied-Postcondition*; these are the places where the problem solver actively seeks a way of resolving an impasse, and in our view, this is where the student will most likely benefit from tutoring, since interruption provides information consistent with the student's current goal. Conversely, if the student does not have a goal to acquire impasse-related information, then tutoring is not likely to be helpful since it may disrupt the student's goal stack by turning attention away from the task. This argues against the frequent interruptions of some early model tracing systems caused by a need to inquire about the student's goals, and it also indicates that waiting until after the problem solving session to give tutorial feedback may be too late; it is preferable to intervene tutorially in the context of an impasse. Our tutoring system design has taken these observations into account by focusing on how to recognize and explicate problem solving impasses (Hill & Johnson, 1993).

(5) *Learning via Failure Impasse.* Failures in problem solving provide strategic opportunities for learning new preconditions. Failure impasses change the problem solver's focus from trying to achieve a task goal to trying to recover from the failure, which is a goal that makes the problem solver amenable to acquiring new knowledge. Recall that the cognitive model commits errors when it lacks knowledge of preconditions or has misconceptions. Once the problem solver detects an anomaly, it recovers by first interpreting the situation and the causes for the failure. Next, it modifies its knowledge by acquiring a declarative precondition for the action, and then it compiles the new knowledge with chunking. Each of these recovery steps suggests tutoring intervention opportunities that would not disrupt the problem solver's goal context.

The cognitive model's ability to recover from errors clearly represents an idealized self-tutoring capability. We do not expect human students to be able to tutor themselves as easily as the cognitive model did in the examples described in this paper. Rather, we view the error recovery process as a strategic opportunity for an intelligent tutor to assist the novice in acquiring new knowledge. Consequently, the design of the intelligent tutor should focus on how to capitalize on error recovery situations. This includes the issues of how to create failure situations, how to recognize failures, how to recognize failure recovery, and how to assist the student in recovering from the error.

We also observe that in our domain some errors may not cause a failure impasse for a significant amount of time. An action taken while performing one procedure may not cause a failure impasse until a much later procedure. If the tutor waits until the novice reaches a failure impasse under these circumstances, then we anticipate that the conditions for learning will not be favorable. The goal context of the original error will be long gone, and the error recovery procedure will potentially be convoluted. We conclude that it is better to force the student to deal with undetected errors prior to completing a procedure, thus the tutor will have to be capable of detecting errors within the goal context of a procedure rather than waiting until after a failure impasse. A challenge from a tutoring design perspective is how to force the student to deal with the error without unduly interfering with the student's problem solving activity.

(6) *Detecting Hidden Knowledge Gaps.* Even as the novice's knowledge of preconditions begins to converge with the expert level, there may still be hidden knowledge gaps that are difficult to detect. This is a case where the action sequence order eliminates the need to verify that a precondition is satisfied because the results of one action satisfy the preconditions of the next. As long as the actions are always executed in the same order the precondition will be satisfied and the precondition will remain hidden in the sense that problem solver will not act to verify that it is satisfied. Whereas some actions in a sequence have preconditions that are independent of the other actions, it is difficult to create device situations that will force the student into a failure impasse. The tutor has to find ways to detect hidden knowledge gaps, perhaps by forcing the student out of a rote action sequence and into situations where the actions are executed in a different order.

(7) *Recovery from Incorrect Knowledge.* When our cognitive model recovers from errors it builds new chunks summarizing what it learned during the failure impasse. If the error was caused by incorrect knowledge then there is a potential conflict between the correct and the incorrect chunks. The Soar cognitive architecture does not excise chunks from recognition memory, so what happens to the incorrect knowledge? Our current cognitive model does not have a satisfactory answer to this question. It does not have a problem recovering from incorrect knowledge, but this is mainly due to some fortuitous aspects of our implementation. Building on Laird's approach to recovering from incorrect knowledge (Laird, 1988), we have in mind a way of fixing this so that the problem solver learns chunks that give reject preferences for incorrect operators; these would be learned while recovering from failure. However, to some extent the existing model has already served its function: it has alerted us to the problem of counteracting incorrectly learned knowledge, and we will take this into account when designing the tutor.

## 7. Conclusion

We have described a cognitive model that accounts for problem solving and skill acquisition in dynamic, interactive task domains. The model has given us a number of insights that have directly affected the design of an intelligent tutor for the DSN Link Monitor and Control System (Hill & Johnson, 1993). The tutor's architecture includes: (1) a communications link simulator that models subsystem directive responses and device states, thereby providing an unpredictable external environment; (2) a task environment in which students learn by doing; (3) a situated plan attribution component that recognizes student impasses and decides when to intervene; and (4) an expert cognitive model<sup>7</sup> that decides what to say during the tutorial intervention. The situated plan attribution component focuses on recognizing failure impasses with respect to a goal context, which contrasts with systems that try to understand every student action. The tutor uses the expert cognitive model to provide explanations to the student about what caused the failure impasse and focuses the student on the necessary situational information that is needed to avoid the error in the future. We are currently in the process of testing the veracity of the cognitive model and our predictions about skill acquisition by evaluating the effectiveness of the training provided with the intelligent tutor.

## 8. References

- Anderson, John R. (1983). *The architecture of cognition*. Harvard University Press, 1983.
- Anderson, John R. (1989). Use of analogy in a production system architecture. *Similarity and Analogical Reasoning*, edited by Stella Vosniadou and Andrew Ortony. Cambridge University Press, New York, 1989.
- Anderson, John R., Boyle, C. Franklin, Corebett, Albert T., & Lewis, Matthew W. (1990). Cognitive modeling and intelligent tutoring. *Artificial Intelligence*, 42, 1990.
- Burton, Richard R. (1982). Diagnosing bugs in a simple procedural skill. *Intelligent Tutoring Systems*, edited by D. Sleeman and J.S. Brown. Academic Press, Inc., 1982.
- Hill, Randall W., Jr. & Johnson, W. Lewis (1993). Impasse-driven tutoring for reactive skill acquisition. *Proceedings of 1993 Conference on Intelligent Computer-Aided Training and Virtual Environment Technology*, NASA/Johnson Space Center, Houston, Texas, May 5-7, 1993.
- Laird, John E., Newell, Allen & Rosenbloom, Paul S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(3), 1987.
- Laird, John E. (1988). Recovery from incorrect knowledge in Soar. *Proceedings of AAAI-88*, St. Paul, Minnesota, 1988. Morgan Kaufmann.
- Langley, Pat & Ohlsson, Stellan (1984). Automated Cognitive Modeling. *Proceedings of the National Conference on Artificial Intelligence*, 1984, Austin, Texas, 193-197.
- Newell, Allen (1990). *Unified Theories of Cognition*. Harvard University Press, 1990.
- Ohlsson, Stellan & Langley, Pat (1988). Psychological Evaluation of Path Hypotheses in Cognitive Diagnosis. *Learning Issues for Intelligent Tutoring Systems*. Springer-Verlag, New York, 1988.
- Reiser, Brian J., Anderson, John R. & Farrell, Robert G. (1985). Dynamic student modelling in an intelligent tutor for lisp programming. *Proceedings of IJCAI-85*, Los Angeles, CA, 1985.
- Rosenbloom, Paul S. & Newell, Allen (1986). The chunking of goal hierarchies: a generalized model of practice. *Machine Learning*, Volume II, pp. 247- 288, edited by Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1986.
- Unruh, Amy & Rosenbloom, Paul S. (1990). Abstraction in problem solving and learning. *Proceedings of AAAI-90*, 1990.
- VanLehn, Kurt (1987). Learning one subprocedure per lesson. *Artificial Intelligence*, 31, 1987, 1-40.
- Ward, Blake (1991). ET-Soar: Toward an ITS for Theory-Based Representations. Ph.D. Dissertation, CMU-CS-91-146, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

## 9. Acknowledgement

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

---

<sup>7</sup> One of the side benefits of the cognitive modeling work described in this paper was the expert cognitive model, which has been adapted for use in the intelligent tutoring system.

Dr. Johnson was supported in part by the Advanced Research Projects Agency under contract number N00013-92-K-2015. Views and conclusions contained in this paper are the authors' and should not be interpreted as representing the official opinion or policy of the U.S. Government or any agency thereof.