Using A Simulated Student for Instructional Design

Joseph S. Mertz

International Journal of Artificial Intelligence in Education (1997), 8, 116-141

Using A Simulated Student for Instructional Design

Joseph S. Mertz, JR.
e-mail: JoeMertz@cmu.edu

Center for Innovation in Learning
Center for University Outreach
Carnegie Mellon University,
Pittsburgh, PA 15213, U.S.A.

Abstract. In this paper, I describe how a cognitive model was used as a simulated student to help design lessons for training circuit board assemblers. The model was built in the Soar cognitive architecture, and was initially endowed with only an ability to learn instructions and prerequisite knowledge for the task. Five lessons, and a total of 81 instructions for teaching expert assembly were developed by iteratively drafting and testing instructions with the simulated student. The resulting instructions were in a canonical form, so they were embellished to create humanly palatable lessons using qualitative insights from the simulated student's model of learning from instruction. The constraints imposed by Soar exposed where learning can be difficult for students. During the design process, six types of problems were found and corrected. The paper concludes by reviewing some interesting characteristics of a cognitive architecture-based simulated student.

INTRODUCTION

The classic educational use of a cognitive model is to define the target knowledge for an area of study -- an expert model. The expert model formally defines what a student should know after he or she has completed a course of study. This knowledge cannot be pumped directly into a student, however. That is, the student can not be programmed with the model as a computer can -- the process is much more complex and error prone. An instructional designer must use the expert model as a target, and then design educational materials (e.g. expository text, instructions, worked-out examples, lectures, diagrams) to lead the student to that target. The student, then, must work with the materials, extract information from them, and apply it to the tasks they are given. If all works well, then the student learns an approximation of the knowledge in the expert model. Ordinarily, testing materials with real students is the only way to evaluate if lessons work well, or where they fail. And evaluation results are necessary before the lesson can be revised.

Cognitive models that incorporate learning can assist in instructional design by providing an immediately available test bed for formative evaluation. For example, a model that has both the prerequisite knowledge for a course, and the ability to learn from instructions, can be used to test lessons as they are designed. If the model is presented with a lesson that it successfully learns, then the lesson is given some validation. Conversely, if it fails to learn the lesson, the model can be examined to find out why, and the lesson can be revised. In any case, the lesson receives some debugging before students are subjected to it.

VanLehn, Ohlsson, and Nason have called this type of a model a *simulated student* (1994). The purpose of this paper is to describe how Assembler-Soar, a cognitive model built in the Soar cognitive architecture, was used as a simulated student to iteratively design and evaluate lessons for teaching circuit board assembly.

This paper is organized as answers to a series of questions concerning the use of a simulated student for instructional design. The first three sections concern the design of the simulated student: what is it, what does it learn, and how does it learn. After this description of the simulated student model, I answer how it is used in instructional design. I conclude with a discussion reviewing other simulated students and describe their interesting characteristics as tools for instructional design.

WHAT IS THE SIMULATED STUDENT MODEL?

Assembler-Soar is a model built in the Soar cognitive architecture. Soar has been proposed as a unified theory of cognition, built on a wide range of experimental results from psychological research (Newell, 1990) . It has been successfully used to model a wide range of tasks of appreciable complexity (Lewis et al., 1990) .

Assembler-Soar works in the domain of circuit board assembly. One aspect of an assembler's job is to insert electronic parts into their correct slots in a printed circuit board. A *parts list* defines the mapping of what part goes in what slot. The parts are initially sorted into *parts bins*. Consequently, the assembly task requires iteratively inserting each part in the parts bins into its slot on the board, guided by the information in the parts list.

Inserting parts in a board is a fairly trivial task. For example, the assembler can just pick up a part, search the parts list to find the label of the slot it should go in, and then search the board for that slot. Then he or she can repeat this procedure while there are still slots to fill. What separates expert from novice assemblers, however, is the correctness and efficiency with which they work. Most notably, expert assemblers use

strategies that avoid difficult searches of the circuit board, and learn where parts go to avoid having to search the parts list.

Figure 1 shows how Assembler-Soar evolves from a simulated student to an expert model. Assembler-Soar begins with only prerequisite assembly knowledge and the ability to learn from instructions.

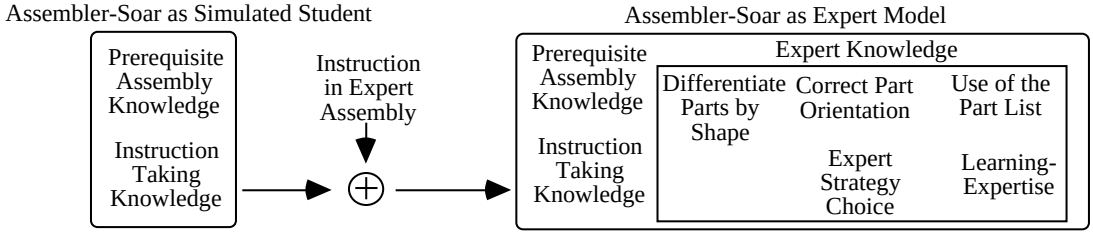

Figure 1. The Evolution of Assembler-Soar into an Expert Model

Its prerequisite assembly knowledge is akin to a novice model of an assembler, and includes knowledge for performing cognitive actions (operators), and control knowledge for choosing between possible operators:

- Operators to initiate motor action - e.g. pickup-part, rotate-part, insert-part
- Operators to control attention - e.g. focus attention on a slot on the board, shift attention to another slot
- Operators to augment internal models - e.g. notice a part's shape and add it to an internal part-model
- Control knowledge to guide when operators apply - e.g. if already holding a part, don't look to the parts bins.

With this prerequisite assembly knowledge, the model knows enough to assemble a board. However, parts will be put in incorrect slots, and their orientations are likely to be wrong. Because the initial model does not bother with correctness, its efficiency (measured in parts-inserted-per-minute) could be quite good.

Assembler-Soar evolves into an expert model, not by directly augmenting its Soar-encoded knowledge, but by having the model interpret and apply the instructions that it "reads." This instruction taking behavior is also knowledge-based; that is, it is not part of the Soar cognitive architecture, rather the model has explicit knowledge of how to learn from instructions. This knowledge is discussed later.

As the model receives instructions and practices assembly, it develops new expert knowledge in assembly. Specifically, Assembler-Soar learns the five expert skills listed in Figure 1:

1. *To differentiate parts inserted by shape.* Correct part placement can sometimes be inferred by finding a slot that matches the shape of a part.
2. *To correctly orient parts*. Some types of parts, such as diodes and integrated circuits, have to be oriented correctly before inserting.
3. *To use the parts list to guide insertions*. The part list defines what the correct part is for each slot.
4. *To apply expert insertion strategies*. Because it is easier to search the ordered part list than it is to search the unordered circuit board, is more efficient to find the part that goes in each slot, than it is to find the slot for each part.
5. *To use learning expertise*. An expert assembler can learn where parts go on the board, in order to avoid having to do any search of the board or part list.

WHAT DOES THE SIMULATED STUDENT LEARN?

Because of the nature of the assembly training task, Assembler-Soar needs to learn operator control knowledge. As was described in the previous section, what separates expert from novice assemblers is the correctness and efficiency with which they work, and the strategies they choose to avoid difficult searches. The basic operators all assemblers use are simple (e.g. pick up part, scan list for item); consequently, they can be considered prerequisite skills that neither Assembler-Soar, nor a real student assembler, needs to learn. But novice assemblers must learn to sequence these operators toward a more efficient strategy (e.g. when to scan the parts-list, or when to stop and try to remember what part goes in a slot). This type of knowledge is operator control knowledge.

In Soar, operator control knowledge is well defined and has three parts: a context, an operator, and a preference. The context is the state of working memory in which the knowledge (expressed as a production) will fire. The operator is a primary unit of cognitive action. And preferences express the desirability of applying an operator, and are considered in the decision procedure for choosing what operator will be applied next. Therefore, given that the task is to learn operator control knowledge, it is easy to work backward from Soar's requirements to define what the content of instruction must be. The instruction given to Assembler-Soar must provide some sense of context, operator, and preference.

Assembler-Soar learns a canonical form of instructions that has three parts: a situation, an action, and advice. The situation describes when the instruction applies. It corresponds to the state of working memory

defining the context in which the control knowledge production will apply. The action names what basic behavior (pickup, insert, scan, compare) the instruction concerns, and corresponds to a Soar operator. And the advice expresses whether the action should or should not be done in this situation. It maps onto Soar's preference options for how desirable it is to choose a given operator in the next selection cycle. An example of a typical instruction is shown in Figure 2. The actual productions that Assembler-Soar learns from the instruction in Figure 2 are listed in detail in the WWW version of this paper at

http://cbl.leeds.ac.uk/ijaied/currentvol.html

Instruction Part	**For Assembler-Soar**	**For a Real Student**
	(instruction (location page)	The instruction on this page says...
Situation -	(situation	When you are
	(look-at bins)	looking at the parts bins,
	(holding nil)	with nothing in hand,
	(attending-to	and attending to
	(isa part)	a part
	(object-model	that you have already compared and found
	(compare (type same)))))	that its type matches the part-model...
Advice -	(advice best)	It is best...
Action -	(action (name compare)	To compare
	(attribute value)))	the part's value with that in the part-model.

Figure 2. Canonical Form of an Instruction with Example

Assembler-Soar does not read a lesson in the same form as a real student. Rather, it reads a nested list describing explicitly a situation, action, and advice. This is a much different form than a real student typically sees. There are at least three ways to view this difference:

1. *Prerequisite Knowledge.* When training assemblers, the ability to comprehend natural language text (and diagrams, tables, graphs, pictures, etc.) is assumed to be a prerequisite skill. That is, just as the ability to pick up a part is considered a simple prerequisite that a student approaching the lesson should have, students are also expected to be able to comprehend natural language text and to extract from it the relevant situation, action, and advice. If a student does not have the prerequisite skill, then they are unlikely to be successful in learning the lesson. This is true whether the prerequisite is comparing shapes of parts, or extracting the key components from natural language instruction.

2. *Parsimony.* A model of natural language text (and diagram, etc.) comprehension is unlikely to provide significant insight into design of instruction for assembly. In any use of a simulated student, the modeling effort should be concentrated where it is most likely to benefit the instructional design process.
3. *State of the Art.* The state of the art in machine vision and natural language processing currently makes comprehending real student instruction infeasible. (Huffman's Soar model of an instructable agent does have a limited natural language ability (1993; 1994) .)

Assembler-Soar used the canonical form of instruction for all three of these reasons.

While the canonical form of instruction is sufficient for the task given to Assembler-Soar, it would be quite desirable for a simulated student to comprehend natural language text. This would free the instructional designer from the task of translating between the canonical and real instruction, and could provide insight into educational problems that happen at the text (or diagram, etc.) comprehension level.

HOW DOES THE SIMULATED STUDENT LEARN?

To be used as a simulated student, Assembler-Soar needed a model of learning instructions that could expose problems similar to a real student's.

Unlike some other cognitive theories, Soar does not have an architectural ability to learn from instructions. For example, ACT (Anderson, 1983; Anderson, 1993) has an architectural means by which declarative examples are adapted by analogy into procedural rules. And more recently (Anderson & Fincham 1994), ACT has moved away from the necessity of declarative examples being first stored in long term memory. As will be evident in the following discussion, any model of learning from instruction in Soar could not currently move away from this necessity.

While Soar does not implement instruction-taking architecturally, it does have a ubiquitous learning mechanism on which all more complex forms of learning can be built. Ohlsson (1987) has reported on another system which proceduralizes declarative knowledge. It has a set of learning mechanisms that are not part of the architecture, but are implemented as productions. These productions are treated slightly differently than task-dependent productions, however. And while the productions that implement the learning mechanism are constrained by psychological theory, the underlying model is a production system, not a model of cognition.

Hayes-Roth, Klahr, and Mostow (1981) have also reported on a significant system that can learn from instructions. Their system could operationalize advice, expressed as LISP expressions, consisting of concept definitions, behavioral constraints, and performance heuristics. They did not make a distinction between declarative and procedural knowledge, rather a concept was operational if it could be evaluated. Advice was made operational by transforming it, using other known concepts, until the result could be fully evaluated. Consequently, it appears that some advice was operational as given, while most advice had to be transformed. They had a rich set of transformational rules (about 200). Hayes-Roth, Klahr, and Mostow were successful in learning a sophisticated card game (Hearts) and they reported having success in other domains, including music. Their research demonstrates the sophistication necessary from a logic standpoint to operationalize advice. It would be interesting to achieve this same level of sophistication, while conforming to constraints of a cognitive theory, to better inform us how humans learn complex instruction. For example, most cognitive theories do make some sort of distinction between declarative and procedural knowledge, at least to the point that received input is never directly operational.

Assembler-Soar builds on previous work on instruction taking in Soar. The earliest work was by Golding, Rosenbloom, and Laird (1987) . They reported on a model that performs simple algebra problems, and asks for advice when it doesn't know how to proceed. The model accepts direct advice, or an illustrative small problem from which it could infer advice.

Lewis, Newell, and Polk (1989) reported on a model that learns instructions for immediate reasoning tasks. Their model performs natural language comprehension, and builds a behavioral model from simple English instruction. This behavioral model is interpreted in order to perform the immediate reasoning task.

Vera, Lewis, and Lerch (1993) reported on a model that learns how to use an automated teller machine (ATM). The ATM's screen provides instructions for each step, and the model is able to comprehend and apply the instructions in order to perform the task. Their research focused on how Soar learned in a simple situated task.

The most significant work so far is reported by Huffman and Laird (1993; 1994). By developing an instructable autonomous agent that performs tasks in a robotic blocks-world, they have provided Soar with a very general theory of instruction-taking called *situated explanation.* As the name implies, the flavor of this instruction is interactive, and initiated by the model.

How do the assembly task and the form of instruction Assembler-Soar learns compare with these other models?

First, Assembler-Soar perceives and manipulates a model of the external world. This is similar to models performing in the ATM and

robotic blocks-world tasks. The models that perform the algebra and immediate reasoning tasks are purely cognitive ones. They do not rely on visual input.

Second, all instructions for performing a task are presented to Assembler-Soar before the task is begun. Only the model that does the immediate reasoning task accepts instructions in this way, the other three models accept instructions interactively while at the task. Table 1 summarizes these two comparisons.

	The Model Interacts with an External World	The Model Performs Cognitive Operations Only
Instruction is Delivered Interactively When Needed	Vera, Lewis, & Lerch (1993) Huffman (1994)	Golding, Rosenbloom, & Laird (1987)
Instruction is Delivered Before it is Applicable	Assembler-Soar	Lewis, Newell, & Polk (1989)

Table 1. Comparison of Soar Models that Learn Instructions

The ramification of these differences is that Assembler-Soar must not only learn what to do, but when to do it. For the model to acquire more efficient strategies, it must know when to apply an instruction to change behavior that otherwise would have been acceptable. In other words, it has to recognize for itself situations where the old behavior is no longer desired, and apply new knowledge from the instruction. Conversely, the algebra model does not have to recognize when the advice applies. It applies immediately to the situation in which it has been requested. Likewise, the instructions in the ATM model say what to do now, not some time in the future.

Instructions given to the immediate reasoning model apply to the future, but it is unclear what the model would do if they advised different behavior than it had previously learned. And finally, the robotic blocks-world model deals only with adding to incomplete knowledge, not augmenting or replacing that which is already complete.

Because of these differences in the task and the form of instruction, Assembler-Soar had to build on the previous Soar work, to add an ability to visualize a future situation in which an instruction would apply. When an instruction is being studied, the model visualizes the situation that it describes. This enables the model to learn to recognize that situation when it perceives it later while doing the task. This idea will be described in more detail later in this section.

Soar theory attempts to define a parsimonious cognitive architecture. In a Soar model, no matter what is being learned, there is only one way to learn. Yet with this ubiquitous learning mechanism, it is possible to model

a wide range of learning behavior. Assembler-Soar learns from instruction, and in doing so, provides a process-level view of what is necessary to learn from instruction, given this unitary learning mechanism.

The theoretical essence of Soar's basic learning mechanism follows:

- Whenever a knowledge impasse prohibits the continuation of normal behavior...
- A subgoal is created, providing an opportunity to reason about that impasse.
- Results returned from the subgoal toward overcoming the impasse are cached as a long term memory production. The memory is an association linking the information in the supergoal that was used to arrive at a result, and the result itself.
- Whenever identical information appears in working memory again, the cached long term memory will apply, avoiding any impasse and its corresponding need for reasoning in a subgoal.

As a high-level analogy of how this learning mechanism works, consider a driver who one morning discovers that the usual route from home to work is blocked by road work. The normal continuation of the task of getting to work is prohibited and a subgoal (i.e. "get around the blocked point") is created to deal with the situation. When a search through the surrounding streets reveals a detour around the blockage, that detour is learned and encoded in long term memory as a production rule. When the same obstacle is encountered on the next morning, the new rule fires and no problem solving is necessary to find the detour.

Assembler-Soar's canonical instructions have the same functional content as Soar productions, so it could appear that the translation from instruction to production should be simple. In fact, the ramifications of Soar's basic learning mechanism exposes just how difficult the process can be for students. These difficulties are interesting, not because they make the modeling task harder, but because they highlight, according to the Soar theory of cognition, what problem solving is necessary to learn from instruction. A knowledge-based, process-level understanding of how instructions are learned can show where students are likely to have problems and can help instructional designers be sympathetic to the learning process.

In building the learning model in Assembler-Soar, I attempted to add only minimal knowledge to Soar's architectural abilities. The alternative would have been to develop the model from human data or existing research. The purpose of the minimal approach was to exploit the constraint the architecture provides in order to see what insight it brings to

understanding instruction. This has been referred to as "listening to the architecture." This approach also provided the parsimonious path to supporting Assembler-Soar as a simulated student.

In the remainder of this section, I will discuss two difficult issues in modeling learning from instruction in Soar, and the problem solving knowledge that Assembler-Soar was given to overcome these problems. An even more detailed description can be found in Mertz (1995) .

Recognizing When an Instruction Applies

Consider again the example of navigating the route from home to work. What if the driver did not hit road work, but the night before had talked to a friend who described a faster path. In this case, the problem solving necessary to learn the new detour would not be forced by the environment. Rather, the driver would, in some other way, have to break from the habit of the former route. The driver would need to somehow create a need to do problem-solving at the appropriate time (i.e. at the appropriate street), and recall and apply the friend's instruction. It would be all-too-easy to just follow the old route by habit (i.e. by compiled knowledge).

Assembler-Soar faced a similar problem. It could perform the task with the knowledge it had. But it needed to learn how to perform the task more efficiently -- to learn a new route. Therefore, recognizing when an instruction applied was one difficulty Assembler-Soar had to overcome. An instruction describes a situation, but the model requires operational knowledge to recognize when the contents of working memory match that description. Without operational knowledge to flag when an instruction applies, the model would continue to behave in old ways, ignoring new instructions.

When viewing an instruction, Assembler-Soar visualizes the situation in which it applies by mimicking what working memory will look like in that situation. Instruction-taking knowledge then bars further progress until two symbols exist in working memory which a) uniquely name the instruction situation and b) flag that the instruction has been studied.

Because progress cannot continue until these conditions are met, an impasse is hit and a subgoal of "study the instruction situation" is created. In this subgoal, the working memory contents relative to the situation are examined, and the two appropriate symbols are returned. In this way, two new productions are learned. One fires when the context of working memory matches the one described in the instruction and adds a unique symbol to working memory. The second fires when viewing the instruction again and flags that the instruction has already been studied.

Returning to the home to work example, the first production would fire when the driver got to the corner that the friend had described was where to turn, and would provide a way of breaking from the normal route (and

from compiled knowledge). The second production is less useful, and would only apply if someone again told the driver about this faster route.

Breaking the Association of Knowledge from its Source

Once the driver knows where to turn, the new-route problem solving only begins. Which way did the friend say to go? The second hurtle Assembler-Soar overcame is related to the difficulty of breaking the association of knowledge from external sources (e.g. an instruction), from those sources. In other words, it is easy to follow instructions while looking at them, and hard when they are taken away. Truly learning instructions requires taking the hard step of following them while not looking at them. In Soar, this difficulty is part of what is called the data chunking problem (Newell, 1990; Rosenbloom, Newell & Laird, 1991) .

This difficulty is manifested in Soar as a consequence of two assumptions. The first is how learning is achieved (as results are returned from a subgoal), and the second is that all visual input enters working memory only into the top goal state. (In other words, visual input cannot enter into a subgoal state.) Consider learning an instruction that has a situation, action, and advice. The declarative form of this instruction that enters the top goal state of working memory is:

Situation, Action, Advice

The desired operational form of this knowledge is in the form of a production:

Situation -> Action, Advice

That is, in a given situation, apply some control knowledge advice to an action. The model of learning from instruction must make this transformation. How can this be done? Recall that long term memories are built as results are returned from a subgoal. First consider the simple case: with a goal of learning an instruction, the model hits an impasse, lacking knowledge of the instruction. The instruction is then examined in a Soar subgoal, and the action and advice returned to overcome the impasse. In this case a memory would be created:

Situation, Action, Advice -> Action, Advice

The action and advice appear on the result (right) side of the association, because they are what was returned from the subgoal, but they also appear on the condition side because it was their existence in the top goal state of working memory (as visual input) that motivated the result. This is a useless memory because it only applies if action and advice are already in working memory! If they were not, then the memory would not apply. Consequently, if the model were to look away from the instruction, say to assemble a circuit board, the memory would never apply.

Returning to the driving example, if the driver's friend is always sitting in the car pointing where to turn, then the driver will not learn the new route. Only by doing problem solving beyond following the friend's immediate instruction will the new route be learned to the extent that the driver can follow it alone. Else, the memory is created: "When seeing the corner of Murray and Forward Avenues and hearing it is best to turn left" -> "turn left".

The condition-side dependence on the action and advice must be broken. (e.g. "When seeing the corner of Murray and Forward -> "turn left".) To do this, the information returned from a subgoal must come from someplace other than the visual (or auditory) input. The only remaining option is from long term memory.

To achieve this, first the information can be put into long term memory in the form of a recognition memory:

Situation, Action, Advice -> symbol-1

In this form, symbol-1 serves merely to recognize the declarative information of situation, action, and advice (e.g. "It is best to turn left at the corner of Murray and Forward" -> symbol-1).

Later, when actually performing the assembly task, operationalize the instruction. The process begins when the production described in the previous section recognizes the instruction situation, and puts a symbol signifying the situation into working memory (e.g. "at the corner of Murray and Forward" -> "situation-1"). Instruction-taking knowledge then bars further progress until the instruction advice has been applied, as flagged by a symbol signifying that the instruction has been applied.

Because progress cannot continue until this condition has been met, an impasse is hit and a subgoal of "recall the action and advice for this situation from instruction" is created. In this subgoal, consider all combinations of alternative operations afforded by the current context, and possible advisements of which there are only two, "it is best", or "reject" (e.g. it is best to turn right, reject turning right, it is best to turn left, reject turning left, it is best to go straight, etc.). The long term memory:

Situation, Action, Advice -> symbol-1

will "recognize" the correct alternative combination (e.g. "It is best to turn left at the corner of Murray and Forward" -> symbol-1). The application of the correct advice to the correct action can then be returned from the subgoal. This will create the desired long memory:

Situation -> Action, Advice

(i.e. "At the corner of Murray and Forward" -> "It is best to turn left.")

Notice that only the situation appears on the condition (left) side of the association, because it is the only information from the supergoal that was

used to recall the instruction. The action and advice were retrieved from long term memory (by generating and testing alternatives).

Whenever the same working memory context is recognized in the future, the new knowledge will immediately apply, avoiding any further impasse. The instruction would have been truly learned and operationalized.

Summary

To recap, Assembler-Soar's canonical form of instruction is not the same as a Soar production, and its model of learning from instruction makes explicit the type of problem solving necessary for a student to operationalize instructions.

The Soar cognitive architecture pushed Assembler-Soar's model of learning from instruction in an interesting direction. Classically, learning has been thought of as something one has direct control over, just as one has control over moving a finger. Soar suggests a somewhat different approach. There is no "learn" operator that directly stores away knowledge. An individual can control reasoning, but not what is learned. Learning can only be controlled indirectly, by reasoning in ways that are sympathetic with how the mind learns. In Soar, learning happens when impasses are hit, when new knowledge is needed to continue. If reasoning is not done in ways in which impasses are hit, then nothing will be learned. If the surface structure of instructions are read without reasoning about their content, then nothing will be learned. But by choosing to reason in ways in which knowledge impasses are hit, new knowledge can be learned.

This process-level model of learning from instruction is useful in two ways. First, as a simulated student, Assembler-Soar will fail in learning instructions when the conditions necessary for applying the process are not met, and in doing so, will flag problematic instructions. Second, once a full set of canonical instructions have been developed and tested with the simulated student, qualitative insights from the learning model can be used to embellish the instructions into a full student lesson. Both of these points will be discussed again later in this paper.

HOW IS THE SIMULATED STUDENT USED?

Designing instructions with a simulated student is an interactive process, and the classic roles of novice and expert models are reinterpreted somewhat. Writing a lesson is done by working with the simulated student in a process of drafting, testing, and revising instructions and exercises. The novice model defines the prerequisites to the sequence of lessons. Instead of a singular penultimate expert model, a sequence of increasingly expert models is created at each step of the design process. And finally,

the penultimate expert model is not a source of instruction, rather it is a validation of the instruction that generated it. In other words, the expert model is not built directly, rather the novice model is taught to be an expert, therefore the expert model does not precede the instruction but is developed alongside it. This does, of course, presuppose that there is some target expert behavior that the expert model can be externally validated against.

The Instructional Design Process

Figure 3 depicts how lessons are designed using a simulated student. It shows the iterative process of designing, testing, and revising a sequence of lessons. The process begins by defining the prerequisites for the lessons. Next, each lesson is drafted, tested, and revised. Once the student is able to successfully learn from a lesson, a new knowledge state exists on which to build future lessons. Eventually, the last lesson brings the student to the fully expert knowledge state.

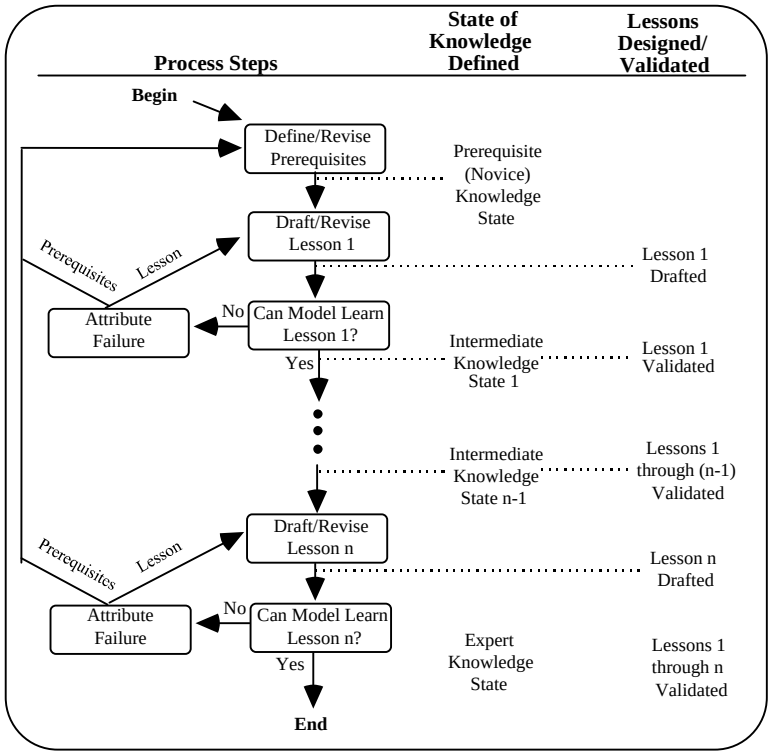

Figure 3. The Instructional Design Process with a Simulated Student

Defining Prerequisites. The instructional design process begins by defining the lessons' prerequisites. This is done by developing a novice model of whatever entry-level skills are necessary. There are two types of knowledge required of the simulated student: Task knowledge and learning-from-instruction knowledge.

Task knowledge is mostly domain specific. For example, Assembler-Soar has knowledge of assembly-specific actions such as pick-up-part and look-at board. Other task knowledge includes the basic general skills on which task-specific skills are built. For example, Assembler-Soar begins with a general attention mechanism and learns to use it to perform assembly-specific searches.

Learning-from-instruction is a more general type of knowledge that does not depend on the task. For example, Assembler-Soar's theory of learning from instruction is not specific to assembly; the ability to learn control knowledge from direct instruction can apply to many domains.

Once the novice model has been built, its contents define the knowledge necessary to learn the lessons. As will be seen, however, this definition is likely to be refined as the instructional design process continues.

Drafting the Lesson. Lessons are drafted by writing a set of instructions in a form that the simulated student can learn. The lesson can only build upon the student's current knowledge state. For the first lesson, this is equivalent to the novice model. For later lessons, it is the intermediate state of knowledge leading into that lesson. In either case, the knowledge that the lesson can build on is well defined.

If the type of instruction is restricted to the form that Assembler-Soar can learn, then the instruction design process is quite constrained. With a target behavioral change in mind, drafting an instruction requires defining the situation in which the behavioral change will occur, defining the action to be affected, and defining the new advice to be applied to that action.

Testing the Lesson: Does the Model Learn? After defining the instructions and exercises, they are tested with the simulated student. The lesson is successful if the simulated student studies the instructions and performs the exercises, and as a result, acquires the lesson's target knowledge. If so, then the instructional designer can move on to the next lesson.

Attributing Failure. The whole point of using a simulated student is to identify points of failure, and to fix them. Failure of the model to learn can be attributed to either the prerequisite knowledge, or to the lesson. In either case, however, the model is available to examine exactly where the failure occurred.

The problem might be with the prerequisite knowledge. For example, in Assembler-Soar, an instruction might refer to an action that was not in the novice model. Because Assembler-Soar can only learn control knowledge for actions it already knows, this action must be added to its prerequisite knowledge.

The problem might be in the instruction. For example, it might describe a situation that is implausible, in which case, the instruction

would never be learned. Consequently, the instruction would have to be changed.

Similarly, the problem might be in the practice exercise. If the exercise does not afford the student with situations that match those in the instructions, then those instructions will not be learned.

The instructional designer often has a choice as to where to attribute the failure, and where to make the necessary changes. In general terms, the decision is: "Is this something that a real student is likely to know beforehand?" If it is, then it should be added to the prerequisite knowledge. If it is not, then the lesson should be changed.

Revising. Finally, once the source of failure has been determined, either the prerequisite knowledge or the lesson can be revised, and the lesson can then be re-tested.

A Characterization of the Knowledge Learned

Assembler-Soar was successful in learning the 5 expert skills listed in Figure 1. As a result, 81 canonical instructions in 5 lessons (one per skill) were designed using this simulated student of assembly. The canonical instructions are listed in detail in the WWW version of this paper at

http://cbl.leeds.ac.uk/ijaied/

Assembler-Soar was capable of starting from its novice state and learning all lessons in a single run. Each lesson required studying the instructions and assembling a circuit board one or more times. The number of assemblies varied for each lesson. For the lessons *Differentiate Parts by Shape, Correct Part Orientation, Use of the Part List, and Expert Strategy Choice*, only one practice assembly was needed to give the simulated student all the situations it needed to operationalize the lessons. The *Learning-Expertise* lesson, however, required at least 2 assemblies of the same board. In the first, the student would learn about the parts that went in each slot. That knowledge would then be used to assemble subsequent copies of the circuit board.

Besides learning from instructions (as exemplified in Figure 2), Assembler-Soar also learned while assembling boards. As the model assembled boards, it continually acquired knowledge about what parts it had inserted and what parts go in each slot. Therefore the model learned both in "training" (i.e. studying and practicing) and "on-the-job. The knowledge built in both ways applied in later situations to varying degrees.

1. **Application to later instruction and assembly**
   When studying instructions, recognition knowledge of instruction situations is built. These memories transfer not only from instruction to practice (as has been described earlier), but from one

set of instructions to another. If a situation is similar to an earlier one studied, the similar parts are immediately recognized and only the different parts must be assimilated.

2. **Application to later assembly**
   Operationalized instructions apply in all subsequent assembly tasks.
3. **Application to later assemblies of identical boards**
   Information about the parts that go in each slot is assimilated as a board is assembled (e.g. slot I47 takes an LC451 chip). This knowledge applies to later assemblies of identical boards, and is used to be able to insert parts without resorting to searching the parts list.
4. **Application only within the assembly of the current board**
   Some memories are relevant only to the currently-being-assembled circuit board. While they apply later in the assembly (e.g. "already inserted the 10K ohm resistor into slot R1), they are not relevant after the current board is finished.

No knowledge is deliberately rigged to *never* apply again, but temporal flags augment some learned knowledge and deliberately limits its transfer to future situations. The practical effect is that some learned knowledge will never apply again. This knowledge does not transfer, not because it is somehow wrong, but rather it is knowledge that applies only to the immediate problem being solved and is not relevant to the future. It can be considered as episodic knowledge that applies while in that episode, but afterward it will not apply unless there is reason to reflect on that episode. The duration of the episodes in Assembler-Soar correspond to the duration of objectives and sub-objectives. As an analog, consider the problem solving necessary to decide which slice of pie to choose from the cut whole. Knowledge is learned in making the decision, but this choice of piece has no practical application to later pies. In Assembler-Soar, the knowledge created is flagged with a unique instance tag, and this tag is never matched again in future problem solving instances.

While learning the 5 lessons, 1747 productions were learned. 724 (43%) applied to some other situation (i.e. fired at least once), the remaining 998 were never used. Of those, only 17 were learned as applying only narrowly to an instance of a sub-objective. 46 productions were tied to an objective, such as assembling a single board, and would have applied if the relevant situation had arisen, but would not apply after the board was completed.

Before learning, Assembler-Soar had no knowledge to guide the correct insertion of parts and therefore made many mistakes (parts were chosen in the order they were found in the parts bins and were inserted in

the next empty slot the model found on the board). After learning the two lessons: *Correct Part Orientation* and *Use of the Part List*, the model made no mistakes in orientation or part placement.

In Soar, a decision cycle is a handy unit of temporal progress. In each decision cycle, an operator is applied, a subgoal is created, or a new problem space or state is selected. Consequently, the number of decision cycles required to complete a task is a rough measure of its cognitive difficulty. Figure 4 shows the difficulty to assemble the same practice board after each lesson. Notice that the difficulty increases with each new lesson, because the instructions, in effect, make the task more difficult. For example, for the board that was assembled before learning any instruction, the model does not have to expend any cognitive effort determining if part is oriented correctly in the slot.

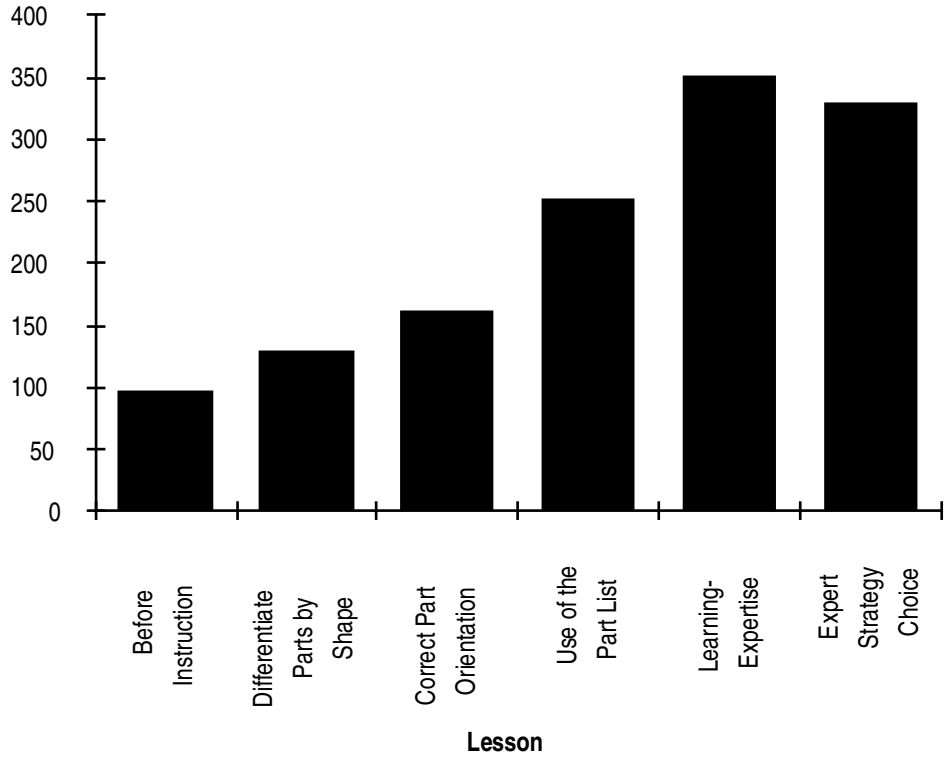

Figure 4. Effort to Assemble Practice Board After Each Lesson

The last two lessons were each meant to make the task easier than after the *Use of the Part List* lesson. The *Learning-Expertise* lesson actually makes the task harder. This is because the practice board that was used each time afforded the opportunity to operationalize the learning-expertise instruction, but did not have enough parts to make the investment in learning where parts go worthwhile. The *Expert Strategy Choice* lesson did show some moderate gain in efficiency. While Assembler-Soar was

able to model behavioral strategies that would be more efficient on larger boards, the model was not run on such boards to be able to empirically demonstrate those gains.

Types of Problems Found Using the Simulated Student

In the course of the design process, several types of problems were found and corrected:

Unknown referent: the instruction referred to a term the model did not yet know about. For example, Assembler-Soar made use of objectives, which were task-oriented goals that the model pursued, such as search, take-note-of-the-parts-shape, or use-the-expert-insertion-strategy. These objectives had to be learned before they could be subsequently used in later instructions. Similar failures would occur if operators that were assumed to be prerequisite knowledge were not already part of the model.

Ill-defined situation: an instruction situation can be either too generally or too selectively defined. When a situation is too generally defined, it would be recognized at times when the instruction should not apply. For example, one of the skills that were taught was *learning expertise*: an expert assembler can learn where parts go on the board, in order to avoid having to do any search of the board or part list. Implementing this strategy requires doing some special problem solving just after a part is inserted. But this situation must be carefully delineated to not do the same problem solving on parts that had been previously inserted, only the part that was last inserted. Conversely, when a situation is too selectively defined, it may not apply in situations where it could.

Inappropriate practice: the practice exercise that accompanied an instruction did not afford a situation in which the instruction could be applied. This was often a problem with the exercise boards that Assembler-Soar practiced on. For example, the learning expertise strategy takes advantage of the fact that identical parts are often located next to each other on a circuit board. If the practice board did not have this feature, then it would not learn the strategy.

Inappropriate behavior: although the instructions were learned successfully, the resulting behavior was not what the designer planned. And example of this arose when designing instructions to apply expert insertion strategies. Because it is easier to search an ordered part list than it is to search the unordered circuit board, is more efficient to find the part that goes in each slot than it is to find the slot for each part. In the less efficient strategy, Assembler-Soar would keep a marker in mind for where it left off in the parts list so that it could continue with the next line. But when searching the whole parts list for the new strategy, it was a problem if the model initiated its search where it had last attended. Consequently,

while the new instructions were successfully learned, interactions with old knowledge sometimes produced incorrect behavior.

Refinement of behavior: the instructions were learned successfully, but when watching the resulting behavior, better ways of doing the task were seen. For example, early instructions directed Assembler-Soar to compare the slot name for each slot it was looking at with the one being searched for. (The slot name is a label stenciled on the circuit board next to each set of holes.) It became obvious from watching the model's behavior that comparing the slot names of slots already containing parts was silly and that only the names of empty slots should be compared. This instruction was then further refined to only compare the slot names of empty slots whose shape matches the part to be inserted.

Slips: the instruction has small oversights. For example, accidentally referring to a *part* instead of a *part slot*.

Creating Real Instruction from Simulated Student Instructions

Because the instructions were in a canonical form, the instructional design process was not complete. Rather, it was still necessary to embellish the instructions into a more humanly palatable form. In order to maintain as much discipline as possible in the process, this was done in two passes. First, I performed the most basic translation of the canonical instructions to English sentences. Because these sentences are brief and direct, they appear to be proverbs. An example of such a proverb is given in the rightmost column of Figure 2.

In the second pass, I further transformed the instructions into general prose and added diagrams. To do this, I anticipated the type of mental work that students would have to do with the instructions, guided by Assembler-Soar's model of learning from instruction, and embellished the instructions to be sympathetic to that process. For example, the model suggests that an instruction must be visualized, therefore the lesson was embellished to support this visualization. This can be done in a variety of ways. Good prose can richly describe a situation, using concise detail to help the learner to visualize it. What is the situation? What is being seen? What would already be known in the situation? What should or should not be done? A diagram can be worth 10,000 words, for they very directly aid the learner's visualization. In any case, the key is to provide information so that the learner can visualize applying the instruction, which is necessary in studying it.

For teaching assembly, the proverbs were used directly to "give away" the essence of the instruction. This was done by including them in the lesson, but in such a way that they stood out from the rest of the text. As a result, the situation, action, and advice, are highlighted, priming the learner for what is important.

The specific referents in the original canonical instructions are critical to the instruction, however, and should not be lost or made too fuzzy to extract. They are the concepts that the simulated student identifies as being necessary to discern the uniqueness of a situation or of an action. The lesson should strive to make these referents clearer, not to obfuscate them, for the latter would negate the discipline brought to the process of instructional design.

DISCUSSION

VanLehn, Ohlsson, and Nason (1994) published a review surveying work that has been done with simulated students. They describe three uses of simulated students: teacher training, student collaboration, and formative evaluation. Assembler-Soar is an example of the third.

The Sierra system (VanLehn, 1987) is another simulated student that has been used in formative evaluation. Sierra can learn arithmetic skills from solved examples. In particular, the model learns an ordered sequence of lessons, where a lesson is an unordered sequence of examples. The examples can be positive examples (of things to do) or negative examples (things not to do). As the Sierra model solves arithmetic problems, it uses solved examples to augment its procedures whenever it reaches an impasse. Sierra can succeed in learning arithmetic skills if it is given a sequence of lessons that lead it to hit impasses in its knowledge, and if the examples are adequate for it to bridge each impasse. The sequence of lessons and the availability of examples must be such that a gradually more sophisticated skill is built. If a lesson leads Sierra to an impasse that a single inference taken from an example cannot bridge, then the simulated student will fail to learn. In this way, lessons and examples can be evaluated for their efficacy.

HS (Heuristic Searcher) has also been used as a simulated student in formative evaluation (Ohlsson, 1992; Ohlsson, 1993; Ohlsson, Ernst & Rees, 1992) . It also learns arithmetic skill, but from instruction, not examples. The instruction is presented to the model interactively, immediately whenever the model makes a mistake. HS learns by repairing over-general rules which are detected when they lead to incorrect behavior. The instructions given to HS are actually constraints that describe erroneous states. If the model makes a decision that leads it to a state described as erroneous in an instruction, the model repairs the knowledge that made that decision so that the incorrect behavior will not be repeated. In this way, the instruction is learned and guides future behavior. In the real world, these constraints would be relayed by a human tutor at the time an error is made. HS was used as a simulated student to compare the performance of learning two alternative strategies for doing multi-column

subtraction. Specifically, it modeled acquisition across two subtraction strategies and two instruction methods. Whenever the model made an error, the researchers would add a new constraint/instruction, and run the model again from the beginning. From this experimentation, they were able to compare the efficacy of the alternative instructional designs.

Like Sierra and HS, Assembler-Soar is a fine-grained, algorithmic model. Also like Sierra and HS, Assembler-Soar learns control knowledge that changes decisions concerning the application of prerequisite primitive operations. Sierra and Assembler can learn positive and negative control knowledge, however, while HS learns only negative. Each of the three models learn from different forms of "instructions." Sierra learns from solved examples, HS from state-constraint instructions, and Assembler-Soar from situation/action/advice instructions.

As simulated students, Sierra, HS, and Assembler-Soar were used in similar, but not identical, types of formative evaluation. Sierra was used to show that one set of examples was more effective than another. HS was used to show the efficacy of one strategy over another. Assembler-Soar was used to develop a sequence of lessons to teach a series of increasingly expert job skills. In effect, Assembler-Soar was used to make a series of instructional decisions, each replacing an ineffective instruction with an effective one.

The major difference between Assembler-Soar and the other two models is that it is built in a general cognitive architecture. HS and Sierra were each developed specifically to model aspects of arithmetic skill, and their implementation decisions are individually supported by empirical evidence and/or developed theory. Assembler-Soar's development was different. Data on expert assemblers was collected and formed the target behavior for the model, but many of the implementation decisions of the model were already constrained by the Soar cognitive architecture. Specifically, Soar's modeling language and execution environment are designed toward expressing only those behaviors that are considered to be cognitively plausible (i.e. there is psychological data to support the type of behavior).

Several benefits are derived from Soar's well-defined capabilities and constraints. The first is quite practical: it can make modeling easier. Many modeling decisions, such as how long term memories are created and retrieved, are already defined and supported by empirical evidence. Modeling is also simplified because Soar's constraints often point to fewer ways to model a given behavior. Another benefit is that Soar bestows on a model some degree of validity. Simply put, if the cognitive architecture is validated by human data, any model built in it inherits that validation. In reality, the inheritance may be a modest, first-order validity, but it is a start.

A final benefit of building a simulated student in a general cognitive architecture like Soar is the packaged ability and mutual constraint that comes from other models built in the architecture. General cognitive models deal with one aspect of cognition, and while researchers make cases for how results of one model correspond to other findings, these claims are not verified by a combined execution of multiple models. Models built within a cognitive architecture are also often built to deal with one aspect of cognition, but because they exist in the architecture, they become part of a larger comprehensive model. For example, NL-Soar is a model of language comprehension, built in the Soar cognitive architecture (Lehman, Lewis & Newell, 1991; Lewis, 1993). Instructo-Soar is a model of learning from instruction that incorporates NL-Soar to comprehend written instructions (Huffman & Laird, 1993; Huffman & Laird, 1994). NTD-Soar is a model of the behavior of a NASA Test Director that uses NL-Soar to comprehend the text of the Director's manual (Nelson, Lehman & John, 1994) . In both cases, NL-Soar provides some natural language capability, and constrains what assumptions the models can make about natural language comprehension.

CONCLUSION

Advances in cognitive science have created new opportunities for using computational models. It is now possible to develop simulated students that can aid teachers, students, and instructional designers. I have described one such simulated student, Assembler-Soar, which was used to design instruction for an adult vocational task. From this work, I've identified a few interesting characteristics of using a simulated student.

The simulated student is an evolving model. Instead of being a single expert performance model, the simulated student is an evolving model. Each intermediate state of knowledge is an expert model for the completed lesson and a novice model for the ensuing one.

A simulated student highlights detailed changes in knowledge. When using novice and expert models to design instruction, the difference between the two gives a gross measure of what must be learned. This approach potentially misses intermediate states of knowledge that are necessary to make the transition to more expert behavior. The simulated student method brings to light those intermediate states.

A simulated student permits the use of multiple methods of learning. Unfortunately, knowledge cannot be directly infused into human long term memory. Consequently, a model of how real students learn from external sources is integral to a simulated student. Assembler-Soar learns control knowledge from direct instructions; other simulated students have learned control knowledge from solved examples, and state-

constraint instructions. There are a variety of other methods of learning that a simulated student could use, if it knew how. Some include learning new actions instead of just control knowledge, and learning by interpreting diagrams, graphs, or tables.

A simulated student can test alternative ways of instructing a learner. The instruction design process depicted in Figure 3 produces a linear sequence of lessons. The same process can be repeated, however, if a variety of different sequences are desired, or if alternative methods of learning are to be tried.

Cognitive architecture-based simulated students inherit capability and validity. Models built in a cognitive architecture can borrow capabilities from, and be constrained by, other models implemented in the same architecture. This mutual constraint adds validity to all the models involved.

As with any cognitive modeling task, the primary disadvantage of using a simulated student is the time and expertise that is necessary to build one. Because Assembler-Soar is built in a general cognitive architecture, it is hoped that further research with simulated students could build on it and on the capabilities being developed by other researchers using Soar. For example, of Assembler-Soar's 59 operators, only 7 are assembly-specific. The rest provide general skills such as controlling attention, learning from instruction, and augmenting situation models with perceived information. By reusing this knowledge, the marginal effort of building new simulated students is reduced.

While simulated students are not applicable to every instruction design task, since existing methods and technologies are effective enough for most teaching, the insights from cognitive psychology and AI together can take on problems where learning has not been effective enough. It is in these cases that simulated students can be employed. As the science of learning and instruction has progressed, first performance was modeled, then learning. Simulated students integrate these two advances into a technology where learning toward better or new performance is modeled. One practical use of such a model is the ability interactively design and test instruction.

Acknowledgments

This research was funded by the James S. McDonnell Foundation through a grant to Jill H. Larkin. This research was conducted and an early version of this paper was written under the her advisorship. I would like to thank Ken Koedinger for his supportive suggestions and my wife Rebecca for her editing help. I would also like to thank Stellan Ohlsson for his suggestions concerning this paper, especially for the concise example of learning a detour around a street blockage.

REFERENCES

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

Anderson, J. R. (1993). *Rules of the Mind.* Hillsdale, NJ: Erlbaum.

Anderson, J. R. & Fincham, J. M. (1994). Acquisition of Procedural skills from examples. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 20*, 1-20.

Golding, A., Rosenbloom, P. S., & Laird, J. E. (1987). Learning general search control from outside guidance. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*.

Hayes-Roth, F., Klahr, P., & Mostow, D. J. (1981) Advice Taking and Knowledge Refinement: An Iterative View of Skill Acquisition. In Anderson, J. R. (Ed.), *Cognitive Skills and Their Acquisition*, 231-253, Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Huffman, S. B., & Laird, J. E. (1993). Learning Procedures from Interactive Natural Language Instructions. *Proceedings of the Machine Learning Tenth International Conference*.

Huffman, S. B., & Laird, J. E. (1994). Learning from Highly Flexible Tutorial Instruction. *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA.

Lehman, J. F., Lewis, R., & Newell, A. (1991). Integrating Knowledge Sources in Language Comprehension. *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*.

Lewis, R. L. (1993). *An Architecturally-based Theory of Human Sentence Comprehension*. Unpublished doctoral dissertation, School of Computer Science, Carnegie Mellon University.

Lewis, R. L., Huffman, S. B., John, B. E., Laird, J. E., Lehman, J. F., Newell, A., Rosenbloom, P. S., Simon, T., & Tessler, S. G. (1990). Soar as a Unified Theory of Cognition: Spring 1990. *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*, Cambridge, MA.

Lewis, R. L., Newell, A., & Polk, T. A. (1989). Toward a Soar theory of taking instructions for immediate reasoning tasks. *Proceedings of the Annual Conference of the Cognitive Science Society*.

Mertz, J. S. (1995). *Using a Cognitive Architecture to Design Instructions*. Unpublished doctoral dissertation, Carnegie Mellon University.

Nelson, G., Lehman, J. F., & John, B. (1994). Integrating Cognitive Capabilities in a Real-Time Task. *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, Massachusetts: Harvard University Press.

Ohlsson, S. (1992). Artificial instruction: A method for relating learning theory to instructional design. In M. Jones & P. H. Winne (Eds.), *Foundations and frontiers in instructional computing systems*, (pp. 55-83). Berlin, Germany: Springer- Verlag.

Ohlsson, S. (1993). Impact of cognitive theory on the practice of courseware authoring. *Journal of Computer Assisted Learning, 9*(4), 194-221.

Ohlsson, S., Ernst, A. M., & Rees, E. (1992). The Cognitive Complexity of Learning and Doing Arithmetic. *Journal for Research in Mathematics Education, 23*(5), 441-467.

Rosenbloom, P. S., Newell, A., & Laird, J. E. (1991). Toward the Knowledge Level in Soar: The Role of the Architecture in the Use of Knowledge (Chapter 4). In K. VanLehn (Ed.), *Architectures for Intelligence*, (pp. 75-111). Hillsdale, NJ: Erlbaum.

VanLehn, K. (1987). Learning One Subprocedure per Lesson. *Artificial Intelligence, 31*(1), 1-40.

VanLehn, K., Ohlsson, S., & Nason, R. (1994). Applications of Simulated Students: An Exploration. *Journal of Artificial Intelligence in Education, 5*(2), 135-175.

Vera, A. H., Lewis, R. L., & Lerch, F. J. (1993). Situated Decision-Making and Recognition-Based Learning: Applying Symbolic Theories to Interactive Tasks. *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, Boulder, CO.