A Development System for Model-Tracing Tutors

John R. Anderson and Ray Pelletier
Department of Psychology
Carnegie Mellon University, Pittsburgh, PA 15213 USA

Abstract: Our model-tracing methodology for tutoring is based on the ACT* theory of skill acquisition. According to that theory a skill can be analyzed into a set of productions rules and instruction can be organized around these rules. Students' problem-solving behavior can be interpreted and tutored by tracing their solution through the production rules. Problems can be selected to optimize mastery of production rules. Because of recent advances in production-system efficiency we have been able to develop a general system for building model-tracing tutors. Developing a tutor in this system involves constructing an interface, defining the production rules, creating declarative instruction, and setting up the knowledge tracer to manage the curriculum.

Since 1984 we have developed a number of tutoring systems which have all exemplified what we have called the model-tracing approach to intelligent tutors. The first three systems were a tutor for the programming language LISP (Anderson & Reiser, 1985), a tutor for proofs in high school geometry (Anderson, Boyle, & Yost, 1985), and a tutor for symbol manipulation in high school algebra (Lewis, Milson, & Anderson, 1987). All three systems have been used in actual classroom situations with generally positive outcomes. Some evaluations of these systems are reported in Anderson, Boyle, Corbett, and Lewis (1990). More recently we have developed and deployed at CMU a single tutoring architecture which teaches LISP, Pascal, and Prolog to undergraduate freshman (Anderson, Corbett, Fincham, Hoffman, & Pelletier, in press) with generally positive results. We have also developed a tutor for high school word algebra problems which has had successful laboratory tests but has not yet been deployed in the classroom (Singley, Anderson, Givens, & Hoffman, 1989). Although specific evaluations produce varying magnitudes of effects our general estimation is that our tutors improved student achievement by about one standard deviation given constant amount of time or at least halved the time to reach the comparable levels of achievement.

Each of these tutors has had a production system model of the skills they were supposed to teach. This involved developing a set of rules which in combination would perform the target skill. A typical tutor would involve as many as 500 of these rules. These rules reflected a cognitive model which we thought could underlie successful problem solving in that domain. Thus, for instance, we would have a production rule for geometry of the following form:

IF the goal is to prove $\triangle ABC$ is congruent to $\triangle DBE$
and AE and CD are collinear
THEN infer $\angle ABC$ is congruent to $\angle DBE$ because they are
vertical angles.

The rule reflects the wisdom that it is useful to make a vertical angle inference when trying to prove triangles congruent.

Organizing a set of production rules which will solve a class of problems is like creating an expert system with the added constraint that it be cognitively plausible. This is not a particularly easy task but having done it we are in a powerful position to deliver instruction. The cognitive model defines the instructional objectives—basically to communicate to the student each of the underlying production rules. Student interaction with our tutors takes the form of solving problems while interacting with the computer system. The tutors could use their production system models to interpret the students' behavior and provide feedback. Basically, our systems try to find some sequence of production rules which will match the students' behavior and use this sequence as an interpretation of the students' problem solving. The approach is called model tracing because it traces the students' behavior with a set of production rules.

Each of the tutors we built was more or less constructed anew. There was only informal transfer from our experience in constructing previous tutors. We felt that we had reached the point where we could articulate what we were doing when we built a computer tutor and create a system that facilitated that process. We thought it would be useful to codify this as a development system for a number of reasons. First, it would provide a precise definition of what our approach is to tutor construction. Second, it could be made available to others who wanted to create model-tracing tutors. Third, we felt we could endow such a system with facilities which would improve the process of developing such a tutor and the performance of the tutor once developed.

We will describe in this paper the system we have developed for building model-tracing tutors (Anderson et al., in press). It has evolved as a generalization of the multiple programming languages tutor. Its generality is currently being tested by having it embody the word problem tutor (Singley et al., 1989). It is best described as a development system and not an authoring system. It does not eliminate the need to go through the same steps as we did when we created our first tutor. Rather it just structures and facilitates the development process.

In this paper we will go through the basic principles which we feel lie behind our approach to tutor development. Then we will describe the essential steps involved in creating a tutor in our system.

Principles of Model-Tracing Tutors

We have previously (Anderson, Boyle, Farrell, & Reiser, 1987) published a list of cognitive principles for the development of tutors based on the ACT* theory (Anderson, 1983) of cognition. We have subsequently realized that the list was overly elaborate and involved issues not really central to the conception of skill acquisition in ACT*. The basic implications of ACT* for instruction are really quite simple but important.

The first principle derived from ACT* is that it is essential to define the target cognitive model as a set of productions rules. Without this one does not have a well-defined educational goal within our approach. The task of building such a model is not easy although we have developed a number of helpful facilities within our system. Our view is that this is the most labor intensive aspect of tutor development. It is also worth noting that there is not necessarily a unique set of production rules which prescribe how students should approach a task. Just as there are multiple possible programs for doing a task there are multiple possible production systems. There is no guarantee that one will come up with the best production system for performing a task. Some of our recent work on geometry tutoring

is predicated on the belief that we have found a more powerful cognitive model for proof construction (Koedinger & Anderson, 1990) than the tutor which underlay the original geometry tutor.

The second principle concerns how these productions are to be communicated to the student. The ACT* theory has a quite specific proposal for how production rules are acquired. According to ACT* one cannot directly tell students the underlying rules. Rather, students form these rules as a byproduct of problem-solving. Their initial attempts to solve a problem involving a rule take the form of analogy to some example that illustrates the rule. Thus, one needs to communicate the rules to students by providing them with examples that illustrate the rules. The examples need to be embellished to communicate the underlying rules. Thus, to communicate the production rule given earlier one might present the student with an example and the accompanying embellishment "When trying to prove triangles congruent that form a vertical angle configuration it is a good idea to infer that the vertical angles are congruent." When the student next comes across a problem that involves this rule, they can solve that problem by analogy to this example. If they do, the production rule will be formed by the knowledge compilation process in ACT* (Anderson & Thompson, 1989).

The third principle of tutoring that derives from ACT* is that one wants to maximize the rate at which students have opportunities to form and practice these production rules. Because of errors in understanding of the examples it may take multiple opportunities before the correct production rule is formed. Also, further practice of a rule increases its strength and so the reliability of its application. We have done a number of experiments looking at different modes of tutoring which vary how long it takes subjects to go through a set of problems and the way in which they go through the problems. We find that what predicts students final achievement is how much practice they have had of these rules and not how that practice occurs. At the extremes, we have compared students who are forced to always stay on a path of correct solution with students who are allowed to explore any path of solution they like with the only constraint that they finally execute the correct solution. Students in the second condition will take two to three times as long to get through the problems but show the exact same level of achievement. The basic implication of this principle is that one wants to minimize floundering time in problem solving and select problems which will offer practice on those productions where students most need practice. Much of the overall organizations of our tutors are motivated to maximize learning rate.

The fourth principle of tutoring that derives from ACT* concerns how to treat errors in student problem solving. Traditionally, much of the intelligence in intelligent tutors has been aimed at error remediation but we find little theoretical justification for extensive efforts at error remediation. There are four considerations which must be addressed in how one responds to errors: The first consideration is that many errors do not reflect misunderstandings or lack of knowledge on the students part but simply slips. The second consideration is the fact that people learn best when they generate the answer for themselves rather than are told (Anderson, 1990). The third consideration is that a great deal of time can be wasted in error states. The fourth consideration is that when students have problems with their knowledge what they need is another opportunity to learn the correct production (rather than a deep appreciation of their error). Thus, our recommended line of action is one in which we start out by simply pointing out an error to a student when it occurs without further comment. This avoids burdening the student with instruction when it is not needed, allows the student to self correct, and gives the students the opportunity to avoid going into an extended error state. We find that the majority of the time students are able to correct their errors without further instruction. When students cannot and request help, they are given the same kind of explanation that would accompany a training example. Specifically,

we focus on trying to tell them what to do in this situation rather than focus on telling them what was wrong with their original conception. Thus, in contrast to the traditional approach to tutoring we focus on reinstruction rather than bug diagnosis. There has been some recent evidence that this is more effective (Sleeman, Kelly, Martinak, Ward, & Moore, 1989). This is not to say that our tutors are incapable of explaining what is wrong with the solution when asked. However, what they give at such points is more like an intelligible error message than it is like an attempt to debug a misconception.

Tutor Development

Within our system tutor development takes place in a characteristic sequence. In this section we would like to explain this sequence, what each step involves, the connection of each step to the general principles of tutor development, and what facilities we are providing to support that step of development. Right now this all takes place within a Macintosh Allegro LISP system and this will undoubtedly color our exposition. However, we hope to see our development system generalized beyond that platform.

Constructing a Tutor Interface

The first thing one must do in creating one of our tutors is to define the interface in which the problem solving is going to occur. This contrasts with the early tutors where we first developed the production system model and then searched for some tutor interface that would "realize" it. This shift in development approach reflects the observation that the interface defines in part the task that the student has to master and there is no definition of the skill independent of the interface. An interesting example concerns our tutors for teaching computer programming which use a structured editor interface that relieves the student of the burden of mastering the syntax of the programming language. The skill to be mastered in this interface is different than one which would require mastery of syntax and students who emerge from this tutoring do not know the syntax although they do not appear to have great difficulty in picking it up.

A number of factors impinge on the design of the interface. One involves decisions about what is important to master. In the case of computer programming we made a decision that syntax was not important and we wanted to focus on the semantics of the language. In general, if one is using a computer system for instruction it makes sense to use a relatively high-tech interface that does the mundane for the student and allows the student to focus on the more significant aspects of the domain. As a consequence our interfaces can serve as useful tools for problem solving even in the absence of a tutor. Another factor in designing an interface is to facilitate the process of model tracing in the tutor. In particular, it is useful to create an interface that involves a high density of rich input from the student to help us diagnose the student's problem-solving state.

The interface itself must be created in LISP although we have provided a number of facilities to assist the user. Creating an interface involves defining a number of interface interactions which have two side effects. One is to produce some transformation to a workspace on the screen. Thus, if a student selects "procedure" from a menu in our Pascal tutor and template for a Pascal procedure will appear. The second side effect is to deposit in the working memory of the production system a representation of this transformation of the screen. The consequence of this coupling is that the working of memory of the tutor always represents faithfully the problem state before the student on the screen. The actions

in our tutor consist of either selecting menu items or typing values into slots created in the workspace.

Writing Production Rules and Model Tracing

The production rules for the system are defined in terms of the interface and its representation in working memory. The production rules match to states of the problem representation in working memory and call for interface actions. The actual problem representations contain more than just what is in the interface but also interpretations of these elements. Thus, production rule for programming might be, informally stated:

IF the goal is to expand the output of a writeln statement
and the output is the product of two numbers
THEN select "*"
and pass the numbers to the arguments of "*".

The first clause "expand the output of a writeln statement" refers to a screen state where the writeln statement has not been completed. This is something of the form:

writeln(**<output>**);

"the output is the product of two numbers" represents an interpretation associated with the unexpanded argument, **<output>**. "select "*" refers to an interface action (select this from a menu). This will transform the code so that it looks like:

writeln(**<arg1>** * **<arg2>**);

"pass the numbers to the arguments of "*" " will serve to annotate **<arg1>** and **<arg2>** with the numbers they should represent.

When the tutor is in a model-tracing mode, it tries to find some sequence of production rules in its internal model that will match up to the behavior of the student. The tutor's goal in model tracing is always to account for the next action of the student. There may be multiple production rules that fire in the student model in order to make the transition between the current state and the next action. There are three possible outcomes when the tutor tries to interpret the next action of the student:

(1) A single sequence of productions is found which will produce that action. This then is taken as the interpretation of that segment of the behavior.

(2) Multiple sequences of productions are found. It is then not possible to interpret (model trace) subsequent behavior of the student until the ambiguity is resolved. This ambiguity can be resolved by presenting students with a menu of possible interpretations of their behavior and letting them determine the correct one.

(3) There is no interpretation of the student's behavior. The presumption is that the student's behavior is in error at this case. One thing we can do is to see if the behavior can be recognized by a buggy production sequence. Again until this behavior is changed, it is not possible to interpret (model trace) subsequent behavior.

What is actually done in states (2) and (3) depends on the choice of tutoring discipline. In immediate feedback mode the tutor will request an interpretation from the student in case (2) and force the student onto a correct path in case (3). However, in other modes it is possible to just suspend further interpretation and let the student continue to try to solve the problem in the interface. We only try to resolve the ambiguity in (2) or force a correct solution in (3) when the student requests further help from the tutor. We have looked at two versions of a less directive tutor. In one version, called flag tutoring, we resolve the ambiguities in (2) as they arise and warn the students of possible errors in (3) as they arise but do not force students to correct errors before going on. In another version, called demand-feedback tutoring, we neither try to resolve ambiguity or correct errors until the student requests help from the tutor.

One of the major technical accomplishments in our new tutoring architecture is that the production system is much more efficient than previous ones which allows us to search through multiple sequences of production rules looking for a match without perceptible pauses. The current system is also largely unaffected by problem complexity. In the case of programming, for instance, we have been able to tutor many-hundred line programs. We have also included a good many facilities to help the production system writer develop and debug the production rules.

Declarative Instruction

Once the production rules have been written we can then engage in the process of attaching to these production rules the declarative instruction which will help serve to communicate to the student the content of these rules. This instruction is meant to accompany the original example-based instruction or to provide correction in the context of a problem. Thus, the instruction is always example-based, with the example either being the training example or the test problem. The instruction takes the form of templates with slots which can be filled by English descriptions of the specific example. Thus, attached to the multiple production above might be a template of the form:

> At this point you want to write out the product of <slot1> and <slot2>. Use the symbol '*' to calculate a product.

where <slot1> and <slot2> corresponded to variables bound in the production. If in this example, the value bound to <slot1> had the English description "the employee's salary" and the value bound to <slot2> had the description "the applicable tax rate" this would become:

> At this point you want to write out the product of the employee's salary and the applicable tax rate. Use the symbol '*' to calculate the product.

Similar templates attached to error productions can be used to give context-specific error messages and descriptions in disambiguation menus.

As noted, there are two places at which we want to deliver such declarative instruction. One is when we are introducing the productions and the other is when the student needs reinstruction at a point of error. It turns out that these situations are not very distinct in practice. We actually deliver the original instruction in the context of a problem. One difference is that when a student gets to a point which requires instruction on a new rule,

we simply volunteer the information right away; whereas, we wait until an error or request for help when it is an old rule. Another difference is that when we are helping the student at a point which they should know, we may want to initially provide less information and only later provide more direct information. Thus, we might first tell the student "At this point you want to calculate the product of the employee's salary and tax rate." Hopefully, this will be enough and the student will enjoy the benefit of partially solving this step. Only if necessary would we provide the additional "Use the symbol '*' to calculate the product." Thus, we have the facility to store successively stronger hints with a production. The first time a student is introduced to a production all of these hints are amalgamated into a unit of instruction.

In preparing declarative instruction, the developer must both attach these templates to production rules and associate English descriptions to elements of a problem. The template construction is a one-time investment but each problem entered into the system needs its own English embellishments. We have been working on facilities to expedite problem entry.

Knowledge Tracing and Mastery Instruction

There is a distinction between what we call model tracing and knowledge tracing. Model tracing refers to trying to interpret a student's solving of a single problem as a sequence of production rules in the tutor. From this attribution we can make inferences about how well the student knows the various productions. Knowledge tracing refers to trying to infer the growth of knowledge about the production rules across a sequence of problems. This is critical to optimizing the curriculum sequence. We can associate with each problem the production rules it involves. Our decision about what problem to next present to the student and when to promote the student onto new material is made in response to our interpretation of how well they know the various productions.

A Bayesian inference model for making such interpretations has been described in Corbett and Anderson (in press). Basically, it allows us to assign to each production rule a probability that the student knows that specific production. We select the problem that involves the maximal learning opportunity where this is defined as the problem with the highest density of productions with low probabilities. We continue presenting problems to students until we reach the point where the instructional gain (expected increase in their probabilities) for all the problems falls below a threshold defined by the length of the problem. Then we promote the student to the next unit in the curriculum sequence.

The overall curriculum is broken up into units where each unit involves the introduction of about a half dozen new productions. The first problem in each unit is meant to explicitly illustrate the new material and later problems are for purposes of practice. One of the tasks of the curriculum designer is to specify this division of the overall curriculum into units and specify the training problems. The other responsibility is to associate a number of parameters with the Bayesian inference scheme which define the difficulty of the various productions and the ability of the students. With this in place the system does a nice job of bringing students to mastery of the productions in each unit.

Summary

While the system is still in development, we have had considerable experience doing curriculum development with it in our lab and hope to make it available to other researchers. The importance of the production rule development comes strongly through in the

experience of curriculum development. It is the component where most of the time is spent and it serves to organize subsequent development such as the design of declarative instruction. Interface design is separate but is largely a one-time initial investment whereas each unit of curriculum requires a fresh, full effort at production-rule modeling. Our system has made it easier to develop these rules and once developed provides a overall system in which they can organize efficient instruction. However, we see no way of relieving the developer of the essential question in our approach which is "How should the student think about solving this class of problems?"

This research was supported by Contract MDA-903-89K-0190 from the Army Research Institute and Grant 87-51890 from the National Science Foundation.

References

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

Anderson, J. R. (1990). *Cognitive psychology and its implications, Third Edition*. New York: W.H. Freeman.

Anderson, J.R., & Reiser, B.J. (1985). The LISP tutor. *Byte, 10*,159-175.

Anderson, J. R., Boyle, C. F., & Yost, G. (1985). The Geometry Tutor. In *Proceedings of IJCAI-85.* Los Angeles, CA: IJCAI, 1-7.

Anderson, J. R., Boyle, C. F., Corbett, A. T., & Lewis, M. W. (1990). Cognitive Modelling and Intelligent Tutoring. *Artificial Intelligence, 42,* 7-49.

Anderson, J. R., Boyle, C. F., Farrell, R., & Reiser, B. J. (1987). Cognitive principles in the design of computer tutors. In P. Morris (Ed.), *Modeling Cognition,* Wiley.

Anderson, J. R., Corbett, A., Fincham, J., Hoffman, D., & Pelletier, R. (In press). General Principles for an Intelligent Tutoring Architecture. In V. Shute and W. Regan (Eds.), Cognitive Approaches Automated Instruction.

Anderson, J. R., & Thompson, R. (1989). Use of analogy in a production system architecture. In S. Vosniadou and A. Ortony (Eds.), *Similarity and analogical reasoning,* 267-297. Cambridge University Press.

Corbett, A. T. & Anderson, J. R. (in press). The LISP intelligent tutoring system: Research in skill acquisition. In J. Larkin, R. Chabay, C. Scheftic (Eds.), *Computer Assisted Instruction and Intelligent Tutoring Systems: Establishing Communication and Collaboration.* Hillsdale, NJ: Erlbaum.

Koedinger, K. R., & Anderson, J. R. (1990). The role of abstract planning in geometry expertise. *Cognitive Science, 14,* 511-550.

Lewis, M. W., Milson, R., and Anderson, J. R. (1987). The teacher's apprentice: Designing an intelligent authoring system for high school mathematics. In G. P. Kearsley (Ed.), *Artificial Intelligence and Instruction*. Reading, MA: Addison-Wesley.

Singley, M. K., Anderson, J. R., Gevins, J. S., & Hoffman, D. (1989). The algebra word problem tutor, *Artificial Intelligence and Education*, 267-275.

Sleeman, D., Kelly, A. E., Marlinak, R., Ward, R. D., & Moore, J. L. (1989). Studies of diagnosis and remediation with high-school algebra students. *Cognitive Science, 13,* 551-568.