

Authoring Model-Tracing Cognitive Tutors

Stephen B. Blessing, *Department of Psychology, University of Tampa,
Tampa, FL 33606, USA*
sbleasing@ut.edu

Stephen B. Gilbert, *Human Computer Interaction and Department of Psychology, Iowa
State University, Ames, IA 50011, USA*
gilbert@iastate.edu

Stephen Ourada, *Clearsighted, Inc., Ames, IA 50010, USA*
s@ourada.org

Steven Ritter, *Carnegie Learning, Inc., Pittsburgh, PA 15219, USA*
sritter@carnegielearning.com

Abstract. Intelligent Tutoring Systems (ITSs) that employ a model-tracing methodology have consistently shown their effectiveness. However, what evidently makes these tutors effective, the cognitive model embedded within them, has traditionally been difficult to create, requiring great expertise and time, both of which come at a cost. Furthermore, an interface has to be constructed that communicates with the cognitive model. Together these constitute a high bar that needs to be crossed in order to create such a tutor. We outline a system that lowers this bar on both accounts and that has been used to produce commercial-quality tutors. First, we discuss and evaluate a tool that allows authors who are not cognitive scientists or programmers to create a cognitive model. Second, we detail a way for this cognitive model to communicate with third-party interfaces.

Keywords. Authoring Tool, Cognitive Tutor, Model-Tracing Tutor

INTRODUCTION

Intelligent Tutoring Systems (ITSs) have been shown effective in a wide variety of domains and situations. Within the different types of ITSs, model-tracing tutors have been particularly effective (e.g., Anderson, Conrad, & Corbett, 1989; Koedinger, Anderson, Hadley, & Mark, 1997; Ritter, Kulikowich, Lei, McGuire & Morgan, 2007, VanLehn, et al., 2005). These tutors contain a cognitive model of the domain that the tutor uses to check most, if not all, student responses. For example, the Andes physics tutor by VanLehn and his colleagues (2005) contains a cognitive model of how to solve problems within physics. This model is referenced for each step in the problem solving process in order to make sure the student stays on path. The model can also be used to provide hints and other assistance to the student learning the material.

This type of intense interaction and feedback has led to impressive student learning gains. Students using the Andes physics tutors have demonstrated large gains in effect size over paper-and-pencil homework on both standardized and more conceptual, experimenter-designed tests (VanLehn, et al., 2005). Results from another model-tracing tutor that instructs students on how to program show

that they can master the material in a third less time (Corbett, 2001). Field trials comprising hundreds of students learning algebra using Carnegie Learning's Algebra I model-tracing tutor have shown significant improvement on standardized tests when students use such a tutor (Ritter, Anderson, Koedinger, & Corbett, 2007).

Despite these successes ITSs, including model-tracing tutors, have not been widely adopted in educational or other settings, such as corporate training. Perhaps the most successful deployment of model-tracing tutors is Carnegie Learning's Cognitive Tutors for math, which are in use by over 1300 school districts and by hundreds of thousands of students each year. After this notable success, however, most educational and training software is not of the ITS variety, let alone a model-tracing ITS, in spite of the impressive learning advantages. There are two clear and related reasons for this lack of adoption: the expertise needed to create such a system and the costs involved.

Both of these issues, expertise and cost, are connected to the fact that to create a complete model-tracing ITS requires many different software pieces: an interface, a curriculum, a learner-management system, a teacher report package, and the piece that provides the intelligence, a cognitive model. Many kinds of expertise are needed to create these pieces, most notably programming (both GUI and non-GUI), pedagogy, and cognitive science. This generally requires a team of highly trained people to work together, thereby resulting in a high cost and a large time expense. Past estimates have found that it takes 100 hours or more to create 1 hour of instruction (Murray, 1999; Woolf & Cunningham, 1987). Taken together, these factors of expertise and cost obviously are a large part of why model-tracing ITSs are not in wider use.

The goal we have for this project is to assist the cognitive modeler to do his or her job better and more efficiently. While team design processes are essential for creating tutors in general, the interdependence of the team can result in bottlenecks. If the cognitive modeler requires programming assistance in order to get a piece of work done, this slows down the effort. A person who desires to create a tutor for a particular domain (be it a cognitive scientist or a master teacher) should not be required to be a programmer to make substantial progress in creating the basics of the cognitive model.

Before discussing our work in detail, we will briefly provide an historical perspective on how cognitive tutors have been authored, so that our current direction can be evaluated with regards to the previous work. Table 1 provides a view of the ancestry of the SDK and a brief comparison of the features to be discussed.

Table 1
The Ancestry of the Cognitive Model SDK

| System | Declarative Knowledge | Procedural Knowledge | Student Interface | User Base |
|-----------------------|-------------------------|----------------------|-------------------|---|
| Tutor Development Kit | Working Memory Elements | Productions | LISP | Cognitive Scientist/programmer |
| TRE/LISP | Type Hierarchy | Rules | Java | Cognitive Scientist/programmer |
| Cognitive Model SDK | Type Hierarchy | Predicate Tree | Java/third-party | Non-Cognitive scientist/minimum programming |

Historical Perspective

The first cognitive tutors were the result of a test of Anderson's ACT Theory of cognition: if cognitive skills can be realized as units of goal-related knowledge, then successful learning should be accomplished by teaching students using those units (Anderson, Corbett, Koedinger, & Pelletier, 1995). After initial success with this hypothesis, a tutor development kit (TDK) was created in LISP (Anderson & Pelletier, 1991). In its representation of the cognitive model the TDK borrowed heavily from the ACT representation. There was a clear delineation between declarative knowledge and procedural knowledge. Programming a TDK-based tutor required much knowledge of cognitive science, particularly with respect to how procedural knowledge could be represented by production rules and declarative knowledge by schematic structures. It also required much programming experience, specifically in LISP. Teams using this system were limited to researchers and graduate students from the laboratory at CMU where it was developed. However, this was the system that created the programming language and algebra tutors that saw great success in the early 1990s and were used by thousands of students.

With commercialization of this system came recognition that cross-platform delivery (including via the web) and performance considerations necessitated a separation between the delivery environment and the development environment. A representation for the tutor delivery environment was developed that is referred to as the Tutor Runtime Engine (TRE). An authoring system was created to produce tutors that output TRE-based systems (see Ritter, Blessing, & Wheeler, 2003, for a fuller description). In accomplishing this goal, a clear separation was created between the student's interface and the underlying cognitive model that provided the tutoring. This enabled the tutor's interface to be in Java, and potentially any other language. Authoring, however, was still done in a LISP-based environment. The distinction was made between declarative and procedural knowledge. Those able to create tutors using this system were restricted to a small number of people, all of whom possessed a fair degree of cognitive science and programming knowledge. The algebra and geometry tutors originally created with the TDK were ported over to this representation, and many new lessons and cognitive models were created using these tools. Furthermore, these tutors have been deployed to over 500,000 students in over 1300 school districts across the United States and the world. This placed additional quality assurance demands on these tutors, and these tools have shown they were up for the task. Indeed, in a very practical sense, we see the ability to easily perform QA tasks on our tutors as an important goal.

The Cognitive Tutor SDK

Soon after Cognitive Tutors were being created and deployed using the TRE/LISP authoring tool and Java interfaces, we began to design and develop the Cognitive Tutor SDK. This SDK is meant to ease the creation of all aspects of an ITS: cognitive model, curriculum, problems, and interface. Across all these pieces, a main challenge in the work is coming up with representations that enable the ITS author to do their work with little or no programming and with a clarity not offered by present systems. We have concentrated our current efforts on two main pieces of the ITS: the cognitive model and the interface.

The aspect of a model-tracing tutor that provides its unique sense of student interaction, the cognitive model, requires arguably the most expertise to produce. As seen above, creating these cognitive models historically required a high level of competence in both cognitive science and computer programming. This limited their creation to mostly Ph.D. cognitive scientists who have specialized in such endeavors. Recent research by various investigators has attempted to create authoring systems that make creating ITSs, particularly the aspect that makes them intelligent, easier (see Murray, Blessing, & Ainsworth, 2003, for a review). In our own efforts, we have attempted to streamline the process for creating a cognitive model, and to eliminate, or at least reduce greatly, the programming aspect. An author who wants to create a cognitive model should not be required to be a programmer. Indeed, our ultimate aim is to allow a motivated master teacher to be able to create the intelligence behind an ITS, or at the very least modify in a meaningful way an already produced cognitive model.

In producing an ITS, one needs not only to create the cognitive model, but also the student interface. Creating this is akin to creating any other piece of software, which adds to the expertise and cost of the overall system. If the ITS team could link a cognitive model to an off-the-shelf piece of software with little or no modification, then this would result in a huge cost and time savings. We have created a system in which an already produced GUI can be integrated with a cognitive model (Ritter & Blessing, 1998). In doing this we concentrated on using already developed protocols for communication with the GUI, in addition to developing some pieces ourselves.

What follows are two main sections that explain more deeply our two-part solution to lowering the bar in creating model-tracing ITSs. The first part discusses our Cognitive Model SDK, followed by a description of our system for leveraging existing interfaces. Within both sections are discussions of our observations and empirical findings from actually using the system in our work.

THE COGNITIVE MODEL SDK

In order to lower the bar in creating the cognitive model component of an ITS, we have concentrated on creating and using representations that we believe to be easier to understand than the working and procedural memory representations used by past cognitive model development environments. In doing so, we hope to enable authors who are not cognitive scientists or programmers to create cognitive models. This goal is similar to that adopted by the Cognitive Tutor Authoring Tools (CTAT) project (Alevan, Sewall, McLaren, & Koedinger, 2006). However, we are approaching the task from a different direction. Their efforts emphasize programming-by-demonstration techniques, whereas our system uses less direct or manipulable representations. There is an obvious trade-off between ease-of-use and the expressiveness of the system, but by pursuing both ends we can find the common ground of these two approaches. Our interests are driven by the need to support already existing tutors in use by hundreds of thousands of students, which places additional needs on the maintainability and robustness on our system.

To more specifically relate the Cognitive Model SDK to CTAT, in the context of Table 1, CTAT would be a sibling of the SDK. That is, they share a common ancestor with the TDK. Authors can create two types of cognitive models with CTAT: 1) Example-based Tutors, and 2) Cognitive Tutors. Example-based Tutors require no programming. The author demonstrates the solution to a particular problem, providing specific help and JIT messages during the process. The end result is a tutor for that specific problem, though there are some provisions made for generalization. A Cognitive Tutor can be

created by first starting with an Example-based Tutor and then examining and generalizing the underlying production rules. This generalization step is similar to the type of activity that an author would perform in the TDK. One can clearly see the two end-points of ease-of-use and expressiveness between these two types of cognitive models. The Example-based Tutors require no programming, but create cognitive models specific to single problems; the Cognitive Tutors require much programming experience, but create very general cognitive models. The goal of the Cognitive Model SDK is to be between these two endpoints. We desire a general purpose tool, but one that reduces or eliminates much of the programming. Lastly, CTAT includes interface authoring, either in Java or Flash. The SDK does not. As discussed in a later section, the SDK interacts with third-party interfaces, as we did not want to constrain our interfaces by our ability to build a robust interface-building tool.

Within the cognitive models for our model-tracing ITSs there are two main components: the type and rule hierarchies. The type hierarchy contains the parts of the domain relevant to tutoring, and this type hierarchy serves as the basis for the rule hierarchy to provide the tutoring (e.g., hints and just-in-time messages) to the student. These two constructs are at the heart of our tutoring architecture. For editing both constructs, we wanted representations that are understandable by non-programmers yet still provide enough power to create useful ITSs. We sought interfaces that have been successful in other software applications, such as tree views, hierarchies, and point-and-click interfaces. Other ITS SDKs have started using these and others to some degree (the VIVIDS system, Munro, 2003, was a particular inspiration to us).

Type Hierarchy Inspector

An important aspect of a model-tracing ITS cognitive model is the declarative structure that is used to refer to the important aspects of the domain and interface during tutoring. As opposed to our previous development environments for a domain's object model (a blank source document in Macintosh Common Lisp), this part of the SDK requires no programming knowledge, thereby lowering the bar considerably and results in a much more understandable, and maintainable, system.

This tool has requirements similar to those of other tools such as Protégé, an open-sourced ontology editor developed at Stanford University. The basic functionality is to display and edit objects consisting of one or more property/value pairs. However, there are additional requirements for Cognitive Tutors that make using any off-the-shelf or previously produced authoring software problematic. In particular, pre-defined object types exist that have their own properties and behaviors. For example, there is a *goalnode* object type, representing a student's subgoal in the problem, that has a set of predefined properties. Attached to these goalnode types is a predicate tree inspector, the subject of the other main tool, representing the rule hierarchy. Also, the values of all properties are typed, and this has implications for other aspects of the system (e.g., the use of Tutorscript, a topic to be discussed later).

Figure 1 shows a screenshot of the SDK. The Type Hierarchy Inspector can be seen in the upper left. The left pane of that window shows the current type hierarchy and the right pane shows information about the currently selected item in the left. There are provisions for adding, moving, and deleting types and properties, as well as maintaining other aspects of the tree. The type hierarchy in the figure is part of a model for fraction arithmetic (this model was developed by one of our participants in the evaluation study to be discussed later). The items next to the disclosure arrows are types, and the items next to the circles are properties. "denominatorAnswer" is a type inheriting directly from goalnode, and the author created twelve properties along with this type. The symbols

next to denominatorAnswer indicate there are five hints and one JIT attached to this child of goalnode. Generally speaking, objects within the interface will be matched to a goalnode type. The properties of the goalnode will be set appropriately (e.g., for this problem, is one answer in the denominator a multiple of the other? If so, what is the multiplier?) by the problem instance that instantiates the types.

The full type hierarchy for Carnegie Learning's existing algebra and middle school math tutors can be viewed and edited using this tool (consisting of approximately 85 types with 365 properties). With our past tools, these hierarchies were viewable only in a standard code window, and the definition of the various objects and properties were often scattered across pages of code and contained in several files. The superior visualization offered by the Type Hierarchy Inspector has encouraged more code-sharing between different tutors, and has helped to identify and correct inefficiencies in the representation of objects. Using this tool to enter, edit, and maintain type hierarchies for Cognitive Tutors is a clear win for the design of the type hierarchy of an ITS cognitive model. It has enabled us to find inefficiencies within existing code and to allow those who are not cognitive scientists or programmers to create and assist with the cognitive models.

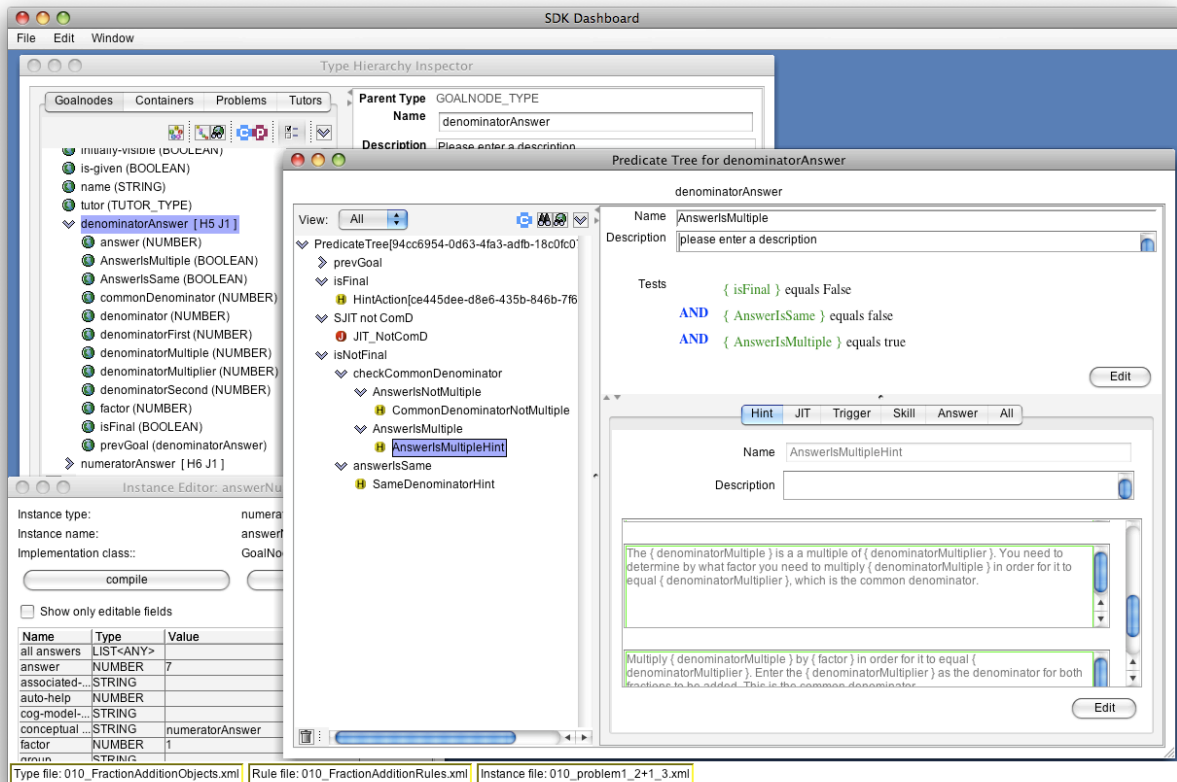


Fig. 1. Cognitive Model SDK screenshot.

Predicate Tree Inspector

The piece of a cognitive model that provides the hints, just-in-time messages, and other important tutoring behavior is the set of rules that work with the type hierarchy. Again, the challenge is to provide something that is powerful enough to meet our requirements for expressiveness, but is still usable by non-Ph.D. cognitive scientists. The system we devised, in many respects, is similar to Feigenbaum and Simon's EPAM system (1984). Our Predicate Tree Inspector for the rule hierarchies uses a hierarchical tree view, where each branch contains a predicate relating to the current goalnode of interest, and most of the actions (the hints, for example) are contained at the leaf nodes. Figuring out which hint to give, then, amounts to sorting down through the predicate tree to see what matches the current set of conditions for the current object.

What is required is a system that lays bare what is needed to produce statements such as "give this help message when the student is in an expression cell of the form $mx+b$ and the problem involves money" and "provide this just-in-time help message if the student enters an mx expression but omits the intercept b ." In addition to providing an intuitive way to enter such expressions when required, the interface needs to guide the author in creating the syntax for each part of the predicate and provide templates for entering help, just-in-time-messages, and the other actions that can be performed by the tutor.

The front-most window in Figure 1 shows the tool. The left pane of the window shows the predicate tree hierarchy for a particular goalnode. These are the rules that will be used when the student is working on an instance of that goalnode type. The upper right pane shows the full set of predicate tests for the selected node at the left, and the lower right pane shows the hints, just-in-time messages, and other tutoring actions attached at this node. Not shown is the rule editor that enables the creation and editing of nodes on the predicate tree. Seen in the figure is one of the rules and hints attached to the denominatorAnswer goalnode. This rule will apply, according to the test, when the student is not working on the final answer, and the two denominators are not the same, but one is the multiple of the other (e.g., the problem is $1/2 + 1/4$). Our student author created those conditions and the hint that is to be presented when those conditions are true for the problem instance.

The language that is used in these predicates and in other parts of the SDK to refer to a working memory state is called Tutorscript. We modeled this scripting language on those found in other object-oriented environments (e.g., Applescript in the Macintosh operating system). It provides an English-like way for non-programmer authors to refer to the needed aspects of the problem state within the rule system. Tutorscript provides a way for the author to refer to the type hierarchy within the constraints of the rules used by the cognitive model. Therefore, it is used not only in the predicates within the rules, but also in the hint and just-in-time message templates, as well as other aspects of the authoring system. In Figure 1, the items enclosed in curly braces are Tutorscript expressions. In addition to providing a way for authors to refer to properties within the type hierarchy, the syntax of Tutorscript also provides simple if/then clauses, arithmetic, and formatted output.

One final piece of the SDK to discuss is the instance editor, which can be seen in the bottom left corner of Figure 1. This instance editor provides a rudimentary way to enter problems into the SDK so that the author can test the system. The functionality is basic: a new instance of an object can be created, and then its values can be edited. Once all the instances needed for a problem have been created, it can be tested in a simple testing environment called the Goalnode Testing Tutor (GNTT). Our vision within the larger Cognitive Tutor SDK is that most cognitive models should have their own interface for entering problems, so that the process is easier, more streamlined, and usable by a wide

variety of people (e.g., teachers and perhaps students). An easy-to-use problem entry tool has been created for the algebra tutor (Ritter, Anderson, Cytrynowicz, & Medvedeva, 1998).

Usability by Authors Who Are Not Cognitive Scientists Or Programmers

We have had some experience now using the SDK and its representations in various settings. Indeed, it is how cognitive models are created and maintained at Carnegie Learning. A major risk associated with this project was not that the resulting tools would not author meaningful cognitive tutors, but rather that the tools would be too complex for authors who are not cognitive scientists or programmers to understand. We conducted a pilot experiment to check whether the proposed representations for the type and rule viewers could be used and understood by people unfamiliar with cognitive models (Blessing, Gilbert, & Ritter, 2006). The results from undergraduates who were not cognitive scientists or programmers indicated that these representations were understandable and usable.

More anecdotally, people who are not cognitive scientists or programmers have been taught to use the system and have created usable tutors. In late 2005 and early 2006, two programmers at Carnegie Learning re-implemented parts of the geometry curriculum. They recreated around 25 hours of curriculum instruction, and it took them about 600 hours working with an early version of the SDK. While 24 development hours per instruction hour is quite good, and we are pleased with it, there are obvious provisos. While not cognitive scientists, they were programmers that had been working with and observing cognitive scientists. Also, they were recreating the cognitive model, not making a new one. Finally, the 600 hours figure was just the time for creating the cognitive model and a few problems—the interface and some number of the problems had already been done. However, because it was an early version of the SDK, they were adding new features and fixing bugs to the system as they worked on the model itself. While this all makes the effort hard to interpret, we do take it as evidence that people who are not cognitive scientists can create a commercial-quality cognitive model via the SDK.

As another example, ClearSighted, Inc. uses student interns as instructional designers and has developed a brief training curriculum to orient them to the SDK and the process of cognitive modeling. During Fall 2006, this curriculum was used successfully to enable two non-programmer graduate students from instructional design and one computer science undergraduate to complete a cognitive model for multi-column addition. They have since assisted in other cognitive modeling tasks.

A recent study provides more substantial results, which we will discuss more fully here. Past studies have tested the effectiveness of other authoring tools (e.g., Ainsworth & Fleming, 2005; Halfff et al., 2003; Martin, Mitrovic, & Suraweera, 2007). This, however, was the first controlled evaluation of the Cognitive Model SDK. Given that, it will serve mostly as a baseline for future evaluations. Our main interest is to determine if authors who are not cognitive scientists or programmers can create usable cognitive models with the tool. Our findings will be placed more fully in the context of previous research efforts in the discussion.

Participants

Seventeen graduate students from Iowa State University participated. Six of these participants were students in the first-year Human-Computer Interaction (HCI) course at Iowa State University who chose to do this assignment for course credit. The other eleven participants were recruited from among

the HCI and instructional design graduate students at Iowa State University. These students were paid \$150 for their participation. None of these participants had cognitive psychology or cognitive science as their home department, nor had any done cognitive modeling before. Some had programming experience, and this will be highlighted in the results.

Materials

An assignment similar to the one used by Suraweera, Mitrovic, and Martin (2007) in their evaluation of a constraint-based tutor authoring system was used. In this assignment, participants were asked to create a cognitive model of a fraction arithmetic task. For our version of the assignment, participants were told that their model had to provide distinct and specific hints for three types of problems: 1) problems in which the two fractions started with the same denominators (e.g., $1/5 + 2/5$); 2) problems where one denominator is a multiple of the other (e.g., $1/5 + 1/10$); and, 3) problems in which the least common denominator is neither of the two given fractions (e.g., $1/5 + 1/7$).

Participants were shown a picture of an interface (see Figure 2) in which students would be given two fractions to add. The students would need to write both fractions in terms of a common denominator and would then need to compute the final, but unreduced, answer. Note that the participants did not actually have access to this interface, but had to use the more rudimentary interface that is constructed automatically by the SDK within the GNTT. In the GNTT goalnodes are simply listed vertically, with one text entry box per goalnode for the author to test tutoring behavior.

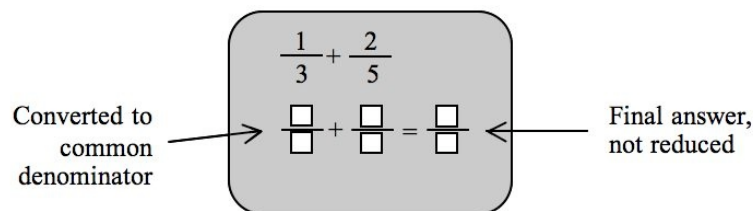


Fig. 2. Example interface shown to participants for the fraction addition task.

Four pieces of background information were provided to the participants. First, they received a demonstration of the Carnegie Learning Algebra I Cognitive Tutor (either done live in class or via a 8.30 min screen capture movie). Second, participants were given a four-page document that introduced the vocabulary and basic concepts of cognitive modeling. It explained the difference between hints and JIT messages, and introduced the concepts of hierarchical types, predicates, instances, and goalnodes. The other two pieces of information amounted to worked examples of cognitive models constructed using the SDK. The first worked example was a completed cognitive model of multi-column addition. The model could add two multi-digit numbers together, including problems involving a carry. A screen capture movie with voiceover (25.97 min) was created that walked the viewer through creating all parts of the model using the SDK: the type hierarchy, the rules, and an instance. An additional problem instance was provided to participants, as well as the movie's transcript. While both this model and the model that the participants were asked to create involved addition, the participants could not simply take this model, make a few simple modifications, and have a fraction addition model. The last piece of information was another screen movie with voiceover (10.70 min) and its transcript that stepped participants through the creation of a very simple tutor. The

tutor in this movie asked students to indicate if the presented number were even or odd. This gave the participants an example of how to construct the bare minimum framework for a tutor from scratch.

The version of the SDK used by the participants was the full version of the SDK used by Carnegie Learning. In addition, it also logged how long they spent performing each action within the SDK interface. In this way we were able to determine not only how much total time it took to complete the model, but also how long each participant spent doing the individual components of authoring a cognitive model, such as creating properties on goalnodes or writing hint messages. Participants were also given an exit questionnaire concerning their experiences, and also asked for certain demographic information such as previous programming experience.

Procedure

Participants were first given the demo of the Algebra I Cognitive Tutor. They were then given the assignment and the two worked examples as described above and asked to complete a cognitive model of fraction addition using the SDK. They could choose to do with these examples as they willed. In all, they were given about 45 min worth of instruction, and as much time as they desired to complete the assignment, though we suggested they plan between 8-12 hours for the assignment.

Results

The results are divided into three parts: 1) quantitative and qualitative measurements of the cognitive models; 2) timing data concerning cognitive model creation; and 3) exit questionnaire data.

Examining first the quantitative and qualitative aspects of the cognitive models, of the 17 participants, 13 of them completed a runnable cognitive model. One person's rule file became corrupted and so could not be scored, and three people (all paid participants) did not complete the assignment. The first author scored the 13 completed cognitive models on a 5-point scale. The criteria used and the number of models within each criterion is in Table 2. The quality of the cognitive model was rated on its behavioral characteristics, not on the particular approach taken to achieve this behavior. A similar metric was used by Martin and his colleagues (2007), though the scoring was based on the model itself. The mean score is 3.31, indicating that the average model at least met expectations. The mean time to complete the assignment was 7.68 hr.

Table 2
How the Cognitive Models Were Scored

| Score | Description | Models Meeting Criterion |
|-------|---|--------------------------|
| 5 | A model that produces behaviors close to an ideal model for fraction arithmetic, in terms of hints and just-in-time messages | 4 |
| 4 | A very good model that is beyond just being sufficient | 2 |
| 3 | A sufficient model, one that provides distinct and specific hints | 3 |
| 2 | An adequate model, but lacking in one or two ways (e.g., hints for only 2 of the 3 problem types) | 2 |
| 1 | Lacking in multiple ways; while it produces hints, it does not meet the specifications of the assignment (e.g., static hints) | 2 |

A split between the better and poorer cognitive models was performed. There were 6 cognitive models that scored either a '4' or a '5.' Seven scored a '3' or below. References to the "better" and "poorer" cognitive models will be made in the following statistics. Three of the better cognitive models came from the class participants, and three from the paid participants.

We will provide a few examples of what constituted a model that was scored a '5' versus a lower scoring model. As mentioned previously, the model pictured in Figure 1 is from a student who scored a '5' on the model's behavior. The hints were above par across the board, and the model had two JITs, both useful. Upon inspection, the properties he created for his goalnodes gave the ability to properly delineate between the three types of problems required by the assignment. For example, one Boolean property indicated if one denominator was a multiple of the other. (This would need to be set properly within the problem instance; another approach would be to do this via arithmetic in the test expression, but we encouraged our authors to use properties in this manner so that their test expressions need not be so complicated.) This property was then used in an appropriate manner in the rule to provide a sufficient hint that would be variablized correctly across different problems.

Participant03 scored a '3' on the model's behavior. The model provided good hints for the denominators in the intermediate fraction sum, but the hints for the numerators and answer fraction were static. Upon inspection, the model had a JIT, but it was not useful nor one a student would likely find. The model had sensible properties allowing it to provide hints for the three problem types in working the intermediate denominators. However, those properties were not used for the other hints. As a last example, Participant17 scored a '2' on the model's behavior. Generally speaking, the hints were adequate for two problem types, but not the other. The hint for the answer's denominator was always wrong. There were no JITs. In examining the model, we found property use that allowed it to do only part of the job. For example, one property was meant to encode the multiplier between different denominators, but not one for different comparisons between the denominators. Moreover, their use in the test expressions and the hints were not conducive to providing adequate hints. In both of these lower scoring models, additional work and further clarification of the model would have been needed in order for the behavior of the model to match what was called for by the assignment.

On almost all quantitative measures concerning the model there were no differences between the better and poorer cognitive models (see Table 3). The number of goalnodes they defined, the number of properties on those goalnodes, and the depth and size of the predicate tree were all essentially the same (in all cases, $t < 1$). The better and poorer cognitive models did differ somewhat on the number of just-in-time message defined (4.50 v. 0.86, $t(11) = 1.73$, $p = .1$), but it was the case that to be considered a better cognitive model it had to have at least one just-in-time message.

Table 3
Quantitative Aspects of the Cognitive Model

| | Goalnodes | Properties | Nodes | Depth | JITs |
|---------|-----------|------------|-------|-------|------|
| Better | 3.00 | 6.83 | 13.17 | 2.17 | 4.50 |
| Poorer | 2.50 | 6.22 | 13.12 | 2.00 | 0.86 |
| Average | 2.93 | 6.27 | 13.14 | 2.08 | |

Second, the timing data provides a snapshot into how participants approached the task. As stated above, the average time to complete a cognitive model was 7.68 hr. The participants who produced the better cognitive models spent on average almost the identical amount of time (7.67 hr, with a range of

4.98 hr to 13.08 hr) to the participants who produced the poorer models (7.68 hr, with a range of 3.42 hr to 13.82 hr). The logging produced by the SDK provided much more detail than this. Each action that the participant performed within the tool's interface was time stamped with millisecond precision. Table 4 shows how much time the participants spent performing the component actions of creating a cognitive model, time spent on 1) the objects (creating objects and defining properties); 2) the rules (predicates, hints, and just-in-time messages); 3) defining instances; and, 4) testing the model. One sees no differences on these measures between the better and poorer participants.

Table 4
Average Time Spent on Various Aspects of Cognitive Model Construction ($n = 13$)

| Category | Time (hr) | Percentage |
|-----------|-----------|------------|
| Objects | 2.45 | 31.8% |
| Rules | 3.07 | 39.9% |
| Instances | 1.65 | 21.4% |
| Testing | 0.52 | 6.8% |
| Total | 7.68 | 100% |

A further analysis was performed that examined what actions the participants were performing during the time course of creating the cognitive model. Did most participants create their type hierarchy at the very beginning, and then turn to rule writing? Or, was there more give-and-take between working on the type hierarchy and the rules? We created graphs for each participant that divided their progress in writing the model into deciles. Within each decile we calculated what percentage of the time was spent on object actions, rule actions, instances, and testing. Figure 3 shows two of these graphs, the top one illustrating a participant who produced a better cognitive model, and the bottom one showing a poorer cognitive modeler. In examining the quantitative and qualitative aspects of these graphs for each participant, one difference stands out and can be seen in Figure 3. The proportion of time spent on the type hierarchy during the first half of the participant's time on task versus the second half differs between participants. Some participants spent relatively less time on this task during the last half of their time than during the first half. That is, these participants appeared to have built a very usable type hierarchy up front, and then did not modify it much after that. To quantify this observation, we counted as an "up-fronter" any participant who did not spend 30% of their time for more than one decile working on the type hierarchy during the last half of their editing. Under this definition, all of the better cognitive modelers were "up-fronters," whereas only 3 of the 7 poorer cognitive modelers were "up-fronters." This is a marginally significant difference by a chi-square test, $\chi^2(1, n = 13) = 2.86, p < .1$. Getting the model "right" (and there are many potential "right" models) early appears somewhat predictive of creating a successful cognitive model. We investigated the possibility that some attribute associated with the cognitive modeler correlated with the ability to produce a "right" model early by examining the exit questionnaire data.

We examined the demographic information such as undergraduate major, current department, and number of programming courses. Regarding undergraduate majors and current discipline in graduate school, there was no clear trend between who created the better and poorer cognitive models (there were no psychologists or cognitive scientists in either group). There were roughly equal numbers of computer scientists and engineers who created better models as who created poorer models. And, one could find the "non-technology" majors represented in both groups. Participants had taken an average of 4.36 programming classes prior to working on the cognitive model. Examining this measure with

regards to the better versus poorer cognitive modelers does yield a significant difference, with the better cognitive modelers having taken more programming classes, an average of 6.83 v. 1.57 classes, $t(11) = 3.37, p < .01$.

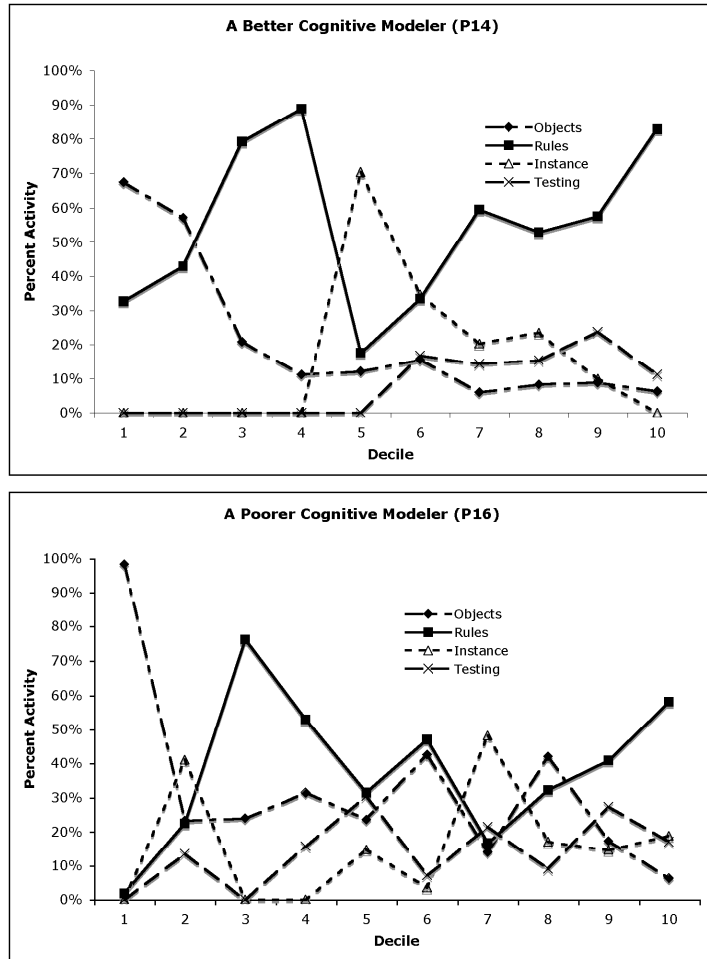


Fig. 3. Activity graphs of two participants.

Participants were also asked free response questions, where they reflected on their experiences doing the activity. In particular, they were asked to consider the challenges they encountered in creating the cognitive model, as well as the benefit of the approach. Almost all participants saw both positive and negative features in these kinds of cognitive models in general, and using this tool specifically. Ten of the 13 participants mentioned either insufficient documentation or bugs in their response, with 5 specifically mentioning Tutorscript (a planned editor to help with Tutorscript entry had not been completed). However, despite some issues with this particular tool, 11 of the 13 participants mentioned the generality of the tool; that is, it could be used to create tutors in a variety of different domains. Most (8) qualified their answer to include only domains with specific, finite answers, such as chemistry, physics, and math (i.e., more “well-defined” problems). Most model-

tracing ITS systems do fall into such domains. However, there have been tutors for domains like computer programming (more “ill-defined”). If the rules of a task can be described, and the strong claim of the ACT theory is that such rules underlie all cognitive skill (Anderson, 1993), then those rules can be created in the SDK. For example, a Java programming language tutor akin to the original programming tutors (Anderson, Conrad, & Corbett, 1989) is definitely within the scope of the SDK. Lastly, a number of participants mentioned that in order to create a cognitive model, one needed to be very explicit about the steps the students should and can take, and that the steps needed to be complete in order to have a useful cognitive model. Interestingly, it was 4 of the 6 better cognitive modelers who made such an observation, none of the poorer cognitive modelers made such a statement. This is particularly interesting in light of the work by Kirschner, Sweller and Clark (2006), in which they argue that having such explicit instructional guidance provides for more efficient learning.

Discussion

We would like to highlight three main results from this study. First, the fact that the majority of participants (13 of 17, 77%) created a usable cognitive model with minimal instruction (less than 1 hr) is remarkable. Of the four who did not create a model, one experienced computer issues resulting in file loss and two had partial models. Historically, creating a cognitive model of this type and generality was relegated to a Masters or Ph.D. level cognitive scientist with much training in ITSs and in the particular tool used to create the ITS. None of our participants were psychologists or cognitive scientists, and none were in their graduate training to produce intelligent tutoring systems. This finding can be compared to research conducted with another ITS authoring tool, WETAS. WETAS creates constraint-based models. In one study (Martin & Mitrovic, 2003), 6 of 11 authors (55%) created a usable model for a multi-week course assignment, and in another study (Martin et al., 2007) 6 of 12 authors (50%) were able to create a simple ontology-based tutor given minimal instruction and only an hour to complete a workshop assignment. The scope of our assignment is more similar to the 2003 study, though the participants in that study were students in a course devoted to ITSs. The 2007 result, though confined to a simpler tutor built during a workshop, is particularly impressive given the more mixed background of the participants (that is, not all were computer scientists). Furthermore, as a second part of that study, some of the participants worked as a group on the second day to complete a larger tutor, which they completed adequately.

In conjunction with this first result, our participants created their models relatively quickly, in under 8 hr on average. Accounting for watching the demos and other instructional activities, these participants went from ITS neophytes to having a cognitive model in about 10 hr. A proper interface, more problems, and refinements to even the best model would still have to be made, so it is unclear how to precisely gauge this effort with regards to hours of development per hour of instruction. In this particular domain, the interface is rather simple and the problem authoring quite routine, so the ratio would be much less than 100:1.

The second notable observation is how similar the better and poorer cognitive models were across a number of measures. In terms of time to construct the model, including an examination of working on certain subcomponents, along with certain quantitative aspects of the model (number of objects and predicates, for example), there were no differences between the better and poorer models. This would make it difficult to look for certain markers (such as average depth of the predicate tree) or time measures and quickly determine if the model appeared to be a quality model, if this observation holds true in future studies.

Perhaps the main difference we found between the better and poorer cognitive models is the third observation. The creators of the better cognitive models had taken more programming courses prior to the experiment. Given the correlational nature of this observation, it is hard to know the causal influences; did having more programming courses help them create better cognitive models, or did the fact that they might be better cognitive modelers to begin with attract them to programming courses? Martin and his colleagues (2007) made a similar observation. In their first study, the participant who completed the best ontology model was a computer scientist who implemented recursion into the model. It seems reasonable to assume that programmers, either by their nature or training, are better equipped to think about tasks in ways that are conducive to putting them into a cognitive model. Specifically, we can think of at least two such reasons. First, much programming now is object-oriented in nature, and this is the representation used by the SDK. If a person is used to thinking about objects, properties, and inheritance, then being able to represent a task in the SDK should be easier. Second, although the rules were also represented in a hierarchy, they still had the “if-then” quality of production rules. Again, this kind of thinking is probably more natural to programmers.

USE OF EXISTING INTERFACES

Almost all Cognitive Tutor interfaces have been designed alongside the cognitive model (though see Ritter & Koedinger, 1996, for description of tutors that use Microsoft Excel and Geometer’s Sketchpad). We desire to enable the construction of model-tracing ITSs around pre-existing software. This would eliminate or greatly diminish the time spent doing traditional interface programming. By allowing authors to create an ITS for off-the-shelf software, this will lower the bar for creating such systems. One can imagine tutoring not only math or statistics using Microsoft Excel as the interface (or some other spreadsheet), but one could also tutor on Microsoft Excel itself (or any other application requiring training). Such a scenario would be a boon to corporate training environments.

What is needed is a way for the Tutor Runtime Engine, the tool that uses the cognitive models created by the SDK, to communicate with third-party software (that is, software that the authors creating the ITS did not program). Even though the interfaces for the current Cognitive Tutors were developed essentially in tandem with their cognitive models, the code for the interfaces was separate from the tutoring code, with the two pieces communicating to each other through a messaging protocol (described in more depth in Ritter, Blessing, & Wheeler, 2003). This separation allows for the kind of SDK described in the previous section, and also allows for the possibility of third-party applications to be the student interface. TutorLink is our solution for allowing the TRE to communicate with outside applications (see Figure 4).

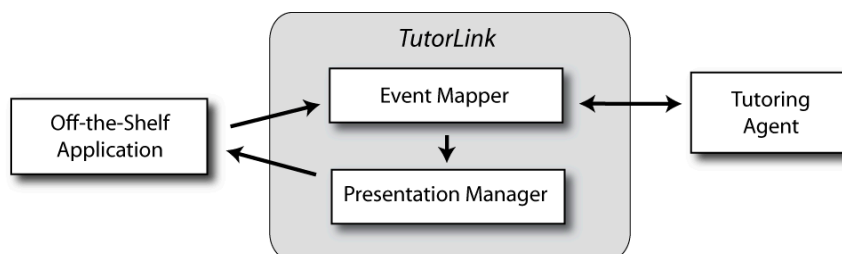


Fig. 4. TutorLink architecture.

TutorLink

In order to provide tutoring, the TRE engine needs to know what the student is doing. Either the interface needs to communicate what is happening or those actions must be inferred by some mechanism. There are three basic ways by which this might occur. First, the developer of the interface application uses tutable widgets (that is, buttons, entry boxes, menus, etc.) that readily broadcast what they are doing in a way that is easily and straightforwardly understood by the TRE. This is what the current interfaces used within the Cognitive Tutors developed by Carnegie Learning do. Also, this is the way that CTAT tutors work (Aleven et al., 2006). Such tutors must use the Java or Adobe Flash widgets the research project provides. Second, and least desirable of the three, occurs when the application is truly a black box, but based on low-level operating system events such as mouse clicks at OS-level coordinates, inferences are made as to what buttons, menus, entry boxes, and other widgets are being manipulated. You could do this with any application, but this solution is brittle, as any change to the application would probably break the tutoring interaction. The third choice is when the interface developer did not use tutable widgets, and the source code is not available to be recompiled with tutable widgets. However, there may still be a way, that does not involve low-level OS events, to know what widgets the user is manipulating and map that to something with which the TRE can work. Different operating systems and architectures have mechanisms like this to different degrees. On the Macintosh platform, *AppleEvents*, if implemented correctly by the programmer, can provide excellent semantic-level events with which to use for tutoring (see Ritter & Blessing, 1998 for a pre-TutorLink implementation of this mechanism). On Microsoft platforms, both Microsoft Foundation Classes and their .NET architecture allow for outside programs at least some insight and control as to what is happening in the interface. Java also allows for this to at least some degree. Secondly, accessibility protocols such as UI Automation used with Windows controls or Mac OS X Accessibility, frequently used by screen readers such as JAWS or by screen-movie capture software such as Adobe Captivate, can also be used to observe a user's behavior in a third-party application. It is Microsoft's .NET architecture with which we have been currently experimenting. Depending on the exact nature of the representation, this third option is more or less brittle, and so is more advantageous than the second option. With regards to .NET, a listener function can be added to the original code that eavesdrops on the event queue within the application and sends a copy of the user's input to the tutor, making the option even less brittle.

Regardless of how exactly TutorLink is monitoring the application that contains the interface, once the user interacts with the interface, that interaction needs to be noted by TutorLink and sent to the TRE for the appropriate tutoring action (the TRE is a Java process; TutorLink could be anything). The part of TutorLink that receives the interaction message from the interface is called the *EventMapper*. As its name implies, this part takes the incoming message and maps the event into something the TRE can understand (that is, a goalnode within the cognitive model). In the case where the interface was built using tutable widgets, this mapping is straightforward. In other cases, there needs to be a more structured mapping table that maps between interface widgets and goalnodes (e.g., both 'color wheel output' and 'list of red, green, blue textbox values' map to the 'answer for the choose-color goalnode). This mapping has to be supplied by the author. Such a mapping application using *AppleEvents* has been described (Ritter & Blessing, 1998). In general terms, the author starts a listening application, then begins to interact with the widgets in the interface in order to identify their names, and then maps those to goalnodes. We developed such an application for the .NET architecture, and by extension, one could be developed for other architectures as well.

Once the event has been mapped for the tutor by the EventMapper, the message is transferred to the TRE. The tutor will decide on the appropriateness of the user action, and offer feedback (be it a hint, a just-in-time message, a change to the skill window, or something else) that will be communicated back to the interface through TutorLink via the EventMapper and a part of TutorLink referred to as the PresentationManager. The PresentationManager determines how to present the tutor's feedback to the user. Again, depending on which architecture is being used, the feedback might be able to be presented within the interface, such that the user would not realize that a different program is really presenting the information (possible when using tutable widgets, AppleEvents, and .NET), or the feedback might need to be presented within its own process.

Current Status

As mentioned above, we currently have a system using this architecture within Microsoft's .NET framework. Figure 5 shows a screenshot of our system in action (the tutor has indicated that the chosen method of resizing an image is inappropriate, and is providing a just-in-time message). The tutoring backend is the same backend that Carnegie Learning uses for its algebra tutors (that is, the TRE). The frontend is an application called Paint.NET, which is image manipulation software akin to Adobe Photoshop. We have developed a curriculum around Paint.NET that teaches several lessons with a model-tracing ITS. Paint.NET is obviously a piece of software that was not designed with a model-tracing tutor in mind, so being able to provide such a rich tutoring experience within it has been the culmination of much previous work. In a proof-of-concept study (Hategekimana, Gilbert, & Blessing, 2008), 75 undergraduate participants learned Paint.NET either through the ITS lessons, screen-movie-based training (common in corporate training), or using printed material (similar to the *How to Learn XYZ in 30 Days* software manuals). Given the short nature of the training time (only 60 min), no real learning differences were detected, though the screen-capture movie group performed worse on test tasks at the end of the experiment. However, we take this as validation that a meaningful ITS can be created using the Cognitive Model SDK that uses a third-party application as its interface.

In addition to the Paint.NET tutor, we have also developed a handful of other tools that assist in creating tutors for existing third-party applications. First, UITest is an application that can identify all the UI components of many third-party applications and determine how much information we can observe about a user's behavior. UITest can also sometimes control these UI components without having access to source code. Second, the Tutorial Recorder significantly eases the mapping of an application's GUI control messages like "mainMenu._1._6:click" to tutor-friendly messages like "Edit Menu:Invert Selection." This makes mapping interface widgets to tutor goalnodes much more manageable. Finally, we have created a Firefox plug-in called WebTutor that enables the creation of tutors on websites by leveraging the DOM model of the pages. We are currently examining the range of complexity on which we can tutor. Flash and AJAX sites may be too difficult to observe robustly, though we have had success tutoring on a complex dynamic site built on python and web forms (Roselli, Gilbert, Raut, Pandian, & Blessing, 2008).

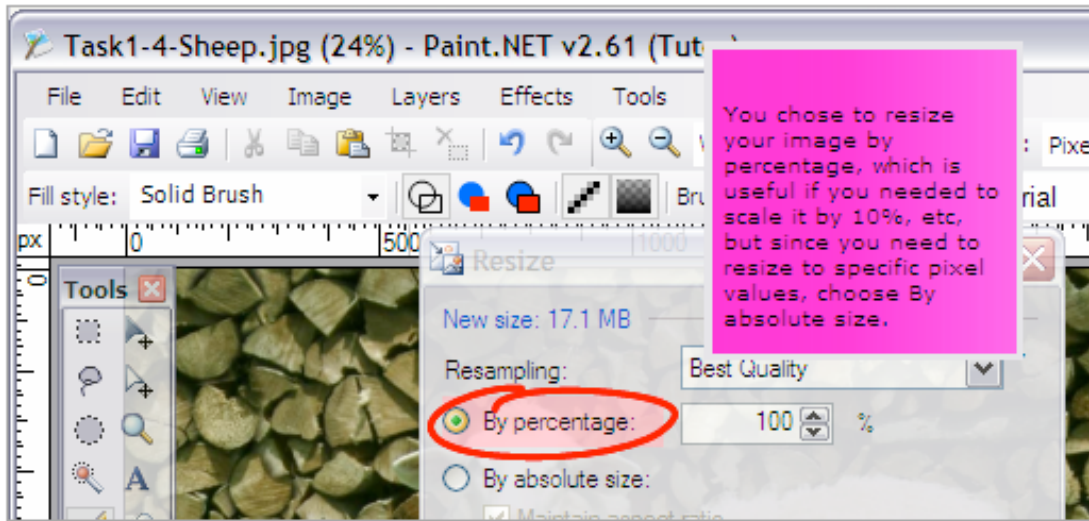


Fig. 5. Tutored Paint.NET screenshot.

Challenges

Technical Challenges

In using TutorLink with Paint.NET, we encountered two problems that would exist when tutoring on many existing third-party applications. First, event handling within applications is not always coincident with the user action, making it difficult to tutor. For example, in Paint.NET, if a user chooses the Resize menu item, the Event Listener learns of that menu event only after the entire Resize dialog is closed, making it too late to tutor the user on the menu choice or even on the choices made within the Resize dialog. The .NET architecture allowed us to work around this limitation. This issue of event handling illustrates a design pattern that could be encouraged as a developers' guideline for building more easily tutorable applications even when the applications do not use tutorable widgets.

The second problem that arose was the tracking of state variables in the application. Storing which windows are open, which tool is selected, and which preferences are set cannot always be inferred by watching the user's actions because there can be initial values and there may be complicated, opaque logic determining the values. State management in tutors adds complexity, as "state" is a programming concept and the goal of our authoring system is to avoid programming. To address these problems in the Paint.NET tutor, we decided to allow for more intrusive application instrumentation whenever we needed the value of state variables in the tutor. In these cases, state variables were treated like invisible controls, with events triggered by any change to the state. Again, this level of programming is not ideal.

Instructional Challenges

Few custom-developed Cognitive Tutor interfaces offer the panoply of buttons, toolbars, menu options, and tool panels found in off-the-shelf productivity applications such as Microsoft Word or

Adobe Photoshop. This more complex GUI environment poses two particular instructional design challenges. First, when there are multiple correct methods of reaching a goal within the interface, which method should the tutor recommend? A model-tracing tutor can accommodate whichever method users may choose, but if a user requests a hint at the junction point of three correct paths to a subgoal, the tutor designer needs to balance helping the user reach the goal versus educating the user.

Second, how much should the tutor allow the user to explore the interface and how much guidance should be given? Cognitive Tutors typically give feedback for each user action, and they will sometimes disable parts of the interface to narrow students' attention to the most appropriate action to take. Doing both of these on a live interface might hinder the type of learning we want to foster. The Paint.NET tutor allowed users to choose an infinite number of different painting tools, e.g. Paintbrush, Paint Bucket, Magic Wand, without intervening, as those selections did not affect the image, and providing feedback on those selections would have been burdensome on the user. However, the tutor did object if users chose menu options that were completely irrelevant to the current subgoal. Also, the Paint.NET tutor blocked and forced a user to undo incorrect file events (GUI choices that would irreversibly affect the currently open document, such as closing it) but allowed users to perform incorrect application events (GUI choices affecting the application only, such as opening the Layers panel or choosing a color).

CONCLUSIONS AND FUTURE WORK

We have described a system that lowers the bar for creating model-tracing ITSs. The system achieves this goal by accomplishing two main objectives: 1) providing a feasible way in which authors who are not cognitive scientists or programmers can create cognitive models; and 2) providing a mechanism by which third-party applications can be instrumented and used as the frontend of a tutoring system. Meeting both objectives lowers the bar necessary to create this effective class of tutor. Current work in pursuing these objectives involves creating learning materials for the SDK, for the first objective, and investigating how to increase the classes of third-party interfaces that will work with TutorLink, for the second objective.

In closing, we would like to highlight two current concerns we are investigating. First, much current effort is focused on ensuring that the workflow presented by the SDK supports efforts to improve and validate the system over time. One of the major challenges in commercializing an intelligent tutoring system is managing quality assurance. A hallmark of a good intelligent tutoring system is that it supports multiple problem-solving strategies, meaning that the behavior of the system can and should be complex. Given that complexity, it is a difficult task to ensure that the system behaves properly under each of the many possible problem-solving strategies. Part of our approach to this problem has been to automatically classify both problems and strategies into equivalence classes, simplifying the space that needs to be tested.

A related issue arises from our need and desire to improve the system over time. A good cognitive model contains powerful rules that apply across many different problem types. If we decide to change a set of rules to better model student performance (following Cen, Koedinger & Junker, 2006), how can we detect all the implications of that change and communicate the most efficient and complete testing plan to the quality assurance staff? Our solution, as yet incomplete, involves building quality assurance into the SDK workflow. Changes in the cognitive model automatically lead to new

test plans. The SDK architecture supports an approval infrastructure, which allows different members of the team to review, comment on and accept proposed changes.

Our second current concern was brought to light in the study examining the Cognitive Model SDK. Consistent with previous studies (Martin et al., 2007), we found that having at least some programming knowledge apparently assisted in creating a cognitive model. Given the nature of the Cognitive Model SDK, particularly in relation to CTAT and its goal of addressing more specifically non-programmers (Alevan, et al., 2006), the association between programming and producing better cognitive models using the SDK is not surprising. However, only half of the people who created the better cognitive models would consider themselves a “programmer.” Future lines of research should investigate this relationship in more depth, to examine which pieces of programming knowledge most help in creating a cognitive model. There are two possible benefits to this line of research. First, we may be able to move towards more people being able to make higher quality cognitive models. Second, and consistent with a discussion in Ainsworth and Grimshaw (2004), we may discover that different authoring tools at different levels of abstraction may be desired. That is, for people more comfortable with the programming concepts, a more general, flexible tool may be available, but, in order to accommodate those without the programming knowledge, a tool that still allows some modification of a cognitive model but not the flexibility may be more appropriate.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grants DMI-0441679 and OII-0548754. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We would like to thank Matt McHenry, Tristan Nixon, Janea Triplett, Leslie Wheeler, and three anonymous reviewers for their help on various aspects of this project. Portions of this work have been presented at the 19th International FLAIRS Conference, Melbourne, FL, the 13th International Conference on Artificial Intelligence in Education, Marina del Rey, CA, and the 9th International Conference on Intelligent Tutoring Systems, Montreal, Quebec.

REFERENCES

- Ainsworth, S.E. & Fleming, P.F. (2005). Evaluating a mixed-initiative authoring environment: is redeem for real? In C-K Looi, G. McCalla, B. Bredeweg & J. Breuker (Eds.) *Proceedings of the 12th International Conference on Artificial Intelligence in Education* (pp. 9-16). Amsterdam, IOS Press.
- Ainsworth, S.E. & Grimshaw, S.K. (2004). Evaluating the REDEEM authoring tool: Can teachers create effective learning environments? *International Journal of Artificial Intelligence in Education*, 14, 279-312.
- Alevan, V., Sewall, J., McLaren, B. M., & Koedinger, K. R. (2006). Rapid authoring of intelligent tutors for real-world and experimental use. In Kinshuk, R. Koper, P. Kommers, P. Kirschner, D. G. Sampson, & W. Didderen (Eds.) *Proceedings of the 6th IEEE International Conference on Advanced Learning Technologies*, (pp. 847-851). Los Alamitos, CA: IEEE Computer Society.
- Anderson, J. R. (1993). *Rules of the Mind*. Hillsdale, NJ: Erlbaum.
- Anderson, J. R. & Pelletier, R. (1991). A development system for model-tracing tutors. In L. Birnbaum (Ed.), *Proceedings of the International Conference of the Learning Sciences* (pp. 1-8). Charlottesville, VA: Association for the Advancement of Computing in Education.

- Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP Tutor. *Cognitive Science*, 13, 467-506.
- Anderson, J. R., Corbett, A. T., Koedinger, K., & Pelletier, R. (1995). Cognitive tutors: Lessons learned. *The Journal of Learning Sciences*, 4, 167-207.
- Blessing, S. B., Gilbert, S., & Ritter, S. (2006). Developing an authoring system for cognitive models within commercial-quality ITSS. In G. Sutcliffe & R. Goebel (Eds.) *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference* (pp. 497-502). Melbourne, FL: AAAI Press.
- Cen, H., Koedinger, K., & Junker, B. (2006). Learning factors analysis: A general method for cognitive model evaluation and improvement. In M. Ikeda, K. Ashlay & T-W Chan (Eds.) *Proceedings of the 8th International Conference on Intelligent Tutoring Systems (ITS 2006)* (pp. 164-175). Berlin: Springer Verlag.
- Corbett, A.T. (2001). Cognitive computer tutors: Solving the two-sigma problem. In M. Bauer, P. Gmytrasiewicz & J. Vassileva (Eds.) *User Modeling: Proceedings of the Eighth International Conference (UM 2001)* (pp. 137-147). Berlin: Springer Verlag.
- Feigenbaum, E., & Simon, H.A. (1984). EPAM-like models of recognition and learning. *Cognitive Science*, 8, 305-336.
- Halff, H.M., Hsieh, P.Y., Wenzel, B.M., Chudanov, T.J., Dirnberger, M.T., Gibson, E.G., & Redfield, C.L. (2003). Requiem for a development system: Reflections on knowledge-based, generative instruction. In T. Murray, S. Blessing, & S. Ainsworth (Eds.) *Authoring Tools for Advanced Technology Learning Environments* (pp. 33-59). Norwell, MA: Kluwer Academic Publishers.
- Hategekimana, C., Gilbert, S., & Blessing, S. (2008). Effectiveness of using an intelligent tutoring system to train users on off-the-shelf software. In K. McFerrin et al. (Eds.) *Proceedings of the 19th Annual Conference of the Society for Information Technology & Teacher Education* (pp. 414-419). Chesapeake, VA: AACE.
- Kirschner, P. A., Sweller, J., & Clark, R. E. (2006). Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based experiential and inquiry-based teaching. *Educational Psychologist*, 41, 75-86.
- Koedinger, K. R., Anderson, J. R., Hadley, W. H., & Mark, M. A. (1997). Intelligent tutoring goes to school in the big city. *International Journal of Artificial Intelligence in Education*, 8, 30-43.
- Martin, B., & Mitrovic, A. (2003). Domain modelling: Art or science? In U. Hoppe, F. Verdejo & J. Kay (Eds.) *Proceedings of the 11th International Conference on Artificial Intelligence in Education* (pp. 183-190). Amsterdam: IOS Press.
- Martin, B., Mitrovic, A., & Suraweera, P. (2007). Domain modelling with ontology: A case study. In A. Cristea and R.M. Carro (Eds.) *Proceedings of the 5th International Workshop on Authoring of Adaptive and Adaptable Hypermedia, User Modeling 2007* (pp. 4-11), Corfu, Greece.
- Munro, A. (2003). Authoring simulation-centered learning environments with RIDES and VIVIDS. In T. Murray, S. Blessing, & S. Ainsworth (Eds.) *Authoring Tools for Advanced Technology Learning Environments* (pp. 61-91). Norwell, MA: Kluwer Academic Publishers.
- Murray, T. (1999). Authoring Intelligent Tutoring Systems: An analysis of the state of the art. *International Journal of Artificial Intelligence in Education*, 10, 98-129.
- Murray, T., Blessing, S., & Ainsworth, S. (2003). *Authoring Tools for Advanced Technology Learning Environments*. Norwell, MA: Kluwer Academic Publishers.
- Ritter, S., & Koedinger, K. R. (1996). An architecture for plug-in tutor agents. *Journal of Artificial Intelligence in Education*, 7, 315-347.
- Ritter, S. & Blessing, S. B. (1998). Authoring tools for component-based learning environments. *Journal of the Learning Sciences*, 7(1), 107-131.
- Ritter, S., Blessing, S. B., & Wheeler, L. (2003). User modeling and problem-space representation in the tutor runtime engine. In P. Brusilovsky, A. T. Corbett, & F. de Rosis (Eds.) *User Modeling 2003: Proceedings of the 9th International Conference* (pp. 333-336). Berlin: Springer-Verlag.

- Ritter, S., Anderson, J., Cytrynowicz, M., & Medvedeva, O. (1998). Authoring Content in the PAT Algebra Tutor. *Journal of Interactive Media in Education*, 98 (9) [www-jime.open.ac.uk/98/9].
- Ritter, S., Anderson, J. R., Koedinger, K. R., Corbett, A. (2007). Cognitive Tutor: Applied research in mathematics education. *Psychonomic Bulletin & Review*, 14, 249-255.
- Ritter, S., Kulikowich, J., Lei, P., McGuire, C.L., & Morgan, P. (2007). What evidence matters? A randomized field trial of Cognitive Tutor Algebra I. In T. Hirashima, U. Hoppe & S. S. Young (Eds.), *Supporting Learning Flow through Integrative Technologies* (Vol. 162, pp. 13-20). Amsterdam: IOS Press.
- Roselli, R.J., Gilbert, S., Raut, A., Pandian, P., & Blessing, S. (2008). Integration of an intelligent tutoring system with a web-based authoring system to develop online homework assignments with formative feedback. American Society for Engineering Education 2008 Conference, Pittsburgh, PA.
- Suraweera, P., Mitrovic, A., & Martin, B. (2007). Constraint authoring system: An empirical evaluation. In R. Luckin, K. Koedinger, & J. Greer (Eds.) *Proceedings of the 13th International Conference on Artificial Intelligence in Education* (pp. 451-458). Amsterdam: IOS Press.
- VanLehn, K., Lynch, C., Schulze, K., Shapiro, J.A., Shelby, R., Taylor, L., Treacy, D., Weinstein, A., & Wintersgill, M. (2005). The Andes physics tutoring system: Lessons learned. *International Journal of Artificial Intelligence in Education*, 15, 147-204.
- Woolf, B. P., & Cunningham, P. (1987). Building a community memory for intelligent tutoring systems. In K. Forbus & H. Shrobe (Eds.) *Proceedings of the Sixth National Conference on Artificial Intelligence, AAAI 1987*(pp. 82 – 89). Menlo Park, CA: AAAI Press.