Lowering the Bar for Creating Model-Tracing Intelligent Tutoring Systems

Stephen BLESSING[a,1], Stephen GILBERT[b,c], Steven OURADA[b], Steven RITTER[d]
[a] *Department of Psychology, University of Tampa, Tampa, FL*
[b] *Clearsighted, Inc.;* [c] *ISU Center for Technology;* [d] *Carnegie Learning, Inc.*

Abstract. The main goal of the work presented here is to allow for the broader dissemination of intelligent tutoring technology. To accomplish this goal, we have two clear objectives. First, we want to allow different types of people to author model-tracing intelligent tutoring systems (ITSs) than can now do so. Second, we want to enable an author to create a tutor for software that was not initially designed with an ITS in mind. Accomplishing these two objectives should increase the number of such ITSs that are produced. We have created the first iteration of an authoring system that addresses both objectives. Non-cognitive scientists and non-programmers have used the system to create a tutor, and the system can interface with third-party software that was not originally designed with the ITS.

1. 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 [1, 4, 13]. These tutors contain a cognitive model of the domain that the tutor uses to check most, if not all, student responses. This type of intense interaction and feedback has led to impressive student learning gains. Results show that students can master the material in a third less time [3]. Field trials comprising hundreds of students have shown significant improvement on standardized tests when students practiced with model-tracing tutors [6].

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 is in use by over 1000 school districts by hundreds of thousands of students. 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. To create these pieces, many kinds of expertise are needed, most notably programming (both GUI and non-GUI), pedagogy,

[1] Corresponding Author: Department of Psychology, University of Tampa, 401 W. Kennedy Blvd. Box Q, Tampa, FL 33606 USA; e-mail: sblessing@ut.edu

and cognitive science. This generally requires a team of highly trained people to work together, thereby resulting in a high cost. Past estimates have found that it takes 100 hours or more to create 1 hour of instruction [7]. Taken together, these factors of expertise and cost obviously are a large part of why model-tracing ITSs are not in wider use.

The work presented here attempts to decrease these factors for creating model-tracing ITSs by reducing the expertise needed to create such a tutor, which will result in reducing the cost as well. Our solution for this "lowering the bar" of ITS creation consists of two parts: simplifying cognitive model creation with an easier-to-use authoring system and linking existing software interfaces to a cognitive model.

Simplifying Cognitive Model Creation. The aspect of a model-tracing tutor that provides its unique sense of student interaction, the cognitive model, has required arguably the most expertise to produce. In the past to create these cognitive models, which often take the form of a production system, one needed a high level of competence in both cognitive science and computer programming. This relegated 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 [8] for a review).

The plan we have for meeting this objective is 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.

Linking Existing Software Interfaces. To create an ITS, one needs not only to produce the cognitive model, but also the student interface as well. 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.

The plan we have here is to be able to integrate an already produced GUI with the cognitive model created in the manner as described above. To do this we will concentrate on using already developed protocols for communication with the GUI, but we will also consider other solutions as well.

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 Software Development Kit (SDK), followed by an original description of our system for leveraging existing interfaces, a feature found in few other systems. Within both sections are discussions of our observations and new empirical findings of actually using the system in our work.

2. 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 are easier to understand than the working and procedural memory representations used by past cognitive model development environments. In doing so, we hope to enable non-cognitive scientists and non-programmers to create cognitive models. This goal is similar to another project [5] in their effort to lower the bar for creating cognitive models. However, we are approaching the task from a different direction. Their efforts emphasize programming-by-demonstration and other techniques beneficial to non-programmers, whereas our system uses more complex 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 exiting tutors in use by hundreds of thousands of students, which places additional needs of maintainability and robustness on our system.

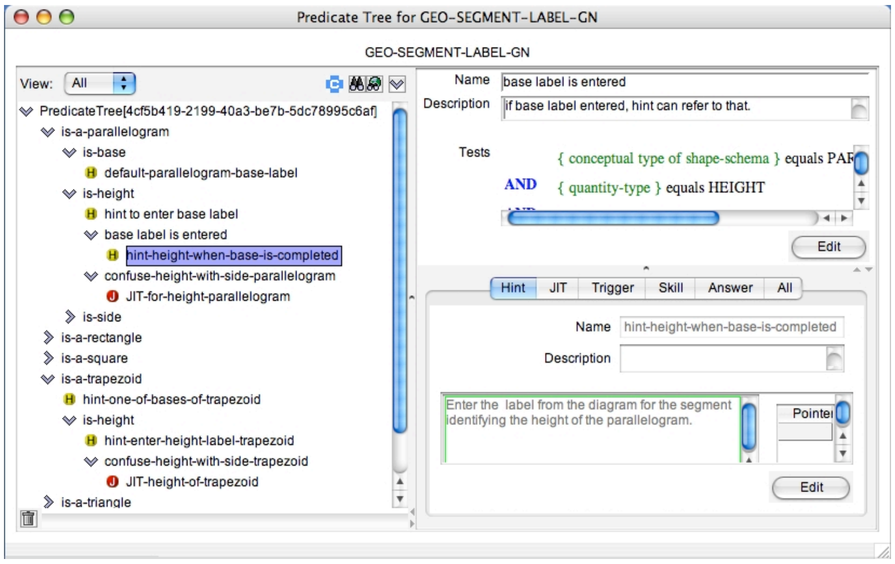
Within the cognitive models for our model-tracing ITSs, there are two main components: an object model and a rule hierarchy. The object model contains the parts of the domain relevant to tutoring, and this object model serves as the basis of the rule hierarchy to provide the tutoring (e.g., hints and just-in-time messages) to the student. These two constructs are at the heart of our tutoring architecture. The particulars of the internal architecture used by us, referred to the as the Tutor Runtime Engine (TRE), has been described elsewhere [10]. A set of tools has been created that work on top of this architecture to provide a SDK for cognitive models. The next two sections provide a summary of how the object model and the rule hierarchy are created within the cognitive model SDK (see [2] for a fuller discussion).

For the object model and rule hierarchy editors, the key is finding 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 [9] was a particular inspiration to us).

2.1 Object Model Editor

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 environment 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. As an aside, both the object model and the rule hierarchy editors are capable of editing the current cognitive models in use by the commercial versions of Carnegie Learning's Cognitive Tutor, thereby demonstrating their robustness.

The object model editor does similar things as already existing tools (e.g., Protégé, an open-sourced ontology editor developed at Stanford University). The basic functionality is to display and edit objects consisting of attribute/value pairs. Our cognitive models, however, contain specific items with particular functionality that made it more attractive to create our own editor. Specifically, pre-defined object types exist that have their own properties and behaviors, and this has ramifications for how the rest of the system operates. For example there is a ***goalnode*** object type (representing a student's subgoal in the problem) that has a set of predefined properties, and attached to these goalnode types is a predicate tree inspector (the subject of the other main tool, representing the rule hierarchy).


Predicate Tree for GEO-SEGMENT-LABEL-GN

GEO-SEGMENT-LABEL-GN

View: All

- PredicateTree[4cf5b419-2199-40a3-be7b-5dc78995c6af]
  - is-a-parallelogram
    - is-base
      - default-parallelogram-base-label
    - is-height
      - hint to enter base label
      - base label is entered
        - hint-height-when-base-is-completed
      - confuse-height-with-side-parallelogram
        - JIT-for-height-parallelogram
    - is-side
  - is-a-rectangle
  - is-a-square
  - is-a-trapezoid
    - hint-one-of-bases-of-trapezoid
    - is-height
      - hint-enter-height-label-trapezoid
      - confuse-height-with-side-trapezoid
        - JIT-height-of-trapezoid
  - is-a-triangle

Name: base label is entered

Description: if base label entered, hint can refer to that.

Tests: { conceptual type of shape-schema } equals PAR

AND { quantity-type } equals HEIGHT

Edit

Hint | JIT | Trigger | Skill | Answer | All

Name: hint-height-when-base-is-completed

Description:

Enter the label from the diagram for the segment identifying the height of the parallelogram.

Pointer

Edit

Figure 1. Rule Hierarchy Editor.

2.2 Rule Hierarchy Editor

The piece of a cognitive model that provides for the hints, just-in-time messages, and other important tutoring behavior is the set of rules that work with the object hierarchy. In our past system, these rules were realized by a production system, which required not only knowledge of cognitive science but also programming. 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. Our rule hierarchy editor (what we call a predicate tree) uses a hierarchical tree view, where each branch contains a predicate relating to the current object of interest (a goalnode, in our parlance), 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.

Figure 1 shows the current state of the tool (the Object Model Editor has a similar look, with objects and properties in a left-hand pane, and a right-hand pane that shows current values). The upper left pane of the design shows the predicate tree hierarchy for a particular goalnode. 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.

2.3 Usability by Non-Cognitive Scientists and Non-Programmers

We have had some experience now using our SDK in various settings. What follows are three indications that the cognitive model SDK and the kinds of representations it uses are indeed usable by non-cognitive scientists and non-programmers.

Usability by non-cognitive scientists. At the very least, we want the SDK to be usable by non-cognitive scientists, so that other people may both create new cognitive models and maintain existing ones. For example, the creation of a new cognitive model or the extensive modification of an existing one had to be done by a cognitive scientist. Furthermore, it would have been impossible for a curriculum designer to make just a simple wording change in one of the hints. Even that simple maintenance task had to be done by a cognitive scientist. Allowing other kinds of people to make these additions and modifications has obvious cost and time benefits.

In late 2005 and early 2006, two non-cognitive scientists at Carnegie Learning re-implemented parts of their 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 the 24 hours per hour instruction is quite good, and we are pleased with it, there are obvious provisos. While non-cognitive scientists, they were programmers that had been working and observing cognitive scientists. Also, they were recreating the cognitive model, not making a new one. Finally, that is 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 a commercial-quality cognitive model can be created via the SDK by non-cognitive scientists.

Usability by non-programmers. There are two pieces of evidence that one doesn't even need programming knowledge to use the SDK. The first was described elsewhere [2] and concerns an experiment investigating whether non-programmers could make sense of the representations used in the SDK. In this experiment, non-computer science undergraduates (nor were any cognitive science undergraduates) demonstrated a high degree of facility and competence at using the hierarchical views used by both the object model and rule hierarchy editors. This gave us confidence that the SDK was at least in the right direction regarding how it represented cognitive models.

We have since gone a step further and have actually had beginning cognitive modelers use the SDK to produce a cognitive model. 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. Although somewhat anecdotal, we again take this as encouragement that we have lowered the bar in creating cognitive models. Using the older system, this kind of result with non-programmers would simply not be possible.

3. Use of Existing Interfaces

Almost all Cognitive Tutor interfaces have been designed alongside the cognitive model (though see [12] for description of tutors that use Microsoft Excel and Geometer's Sketchpad). It is desirable to enable the construction of model-tracing ITSs around pre-existing software. This would eliminate or at least greatly diminish the time spent doing traditional interface programming. By allowing authors to create an ITS for off-the-shelf software, this will obviously 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 model 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 (this is described in more depth in [10]). 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 2).

3.1 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 tutorable 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. Second, and least desirable of the three, occurs when the application is truly a black box, but based on low-level operating system events 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 obviously very brittle, as any change to the application will probably break the tutoring interaction. The third choice is when the interface developer didn't use tutorable widgets, and the source code is not available to be recompiled with tutorable 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 [11] 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. 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.

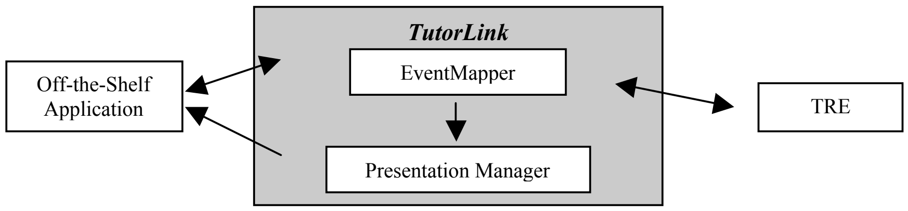
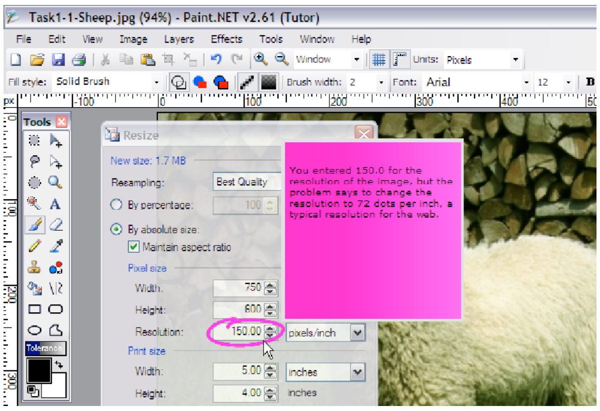


Figure 2. *TutorLink Architecture*


Task1-1-Sheep.jpg (94%) - Paint.NET v2.61 (Tutor)

Resize

New size: 1.7 MB

Resampling: Best Quality

By percentage: 100

[✓ By absolute size:]

[✓ Maintain aspect ratio]

Pixel size

Width: 750

Height: 600

Resolution: 150.00 pixels/inch

Print size

Width: 5.00 inches

Height: 4.00 inches

You entered 150.0 for the resolution of the image, but the problem says to change the resolution to 72 dots per inch, a typical resolution for the web.

Figure 3. *Paint.Net as a tutored application.*

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 tutorable 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 [11]. 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. A part of TutorLink referred to as the PresentationManager will decide 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 tutorable widgets, AppleEvents, and .NET), or the feedback might need to be presented within its own process.

3.2 Current Status

As mentioned above, we currently have a system using this architecture within Microsoft's .NET framework. Figure 3 shows a screenshot of our system in action (the tutor has indicated that the entered value of 150 for resolution is incorrect, 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 Engine). The frontend is an application called Paint.NET, image manipulation software akin to Adobe

Photoshop. We have developed a curriculum around Paint.NET that teaches several lessons with a model-tracing ITS. We are currently testing its effectiveness. 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.

4. Discussion

We have a 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 non-programmers and non-cognitive scientists 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 effort is focused on ensuring that the workflow presented by the SDK assists both novice and expert users of the system, as well as enlarging the classes of systems with which TutorLink will work.

5. Notes

This material is based upon work supported by the National Science Foundation under Grant No. 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.

6. References

[1] Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP Tutor. *Cognitive Science*, *13*, 467-506.

[2] Blessing, S. B., Gilbert, S, & Ritter, S. (2006). Developing an authoring system for cognitive models within commercial-quality ITSs. In *Proceedings of the Nineteenth International FLAIRS Conference*.

[3] Corbett, A.T. (2001). Cognitive computer tutors: Solving the two-sigma problem. In the *Proceedings of the Eighth International Conference of User Modelling.*

[4] 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.

[5] Koedinger, K. R., Aleven, V., Heffernan, N., McLaren, B. M., & Hockenberry, M. (2004). Opening the Door to Non-Programmers: Authoring Intelligent Tutor Behavior by Demonstration. In the *Proceedings of the Seventh International Conference on Intelligent Tutoring Systems*.

[6] Morgan, P., & Ritter, S. (2002). An experimental study of the effects of Cognitive Tutor® Alegbra I on student knowledge and attitude. [http://www.carnegielearning.com/research/research_reports/morgan_ritter_2002.pdf].

[7] Murray, T. (1999). Authoring Intelligent Tutoring Systems: An analysis of the state of the art. *International Journal of AI in Education (1999), 10*, 98-129.

[8] Murray, T., Blessing, S., & Ainsworth, S. (2003). *Authoring tools for advanced technology educational software*. Kluwer Academic Publishers.

[9] Munro, A., Johnson, M.C., Pizzini, Q.A., Surmon, D.S., & Wogulis, J.L. (1996). A tool for building simulation-based learning environments. In *Simulation-Based Learning Technology Workshop Proceedings*, *ITS96*.

[10] 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* (pp. 333-336). Springer-Verlag.

[11] Ritter, S. & Blessing, S. B. (1998). Authoring tools for component-based learning environments. *Journal of the Learning Sciences*, *7*(*1*), 107–131.

[12] Ritter, S., & Koedinger, K. R. (1996). An architecture for plug-in tutor agents. *Journal of Artificial Intelligence in Education*, 7, 315-347.

[13] 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 and Education*, *15*(*3*).