Developing an Authoring System for Cognitive Models within Commercial–Quality ITSs

Stephen Blessing
Department of Psychology
University of Tampa
401 W. Kennedy Blvd., Box Q
Tampa, FL 33606 USA
sblessing@ut.edu

Stephen Gilbert
Clearsighted, Inc.
2325 Van Buren Ave.
Ames, IA 50010 USA
stephen@clearsighted.org

Steven Ritter
Carnegie Learning, Inc.
1200 Penn Avenue, Ste. 200
Pittsburgh, PA USA
sritter@carnegielearning.com

Abstract

Producing Intelligent Tutoring Systems (ITSs) is a labor-intensive process, requiring many different skill sets. A major component of an ITS, the cognitive model, has historically required not only cognitive science knowledge but also programming knowledge as well. The tools described in this paper were created to relieve this bottleneck in producing commercial-quality ITSs.

1 Introduction

ITSs are comprised of many different parts: an interface, a learner-management system, the curriculum, a teacher report system, and a cognitive model. It is this last piece, the cognitive model, which enables the tutor to provide help to the student, assisting when the student veers off track and staying out of the way when the student is doing the right thing. The focus of the work presented in this paper is the design and creation of tools centered on the cognitive model of an ITS. More specifically, we are interested in authoring cognitive models appropriate for model-tracing ITSs, where student input is checked on every student interaction.

While there have been some great ITS successes (e.g., Anderson, et al., 1989, Koedinger et al., 1997, Graesser et al., 2004, VanLehn et al., 2005), their development is a costly endeavor. Studies have shown that students who use an ITS to learn can master the material in a third less time (Corbett, 2001). In controlled studies in school settings, ITS-based curricula have been shown to be more effective in preparing students for standardized tests (Morgan & Ritter, 2002; also see www.whatworks.ed.gov). However, estimates for the creation of the tutored material range to 100 hours per hour of instruction or more (Murray, 1999). To this point, even with their promises of drastically reducing learning time, there have been few commercial ITS successes, due to this development barrier. For ITSs to become mainstream and realize their full potential, the creation of authoring tools is critical.

The research into authoring tools for ITSs is still quite new (see Murray, Blessing, & Ainsworth, 2003 for a review). In the past ITS researchers have had to design systems from scratch, using standard software development packages to construct the system. This requires not only knowledge of cognitive science in order to create the cognitive model, but also a fair amount of programming knowledge. Coupled with the domain knowledge requirements, any sizable ITS system requires a team of designers, each with a different skill set. While such team design processes are essential, the interdependence of the team can result in bottlenecks. If the cognitive scientist requires programming assistance in order to get a piece of work done, this slows down the effort. The goal we have for this project, which we feel is plausible, is not to enable an individual to create a whole ITS, but rather to better assist the person taking on a particular role to do his or her job better and more efficiently. A person who desires to create a tutor for a particular domain (be it a cognitive scientist or perhaps even a master teacher) should not be required to be a programmer to make substantial progress in creating the basics of the cognitive model.

The main challenge in this work is coming up with representations that enable the cognitive model designer to do their work without any programming and with a clarity not offered by present systems. Furthermore, this system is to be used in the context of creating and maintaining cognitive models in commercially available tutors. This work complements the research of Koedinger and his colleagues (2004), who are investigating other ways in which non-programmers can create cognitive models for model-tracing tutors. The Cognitive Tutors created by Carnegie Learning are in use in over 800 schools across the United States and by hundreds of thousands of students. This places additional requirements on the tools with regards to robustness, maintainability, and integration within a QA process.

In our conceptualization, the two main components of a model-tracing ITS cognitive model are the object model and the rule hierarchy. The object model represents the

This material is based upon work supported by the National Science Foundation under Grant No. DMI-0441679.

Copyright © 2006 AAAI

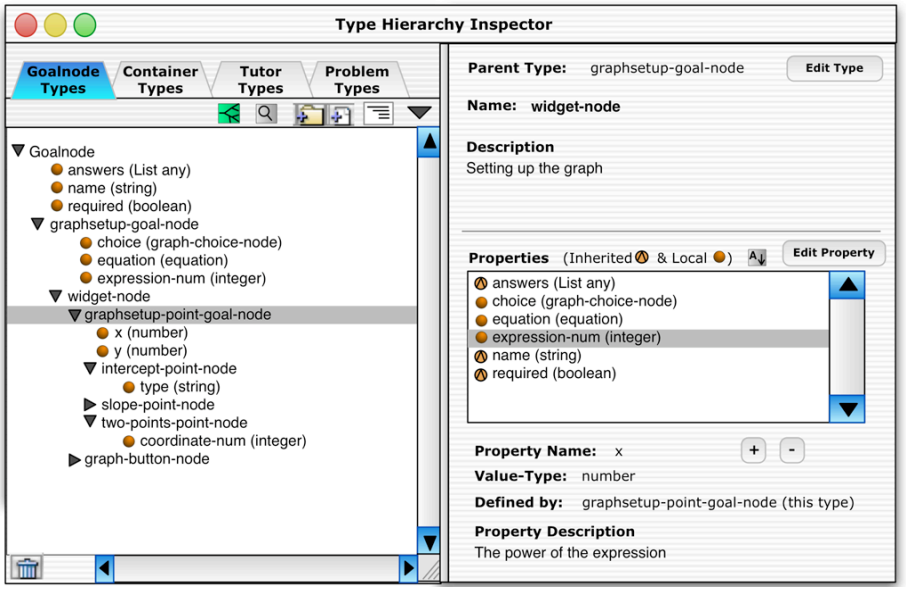


Figure 1. Object Model Editor.

pieces of the domain to be tutored, and this object model is used by the rules to provide the tutoring to the student. In the traditional approach to authoring model-tracing tutors (e.g., Anderson & Pelletier, 1991), working memory elements correspond to the object model and a flat representation of production rules correspond to our rule hierarchy. The particulars of the internal architecture used by us, referred to the as the Tutor Runtime Engine (or TRE), has been described elsewhere (Ritter, Blessing, & Wheeler, 2003). A set of tools has been created that work on top of this architecture to provide a software development kit (SDK) for cognitive models, and the purpose of this article is to describe the main pieces of this cognitive model SDK. The plan behind the SDK is to apply concepts that have been successful in other aspects of computer applications, such as tree views, hierarchies, and point-and-click interfaces, to the design of model-tracing ITSs. Development of such representations so they are usable by non-programmers in this context is difficult (to date, few ITS systems have employed them; the VIVIDS system, Munro et al., 1996, is an exception that has inspired part of the present work), but critical to lowering the bar in terms of both time and money in the creation of such systems. While nothing is automated yet, these tools provide much more support and structure for creating a cognitive model than what existed before (which for Cognitive Tutor Algebra I was essentially a blank document page).

2 Object Model

One of the main components of a cognitive model is the declarative structure used to refer to the objects and their properties within the domain being tutored. As such, a main issue was the creation of such a tool appropriate for a Cognitive Tutor. Traditionally this had been accomplished via code, requiring both programming and cognitive science knowledge. The main accomplishment here is in lowering the bar so that no programming knowledge is required. Moving to more of an object-oriented, object hierarchy based view is the key to cost efficiency in creating ITSs, both in terms of initial development and in on-going maintenance.

A main concern in the design of not only this tool, but also all of the tools that comprise the system, is that it support the viewing and editing of the existing cognitive models that have been produced by Carnegie Learning. This would ensure that the tools were of sufficient value to produce commercial-quality cognitive models. To this end, all tools described here meet that goal.

The requirements for this particular tool are similar to other existing tools (e.g., Protégé, an open-sourced ontology editor developed at Stanford University), in that the basic functionality is to display and edit objects consisting of attribute/value pairs. However, there are additional requirements for Cognitive Tutors that makes using any off-the-shelf or previously produced 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, and 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 in particular (e.g., the use of Tutorscript, a topic to be discussed later).

Figure 1 shows the design of the tool. The left pane of that design shows the currently loaded object hierarchy (including objects and properties) and the right pane shows information about the currently selected item in

Copyright © 2006 AAAI

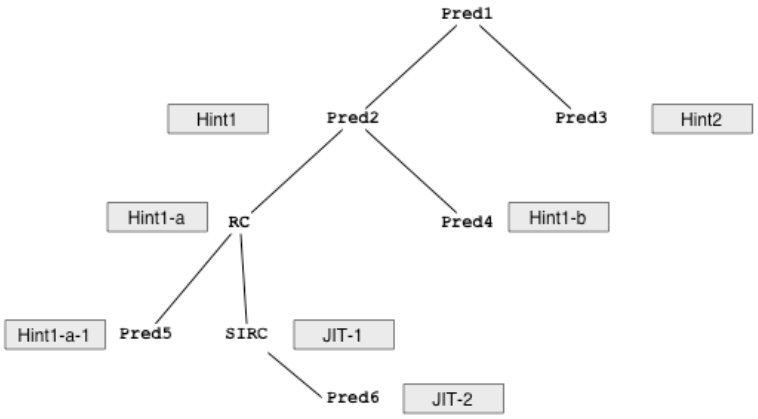


Figure 2. Predicate Tree Schematic.

the left pane. There are provisions for adding, moving, and deleting objects and properties, as well as maintaining other aspects of the tree.

The final, working version follows quite closely from this design. The full object hierarchy for Carnegie Learning's existing algebra and middle school math tutors can be viewed and edited using this tool (consisting of approximately 85 objects with 365 properties). In the past, 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. In addition, object hierarchies for other tutors have been entered using this tool. The superior visualization offered by the Object Model Editor has encouraged more code-sharing between different tutors, and has helped to identify and correct inefficiencies in the representation of objects. We feel that using this tool to enter, edit, and maintain object hierarchies for Cognitive Tutors is a clear win for the design of the object model of an ITS cognitive model. It has enabled us to find inefficiencies within existing code and to allow non-cognitive scientists to create cognitive models.

3 Rule Hierarchy

In addition to the object model, the other main piece of a cognitive model specifies the goal-state behaviors, such as right answers, hints, and just-in-time-messages—the backbone of a model-tracing tutor. The task is to design an editor for these rules, hints, and actions. As previously stated, one goal was that the viewer needed to display the existing rule sets that Carnegie Learning has developed (as a reference, Carnegie Learning's Algebra I cognitive model has approximately 500 rules).

Again, the main challenge is to come up with a repre-

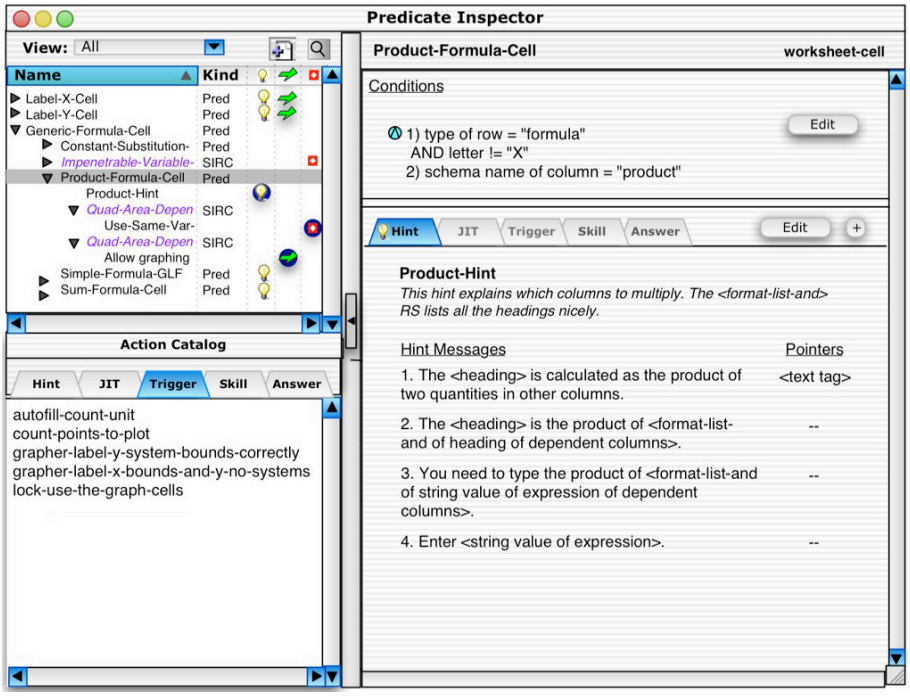


Figure 3. Predicate Tree Inspector.

Copyright © 2006 AAAI

sentation that is natural and understandable by non-programmers. Like with the object model, the traditional method of creating these rules was through code. However, we developed a non-code based representational scheme for these rules. The rules used by the cognitive modeling system share some features in common with the EPAM system (Simon & Feigenbaum, 1984), in that the predicates for the rules form a hierarchical tree, with most of the actions appearing at the leaf nodes. The nodes contain the tests (predicates) used to determine the behavior of the system. This behavior is dictated by how properties of the problem are coded in the object hierarchy and the subsequent actions of the student.

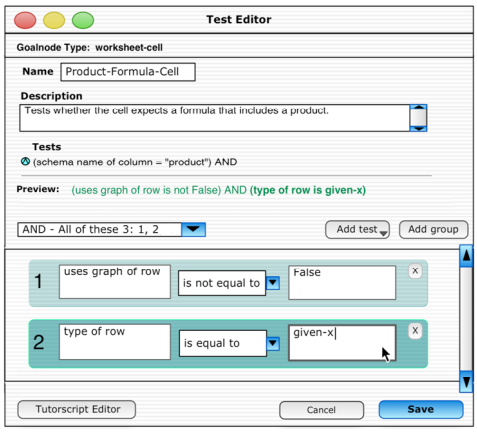
Figure 2 shows a schematic diagram of one of one of these trees. Each tree is particular to one type of goal state, and so the tests apply only to that goal state. For example, one test, or predicate, within the algebra cognitive model is if the cell-type (a property) of a worksheet-cell (a goalnode) is "expression." Later predicates within the tree test whether the expression cell is of the form "mx" or "mx+b". These properties are determined when the problem is authored. As seen in the schematic, the cognitive model is structured around these kinds of predicates. When a given predicate is satisfied, the student may see hints or "JITs" (just-in-time feedback messages). However, not all properties that are important for tutoring can be determined at authoring time, so some properties must wait until the student is in the midst of solving the problem, or at runtime. In the diagram, "RC" notates a Runtime Condition and "SIRC" notates a Student Input Runtime Condition. A Runtime Condition might, for example, specify that the hint to be given depends on whether the student has already completed part of the problem. A just-in-time message is triggered by SIRCs, because they depend on what the student entered.

Figure 3 shows the design for this Predicate Tree Inspector. 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. Finally, the lower left pane shows the Action Catalog, to be used for repeated tutoring actions within a tree. As desired, the Predicate Tree Inspector has the functionality to view all the rules and all their parts that comprise Carnegie Learning's Algebra I cognitive model.

3.1 Rule Editor

As shown in Figure 2, the nodes contain one or more predicates that test certain aspects of the current state of the problem. This is akin to a typical production that contains working memory tests in a rule-based system. As with the Object Model editor, the challenge here is to make such a representation understandable and usable by a non-programmer. The original Algebra I rules were built using a toolkit constructed on top of Common Lisp (the Tutor Development Kit, Anderson & Pelletier, 1991). While usable for simple tutors by non-Lisp programmers, the TDK still had a deep learning curve.

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 expression but leaves off the intercept." 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.

Figure 4. Rule Editor.

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.

3.2 Tutorscript Editor

Tutorscript provides a way for the author to refer to the object hierarchy within the constraints of the rules used by the cognitive model. Therefore, it is used not only in the predicates used within the rules, but also in the hint and just-in-time message templates, as well as other aspects of the authoring system. In addition to providing a way for authors to refer to properties within the object model, the syntax of Tutorscript also provides simple if/then clauses, arithmetic, and formatted output.

Copyright © 2006 AAAI

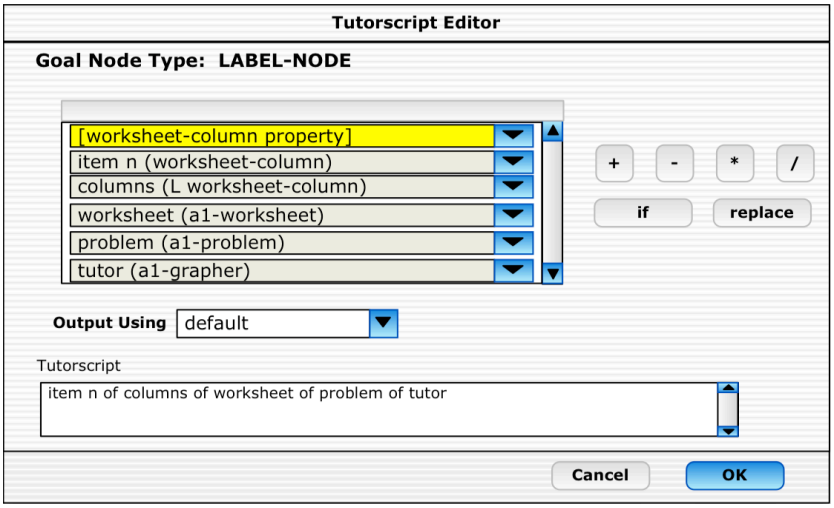


Figure 5. TutorScript Editor.

The only provision within the older authoring tools for Tutorscript was for the author to type out the reference. This placed a burden on the author to remember the syntax and the route through object space to reach the desired property. This added greatly to the programming aspect of what should be a cognitive modeling task.

Designing a Tutorscript editor is a challenge because a Tutorscript phrase is read from left to right, but is more properly constructed from right to left. A piece of Tutorscript like displayed in Figure 5, "item n of columns of worksheet of problem of tutor [of label-node]" starts with all the properties accessible from the goalnode *label-node*, of which *tutor* is one. The *tutor* property itself resolves to an object which has certain properties, of which *problem* is one. The Tutorscript phrase is built up like that until the author finds a path to the desired property, in this case *item n* (the author will need to indicated which exact item) of a list of *worksheet-column* object.

As can be seen in Figure 5, our design calls for the Tutorscript phrase to be built from the bottom up, with all the properties of the base goal-node (*label-node* in this case) selectable from a pop-up menu. Once a selection is made, the appropriate properties for the next part of the Tutorscript phrase appears in a pop-up menu above the initial menu, and so on until the author arrives at the terminal property. The dialog also includes the other features of Tutorscript such as arithmetic and simple if-then statements. The Tutorscript editor is available everywhere in the system that Tutorscript is allowed, which includes predicates, hint messages, just-in-time-messages, and other actions taken by the tutor.

4 Usable by Non-programmers

The main concern with this work is if the underlying representation used to represent the objects and rules are understandable by non-programmers (and indeed, even by non-cognitive scientists). Towards that end, a basic study that assessed the understandability of the object and rule model was conducted. A major risk associated with this project was not that the resulting tools could not author meaningful cognitive tutors, but rather that the tools would be too complex for a non-cognitive modeler to understand. An experiment was conducted to ensure that the representations developed for the object and rule viewers could be used and understood by people unfamiliar with cognitive models.

Sixteen undergraduate participants at a middle-sized liberal arts university took part in the study. These students had no computer or cognitive science background. Participants were instructed in one of two ways to represent information. One of these ways was more consistent with the older implementation of the authoring system (that is, a flat, programming-based representation), and the other way was more consistent with the representation embodied in the tools discussed above. Participants were first given instruction in objects, properties, and inheritance, as well as representing rules relating objects and actions. They were then given a test of their understanding. The instruction used examples of an animal hierarchy for objects, and a banking example for rules. The testing was designed to show whether students could generalize from their instruction using animals and banking to a new, foreign domain using similar representations. The new domain was based on the existing algebra system developed by Carnegie Learning for both objects and rules. The questions asked were ones of identification, such as "What are the inherited properties of this object?" "What are the children of this object?" and "Under what conditions will this message be displayed?"

Across both conditions, the self-paced instruction time was minimal to get them up to speed, amounting to less than 16 min. There were no significant differences between the two conditions in terms of time or accuracy (average of 75.0%), and all but one of the participants was significantly above chance in accuracy. The one participant not above chance was in the flat representation condition. We take this as evidence that these representa-

Copyright © 2006 AAAI

tions are readily learnable and usable by non-cognitive scientists and non-programmers. While we expected a stronger difference between the two conditions, with the hierarchy requiring less time, a difference might not arise until participants have to produce cognitive models, fix errors, and interact more deeply with the representations. Furthermore, the presentation of the information may not have had a high enough fidelity or been intuitive enough for the participants to accurately gauge differences.

A second result comes from a final task in which participants were shown a short narrative about objects, properties, and rules in the knowledge domain of cell phones and calling plans and asked to draw a representation of this domain freehand, based on the instruction and testing they had received. All but 2 of the 16 subjects produced a representation that was tree-based, like Figure 2, as opposed to using a flat representation or the hierarchical file-explorer-style representation (Figure 3). This suggests that a tree structure might be a natural representation for representing hierarchies. Future work will be needed to determine whether tree diagrams work well in a computer-based context both for representing and understanding hierarchical information.

5 The Rest of the ITS Authoring System

What we described here has been the authoring system for the cognitive model. In order to create a complete ITS system, many more pieces must be in place: the interface, the curriculum, and problems that fit within the curriculum. We are at various stages of work on these pieces. Ritter et al. (1998) described a problem authoring system for algebra problems. The tools described here have a simple mechanism for inputting the problem attributes problem, but we advocate the creation of special authoring tools for complex problems to open up the problem authoring process to a wider audience, including students. We have an in-house curriculum authoring system that enables problems to be put into sections so that tutor has the information available to provide a curriculum to the student. The curriculum authoring tool also allows the author to pick and arrange the skills that the students will learn. Finally, we currently have a simple interface toolkit available in java that allows for the construction of interfaces that can communicate to the tutor backend (described in Ritter & Koedinger, 1996). We envision a tutor GUI construction kit, but that work is in the future (the system described by Koedinger et al. does have such a provision).

6 Conclusions

The work described here is still ongoing, but the accomplishments have been sufficient for our team to do real work with the tools. Carnegie Learning's current Algebra I system is represented successfully using these tools. A current project is the re-implementation of the geometry cognitive model. While there is some overlap, geometry offers enough difference for us to gain perspective as to what the missing pieces are in the current implementation. Next steps include widening the domains of ITSs constructed using these tools, and integrating the various components of the total ITS authoring system.

7 References

Anderson, J. R. & Pelletier, R. (1991). A development system for model-tracing tutors. In *Proceedings of the International Conference of the Learning Sciences*, 1-8. Evanston, IL.

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

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

Feigenbaum, E., & Simon, H.A. (1984). EPAM-like models of recognition and learning. *Cognitive Science*, *8*, 305-336.

Graesser, A.C., Lu, S., Jackson, G.T., Mitchell, H., Ventura, M., Olney, A., & Louwerse, M.M. (2004). AutoTutor: A tutor with dialogue in natural language. *Behavioral Research Methods, Instruments, and Computers*, *36*, 180-193.

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*.

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].

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.

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

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*.

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.

Ritter, S., Anderson, J., Cytrynowicz, M., & Medvedeva, O. (1998) Authoring Content in the PAT Algebra Tutor. *Journal of Interactive Media in Education*, 98 (9).

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

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*).

Copyright © 2006 AAAI