Evaluating an Authoring Tool for Model-Tracing Intelligent Tutoring Systems

Stephen B. Blessing[1] and Stephen Gilbert[2,3]

[1] University of Tampa, Department of Psychology, 401 W. Kennedy Blvd., Tampa, FL USA
[2] Clearsighted, 2321 N Loop Dr Ste 110, Ames, IA USA
[3] Iowa State University, Department of Psychology, 1620 Howe Hall., Ames, IA USA
sblessing@ut.edu, gilbert@iastate.edu

Abstract. We have been creating an authoring tool, the Cognitive Model SDK, which allows non-cognitive scientists and non-programmers to produce a cognitive model for model-tracing tutors [1, 2]. The SDK is in use by developers at Carnegie Learning to produce their commercial Cognitive Tutors for math. However, it has never been evaluated with regards to the strong claim that non-cognitive scientists and non-programmers could, without much effort, produce useful cognitive models with it. The research presented here shows that this can be done, using a task that past researchers have used [3]. The models are evaluated across several metrics to see what characteristics of either them or their creators may distinguish better models from worse models. The goal of this work is to establish a baseline for future work examining how cognitive modeling can be opened up to a wider class of people.

1 Introduction

Model-tracing tutors have shown themselves to be one of the more effective types of Intelligent Tutoring Systems (ITSs) in terms of student learning gains [4, 5, 6]. However, they are very labor intensive to create, typically requiring a highly-qualified team of people to produce the end product. The experts needed on this team include cognitive scientists, programmers, and pedagogy and content experts. Estimates of how long it takes to create such a tutor range as high as over 100 hours of development time for 1 hour of instruction [7, 8]. For these tutors to see widespread use, this ratio needs to be greatly decreased. We see two ways of doing this: provide authoring tools that are 1) not only easier to use, but 2) also do not require high levels of expertise. The work presented here describes an evaluation of such a tool, the Cognitive Model Software Development Kit (SDK). While perhaps no authoring tool will obviate the need for a team of people from different disciplines to collaborate on building an ITS, the results from our evaluation indicate that non-cognitive scientists, historically the class of people who created the cognitive model, can produce a basic cognitive model.

1.1 The Cognitive Model SDK

The Cognitive Model SDK assists in the development of the cognitive model that forms the backbone of a Cognitive Tutor, a model-tracing tutor that is based on the

B. Woolf et al. (Eds.): ITS 2008, LNCS 5091, pp. 204–215, 2008.
© Springer-Verlag Berlin Heidelberg 2008

ACT Theory of cognition [9]. Carnegie Learning is a commercial company that produces Cognitive Tutors, primarily for topics in high school math. They have had great success in this endeavor, both commercially and in terms of student learning gains. The company currently uses the SDK to develop their Cognitive Tutors. To date six large-scale cognitive models have been built within the SDK, providing tutoring on over 5500 problems. However, no formal evaluation of this tool had been conducted until the present study.

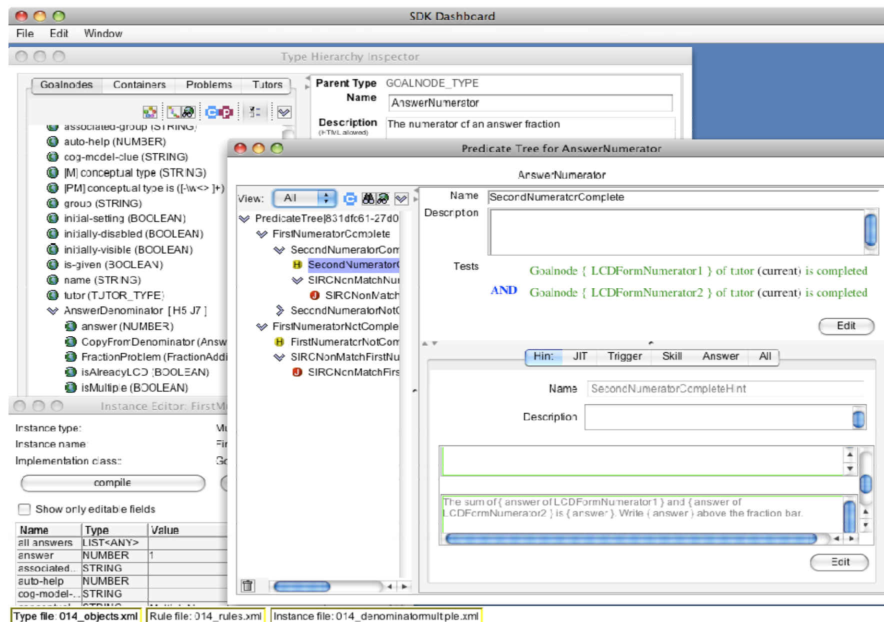

Fig. 1. Screenshot of Cognitive Model SDK

A full description of the Cognitive Model SDK is beyond the scope of the current paper, but can be found elsewhere [1, 2]. In short, there are three main pieces to the SDK and to creating a cognitive model. First, the type hierarchy used by the tutor must be defined. In ACT Theory terminology, this is the declarative knowledge of the model, containing the objects and properties that the student needs to be aware of in order to solve problems. For instance, in a domain like fraction arithmetic, the type hierarchy would contain information about what fractions are, their numerators and denominators, and other aspects of the task. “Goalnodes” are one important piece of this hierarchy. They represent the subgoals within the problem. There is typically an interface element (e.g., an entry box) that will allow the problem solver to complete a goalnode. The second SDK piece is the Rule Predicate Editor, where the rules of the task are defined (the procedural knowledge, in ACT parlance). This is where one can discriminate between different problem types in the domain (e.g., adding fractions where the denominators are already the same versus adding fraction that have different denominators). The help and just-in-time messages that are common to model-tracing tutors are stored here. The last

piece of the SDK is the instance editor, where a cognitive modeler can define problem instances. The SDK contains a simple testing interface by which the cognitive modeler can ascertain if the cognitive model is producing the correct behaviors, without attaching the cognitive model to the interface. Figure 1 contains a screenshot of the SDK, where the cognitive modeler is currently working on part of the predicate tree. Windows for the Type Hierarchy and instance editor can be seen in the background.

1.2 Past Evaluations of Authoring Systems

Evaluations of tutors built using authoring systems have been conducted in the past (e.g., [7, 10]), but evaluations of the authoring systems themselves are much more rare. Henry Halff and his colleagues conducted an early set of studies on the XAIDA authoring system [11]. While XAIDA did not produce model-tracing tutors, the studies that examined how authors used the system serve as a good model and can inform for future research in terms of the qualitative and quantitative data they collected. These studies showed very promising results, with findings like 30 hours of development time for 1 hour of instruction. Other authoring systems have also been evaluated in a similar manner (e.g., [7,12]).

We were motivated by two recent studies. First, Suraweera and his colleagues [3] performed a study looking at their Constraint Authoring System (CAS), a tool for developing a constraint-based tutor. In the context of a graduate student ITS class, 12 of 13 students were able to produce a constraint-based tutor for fraction arithmetic in an average of 31.3 hr. That almost all of the students could create an ITS, when creating ITSs may or may not have been the focus of their graduate studies, shows that the tool lowered the bar with regards to the expertise needed to create an ITS. We used the procedure outlined in this study to model the design of our own.

Second, researchers at Carnegie Mellon University are also developing an authoring tool for Cognitive Tutors. This tool, the Cognitive Tutor Authoring Tools (CTAT), approaches the task of developing a cognitive model from a different angle than ours [13]. Whereas a focus of their tool is to make it easy to create focused, more specialized tutors that center on a particular problem, what they term Example-Tracing Tutors, our authoring tool focuses on creating cognitive models that are more generalizable in nature. In a sense, our two different approaches are complementary in nature, perhaps with the long-term goal of finding that middle spot that exists between generality and ease-of-use.

1.3 This Evaluation

What follows is a description of an initial evaluation we did of the Cognitive Model SDK. We had conducted a study that examined whether or not the representations used in the SDK (e.g., the object/property view when working with types) were usable by undergraduates [1]. We found that they were. Participants in that study, however, did not create any working cognitive models. Furthermore, we have anecdotal evidence that non-cognitive scientists at Carnegie Learning and Clearsighted could create working cognitive models in a fraction of time that previous cognitive models were constructed [2]. This current study is 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 non-cognitive scientists/non-programmers can create usable cognitive models with the tool.

2 Method

2.1 Participants

Seventeen graduate students from Iowa State University participated in this study. Six of these participants were students in the first-year 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.

2.2 Materials

An assignment similar to the one used by [3] was used. In this assignment, participants were asked to create a cognitive model, consisting of the needed goalnodes, properties, hints, and just-in-time messages, 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) ones in which the two fractions started with the same denominators (e.g., 1/5 + 2/5); 2) ones where one denominator is a multiple of the other (e.g., 1/5 + 1/10); and 3) ones 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 itself in its Goalnode Testing Tutor (GNTT) interface. For this particular task, the two differ only by the placement of the boxes. The GNTT provides just a linear list of the goalnodes, represented by entry boxes, defined by the cognitive model and the current problem instance.

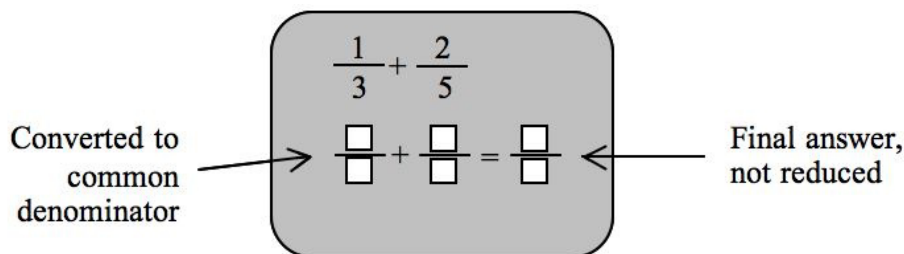

Fig. 2. Example interface shown to participants for the fraction addition task

Four pieces of background information were provided to the participants. First, some general information concerning Cognitive Tutors was given. For the students in the HCI class, this was a short in-class demonstration of the Carnegie Learning Algebra I Cognitive Tutor. For the other participants, this was a similar demo done via a screen capture movie (8.30 min in length) of the same demonstration. 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, 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 (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. The purpose of this information was to provide students a reasonably real-world example of a cognitive model within the SDK. While both this model and the model that the participants were asked to create both involved addition, the participants could not simply take this model, make a few simple modifications, and have a fraction addition model. The creation of the fraction addition model required the student to start from the beginning, constructing all new goalnodes, hint messages, and instances. This model served as inspiration and reference to show what is possible to do within the SDK at an appropriate level.

The last piece of information was another screen movie (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 is even or odd. This gave the participants an example of how to construct the bare minimum framework for a tutor from the start. That is, this example had one goalnode, one defined property, and one hint, plus enough glue to test the tutor within the GNTT.

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. 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.

2.3 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.

3 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.

3.1 Cognitive Model Measurements

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 did not complete the assignment, all of which were paid participants.

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 1. We rated models on behavioral characteristics, not on implementational aspects. The average score is 3.31, indicating that the average model at least met expectations. The mean time to complete the assignment was 7.68 hr.

We split models based on model quality. There were 6 "better" cognitive models that scored either a '4' or a '5.' Seven "poorer" models scored a '3' or below. These categories will be used in later analyses. Three models each came from the class and paid participants.

Table 1. 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., hints largely static)	2

Participants had to write a model that provided hints on 6 different entry boxes (see Figure 2). A cognitive modeler could use a single goalnode type to provide all hints, creating properties to distinguish them, or the modeler could use up to 6 goalnode types. Participants in our study used all possible numbers of goalnodes, with an average of 2.93. There were no statistically significant differences between the better and poorer cognitive models (2.50 v. 3.00, $t < 1$). All participants produced a flat object

model, meaning all defined goalnode types hung directly from the main default goalnode type. Participants defined 6.27 properties per goalnode type on average (the better and poorer cognitive models did not differ on this measure, 6.83 v. 6.22, $t < 1$).

We also examined participant's rule trees. The tree contains the hints that students will receive, as well as the just-in-time messages. Each node of the rule tree contains a predicate that tests aspects of the current problem's object instantiation and state. On average, the trees contained 13.12 nodes and were 2.08 nodes deep. The better and poorer cognitive models did not differ on either of these measures (for number of nodes, 13.17 v. 13.14, and for depth, 2.17 v. 2.00). 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.

It is hard to be evaluative as to whether better models should be deeper on either the object model or rule tree. We have debated with other modelers whether "bushes" (flat models with more properties) or "trees" (deeper models with more object types) are better with regards to the object model, which has consequences for the predicated tree. Among our colleagues, there are a variety of opinions, so perhaps it is not surprising there is little difference on these measures.

3.2 Timing Data Concerning Cognitive Model Creation

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 same amount of time (7.67 hr, with a range of 4.98 hr to 13.08 hr) than 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 2 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 2. 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 object model at the very beginning, and then turn to rule writing? Or, was there more give-and-take between working on the object model 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 one on top illustrating one of the participants who produced a better cognitive model, and the one on the bottom showing a poorer cognitive modeler.

The average participant produced 1156 actions (e.g., working on a hint, defining a property) as they worked on their cognitive model. These include edits and re-edits to the same entity. Again, there are no differences between the better and poorer participants on this measure (1181 v. 1205).

We examined the quantitative and qualitative aspects of these graphs for each participant. One difference that stands out, and can be seen in Figure 3, is the proportion

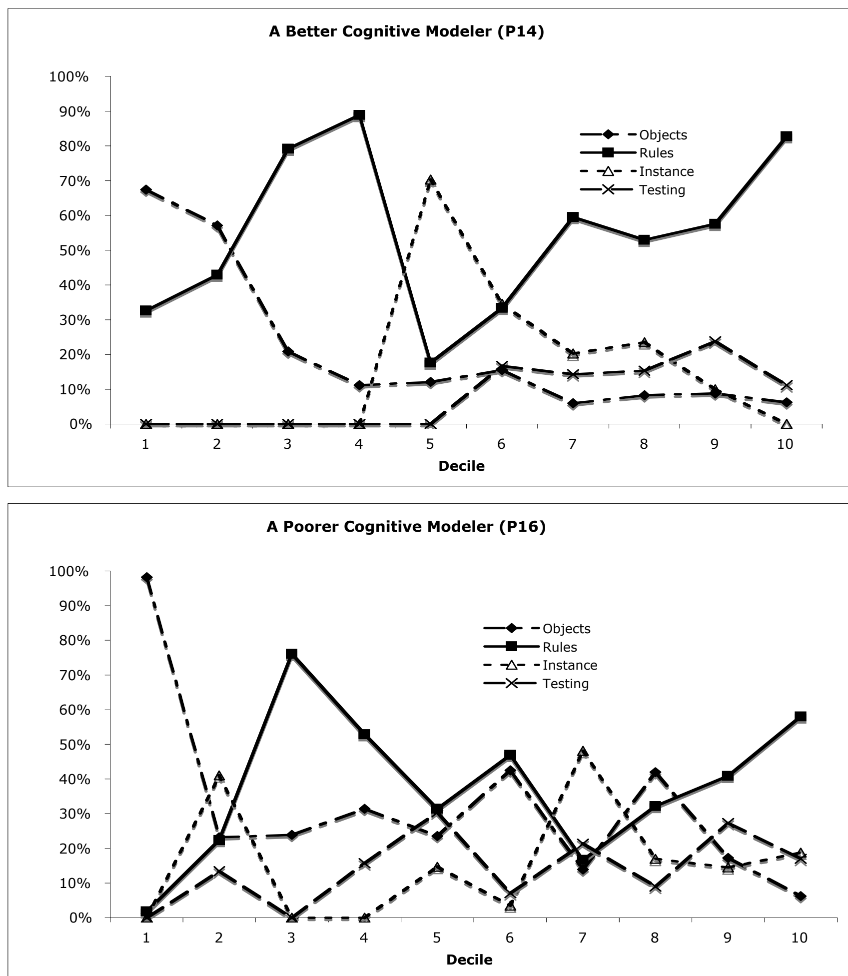

Fig. 3. Activity graphs of two participants

of time spent on the object model during the first half of the participant's time on task versus the second half. 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 gotten the object model mostly right 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 object model 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. This is a marginally significant difference by a chi-square test, $\chi^2(1, n = 13) = 2.86, p < .1$. Getting the model "right" (there are many potential "right" models) early appears important in 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.

3.3 Exit Questionnaire Data

Participants completed an exit questionnaire. We asked demographic information such as undergraduate major, current department, and number of programming courses. Examining undergraduate majors, there was 1 communication major, 1 journalism major, and 2 who did not specify a major. The rest had majors associated with technology (5 computer scientists, 2 engineers, and 2 information technologists). Their graduate degree programs reflect a similar trend (5 HCI, 2 engineers, 1 CS, 1 economics, 1 journalist, 1 information systems, 1 instructional technology, and 1 did not specify). There were no psychologists or cognitive scientists in the group. In looking across who created the better versus the poorer cognitive models, there is no clear trend. There were roughly equal numbers of computer scientists and engineers who created better models than 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, 6.83 v. 1.57, $t(11) = 3.37, p < .01$.

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 in the approach. Almost all participants saw both positive and negative features of these kinds of cognitive models in general, and using this tool specifically. The software is still somewhat of a beta quality, and the documentation is not complete. Indeed, the screen capture movies and example models were created specifically for this assignment in order to obviate the need for more complete documentation. Furthermore, the means by which properties are referred to in predicates and hint messages, tutorscript (in Figure 1, one can see a couple examples of tutorscript inside curly braces), is not well defined. In all, 10 of the 13 participants mentioned either insufficient documentation or bugs in their response, with 5 specifically mentioning tutorscript. 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. In actuality, this tool can be used to create any tutor appropriate for a model-tracing tutor. 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. This is an accurate statement regarding these kinds of cognitive models, and we believe can be taken as both a strength and a weakness. 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. A quote from one of the better cognitive modelers sums up many of our participants' reactions, "I was skeptical at first. It seemed like it took much more time to think through the model than just design a more direct system that would provide feedback to the user for each empty box…. However, my opinion changed when I got to the phase of the project where I could create instances of the problems. It went much faster than I expected, and I began to see that this system is much more flexible than I gave it credit for."

4 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 required a Masters or Ph.D. level cognitive scientist with much training in ITSs and in the particular ITS tool. Indeed, the tool used to create the cognitive model was often directly within a programming language or within rudimentary tools built on top of a programming environment, requiring much programming knowledge. None of our participants were psychologists or cognitive scientists, and none were in their graduate training to produce ITSs. They created their models quickly, in under 8 hr on average. Accounting for watching demos and other instructional activities, these participants went from ITS neophytes to having a model in about 10 hr. A proper interface, more problems, and refinements to even the best model would still have to be made (in general, the "better" cognitive models would require little work to be usable by a student, but the "poorer" ones would have needed much additional help), so it is unclear how to gauge this effort with regards to hours of development per hour of instruction. However, it would be closer to 10:1 than it would 100:1. While the tools and methodology are different, it is somewhat interesting to compare these results to those of [3]. In their study, 12 of 13 graduate students (92%) taking a class specific to ITSs completed constraint-based tutor for a similar fraction addition task. It took them on average 31.3 hr to do so, but that included much time to manually transfer pseudo-code into a runnable system. To do the initial constraints, it took them 6.5 hr. However, that time does not include writing hints and the just-in-time messages that our participants wrote, in addition to the predicates. On par then it seems like these are at least somewhat similar results.

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 appears 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? It seems reasonable to assume that programmers, either by their nature or because of their training, are better equipped to think about tasks that are conducive to putting them into a cognitive model. Specifically, we can think of at least two reasons why this might be so. 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 are also represented in a hierarchy, they still have the "if-then" quality of production rules. Again, this kind of thinking is probably more natural to programmers.

Given the nature of the Cognitive Model SDK, particularly in relation to CTAT and its goal of addressing more specifically non-programmers [13], the association between programming and producing better cognitive models using the SDK is not surprising. Future lines of research should investigate this relationship in more depth, to examine which pieces of programming knowledge most helps in creating a cognitive model. In such a way we can move towards more people being able to make higher quality cognitive models.

Acknowledgments. 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. We would also like to thank Matt McHenry, Tristan Nixon, Steven Ritter, Janea Triplett, and Leslie Wheeler for their help on this project.

References

1. Blessing, S.B., Gilbert, S., Ritter, S.: Developing an authoring system for cognitive models within commercial-quality ITSs. In: Proceedings of the Nineteenth International FLAIRS Conference, pp. 497–502. AAAI Press, Melbourne (2006)
2. Blessing, S., Gilbert, S., Ourada, S., Ritter, S.: Lowering the bar for creating model-tracing intelligent tutoring systems. In: Proceedings of the 13th International Conference on Artificial Intelligence in Education, Marina del Rey, CA, pp. 443–450. IOS Press, Amsterdam (2007)
3. Suraweera, P., Mitrovic, A., Martin, B.: Constraint authoring system: An empirical evaluation. In: Proceedings of the 13th International Conference on Artificial Intelligence in Education, Marina del Rey, CA, pp. 451–458. IOS Press, Amsterdam (2007)

4. Corbett, A.T.: Cognitive computer tutors: Solving the two-sigma problem. In: Bauer, M., Gmytrasiewicz, P.J., Vassileva, J. (eds.) UM 2001. LNCS (LNAI), vol. 2109. Springer, Heidelberg (2001)
5. Koedinger, K.R., Anderson, J.R., Hadley, W.H., Mark, M.A.: Intelligent tutoring goes to school in the big city. International Journal of Artificial Intelligence in Education 8, 30–43 (1997)
6. VanLehn, K., Lynch, C., Schulze, K., Shapiro, J.A., Shelby, R., Taylor, L., Treacy, D., Weinstein, A., Wintersgill, M.: The andes physics tutoring system: Lessons learned. International Journal of Artificial Intelligence and Education 15(3) (2005)
7. Murray, T., Blessing, S., Ainsworth, S.: Authoring tools for advanced technology educational software. Kluwer Academic Publishers (2003)
8. Woolf, B.P., Cunningham, P.: Building a community memory for intelligent tutoring systems. In: AAAI 1987, pp. 82–89 (1987)
9. Anderson, J.R.: Rules of the Mind. Erlbaum, Hillsdale (1993)
10. Ainsworth, S.E., Fleming, P.F.: Evaluating a mixed-initiative authoring environment: is redeem for real? In: Proceedings of the 12th International Conference on Artificial Intelligence in Education, pp. 9–16. IOS Press, Amsterdam (2005)
11. Halff, H.M., Hsieh, P.Y., Wenzel, B.M., Chudanov, T.J., Dirnberger, M.T., Gibson, E.G., Redfield, C.L.: Requiem for a development system: Reflections on knowledge-based, generative instruction. In: Murray, T., Blessing, S., Ainsworth, S. (eds.) Authoring tools for advanced technology educational software, Kluwer Academic Publishers (2003)
12. Mathan, S., Koedinger, K., Corbett, A., Hyndman, A.: Effective strategies for bridging gulfs between users and computer systems. In: Proceedings of HCI-Aero 2000: International Conference on Human Computer Interaction in Aeronautics, Toulouse, France, September 27- 29, pp. 197–202 (2000)
13. Aleven, V., Sewall, J., McLaren, B.M., Koedinger, K.R.: Rapid authoring of intelligent tutors for real-world and experimental use. In: Kinshuk, R., Koper, P., Kommers, P., Kirschner, D.G. (eds.) Proceedings of the 6th IEEE International Conference on Advanced Learning Technologies (ICALT 2006), pp. 847–851. IEEE Computer Society, Los Alamitos (2006)