A Programming-by-Demonstration Tool for Retargeting Instructional Systems

Steven Ritter
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213
USA
412-268-3498
sritter@cmu.edu

Stephen B. Blessing
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213
USA
412-268-7136
blessing+@cmu.edu

Abstract: Although intelligent learning environments have proven to be successful instructional tools, they have seen little commercial success. Part of the problem has been the difficulty of authoring them, even using authoring tools specialized to the task. We argue that the authoring task can be redefined for environments based on the "plug-in" instructional architecture described by Ritter and Koedinger (1995). In systems using that architecture, specialized authoring tools can be developed for each component of the architecture, greatly simplifying both the authoring process and the task of creating authoring tools. We describe an authoring tool directed at creating the "translator" portion of such an architecture. This authoring tool uses programming by demonstration to allow a non-programmer to re-use existing tutoring agents with new interface tools.

Introduction

Intelligent tutoring systems have proven to be remarkably effective instructional tools (Anderson, Corbett, Koedinger and Pelletier, 1995; Koedinger, Anderson, Hadley and Mark, 1995), but the cost of developing them have proven to be a major obstacle to their commercialization.

There have been several solutions proposed to this problem. One class of solutions involves building programmer-level authoring tools for tutoring systems. An example of this type of solution is the tutor development kit (TDK) described by Anderson and Pelletier (1991). The authoring process using this tool involves studying people performing the target task and identifying the strategies used for accomplishing it. These strategies are then translated into a set of production rules that constitute an executable expert model for performing the task.

This expert model can be augmented by "buggy rules," which represent incorrect but frequently-attempted strategies. These buggy rules allow the system to identify cases where a student has strayed down a particular incorrect path. The rules in the expert model are also annotated with "help text," which is used to construct messages to users in response to requests for help.

In addition to constructing the expert model, programmers must design and implement a computer interface for students to use to perform the task. This interface is developed with the TDK and is closely tied to the knowledge representations underlying the production rules in the expert model. This design guarantees that the visual representation shown to the student corresponds to the underlying representation being used by the expert system, since any change to the underlying knowledge representation is immediately reflected in the user interface and every user input is immediately reflected in the underlying knowledge base.

An alternative approach to authoring tutoring systems is discussed by Blessing (1995). This "next generation" method uses programming by demonstration (PBD; see Cypher, 1993) to define the instructional system. Authoring a system in this way requires two steps. The first step is to define a domain-specific authoring tool. The next step is to use this domain-specific tool to define a task-specific tutor.

In Blessing's example, the domain-specific authoring tool can be used to author tutors for instructing students in simple arithmetic. Building the authoring tool consists of creating an interface appropriate to the domain. In this case, the interface consists of a grid of cells in which placeholders for numbers can be entered. The domain-specific authoring system must also be augmented by domain-specific knowledge, such as addition and subtraction tables and by knowledge about how to infer underlying rules when presented with actions in the interface.

Once the domain-specific authoring tool exists, it is very simple to build a tutor for a specific task, such as teaching subtraction using the right-to-left, immediate borrowing algorithm. The author of this tutor need

only solve a few different subtraction problems using the interface and supply appropriate "help text" when prompted by the system. Any productions that are wrongly inferred can be easily fine-tuned at this point.

The biggest payoff from this type of system comes, of course, when the domain-specific authoring tool can be used to build many different task-specific tutors. In these cases, the payoff can be substantial. Using the arithmetic authoring tool Blessing describes, it is possible to create a complete tutor for right-to-left subtraction in less than half an hour.

Each of these approaches has advantages and disadvantages. The TDK approach is very flexible, but it is essentially a specialized programming task which is best accomplished when the programmers are skilled in interface design, cognitive science and standard programming. Although the close connection between interface and knowledge representation ensures consistency, it also requires that development of these two aspects of the system proceed in parallel. As a practical matter, the interface needs to be composed with elements that are present or easily implemented in the TDK itself.

The PBD approach succeeds by shifting much of the work in tutor development to authoring of a domain-specific authoring tool. Once this is accomplished, creating a task-specific tutor is a small and simple task that does not require any programming ability. Still, substantial research needs to be done to identify general and powerful methods of inferring production rules from actions in the domain-specific authoring tool. This PBD approach also suffers from one of the problems of the TDK approach: the close connection between interface and knowledge representation. In this case, the interface that an author uses to create a task-specific tutor from a domain-specific tool becomes the same interface that the student uses in learning a task using the tutor. While this may be appropriate for some domains, it may not always be the case that the same interface can effectively serve for both tasks. In general, it would be beneficial to allow the possibility of a different interface for authoring than for teaching.

Recently, Ritter and Koedinger (1995) proposed a "plug-in" architecture for educational systems that, while not dictating a specific approach to authoring, can help to redefine the authoring task. Ritter and Koedinger separate the learning environment into three parts (see Figure 1): the user interface or tool, the tutoring agent and the translator (an additional part, the curriculum manager, is not relevant to this discussion). The user tool is the element of the system that users see and manipulate. Unlike in the TDK approach, the user tool is not just a way of presenting and allowing users to manipulate knowledge elements. In some systems, the tool is responsible for a substantial amount of interaction. For example, the tool could be a spreadsheet, with all the capabilities of a program like Microsoft's Excel. However, the tool's major responsibility is to accomplish a task, not to teach the user how to accomplish the task. The tutoring agent is responsible for providing instruction and correcting errors. The complete learning environment augments the user tool with the ability to teach the user how to use the tool in order to accomplish the task. For example, a learning environment may be developed to teach budgeting, using Microsoft Excel as the user tool and providing help and feedback in using that tool to create a budget. The translator handles all communication between the user tool and the tutoring agent.

One of the advantages of the plug-in architecture is that it allows us to create learning environments which incorporate off-the-shelf interfaces, including tools like Microsoft Excel. This can save many hours of development time. If no appropriate off-the-shelf tool exists, authors must develop their own interfaces, as they do in current systems. However, in this case, the interface can be developed independently of the cognitive model used in the tutoring component. Thus, we can build cognitive models using either a programming tool like the TDK or a PBD tool like Blessing's arithmetic authoring tool and then retarget the cognitive model to be used with a different interface. This retargeting may include, if appropriate, an interface consisting of an off-the-shelf tool. For example, the interfaces for both the algebra tutor described in Koedinger, et. al., 1995 and Blessing's (1995) subtraction tutor consist, in large part, of a table. Retargeting would allow us to use a worksheet from Microsoft Excel in place of the home-grown interfaces used in these tutors. In fact, the initial version of the Excel-based algebra tutor described in Ritter and Koedinger (1995) was retargeted in exactly this way. The cognitive model (and an accompanying interface) was developed using the TDK, and this cognitive model was then retargeted to use Microsoft Excel as its interface.

The implication of this architecture for authoring tools is that, instead of using a single authoring tool to build the whole system, we can use three different authoring tools, each specialized for a specific part of the system. For the user tool (assuming that we cannot use one off-the-shelf), there are a wide variety of visual programming languages, multimedia authoring tools and traditional programming environments that might be appropriate. The tutoring agent can be created with either a programming environment, like the TDK, or a PBD tool, like Blessing's environment.

The remaining part of the plug-in architecture, the translator, is the link between tutoring agent and user tool, so it is the key to retargeting a tutoring agent. The remainder of this paper describes a programming by

demonstration system that we have developed which allows non-programmers to create the translator portion of the architecture, helping them retarget cognitive models to new user tools.

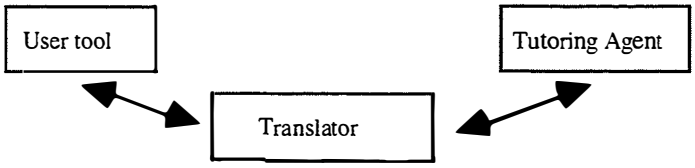

Figure 1: Overview of the "plug-in" architecture

The plug-in architecture provides a protocol for communication between the user tool and the tutoring component. The "translator" portion of the plug-in architecture translates the user tool's messages into messages following the protocol. It also translates the tutoring agent's messages into messages that can be understood by the user tool. This allows the tutoring agent to use an abstract, tool-independent language for communicating with the user, enabling the tutoring agent to be used with different tools that serve the same function. For example, when the tutor indicates that it would like to "point out" some text to the user, the translator is responsible for deciding how to implement the "pointing." With a certain tool (and in a certain context) it might be appropriate to print the text in italics. In another tool (or another context), it might be appropriate to circle the text. The authoring tool described here provides a means for defining communication within this protocol. From the author's perspective, the task is to define a set of if/then rules which specify the tool-specific messages to send when certain user or tutor actions are taken.

Representation

A difficult issue in PBD system is how to represent the information that the system infers from user actions (Myers, 1993). One option is to present information graphically, in a way that maximally corresponds to the user's action (e.g. Cypher, 1993). This method has the advantage of being easy for the user to understand, but it can get unwieldy if the user needs to modify the systems inferences. Consider a graphic way of representing the action of typing into a cell in Microsoft Excel (Figure 2). For tutoring purposes, this action needs to be described as a selection, an action and an input. The diagram makes it clear how the system infers a selection, action and input from the user's action. Suppose, however, that the tutor wanted the selection in R1C1 format (that is, as R2C3 instead of C2). How could the user indicate this desire to the system?

While there are many solutions to this particular case, there is no general solution which allows the user can refer to concepts (like column number 3) that are not directly visible in the interface. All solutions must involve (graphically) augmenting the interface to make these concepts visible. Even then, it is a significant task to define a PBD system which allows the user to edit the system's graphical inferences. As a general rule, presenting information graphically works best when we expect minimal editing of rules by users.

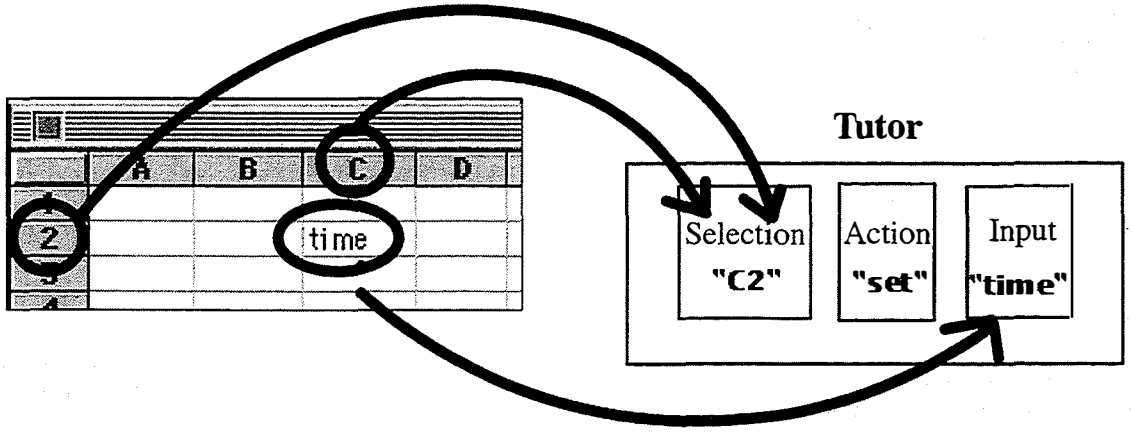

Figure 2: PBD System employing graphic feedback

A contrasting approach is to present system inferences in text. Many macro programming systems allow demonstration of an action which is then presented back to the user as text. This allows for easy modification of the macro but such modification typically requires knowledge of the macro programming language.

Our system follows this approach by presenting its inferences as rules composed in AppleScript, a natural-language-like scripting language that is in wide use on Macintosh computers. AppleScript is easy to understand (since it is close to natural language), so users should have little trouble understanding the system's inferences. In addition, simple modifications (like substitution of nouns and constants) is fairly simple. For the most part, the user does not need to construct new AppleScript statements, although the system allows that. Instead, the user demonstrates activities in the user tool, and the system generates AppleScript rules which the user can then edit. In future versions of the system, we hope to provide ways for users to demonstrate actions which result in rule modifications. This would reduce the programming requirements of the system further.

Defining communication from the user to the tutoring component

The protocol for communicating from the user tool to the tutoring component is very simple. There are four basic messages: *start-problem*, *process-tool-action*, *process-help* and *process-done*. The author's task, then, is to define a mapping between everything the user can do within the user tool and one of these messages. This mapping is represented by if/then rules, in which the condition ("if") part represents a user action and the action ("then") part represents the message to be sent to the tutoring component.

To define an if/then rule representing the mapping between a user action and a message to the tutoring component, the author first enters "recording" mode. This directs the Macintosh operating system to instruct all applications (user tools) to produce AppleScript corresponding to each user action.

The author then goes to a user tool and performs an action, just as the user would. For example, to define a rule to be used when the user types into a cell in Excel, an author would go to Excel, select an appropriate cell and type in a value. Figure 3 shows the display in the authoring tool after returning from typing "999" into cell "R2C2" in Excel.

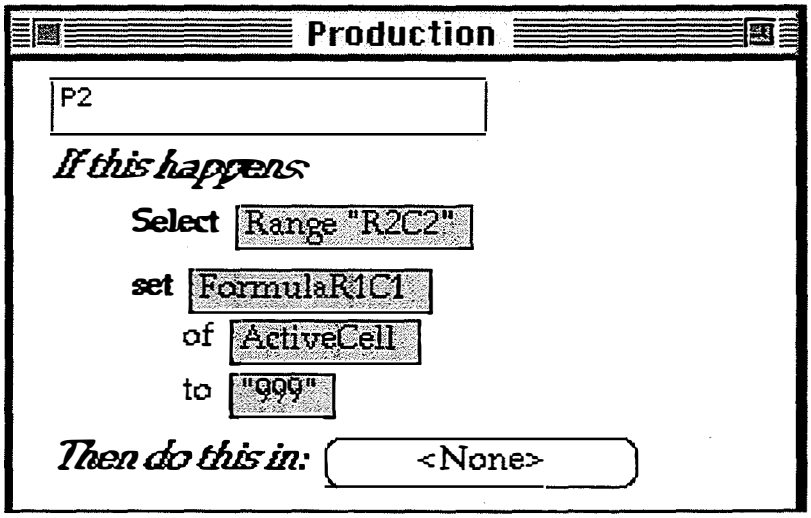

Figure 3: Display after typing "999" into Excel's cell "R2C2"

The system has recorded actions demonstrated in Excel and identified portions of each action that may be variables (those portions are displayed in the shaded boxes). Typically, we would want to generalize this rule to apply when any value is typed into any cell.The user can, by double-clicking, change elements of the AppleScript statement to variables (called "wildcards") or into "don't care" elements. If the goal was to define a rule that defines an action specific to the user typing "999" into cell "R2C2", the condition could be left as it is. To generalize the rule, we would double-click on "R2C2", turning it into "wildcard1" and then double-click on "999", turning it into "wildcard2". The rule can also be given a name. The completed rule, shown in Figure 4, is named "Set Cell".

Note that Excel records the action of typing into a cell as two actions:

```
select range "R2C2"
```

and

```
set formulaR1C1 of activeCell to "999"
```

Other applications might record this action in a single step as

```
set formulaR1C1 of cell "R2C2" to "999"
```

The translator authoring system will operate in the same manner, regardless of whether the action is recorded as a single step or as two steps. In fact, these differences in the way that similar applications describe their actions is one of the reasons that the translator is required in the plug-in tutoring architecture. The tutoring agent need not know the details of how the user tool communicates user actions.

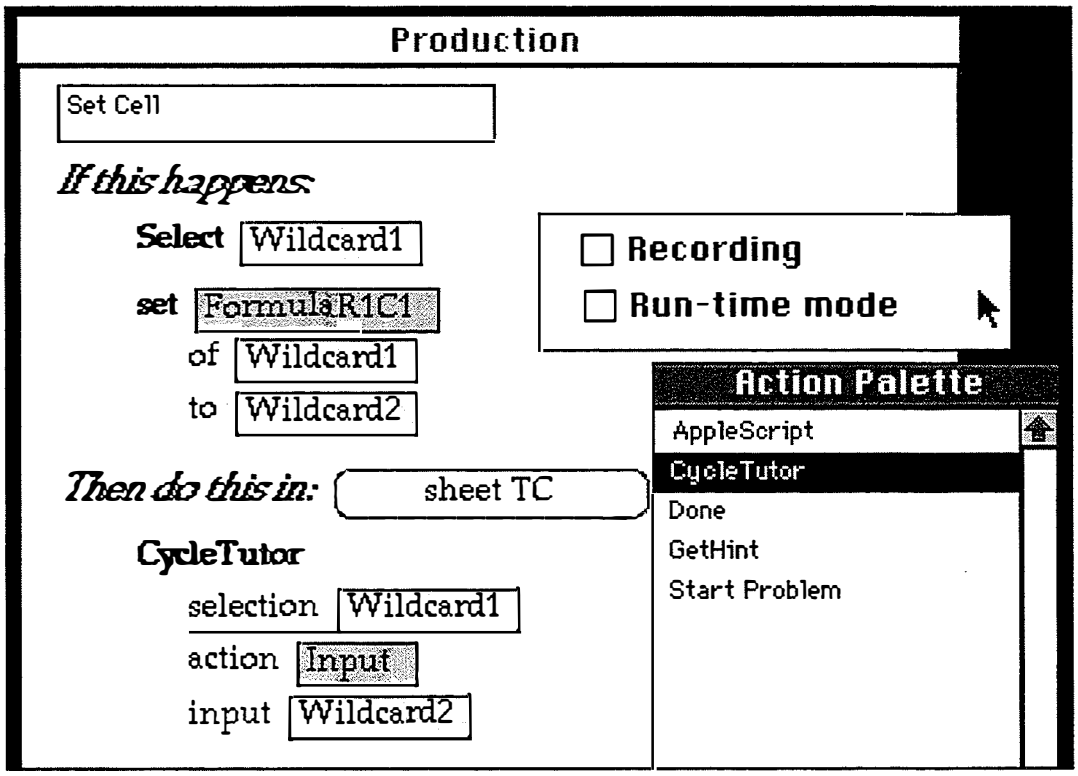

Figure 4: Completed "Set Cell" rule

To define the "action" part of the rule, we first must identify the tutor application to which the rule applies. To do this, we click on the button next to "then do this in." The system presents a dialog with which we can select the tutor application (in this case, "sheet TC"). Note that this system allows different rules to target different tutoring applications.

Since the possibilities for the action part of the rule are limited to the four messages, the author need only drag-and-drop one of these actions from the "action palette" to the action part of the rule (see Figure 4). External names for the messages are in plain English, so the four messages are phrased "start problem", "cycle tutor", "give help" and "done." Once the action is selected and placed into the rule, the system provides places to define arguments. For example, the "cycle tutor" message provides places to define the "selection", "action" and "input" arguments. If an argument is to be represented by a constant, the user need only type in the appropriate string. Alternatively, the user may drag-and-drop a constant or variable from the condition to use an argument.

Rules for start-problem, process-help and process-done can be easily defined, since they usually each correspond to a single action in the user tool (selecting "new problem", "help" and "done" buttons or menuitems, respectively). To define the mapping from Excel to the tutor for a simple application, four rules would need to be defined: one for typing into cells and one each for start-problem, process-help and process-done. If we wanted to incorporate other Excel actions (like creating a chart), we'd need to define more rules. If a user performs an action that is not recognized by any rule, that action is ignored by the tutoring agent. It is

possible to prevent the user from performing some actions in Excel by defining rules that "undo" those actions.

Defining communication from the tutoring component to the user

The next step in the tutoring process is to define the mapping from the tutoring component to the user interface. This process operates in much the same manner as defining the mapping from user interface to tutoring component except that, in this case, the conditions are selected from a "condition" palette and the actions are defined by demonstrating activity in the user tool.

The plug-in architecture defines a limited set of ways that the tutoring component can communicate back to the user. The purpose of creating a mapping from the tutoring component to the user tool is to define the specific way in which the user receives feedback. For example, in one application using Excel as the user interface, we might want to indicate errors by showing erroneous cells in red. In another application, we might want to beep and present the erroneous cell in italics. The plug-in architecture specifies ten messages from the tutoring component to the user interface: *flag*, *unflag*, *point-to*, *send-message*, *undo*, *select*, *verify*, *update-assessment*, *perform-user-action*, *start-activity* and *get-user-value*. For a typical application, one rule needs to be defined for each of these messages.

Suppose that we are defining the rule for "flag" and that we want the system to display flagged cells in red. To define this rule, we would first drag the word "flag" from the "condition" palette to the condition section of a rule. The flag argument (which represents the thing to be flagged) initially appears as a variable, "wildcard1". In most cases, it is desirable to leave the argument as a variable, but the argument can be changed to a constant if there are different ways of flagging different types of items.

Once the condition part of the flag rule is defined, we would check the "recording" box, go to Excel, click on a cell (any cell will do) and change the color of that cell to red. When we switch back to the authoring tool, the system has recorded the following two statements in the "action" portion of the rule:

```
select Range "R2C2"
set ColorIndex of Font of Selection to 3
```

In this case, the "select" statement is extraneous, so we delete it by clicking on it and pressing the "delete" key. In order to create the appropriate "set colorIndex" statement, we drag "wildcard1" from the condition part of the rule to the word "selection". Releasing the mouse button produces the full, correct action:

```
set ColorIndex of Font of wildcard1 to 3
```

The rule is now sufficient to direct the system to present all flagged cells in red. Note that we did not need to know the specifics of Excel's scripting syntax or the fact that colorIndex 3 corresponds to red. The other nine rules defining the mapping between tutoring component and user tool can be defined in a similar manner.

Specializing the system

The authoring process described so far may be sufficient for many retargeting tasks, but the system is designed to be more general in order to handle more complex situations. To accomplish this, the system allows the user to edit any recorded scripts.

For example, in one retargeting application, we were using a tutoring component that expected a cell-based user interface similar to that used by Excel but which described cells in a zero-based system (i.e. the first cell was R0C0, not R1C1). In this case, we cannot define the "set cell" rule simply by passing the name of the active cell to the tutoring component. Instead, we use the "AppleScript" action to allow entry of AppleScript code to perform this manipulation (see Figure 5).

The initial definition of this rule can be created by demonstration, but the user will have to augment it by adding the statements contained in the box. Of course, defining a rule like this requires some knowledge of AppleScript, but the programming is kept to a minimum.

Testing the System

Once the proper rules have been created, the system can be tested by entering "run-time mode". Run-time mode starts AppleEvent recording but, instead of using the recorded AppleEvents to define a rule, the recorded events are checked against rule conditions the user has defined. When a match is found, the appropriate rule actions are executed.

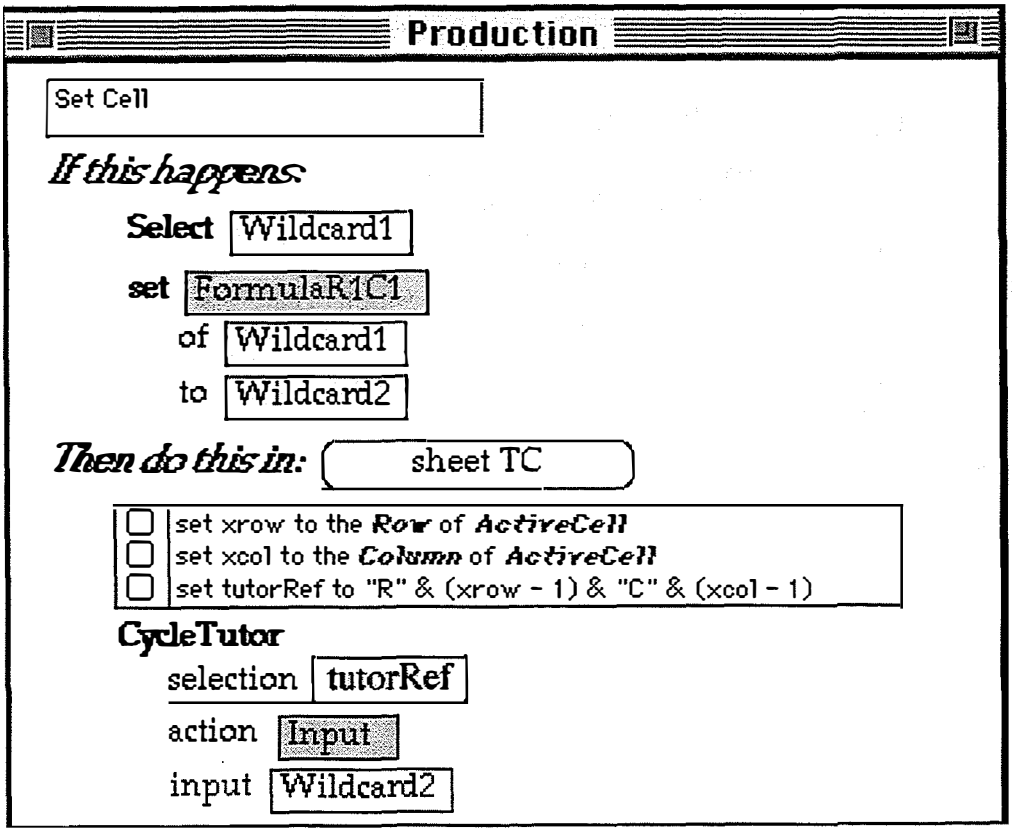

Figure 5: Rule showing use of AppleScript extension to rule action

One potential complication arises when using a tool that communicates these actions the way that Excel does. The "select" statement is, essentially, setting a state variable ("activeCell") which is then referenced in the "set formulaR1C1" statement. Currently, our system will recognize the rule conditions as being satisfied even if other actions occur in between the two steps (such as if the user selected cell "R2C2", set the text style to "bold" and then entered some value as the input). This works fine, under the assumption that no actions other than "select" actions will change the value of "activeCell". In fact, there are some actions (such as opening a new workbook) which would change the value of this variable without recording a "select" statement. An improved version of our system would recognize the use of state variables (which are publicly available in the application's AppleEvent dictionary) and would query the application when a rule fires to get the value of any such variables used in a rule's actions.

Retargeting and tutoring components

Although we have discussed retargeting as a way of defining complete tutoring systems, we believe the real benefits of retargeting are in being able to re-use tutoring components and portions of tutoring components in new contexts. We have used retargeting to take a tutoring component that was written for a custom interface and use it with a Microsoft Excel interface. Ultimately, we expect to be able to build a library of small, focused tutoring components. For example, one tutoring component might know how to tutor algebraic symbol manipulation, another would know how to compose data in a table and yet another to create an appropriate graph of data. The plug-in architecture and the retargeting tool described here would allow us to develop each of these tutoring components independently and test each with a simple, custom-built interface. Using the retargeting tool, we can then retarget these tutoring components to operate with a customized interface within a learning environment for introductory algebra and also, separately, to operate with Microsoft Excel in a learning environment for business accounting. The real benefit of retargeting, then, goes beyond the ability to attach new interfaces to existing tutoring components; it allows us to reuse pieces of tutoring knowledge in new contexts.

Future Enhancements

We have tried to make the system described usable by people not familiar with programming concepts. However, the idea of a "Wildcard" (that is, a variable) used in generalizing the rule may be difficult for people with no programming experience to grasp. One addition to our system that we have considered would automatically induce which statements should be generalized. It would do this by comparing across multiple instances of the same user action, variablizing the structures which are different between the instances.

To give an example, consider the rule in Figure 4. The first time the author created the rule, the condition side recorded that the author selected a specific cell and then typed something particular into that cell. The author could then construct the action side using the specific values as initially recorded. It may not be clear to the author, particularly a first–time user, which values should be variablized. Using the proposed extension, the author could indicate to the system that he or she is going to redo the actions of that rule. The system would then compare those actions to the current rule. Where the current rule differs from the incoming actions, those structures would be variablized. Of course, the author is now going to have to be sensitive to the fact that when redoing the steps, the actions will have to occur in a different context (e.g., a different cell) and perhaps with different inputs (e.g., a different value). Otherwise, the rule will not variablize correctly and be too specific.

A similar extension can also be applied to the few cases where AppleScript will need to be produced to augment the rule. By comparing across multiple instances of a rule, the AppleScript could automatically be generated. Examining the rule in Figure 5, the gist is that the row and column numbers given by the application need to be decremented by one in order to be used by the tutor. In a process similar to ACT–R's analogy mechanism (Anderson, 1993), this pattern can be induced within the actual instantiation of this rule, and the AppleScript which does the transformation could generated.

Conclusion

It is clear that, for the benefits of intelligent tutoring systems to be realized in commercial systems, we need to reduce the cost of their development. Several approaches to solving this problem were discussed in this paper. Finally, we have defined a programming-by-demonstration system that, working with the plug-in architecture described by Ritter and Koedinger, allows non-programmers to retarget tutoring information in new contexts.

References

Anderson, J. R. (1993). *Rules of the Mind.* Hillsdale, N. J.: Erlbaum.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences, 4* (2) 167-207.

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

Blessing, S. B. (1995). ITS authoring tools: The next generation. In *Proceedings of the Seventh World Conference on Artificial Intelligence in Education* (p. 567). Charlottesville, VA: Association for the Advancement of Computing in Education.

Cypher, A. (1993) Eager: Programming repetitive tasks by demonstration. In A. Cypher (Ed.) *Watch what I do: Programming by demonstration* (pp. 204-217). Cambridge, Mass: MIT Press.

Koedinger, K. R., Anderson, J. R., Hadley, W. H., & Mark, M. A. (1995). Intelligent tutoring goes to school in the big city. In *Proceedings of the Seventh World Conference on Artificial Intelligence in Education*, Charlottesville, VA: Association for the Advancement of Computing in Education.

Myers, B. A. (1993). Demonstrational interfaces: A step beyond direct manipulation. n A. Cypher (Ed.) *Watch what I do: Programming by demonstration* (pp. 204-217). Cambridge, Mass: MIT Press.

Ritter, S. and Koedinger, K. R. (1995). Towards lightweight tutoring agents. In *Proceedings of the Seventh World Conference on Artificial Intelligence in Education* (pp. 91-98). Charlottesville, VA: Association for the Advancement of Computing in Education.

Acknowledgments

This material is based upon work supported by the National Science Foundation and the Advanced Research Projects Agency under Cooperative Agreement No. CDA-940860