Jl. of Artificial Intelligence in Education (1996) 7(3/4), 315-347

An Architecture For Plug-In Tutor Agents

STEVEN RITTER
Department of Psychology, Carnegie Mellon University
Pittsburgh, PA 15213, USA

KENNETH R. KOEDINGER
Human-Computer Interaction Institute
School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213, USA

This paper outlines the authors' efforts to build new learning environments that incorporate tutoring elements into pre-existing software packages. Two systems are described; one provides tutoring support in the *Geometer's Sketchpad* and the other supports students using *Microsoft Excel*. Although the implementation of these two systems was somewhat different, they share many basic components. An analysis of their similarities and differences allows us to move toward a set of standards for tutor agents that interact with complex tools. By constructing learning environments in this manner, we can leverage the power of existing workplace software and educational microworlds to create more powerful learning environments.

INTRODUCTION

There are clear pedagogical benefits of well-designed intelligent tutors as demonstrated by numerous studies (cf. Anderson, Corbett, Koedinger, & Pelletier, 1995; Lajoie & Lesgold, 1989; Koedinger, Anderson, Hadley, & Mark, 1995; Mark & Greer, 1995). Nevertheless, the cost of developing large-scale intelligent tutoring systems and the difficulty of adapting them

o curricular objectives other than those of the original designers has im-
peded their use in many domains (McArthur, Lewis, & Bishay, 1996; Brusilovsky, 1995). We are moving toward a new architecture for "plug-in tutor agents" to address these problems and to further increase the flexibility and power of software environments that can provide intelligent learning support.

In contrast to the traditional "all-inclusive" architecture of intelligent tutoring systems (Anderson, 1988), plug-in tutor agents do not contain all aspects of the learning environment. Instead, we are working toward a vision where tutor agents can be easily embedded within existing workplace software tools or existing educational microworlds that are lacking in intelligent learning support. By leveraging the power of existing tools, we can reduce our development cost, and the resulting systems can provide better tools for the workplace and better learning environments for the school.

New technologies are changing the landscape of necessary skills in the workplace and academics (SCANS, 1991) making it increasingly important to aid students in acquiring software skills. The rising use of symbolic calculators, for example, is reducing the need for paper-based symbolic manipulation skills, just as numeric calculators have reduced, or even eliminated, the need for paper-based algorithms like finding the square root of a number. Mathematics curricula, in particular, are beginning to reflect the expectation that powerful tools (e.g., spreadsheets, graphs, and symbolic algebra tools) will be available to workers, so these curriculum reforms are focusing on support of reasoning skills, like expressing problems in algebraic form or writing computer programs (NCTM, 1989). In order to properly use computational tools, workers need to both construct appropriate tool inputs and then interpret outputs of the tools in light of their needs.

While many software tools provide on-line help, the focus is on lower-level specifics of tool use. Such tools do not provide learning support for the higher-order reasoning that is necessary to use the tool in real problem solving. Plug-in tutor agents can fill this role.

EXAMPLES OF COMBINING TUTORS AND EXISTING SOFTWARE TOOLS

As first steps toward building plug-in tutor agents, we have created two prototype learning environments that use an existing software tool as the interface to an intelligent tutor. We consider these initial attempts to be important in identifying the desired architecture and capabilities of tutor

agents. In addition, our experience in building these systems provided some indications of the practicality of building such agents using currently available technology. As a way of illustrating these issues, we provide a description of the design and use of our first prototype systems. Later, we step back to generalize their properties as a way of standardizing the construction and behavior of such systems. Finally, we present an initial specification of a plug-in tutoring agent and the communication protocols for interacting with other tools in a complete learning environment.

Both of the prototype systems to be described were cognitive tutors (Anderson et al., 1995) implemented using the Tutor Development Kit (Anderson & Pelletier, 1991). The Tutor Development Kit (TDK) is a Lisp-based system that includes interface facilities, a production-rule engine, and facilities to maintain a student model and perform knowledge tracing. Since these plug-in systems used other tools for their interfaces, we did not take advantage of the TDK's interface facilities in these systems.

A TOOL-TUTOR FOR GEOMETRIC CONSTRUCTION USING *GEOMETER'S SKETCHPAD*

Overview of the System

The *Geometer's Sketchpad* (Jackiw & Finzer, 1993) is a commercial software tool for creating geometric constructions and dynamically investigating them. It functions both as an educational microworld for exploring and discovering properties of geometric figures (cf. Schwartz, Yerushalmy, & Wilson, 1993) and also as a tool for mathematical research. In previous work, Koedinger and Anderson (1993b) proposed a learning environment that would integrate tutor-supported geometric conjecturing with existing tutor support for conjecture evaluation (Koedinger & Anderson, 1993a). The first step in geometric conjecturing is to construct figures to investigate. We built a prototype tutor for geometric construction that uses *Sketchpad* as the student's workplace.

Figure 1 shows the screen as it looks to a student in the middle of a problem. The problem statement appears as text in a *Sketchpad* file that is loaded when the student starts the problem (see the text "Construct a segment ...." just below the circle). This problem asks the student to draw a segment and then construct a second segment to equal to the first. The student has drawn the first segment (AB), drawn one end point of the second

segment (C), and taken the crucial step of constructing a circle whose radius is equal to the first segment. At this point the student asks for help and the tutor decides, based on the student's progress so far, what it would do next and gives an initial hint. As with other cognitive tutors, further help requests yield more specific hints.

From the user's point of view, the system looks almost identical to the *Geometer's Sketchpad* application. The only differences are that there is an additional menu labeled Tutor which students can use to start and quit the tutor, to request help, and to select the next problem. There is also a Messages window used by the tutor to provide hints and give feedback.

Implementation

Sketchpad has the ability to "demonstrate" a sketch to be drawn on one computer to any other computer in the classroom. When the teacher's program is demonstrating a construction, it sends out messages in the form of AppleEvents (Apple Computer, 1993) to the students' programs, telling them to duplicate the actions of the teacher. When used with our tutor, *Sketchpad* was run in "demonstration" mode, and the tutor intercepted messages intended for "student" machines. Since the demonstration messages provide information about what the student is doing, the tutor has the information it needs to interpret student actions.

In practice, the demonstration facility did not always provide information at the appropriate grain-size for tutoring, so we had to aggregate certain events into a single event which provided more appropriate information for the tutor. For example, the *Sketchpad* action of drawing a segment resulted in four AppleEvents: one for each of the two end points created, one for the connected line segment, and one to indicate the completion of the action. In response to this last AppleEvent, the three proceeding AppleEvents were aggregated into a single action appropriate for tutoring.

In some cases, the demonstration facility did not provide all the communication capabilities we needed, but the *Sketchpad* developers graciously made experimental modifications to the software tool to help us build this prototype. One desired modification was to inform the tutoring agent which object was selected at the time a hint is requested.

Another modification to *Sketchpad* was to include a Tutor menu that students can use to start the tutor, to request help, and to select the next problem. The *Sketchpad* developers added the ability to respond to a special AppleEvent that requests the addition of a menu and a list of menu

items. The result of the tutor (or any other application) sending this AppleEvent request was a new menu appearing in *Sketchpad*. When an item was selected from this menu, *Sketchpad* sent out an AppleEvent indicating which item was selected. The tutor received this message and was able to respond appropriately (by providing help, for example).

Sketchpad did not provide any way for the tutor to communicate back to the student. We worked around this problem by configuring the screen so that, when *Sketchpad* was the front-most application, a Messages window was visible in the background Lisp process, which was running the tutor. This proved to be a reasonable work-around, but its drawback was that users had to be instructed not to rearrange the windows on the screen (or else they might hide the Messages window). If users clicked on the Messages window (bringing Lisp to the front), they were instructed to return to the *Sketchpad* application.

Since we were unable to instruct *Sketchpad* to display graphic items in different forms (like highlighting them in bold), we were limited to text feedback to the user through the Messages window. One consequence of this was that we could not "flag" incorrect objects, as we have done in other systems (Anderson, Conrad, & Corbett, 1993). In response to user errors, we instructed the user (through the Message window) to use *Sketchpad's* Undo menu item to return to a correct solution path.

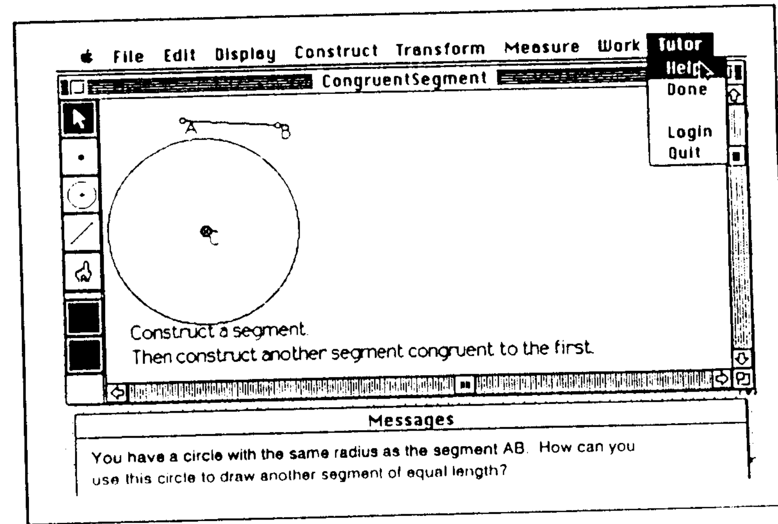

Figure 1. *Geometer's Sketchpad* and a tutoring agent for geometric construction

A TOOL-TUTOR FOR ALGEBRA PROBLEM SOLVING USING *MICROSOFT EXCEL*

Overview of the System

The *Excel* Algebra Problem Solving Tutor is a variation of a learning environment that we have been using to teach word problems in ninth-grade algebra (Koedinger, Anderson, Hadley, & Mark, 1995). In this system, students are given a word problem such as: "A sitka spruce in the Arctic Circle grows only 0.3 centimeters per year. How many centimeters would this tree grow in a day? How long would it take for this tree to grow 1 meter?" Students answer these questions by completing a table that contains two columns (one for the time the tree grows and another for its height) and two rows (one for growing a day and one for reaching 1 meter). In addition to completing the specific questions, students are asked to come up with an algebraic expression for the height, to indicate the unit of measure for each of the variables, and to create a graph of the line that relates height to time.

The original version of the learning environment used a tool that we created specifically for this task, but it is clear that the graphing and table functions are similar to those that we might find in a spreadsheet. In addition, there is a clear mapping between the algebraic expression for height and (at least one version of) the *Excel* formula used to calculate cells in the height column. For this reason, we defined a system which uses *Excel 4.0* for the Macintosh in place of our own spreadsheet window.

From the user's perspective the system looks much like *Excel*. There is an additional Tutor menu from which the user can ask for help or exit the tutor. When the system starts up, an *Excel* worksheet is opened containing the problem statement. On a separate worksheet, borders are drawn around cells to create a table similar to the one they would see in our own table tool.

Users have access to all of *Excel's* capabilities, although the features needed to perform their task are quite limited. They fill in the table by typing the appropriate headings, numbers, and algebraic expressions. In the "height" column, users may enter an *Excel* formula to calculate the height, given the time the tree grows. When the tutor needs to display a message to the user, an *Excel* dialog appears.

Implementation

We used *Excel 4.0* for the Macintosh, which does not normally provide any descriptions of user actions. In order to provide the tutor with information about user actions, we took advantage of *Excel's* customization facilities. We constructed a front-end to *Excel* (called "pseudo-*Excel*") that looked just like the usual version and duplicated all of the usual functions. From the user's perspective, pseudo-*Excel* was identical to regular *Excel*. In pseudo-*Excel*, however, any menu choice or cell entry caused an AppleEvent describing the user action to be sent to the tutor. This is essentially the same approach used by Cypher (1993) to implement Eager (using the *Resolve* spreadsheet) and by Fox, Grunst, and Quast (1994) for their *ExcelInExcel*. We also used *Excel's* customization facilities to create a Tutor menu that contained a Help item. Choosing an item from that menu created an AppleEvent similar to that generated for any other action in pseudo-*Excel*.

Since a cognitive model and tutor for algebra problem solving already existed (in Lisp, using the TDK described earlier), conceptually all that needed to be done was to unplug our home-grown table tool and plug in *Excel* instead. Most of the implementation work, then, involved creating a module that could communicate between *Excel* and the existing tutoring agent. The expert system component of the TDK expects user actions to be described as a "selection-action-input" triple. This description corresponds closely to the AppleEvent description of a user action (with the AppleEvent itself corresponding to the action, the direct object corresponding to the selection, and some other parameter representing the input). *Excel* and the tutoring agent use different languages to refer to these elements, however, so it was necessary to translate from one language to another. In our case, the translator changed *Excel's* references into data structures used by the tutor. For example, "Cell R2C3 in Worksheet 1" (an *Excel* description of a cell) was translated into an identifier, "Cell23," for the working memory element representing that cell in the tutor's production system memory.

Communication between the tutor and *Excel* was straightforward. *Excel* is fully scriptable through AppleScript, and we used this ability to provide feedback to the user. For example, if the user entered a value into the table which the tutoring agent determined to be incorrect, it sent AppleScripts instructing *Excel* to change the font in that cell to outline form. In cases where the user performed some action that was disruptive to the solution (such as sorting the table), we instructed *Excel* to use its "undo" capability to restore the earlier state. When we wanted to present an error message,

the tutoring agent passed the message to the translator, which generated AppleScript code telling *Excel* to display a dialog containing the message. In response to any user action, the tutoring agent informed the translator of any items that should be highlighted or unhighlighted, any messages to dis- play, and whether the action should be undone. The translator expressed these requests in AppleScript, which was then used to modify the *Excel* interface.

BASIC ELEMENTS OF A LEARNING ENVIRONMENT

These prototypes led to the understanding of three elements of a tool that need to be considered in order to implement a tutoring agent architec- ture. First, we need to be able to monitor user actions. Second, we need to be able to give feedback, whether through text, images, or modifications to the tool's display. Finally, we need to be able to make some parts of the tu- toring agent visible to the user. This last feature allows us to augment the tool with a Help menu, for example.

In both the *Sketchpad* and *Excel* tutors, we were able to monitor user actions through AppleEvents. In the case of *Sketchpad*, the AppleEvents were, for the most part, provided by the application itself, although some modification by the application designers was required. In *Excel*, all of the AppleEvent abilities had to be added to the program using its built-in cus- tomization facilities.

In *Sketchpad*, we were unable to give feedback to the user through the tool itself, so we were limited to presenting messages in a window that was outside of *Sketchpad*. *Excel*, in contrast, was fully scriptable, allowing us to display text in a dialog or change the display and contents of any screen elements. In essence, the *Excel* tutor was allowed to do anything in *Excel* that a user could do using a mouse and keyboard.

The *Geometer's Sketchpad* did not ordinarily allow augmentation of the tool so that we could add a Tutor menu to it, but we were able to ac- quire a special version that allowed us to do so. *Excel* provided this ability as part of its normal operation.

In addition to these elements of the tool, our learning environments need to track student performance across problems. Since the TDK provid- ed integrated student modeling and knowledge tracing, we were able to take advantage of those aspects of the system without any changes from our practice of writing TDK-only tutors. Still, a general plug-in tutoring archi- tecture needs to consider these functions.

In general, a full tutoring system will consist of four parts: a tool, a tu- toring agent, a curriculum manager, and a translator. The tool is the part of the system that the user sees and operates. The tutoring agent is the part of the system that contains the domain knowledge necessary to provide help to a student. The curriculum manager is responsible for maintaining the student model and for guiding the student through the elements of the curriculum. The translator is responsible for ensuring that the parts of the system can communicate with each other. The following sections describe each of these objects in more detail.

Tool

A tool is a piece of hardware or software that can be used to perform some work in a particular domain. There are several issues involved in choos- ing (or designing) an appropriate tool for an educational system. One consid- eration is that the tool be neither too weak nor too strong for the student's lev- el of knowledge. Although a symbolic algebra tool like *Mathematica* might be appropriate for calculus students, it is probably too strong a tool to be used with beginning algebra students. Among other things, it automatically simplifies algebraic expressions and presents them in a canonical format. These changes in the surface appearance of expressions might be confusing to beginning students.

Another consideration is that the educational task, to the greatest ex- tent possible, be performed with the tool. In part, this is a good educational practice because it reifies abstract thought in specific activities with the tool (Koedinger & Anderson, 1993c). For our purposes, though, this factor is important because any planning or other activity that takes place outside of the tool cannot be monitored by the tutoring agent.

The most important restriction on our solution to implementing plug- in tutor agents is that we cannot expect tool authors to modify the function or implementation of a tool in order to allow a tutoring agent to be used with it. This follows for two reasons. First, we want to be able to work with software that exists right now. Second, tool authors cannot and should not be expected to fully anticipate the type or function of tutor agents that might be used with their software. This allows us maximum flexibility in designing tutor agents to perform different tasks. Our approach, then, is to require tools to provide information about what the user is doing in the tool. To the extent possible, we want that information to be neutral about the use to which that information will be put.

In direct conflict with this principle is the fact that information must be provided at some level of detail, and the choice of the appropriate level does affect the use to which that information can be put. Consider a case where a *Microsoft Excel* user pushes the mouse button at screen location (100,200) and then moves the mouse and lets go of the button at screen location (148,293). Suppose also that *Excel* responds to this action by moving the value of cell R4C5 to cell R8C7. A low-level description of this action would describe it in three (or even more) steps: The user pushed the mouse button at (100,200); the user dragged the mouse to (148,293); and the user let go of the mouse button. This level of information can be provided by the operating system.

However, since we want to monitor the user's actions with respect to some high-level goal, we need to receive information about the semantics of the user's action, rather than the details of the manipulation of the interface. In order to interpret this user's action at the semantic level, the tutor would have to know that (100,200) corresponds to the selection border (with the exception of the lower-right corner, which is used for extending the selection) of cell R4C4 and that (148,293) corresponds to some point within cell R8C7. To perform this translation, we would need complete information about what interface actions result in a change of selection, the size of the selection border, the size of the lower-right corner of the selection border, whether the "allow cell drag-and-drop" preference was set, and so forth. Clearly, this translation is a difficult task, and the details of the translation are likely to change between different versions of the software.

A more appropriate description of the same event for tutoring purposes would refer to the action in terms of the work accomplished, not the user interface actions that were performed. In this case, we might describe the user as having "Moved the value of cell R4C4 to cell R8C7." This description differs from the user-interface description in several ways. It combines three discrete user-interface actions into a single event. In addition, it refers to application-specific objects (like "cell R4C4") instead of user-interface elements. The use of application-specific objects suggests that this kind of description can only be provided by the application, not the operating system.

Fortunately, the need for this kind of semantic description has long been recognized (Gates, 1987) and is becoming standard in newly developed applications. In fact, the kind of descriptions that a tutoring agent requires are also required by OLE and OpenDoc in order to support scripting. AppleScript and OLE Automation are typically used to describe actions at this level of detail. To support macro recording, OpenDoc requires

that applications, on request, describe all user actions in such terms. OLE 2.0 does not support recordability, but future versions are expected to incorporate this feature. A "recordable" application which does so would be a prime candidate for the tool component of a learning environment. The existence of recordable applications allows us to achieve the goal of "add-on" systems described by the Eurohelp group (Breuker, 1990).

While guidelines for recordable applications allow some flexibility in the way that actions are described (Apple Computer, 1993; Olsen & Dance, 1988), the goals of providing reasonable descriptions for macro recorders and of providing reasonable descriptions for tutor agents are, for the most part, quite compatible. We can expect that, as plug-in tutor agents become more prevalent, application designers will take their existence into account in constructing semantic descriptions, reducing the number of cases where semantic events developed for macro recording are inappropriate for constructing tutor agents.

We need not commit to a particular component architecture (e.g., OpenDoc or OLE) or a particular communication protocol (e.g., AppleEvents or OLE Automation) as long as we are assured that the information is at the appropriate level of detail. Typically, this description takes the form of a predicate describing the action, an argument describing the object being acted upon, and any other parameters that are necessary to understand the action. In most cases, the predicate will have three places describing the subject, verb, and a direct object. For example, the action of dragging a cell in *Excel* might be described as [Subject="cell R4C4"; verb="move";move-location="cell R8C7"].

Another consideration for a tool is that we be able to programmatically manipulate the tool's content. This is important for giving feedback. For example, it would allow us to display a cell containing an erroneous value in red or to highlight a cell as a way of drawing the user's attention to it. If we wish to disallow some actions in the tool or to immediately remove the user's erroneous steps, the tool must support programmatic access to an undo command.

In domains where the problems to be solved take a significant amount of time, it is necessary to be able to tell the tool to restore some previous state, so that the user may quit the learning environment without losing work. If we have control over the tool's objects, we can easily restore the state of a partially completed problem by manipulating the tool's content to reflect that state.

In practice, we have found this requirement much easier to satisfy than the requirement that the tool communicate appropriate information to

the tutoring agent. Many tools are now controllable through some scripting language (like AppleScript) and, in cases where that language is inadequate, there are usually acceptable alternatives to communicating with the tool.

Although it is convenient to display messages through the tool, we have found this to be a less important consideration, since there are often alternate ways of displaying text messages. For example, under a component architecture like OpenDoc, we can always embed the tool in a container which provides a message window.

Tutor Agents

Tutor agents are pieces of software that monitor users' actions. The purpose of the monitoring may be to determine whether the user is performing a task correctly or to provide advice (either as a response to a user's request or due to recognition of some appropriate opportunity). Note that the goals of recognizing errors and giving advice are independent. In this broad definition, tutor agents may include context-sensitive help systems (which give advice but do not recognize errors) and compilers (which typically point out errors but do not give advice). We believe that plug-in tutor agents could be developed to address many of the kinds of help systems described in the Eurohelp project (Breuker, 1990).

Although the tutor agents in the systems we have built are implemented as model-tracing expert systems, our proposal does not require a commitment to tutor agents of this form. All it requires is that the tutoring agent be able to evaluate and respond to user actions in some manner. The tutoring agent needs to be able to accept information about user activity at the grain-size discussed earlier, but some tutor agents may choose to interpret actions at a higher grain-size. Similarly, some tutor agents may give immediate feedback and require users to correct errors before continuing, while others might remain silent until explicitly asked for help by the user.

For the most part, the tutoring agent has no visible user interface, so users of a tool are not aware of whether a plug-in tutoring agent is operating. However, there are a few elements of the tutoring agent that should be visible to the user. For example, we may want to present a Help menu that directs the tutoring agent to provide help. Similarly, we may want to provide menu items that the user can access to ask for a new problem or indicate completion of the current problem. In some implementations, the tutoring agent may also be visible through a window used for presenting messages.

In highly adaptable tools (like *Microsoft Excel*), it is possible to provide an interface to the tutoring agent through the tool itself. Component architectures (like OpenDoc and OLE) allow the ability to embed such controls in existing tools, accomplishing the same goal. This tends to blur the distinction between the tool and the part of the tutoring agent visible through the tool. Throughout this paper, we will refer to elements of the tutoring agent visible to the user as part of the "tutoring agent interface," but the reader should be aware that, from the user's perspective, these elements may appear to be part of the tool. Also, it is important to remember that any user actions in the tutoring agent interface result in messages being sent to the translator, not the tutoring agent (although the translator will typically route the message to the tutoring agent). This implementation allows a consistent manner of handling all user actions.

Curriculum Manager

The curriculum manager is responsible for deciding which problem to present to the student. This decision can be as simple as picking the next problem on a pre-set list or can be as complex as picking a task based on how well its features match a profile of user skill accomplishments. The latter is achieved by the knowledge-tracing mechanism in traditional cognitive tutors (Anderson et. al, 1995). In a lighter weight, though less pedagogically supportive environment, the curriculum manager might be replaced by a problem entry dialog where the student selects or constructs the problem. Such dialogs have been implemented in cognitive tutors for algebra equation solving (Ritter & Anderson, 1995) and geometry theorem proving (Koedinger & Anderson, 1993a).

Translator

The translator handles all communication between the tool and the tutoring agent. When necessary, it translates information from the language of the tool into that of the tutoring agent (and vice versa). If more than one tool or tutoring agent exists, the translator is responsible for routing messages to the correct component. Since the translator needs to know about the communication requirements of the tool and the tutoring agent, it needs to be customized for each learning environment.

The advantage of this design is that the tool and the tutoring agent can be fully designed without knowledge of the details of the other. The ability

to change the tutoring agent independent of the tool is crucial in domains in which we don't develop the tool ourselves. In many domains, it makes sense to use commonly available software tools. For example, we might use either *Mathematica* or *Maple* as tools for solving calculus problems, and we might use either *Excel* or *Lotus 1-2-3* as tools for a business learning environment. Different applications within a class typically have different interfaces, but they share actions at the level of the semantic event. To the degree that two tools share semantic events, a particular tutoring agent will work equally well with either of them.

The existence of a translator also allows us to be more flexible in the implementation of feedback. Since the tutoring agent describes feedback in terms of semantic events, the specific method of presenting feedback can be customized for different audiences, for different system configurations, and for different tools using the same tutoring agent. For example, the semantic command "Flag cell R4C4" could be translated to *BorderAround Cell "R4C4" ColorIndex 3* (which, in *Excel*, draws a red border around the cell) if the user had a color monitor and *set OutlineFont of Font of Cell "R4C4" to true* (which displays the contents in outline font) if the user had a monochrome monitor.

In the future, we would expect the translator to include natural language translation facilities as well. This would allow us to present text feedback in the user's native language. With current technology, we can direct the translator to display a certain numbered message, and the translator can select the text representation of that message based on the user's language.

Yet another advantage of the translator design is that it allows multiple tools and multiple tutors to be operating in a single system. That is, it is possible to have multiple tools sending messages to the translator. The translator may then relay those messages to a single or to multiple tutors (depending on the tutoring agent design). It is also possible through this architecture for a single tool to be supported by multiple tutors. Ritter and Anderson (1995) describe an equation-solving tutor which provides step-by-step feedback. With this architecture, such a system could be supplemented by an "issue-based" tutor (Burton & Brown, 1982) which identifies solution patterns across problems. In this case, the translator would send a copy of the message to each of the relevant tutors. There are many issues still to be resolved about how the translator could mediate disagreements between the tutors (both over the correctness of an action and over access to message windows and flagged items).

TOWARD STANDARDIZATION

Based on our experiences in building the *Sketchpad* and *Excel* tutors, we are able to take some steps toward standardizing the components of a learning environment. Figure 2 illustrates the basic interaction within the learning environment. The architecture consists of the four objects previously described. The rest of this section describes the specific routines used to implement these objects.

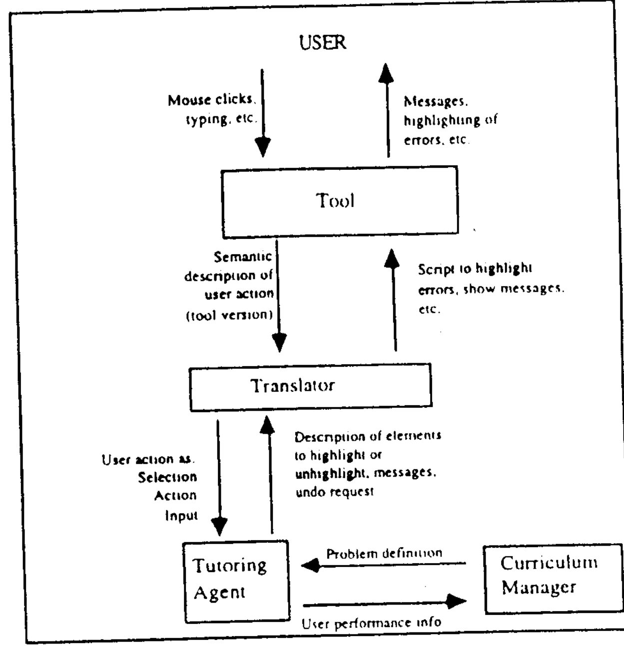

Figure 2. General architecture for tutor agents

Tool

In keeping with our desire to work with tools without imposing a particular architecture on them, we do not define any particular properties for the tool other than the general properties described previously. The tool is required to emit semantic events corresponding to all user actions. These events should be at the intermediate grain-size discussed earlier. We do not commit to any particular format for these events, but our systems have used AppleEvents for this purpose.

Translator

The translator object needs to intercept all actions taken by the user when using the tool. Typically, it translates these actions into selection-action-input triples and passes the information to the tutoring agent. Although the translation of the incoming semantic event into selection, action, and input depends on the particular communication protocols for the tool and the tutoring agent, we can define a standard variant of the translator which translates an AppleEvent into a selection-action-input form. For example, in response to a SetData AppleEvent, this method would set the *selection* to the value of the keyDirectObject parameter, the *action* to "SetData," and the *input* to the value of the keyAEData parameter. Similar standard translations can be defined for all AppleEvents.

The translator also needs to intercept messages from the tutoring agent to the user. This allows the tutoring agent to describe feedback as semantic events, which the translator can implement in a manner consistent with the tool's capabilities. For example, some tools display messages in a fixed window, while others may use a dialog which appears with the message (as did the *Excel* tutor) and others may use no message display at all (as with the *Sketchpad* tutor). If the tool does not support message display, the translator must be able to display messages in some other fashion. One possibility is for the translator to display its own window; another is for it to ask the tutoring agent to display a window (as the *Sketchpad* tutor did).

The implementation of highlighting and unhighlighting elements clearly depends on the tool's ability to display elements in different ways. We allow complete flexibility in this implementation. The *Excel* tutor displayed text in a different style, although it could just as easily put a border around an errant cell. The *Sketchpad* tutor did not highlight elements at all.

Tutoring Agent

Messages received by the tutoring agent. The tutoring agent needs to respond to six messages: *start-problem, process-tool-action, process-help, process-done, get-next-step,* and *reproduce-tool-state*.

Start problem. The start-problem message takes two parameters, corresponding to the initial and goal state of the problem. This message is typically sent by the curriculum manager. Systems which allow users to choose a problem may send this message using the tutoring agent interface. Systems

which do not require users to specify their goals may not use this message at all.

Process tool action. The process-tool-action message takes three parameters, corresponding to the user's selection, action, and input. These parameters correspond closely to the standard semantic event description as subject, verb, and direct object. A fourth parameter is a unique action ID, which the tutoring agent can use to refer to this action.

In response to the process-tool-action message, the tutoring agent should update its representation of the problem state and decide how to (and whether to) respond to the user's action. If the tutoring agent decides to respond, it sends one or more of the following messages to the translator: *flag, unflag, point-to, send-message, undo, select, verify, update-assessment,* and *start-activity*. These events are described below.

Process help request. The process-help-request message is typically initiated by the user, by pressing a Help button or choosing a Help menu item made visible through the tutoring agent interface. The translator routes this message to the tutoring agent. The process-help-request message takes three parameters: the user's selection, input, and a unique ID. The selection is provided so that systems can be sensitive to the user's current focus of attention. The user's input will contain the topic in systems which allow users to ask for help on specific topics. The input parameter can also be used to support systems which provide different ways of asking for help (e.g., the input could be "why" in one case and "how" in another). Typically, the system will respond to a process-help-request message with *send-message*. In some cases, this will be accompanied by point-to, select, or start-activity messages.

Process done. The process-done message takes no parameters. This message is typically initiated by the user, by pressing a Done button or choosing a Done menu item made visible through the tutoring agent interface. The translator routes this message to the tutoring agent. In systems where assessment is updated upon completion of a problem (as opposed to after each action), the tutor will respond to this message by sending update-assessment. Other systems may use send-message to send a congratulatory message. In some learning environments, the tutoring agent may detect completion of a problem and automatically move to the next one (without user input). In such systems, the tutoring agent interface need not support a Done button (or menu item), so the process-done message is not used.

Get next step. In some systems, it is desirable for the user to be able to ask the system to perform some action, either instead of or in addition to providing help on an action to perform. In previous implementations of our

tutoring systems (Koedinger & Anderson, 1993c), for example, we allowed users to ask the system to "do this step for me." The get-next-step message results from such a request. It takes the selection as a parameter (so that the user's current focus of attention can be a factor in choosing the next step).

Get-next-step is initiated by the user through the tutoring agent interface and sent to the translator. The translator sends the message to the tutoring agent and waits for a reply. The tutoring agent's reply should contain (up to) three parameters: the selection, action, and input for the next step. The translator then changes these parameters into a scripting instruction for the tool. Clearly, this message can only be implemented by a tutoring agent which is able to provide a correct user action. The tutoring agent can return two error codes: kNoAction (the equation solver knows of no correct action) and kDone (the problem has been completed).

Reproduce tool state. This message is a request to the tutoring agent to reproduce a state of the tool and is typically initiated by the user through the tutoring agent interface. It takes one parameter, which is either a description of the state to reproduce (e.g., a statement of the form "student A's state on problem B") or the constant kCurrentState. The tutoring agent should respond with a sequence of *perform-user-action* messages. The primary purpose of this message is to recover from fatal errors in the tool. If the tool and tutoring agent are running in different processes and the tool's process crashes, the user should have access to a control which generates a reproduce-tool-state message. This would allow the user to recover lost work. In some systems, it may be possible for the tutoring agent or translator to detect an error situation and generate this event without intervention by the user.

Messages sent by the tutoring agent. The tutoring agent has 10 commands it can use to communicate back to the user in response to either a help request or a user action. These correspond to 11 messages: *verify*, *flag*, *unflag*, *point-to*, *undo*, *send-message*, *select*, *update-assessment*, *start-activity*, *perform-user-action*, and *get-user-value*.

Verify. The verify message is used to indicate that the tutoring component has received, understood, and responded to the user's action. If the tutoring component intends to send more than one message in response to a user action (or request for help), the verify message is the last message sent.

This message exists for several reasons. First, it can be used in systems that require strict synchronization between user actions and tutoring component evaluation. In such systems, it is desirable to prevent the user from initiating an action until the tutor has fully responded to the previous action (this is only an issue if the speed of the tutoring agent is slow relative to the speed of the user). In such systems, the translator may lock the tool after each user action (using tool-appropriate scripting) and unlock the tool upon receiving a verify event from the tutoring agent.

In some systems, the verify event is used to indicate to the user that the tutor has completed its evaluation of an action, even if such completion is not required before the user can continue. In a revised version of the *Excel* tutor (see Figure 3), cell values were displayed in blue but were changed to black upon verification by the tutor.

The verify event can also be used to implement corrections. For example, it may be desirable to correct spelling errors for the student or to always round off decimals to a fixed number of digits. The verify event can provide the corrected spelling (or rounded-off number).

Finally, in systems with multiple tutor agents, the verify event provides a way for the translator to ensure that it has received input about a particular user action from every relevant tutoring agent before it relays that feedback to the user. Systems which require none of these features may not use the verify event at all.

The verify message takes three parameters: the ID of the event being verified (as passed in the process-tool-action and process-help-message messages), a Boolean indicating whether the action being verified was considered correct or incorrect, and a parameter indicating a value to use to replace the user's input (for corrections). The latter two arguments are optional.

Flag. The flag message is a request to flag a particular item or set of items in the tool. Flagging involves some visible and persistent change in the display of the item. For text items, this may involve a change in font. For other items, this may involve drawing a border around the item or displaying the item in a different color. The particular implementation of flagging is up to the translator. The flag message takes one parameter, which is a description of the item to flag.

Unflag. The unflag message is a request to return a particular item or set of items in the display to their unflagged state. This would be used in response to the user's correction of a flagged item. The unflag message takes one parameter, which is a description of the item to unflag.

Point-to. The point-to message is a request to highlight or otherwise point out a particular item or set of items in the display. The distinction between pointing and flagging is that pointing is a temporary event used to draw the user's attention to something. Flagged items will become unflagged only when the user corrects an error. Pointed-to items can return to their normal state once the user has noticed them.

It is up to the translator to decide how to implement the pointing. That is, the translator decides whether pointing to a text object results in setting the text font to outline, circling it, or drawing an arrow pointing to it. Similarly, the duration of pointing is up to the translator. Conceptually, the highlighting of an object should last until the user changes focus. Typically, this means that the object stays highlighted until the user changes the current window or the selection within the window, but the particular actions that define a change of focus may depend on the capability of the tool or display device. In some cases (like when "pointing to" involves blinking the item three times), the pointing may stop after a fixed period of time rather than any action by the user. In some cases (like when AppleGuide is used to circle an object), pointing will end without any action by the translator. In others, the translator might have to determine the change of focus (by interpreting a message sent from the tool) and send a message back to the tool to end the highlighting. The point-to message takes one parameter, which is a description of the item to highlight.

Undo. The undo message is a request to undo the user's action (that is, to return the user tool to a previous state). This message is used in systems (or portions of systems) which do not allow the user to continue after an error.

The undo message takes two parameters: the action ID (as passed in the process-tool-action message) and a Boolean indicating whether subsequent actions (if any) should also be undone. The second parameter is optional. Depending on the tool and the cleverness of the translator, undoing subsequent actions may be required.

Send-message. The send-message message is a request to present some text to the user. The text can be specified as a single string or as a list of strings. If a list is given, the translator should display the messages in the list in order, at the request of the user. That is, the user should see the first message and be provided with a More button that allows the user to see subsequent messages (a Previous Message button is also desirable). The us^r may chose not to see all the messages in the sequence. We typically use a series of messages in this fashion to give help. The initial help message is a high-level goal, and subsequent messages give more specific information. The message text may be a "styled" string (or list of styled strings), which includes information about typeface.

In addition to the text itself, this message can include three parameters: pointers, presentation format, and message place. The pointers argument provides a way to implement a point-to mechanism that is synchronized with message presentation. This takes the form of a list equal in length to the number of text messages. Each element of this list can be null (indicating that no pointer is to accompany this message) or a list of objects that are to be pointed-to when the user is viewing the corresponding part of the message.

The presentation-format parameter can be used to specify a preferred way of displaying the message. Its value can be either kUseDialog, kMessageArea, kUrgent, or kBestWay. kUseDialog specifies that the message should be displayed in a movable modal dialog. kMessageArea specifies that the message should be displayed in a message area (such as a space at the bottom of a window or a window dedicated to displaying messages). kUrgent specifies that the message should be displayed in a modal dialog and accompanied by a beep. If the operating system supports it, the translator should try to present kUrgent messages to the user, even if the current application is not part of the learning environment. kUrgent messages are intended for messages that indicate serious problems, such as system errors. They should not be used for typical feedback. kBestWay specifies that the translator (or display device) should decide how best to display the message. Depending on the capabilities of the tool, the translator may decide to ignore this parameter.

The message-place parameter gives further information when kMessageArea is used as the presentation-format. This parameter specifies the application and/or the window within that application in which to display the message. For example, it might specify that the message should display in the "graph window of the grapher." If the presentation format is kMessageArea and this parameter is omitted, it is up to the translator to select an appropriate message area.

As with pointing, messages shown with kMessageArea should disappear with a change of context. Messages shown with a dialog will be dismissed by the user. The detection of the change of context is subject to the same issues discussed with regard to the point-to message.

Select. The select message is used to change the focus of attention in the tool. For most tools, this is interpreted as the place where keyboard input would go. This message should be used sparingly, since it can be disconcerting to users. In most cases, the proper response to a user who has asked for help with an invalid selection is to point to an appropriate selection (through either the send-message or point-to messages), rather than to select it. The select message is a more appropriate response to a direct query of the form "Where do I go next?" This message takes a single argument: the object to select.

Update assessment. The update assessment message is used to indicate that the tutoring component wants to change its assessment of the student's progress. In some systems, progress is assessed after every user action; in

others, this update occurs only at the end of a problem (or at the end of a series of problems). The message takes three arguments: the measure to update, the amount to change it, and the direction of the change. In our systems, the measure to update is given as a string specifying the skill to update. The amount to change and direction of change parameters are optional. Systems which always increment or decrement by a fixed amount can omit the amount to change parameter. Systems which can provide either a positive or negative amount to change may omit the direction parameter. The translator forwards this message to the curriculum manager.

Start activity. The start-activity message is a request to start some activity for the user. This is something of a catch-all, which allows the tutoring agent to initiate any activity. It is intended for instructional activities that can take place without the intervention or knowledge of the tutoring agent. Activities might include showing a picture or movie, pulling up a calculator, and so forth. This message takes one parameter: a description of the activity to start.

Perform user action. The perform-user-action message provides a way for the tutoring agent to affect the tool in the same way that the user does. This message takes three parameters: the selection, action and input. The translator must convert these parameters into a script suitable for controlling the tool. This message is typically sent in response to a reproduce-tool-state message, but it can also be used in systems in which the user can ask for a demonstration of a step or a set of steps.

Get user value. The get-user-value message is used in cases where the tutoring agent needs more information before it can determine the correctness of a user action. This might be the case if the tool being tutored is not fully emulated by the tutoring agent. For example, a tutoring agent for a spreadsheet tool need not be fully able to calculate the values of formulas in the spreadsheet's formula language. If the user typed "=SQRT(R3C6)" in cell R8C8 and then entered 16 into cell R3C6, the tutoring agent may not be able to determine that the value of cell R8C8 is now 4. The tutoring agent could use the get-user-value message to determine the value of cell R8C8. The get-user-value message takes three arguments: the object, the property that the tutoring agent wants to query, and a code indicating the data type that the tutoring agent wishes to receive.

Curriculum Manager

The curriculum manager accepts the update-assessment method. We leave the specifics of the description of the measure to update to the particular implementation, since this will depend on how the tutoring agent measures performance. In our tutors, we would send a list of skills and, for each skill, the tutoring agent's estimate of the user's progress on that skill. The curriculum manager would then choose a problem which emphasized the student's weakest skills. Other systems might simply take a count of the errors made by the student. In systems which externalize the student model (Paiva, Self, & Hartley, 1995), the curriculum manager is responsible for storing information in an appropriate format. The use of a single update-assessment method to maintain the student model assumes a fairly simple representation. Systems with more extensive student models may need to add additional methods.

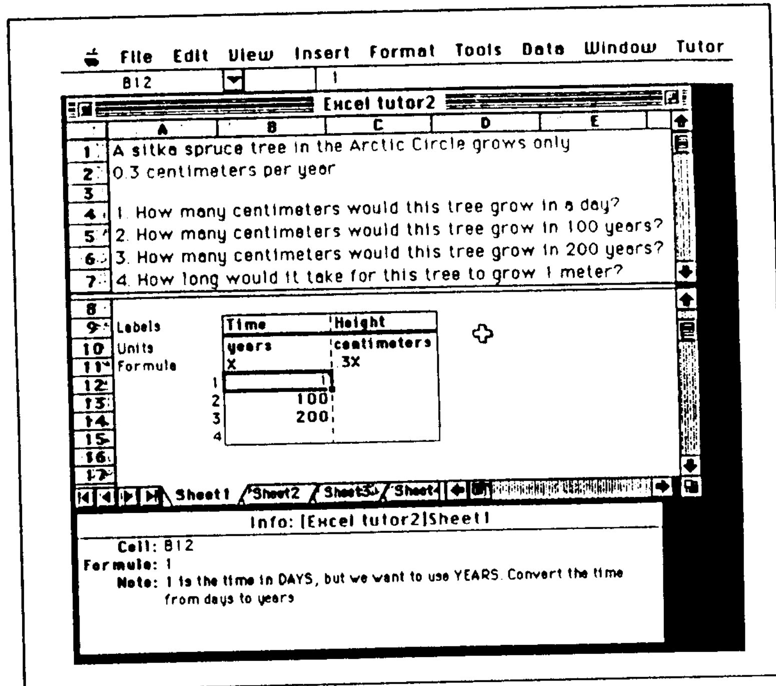

Figure 3. A view of the revised version of the *Excel Tutor* (using *Excel 5.0*). Note that, on the user's color monitor, the number in cell B12 will be displayed in red, and *Excel* displays a red square in the upper right corner of that cell, indicating that there is a note attached to it.

OTHER ISSUES

We have developed several prototype systems based on (incomplete) implementations of this architecture. Through this work, we have identified several issues which test the limits of the architecture.

Side Effects

Monitoring the user's actions is not the same as monitoring the tool's activity. For example, in a spreadsheet, when the user types into one cell, the tool might respond by updating values in 50 cells. However, the translator (and thus the tutoring agent) will only record one event: the user typing into a single cell. From the tutor's point of view, the fact that 50 cells changed is just a side effect of the user typing into the one cell.

For most tutor agents, this is not a problem at all. The tutoring agent will have recorded events for each of the 50 formulas the user had earlier typed in. If, at that time, the tutoring agent verified that the formulas were entered correctly and if the user's current entry is also correct, then the agent may not need to check (or even know) the value of each of the 50 cells.

Even so, a sophisticated agent could, in principle, determine which cells the spreadsheet would need to update on the user's current action. If the agent has enough knowledge about the spreadsheet's formula language, it could calculate the values put into each of the 50 cells. Alternatively, the agent could use get-user-value to determine the values of the 50 cells. In a spreadsheet that supports such a property, the tutoring agent may be able to use get-user-value to ask the tool for the dependents of the current cell, so that it does not have to determine which cells changed in response to the user's action.

On some occasions, the data maintained by the tool may be different from the data required by the tutoring agent. In such cases, the tool considers the information relevant to the tutoring agent to be a side effect of the data maintained by the tool. Consider a drawing tool like the *Geometer's Sketchpad* in which the user wants to construct an angle of a certain measure. The user joins two line segments at a single "pivot" point. The user then drags the end of one of the line segments to change the angle between them. The tool might reasonably describe this activity as: [selection = point1, action = "Move," input = 50,50], representing the activity of moving the end point of the line segment. However, the tutoring agent is concerned with the angle this action formed between the two line segments.

If the tool supports a "measure angle" property, the tutoring agent might be able to use get-user-value to determine the angle formed by the user's action. If not, then there is a slight mismatch between the abilities of the tool and the desires of the tutoring agent. In this case, we would have to work around the problem by having either the tutoring agent or the translator calculate the angle based on the information it has about the position of the three points defining the line segments. The decision about whether the tutoring agent or translator is responsible for this calculation affects the generality of these components. If the translation is done in the translator, we could more easily adapt the tutoring agent to a tool which did support a measure angle property.

Grain-Size

There are some situations where user actions at the level of semantic events are not sufficient. If we were concerned with the user's typographic errors or the specific way that the user accomplished a task (by a menu or through a control key, for example), we would need access to user-interface events.

It is fairly easy to see how the architecture could be augmented to support this goal. In an operating-system-dependent manner, the translator could monitor user-interface events and send them to the tutoring component along with (or instead of) the semantic events. The method of describing these events would differ from that used for semantic events, of course, and it would be appropriate to define a separate message to be used for sending events of this kind. Some systems might require that user-interface and semantic events be coded so that the tutoring agent can determine which user-interface events go with which semantic events. The KATIE system (Kosbie & Myers, 1993) integrates semantic and user-interface events in this way.

Synchronization

Since the tool operates unhindered by the tutoring agent, there is potentially a problem of synchronization between the tool and tutoring agent. Specifically, a tutoring agent which is slow relative to the speed of the user of the tool may allow the user to perform multiple actions before evaluating the first one. This can cause two problems. First, it can be difficult to give

feedback to the user on an action that happened several steps ago. It is usually unacceptable to interrupt the user with a dialog referring back to previous steps. Second, in some tools, it can be difficult for the user to correct actions that have been performed on the assumption that previous actions have been correct. This is especially the case for tools which do not provide a large undo stack. A similar problem arises in tools which may allow the user to put the system in a logically impossible state. This was the case in our equation-solving system, where an error in arithmetic could create an equation that was not consistent with the one being solved. Although the system could let the user continue at this point, the user could never reach a correct answer (except by a fortuitous second error). This is a case where it is probably best to correct the error before continuing.

The feedback problem can largely be solved through good interface design. In the original version of the *Microsoft Excel* tutor, if the user entered an incorrect value, the tutor immediately displayed a dialog with an error message. In a revised version of the tutor (see Figure 3), the message was attached to the errant cell in the form of a "note." In *Excel*, cells with attached notes appear with a red square in the corner. The note is visible in an accompanying window when the cell is selected. Help messages (that is, those messages that result from a user's request for help) were still displayed in a dialog. If the message to be displayed resulted from a mistaken user action, the tutoring agent sent the send-message message with the *presentation format* parameter set to kMessageArea; when the message resulted from a request for help, it used kUseDialog.

The second synchronization problem arises in cases where we do not want the user to continue using the tool until an error has been corrected. In this case, we want the tool to act as if the tutoring agent is an integral component of the tool, rather than a more passive commentator.

The proposed architecture provides some mechanisms for supporting this mode of interaction, but the specific solution will depend on the capabilities of the tool. The basic technique is for the translator to prevent the user from acting (or accomplishing any work) between the time it receives a semantic event and the time it receives a verify message from the tutoring component for that event. One way to accomplish this is for the translator to make the user's document read-only when it receives a semantic event and to return the user's document to read-write after receiving the verify message. Another technique would be to have the translator send undo events to the tool for any semantic events that arrive between the time the original semantic event arrives and the verify message is received.

Clearly, neither of these techniques is ideal because both amount to slowing down and interrupting the user in order to allow the tutoring component

to function. However, it is important to recognize that such synchronization techniques need be employed only in special situations and, even in such situations, they will be visible to the user only if the tutoring component is much slower than the user's ability to operate the tool.

Goal Orientation

An important consideration in a learning environment is whether it is aware of the user's goals in performing some action. Our systems present a problem to the user, thus defining the user's high-level goals. Our proposals here are based on our experiences with such systems, but we believe this architecture could be applicable to systems which need to infer user's goals from context.

For example, Eager (Cypher, 1993) monitors sequences of user actions at the semantic level and recognizes when the user is performing a repetitive task that can be more efficiently done with a macro. Such a system could be built with this architecture, although the curriculum manager would be absent.

Similarly, Tip Wizard in *Microsoft Excel 5.0* looks for more efficient ways for the user to navigate through the interface. For example, it might detect that a user has moved through several levels of dialogs to perform an action when the same action could be accomplished by pushing a function key. Since this system is aimed at giving advice on manipulating the interface, our architecture would need to be supplemented with information about user-interface events, but the same principles would apply.

Relationship to Tool-Specific Help Systems

Many applications come with help systems. Plug-in tutor agents may be seen as a special kind of help system, with the advantage that they are tracking the user's activity step-by-step, and thus can target advice to individual users.

A tool-specific help system and a plug-in tutoring agent can complement each other well. If the user recognized that they wanted help with operating the tool, the user could ask for tool-specific help. If the user wanted to know what action to take next, the user could ask the tutor. If the tutoring component was developed to be tool-independent, its advice would stop at the semantic level. For example, it might say "Enter 10 in the cell." If

the user still didn't know how to proceed, the tutoring agent might call on the tool-specific help system to give advice on using the tool (in this case, the tutoring agent would want to present the tool's advice on "how to enter a value in a cell"). This tool-based advice may be different for different spreadsheets. In this way, the tutoring agent can direct the user to tool-specific instructions, even if the tutoring agent is written in a tool-independent manner.

AppleGuide is an example of an advanced help system that could reasonably be adopted to be used with a plug-in tutoring agent. The use of AppleGuide is especially promising because it, like the plug-in agent architecture described here, works as a plug-in addition to an application.

Relationship to ESSCOTS

McArthur, Lewis, and Bishay (1996) provide an excellent argument for using off-the-shelf software as a basis for educational environments (which they call ESSCOTS). We share their goals and enthusiasm for this approach and believe that ESSCOTS and plug-in tutor agents illustrate two different ways to take advantage of off-the-shelf software. The ESSCOTS approach takes a complex software system and provides scaffolding so that students can better take advantage of it. Through intelligent monitoring of students' skills, it is possible to structure the system so that it adapts to the student's current skill level.

The needs of our intelligent tutor agents have led us to focus on somewhat different issues in our communication with off-the-shelf software. Tutor agents require a fairly fine-grained description of the student's activity, so much of our activity has been concerned the content and format of this information. In addition, we have tried to develop abstract specifications of off-the-shelf software, so that it is possible to substitute one off-the-shelf tool for another, similar one. It is clear that the tool adaptability offered by ESSCOTS would benefit our plug-in tutors, and, in future work, we intend to explore ways in which we can offer adaptability in this framework.

Domain-Independence of the Translator

Despite our desire to write tutor agents for off-the-shelf tools, we have sometimes been unable to identify an appropriate off-the-shelf tool for the subject we want to teach and so have had to write our own tools. Even in

such cases, we have found it helpful to follow this architecture. In part, the architecture helps to separate the responsibilities of the programmers on the project. In part, the architecture allows us to define a series of tools to work with a single tutoring component. This gives us the option of having users work on a very restricted tool at the beginning and then gradually advance to a less restrictive but more powerful tool as they progress through the curriculum.

One of the issues that arises in designing such a system is how much to rely on the translator to understand the operation of the tutoring component. These issues relate to the expected domain of the tool. We have now built several spreadsheet tools (and accompanying translators) to be used with our algebra tutors. In these tutors, students use one column to represent the X variable and one row to represent the column heading.

Our spreadsheet tools generate cell references within semantic events in R1C1 format. In some translators, we keep this format when we send information to the tutoring agent. The translator just offsets the specific row and column numbers to account for the user's placement of their table within the full spreadsheet. From the tutoring component's point of view, the user always starts the spreadsheet at Row 0 and Column 0. This approach is very general, and it is easy to see how we might easily adapt a translator written for our own tool to use *Microsoft Excel*. The tutoring component has the responsibility of figuring out that Column 0 corresponds to the X-variable column. Under this approach, authoring the translator is largely mechanical and, in fact, we are working on a user-level authoring tool for translators of this kind (Ritter & Blessing, 1996).

Another approach is to tailor semantic events to what we know about our tutoring component. A different spreadsheet tool written for use with our algebra tutors uses a translator which describes cells as (for example) "the cell in the X-variable column and the column-heading row." Since this description refers to elements in the tutoring component's descriptive language, it requires that the tutoring component communicate some information back to the translator and that the translator store some state information. In this system, if the user chooses to put the X-variable column in Column 2 of the spreadsheet, the tool tells the tutoring component (through the translator) that the user put the string " X " in Row 2 of Column 2. The tutoring component then instructs the translator to name this column the "X-variable" column in subsequent references. The advantage of such a system is that the tutoring component does not need to compute or maintain the mapping between row and column coordinates and semantic references.

Either approach can be accommodated by the architecture described here. The choice of one approach over the other depends on the expected use of the tutoring component and translator.

Plug-In Tutors and the Internet

The desire to provide educational materials over the Internet has led us to consider extensions to the architecture described here. We could imagine the communication between any of the modules in a plug-in tutoring agent taking place across the Internet, rather than between processes on the same machine.

Because of the density of communication between the tool and tutoring agent, feedback from the tutor may be delayed if the communication between these elements travels over the Internet. Since the tool runs independently of the tutor, however, students can continue to work with the tool, regardless of tutor and network speed. When students make error-free progress toward a solution, there will be no performance delays as the tool processes each action locally while the remote tutor agent works to keep up. During student thinking time the tutor can often catch up, but in some cases students may want to wait for feedback or help messages. It is important to note, however, that such delays, measured in seconds, will be much shorter than the typical homework situation where the student must wait until the next day for help from the teacher. For some applications, this solution may be a reasonable one.

A second solution is to put both elements (along with the translator) on the server side of the link. We are currently experimenting with a system that uses an HTML form as the interface tool. This form communicates with a CGI front-end, which forwards information to the translator. This architecture allows complete re-use of the tutoring agent, but the nature of HTML forms prevents us from implementing the kind of fine-grained feedback that we typically use in our tutoring systems.

A third approach is for the tool, translator, and tutor to run on the client side (perhaps as Java programs). This would allow (though not require) more fine-grained feedback. If the curriculum manager were on the server, communication over the Internet would only be needed when the student wanted to load a problem or save work. The advantage of having the curriculum manager on the server is that students would have access to their work from any machine capable of connecting to the Internet.

CONCLUSION

In our development of two systems that add tutor agents to pre-existing tools, we have begun to define a set of standards that will simplify the creation of learning environments of this type. We believe that technology has advanced to the point where it is practical to think about building systems this way. Furthermore, the resulting systems combine the best elements of workplace tools and educational microworlds with guided instruction that has proved to be effective.

Although the specifications described here are sufficient for the tutors we have built in the past, we need to consider ways in which they could apply to the next generation of tutoring systems. Our architecture defines a tutoring agent as a robot, which has certain senses for monitoring user performance and certain effectors for communicating with others. The result of our work on this architecture has been to describe a way of implementing these senses and effectors using widely available technology.

By separating the tutoring agent from the other components of the learning environment, we can imagine, for example, developing a robotic tutoring agent which interacts with students over the Internet in rooms of a multi-user domain (MUD). While the standards introduced here define senses and effectors sufficient for the kind of learning environment we have already built, more work needs to be done to determine whether the type of interaction described here is appropriate in a wider variety of environments.

References

Anderson, J.R. (1988). The expert module. In M.C. Polson & J.J. Richardson (Eds.), *Intelligent tutoring systems* (pp. 21-53). Hillsdale, NJ: Lawrence Erlbaum.

Anderson, J.R., Conrad, F.G., & Corbett, A. T. (1993). The Lisp tutor and skill acquisition. In J.R. Anderson (Ed.), *Rules of the mind* (pp. 143-164). Hillsdale, NJ: Lawrence 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.

Apple Computer, Inc. (1993). *Inside Macintosh: Interapplication communication.* Reading, MA: Addison-Wesley.

Breuker, J. (1990). *Eurohelp: Developing intelligent help systems.* Copenhagen: EC.

Brusilovsky, P. (1995). Intelligent learning environments for programming: The case for integration and adaptation. In *Proceedings of the Seventh World Conference on Artificial Intelligence in Education,* (pp. 1-8). Charlottesville, VA: Association for the Advancement of Computing in Education.

Burton, R.R., & Brown, J.S. (1982). An investigation of computer coaching for informal learning activities. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems,* (pp. 79-98). New York: Academic Press.

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

Fox, T., Grunst, G, & Quast, K. (1994). HyPLAN: A context-sensitive hypermedia help system. In R. Oppermann (Ed.), *Adaptive user support,* (pp. 126-193). Hillsdale, NJ: Lawrence Erlbaum.

Gates, B. (1987). Beyond macro processing. *Byte, 12*(7), 11-16.

Jackiw, R.N., & Finzer, W.F. (1993). The *Geometer's Sketchpad*: Program- ming by geometry. In A. Cypher (Ed.), *Watch what I do: Programming by demonstration,* (pp. 292-307). Cambridge, MA: MIT Press.

Koedinger, K.R., & Anderson, J.R. (1993a). Effective use of intelligent software in high school math classrooms. In *Proceedings of the Sixth World Conference on Artificial Intelligence in Education,* (pp. 241-248). Charlottesville, VA: Association for the Advancement of Computing in Education.

Koedinger, K.R., & Anderson, J.R. (1993b). *A cognitive tutor for mathematical investigation and reasoning.* Unpublished proposal manuscript.

Koedinger, K.R., & Anderson, J.R. (1993c). Reifying implicit planning in geometry: Guidelines for model-based intelligent tutoring system design. In S. Lajoie & S. Derry (Eds.), *Computers as cognitive tools.* Hillsdale, NJ: Erlbaum.

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.

Kosbie, D.S., & Myers, B.A. (1993). A System-Wide macro facility based on aggregate events: A proposal. In A. Cypher (Ed.), *Watch what I do: Programming by demonstration* (pp. 433-444). Cambridge, MA: MIT Press.

Lajoie, S.P., & Lesgold, A. (1989). Apprenticeship training in the workplace: Computer coached practice environment as a new form of apprenticeship. *Machine-Mediated Learning, 3,* 7-28.

Mark, M.A., & Greer, J.E. (1995). The VCR tutor: Effective instruction for device operation. *The Journal of the Learning Sciences, 4*(2), 209-246.

McArthur, D., Lewis, M.W., & Bishay, M. (1996). ESSCOTS for learning: Transforming commerical software into powerful educational tools. *Journal of Artificial Intelligence in Education, 6*(1), 3-33.

National Council of Teachers of Mathematics (1989). *Curriculum and evaluation standards for school mathematics.* Reston, VA: Author.

Olsen, D., & Dance, J. (1988). Macros by example in a graphical UIMS. *IEEE Computer Graphics and Applications, 8*(1), 68-78.

Paiva, A., Self, J., & Hartley, R. (1995). Externalising learner models. In *Proceedings of the Seventh World Conference on Artificial Intelligence in Education.* Charlottesville, VA: Association for the Advancement of Computing in Education.

Ritter, S., & Anderson, J.R. (1995). Calculation and strategy in the equation solving tutor. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society.*

Ritter, S., & Blessing, S.B. (1996). A programming-by-demonstration tool for retargeting instructional systems. In *Proceedings of the Second International Conference on the Learning Sciences.* Charlottesville, VA: Association for the Advancement of Computing in Education.

Ritter, S., & 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.

SCANS (1991). *What work requires of schools: A SCANS report of America 2000.* Secretary's Commission on Achieving Necessary Skills. U. S. Department of Labor.

Schwartz, J. L., Yerushalmy, M., & Wilson, B. (1993). *The geometric supposer: What is it a case of?* Hillsdale, NJ: Erlbaum.

Acknowledgements

This material is based upon work supported by the National Science Foundation and the Advanced Research Projects Agency under Cooperative Agreement No. CDA-940860 and the CAETI project. This paper is an extension of work first reported in Ritter and Koedinger (1995).