HAL
open science

A Programming by Demonstration Authoring Tool for Model-Tracing Tutors

Stephen B. Blessing

To cite this version:

Stephen B. Blessing. A Programming by Demonstration Authoring Tool for Model-Tracing Tutors. International Journal of Artificial Intelligence in Education, 1997, 8, pp.233-261. hal-00197385

HAL Id: hal-00197385

https://telearn.archives-ouvertes.fr/hal-00197385

Submitted on 14 Dec 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

International Journal of Artificial Intelligence in Education (1997), 8, 233-261

A Programming by Demonstration Authoring Tool for Model–Tracing Tutors

Stephen B. Blessing

Department of Psychology
University of Florida
Gainesville, FL 32611
e-mail: blessing@psych.ufl.edu

Abstract. Model–tracing tutors have consistently been among the most effective class of intelligent learning environments. Across a number of empirical studies, these tutors have shown students can learn the tutored domain better or in a shorter amount of time than traditionally taught students (Anderson et al., 1990). Unfortunately, the creation of these tutors, particularly the production system component, is a time–intensive task, requiring knowledge that lies outside the tutored domain. This outside knowledge—knowledge of programming and cognitive science—prohibits domain experts from being able to construct effective, model–tracing tutors for their domain of expertise. This paper reports on a system, referred to as Demonstr8 (and pronounced "demonstrate"), which attempts to reduce the outside knowledge required to construct a model–tracing tutor, within the domain of arithmetic. By utilizing programming by demonstration techniques (Cypher, 1993; Myers et al., 1993) coupled with a mechanism for abstracting the underlying productions (the procedures to be used by the tutor and learned by the student), the author can interact with the interface the student will use, and the productions will be inferred by the system. In such a way, a domain expert can create in a short time a model–tracing tutor with the full capabilities implied by such a tutor—a production system that monitors the student's progress at each step in solving the problem and gives feedback when requested or necessary, in either an immediate or delayed manner.

Model–tracing tutors have proven extremely effective in the classroom, with the most promising efforts demonstrating more than a standard deviation's improvement over traditional instruction (Anderson et al., 1990; Koedinger & Anderson, 1993a). They are referred to as model–tracing tutors because they contain an expert model which is used to trace the student's responses to ensure that the student's responses are part of an acceptable solution path. The creation of such tutors, particularly the expert models that underlie them, is a time–intensive task, requiring much knowledge outside of the domain being tutored. Anderson (1992) estimated

that 100 hours of development time yields about 1 hour of instruction. The goal of the authoring tool described in this paper is to drastically reduce the amount of time and knowledge needed to create a model–tracing tutor. Since this authoring tool produces a tutor which is functionally equivalent to one of Anderson's ACT tutors, what follows is a description of those tutors, and then a discussion of how an ACT tutor is actually created (see Anderson et al., 1995, for a more complete discussion of the ACT tutors).

A DESCRIPTION OF THE ACT TUTORS

The ACT Tutors developed by Anderson and his colleagues (Anderson et al., 1995; Corbett & Anderson, 1990; Anderson, Conrad, & Corbett, 1989) all operate by the same basic mechanism. What forms the backbone of the system is an expert model, realized as a set of production rules, that contains all the knowledge needed to solve problems within the domain being tutored. A production rule is a statement with a condition and an action, and a set of such statements can specify the procedural knowledge needed to perform a task. Within the algebra equation solving tutor, for example, this expert module is a set of around 150 productions (S. Ritter, personal communication, December 10, 1996). The tutors use this expert model to check the student's answers. In all of the ACT Tutors, this production set is general enough to be able to solve novel problems. That is, a programmer, a teacher, or even a student can enter a new problem into the system, and then the tutor will be able to solve it and to provide tutoring on it. This generality is part of what makes the architecture of the ACT Tutors so powerful.

Another feature of the ACT Tutors which is consistent across the tutors is the student model. The student model is the assessment by the tutor of the student's current knowledge state. As the student interacts with the system, getting some of the answers right and others wrong, this student model is updated. The ACT Tutors contain a list of skills which make up the tutored domain. Skills correspond to a sequence of production rules which result in a student action. Each skill is considered to be in either a learned or unlearned state, with a probability assigned to it that it is currently in the learned state. As students demonstrate proficiency (or a lack of proficiency) in a skill by corresponding to (or not corresponding to) a set of productions, this probability is adjusted according to a Bayesian algorithm. When this probability gets above a certain level, generally 95%, that skill is assumed to be in the learned state. Many of the tutors have an external realization of this student model, referred to as the Skillometer. This Skillometer is essentially a bar chart, with each bar representing a skill, and each bar displaying the current probability that the skill is in the

learned state. When the bar gets above the criterion level for that skill to be assumed to be in the learned state, a little checkmark is placed next to it.

The third main feature of the ACT Tutors is the student's interface, which must vary across the different tutored domains. The interface that the student interacts with in the geometry tutor is very much like a typical computer drawing program, whereas the interface used by the programming tutors is a structured text editor. However, no matter the specifics of an interface, for each student action within the interface (clicking a button, typing in a text field, selecting a menu option, etc.), that action can be checked against the expert model. This process of checking student actions using the expert model is referred to as model tracing. The outcome of this checking can either be displayed immediately, right after the student does the action, or it can be delayed until the student requests the feedback. Also, in the case that the tutor displays the feedback right away, the tutor can either not allow the action or somehow indicate within the interface that the action is not correct (perhaps by displaying the student's incorrect response in red). Depending on what skill the student's action corresponded to, the student model is also updated, in a process called knowledge tracing. Once all the skills that make up the current lesson are assumed to be in the learned state (i.e., all the skills in the Skillometer are "checked"), the student is graduated to the next lesson, which will add skills to be mastered.

CREATING AN ACT TUTOR

Anderson and Pelletier (1991) described a development tool for the creation of the ACT Tutors. This tool is referred to as the Tutor Development Kit, or TDK. The TDK is based in Macintosh Common LISP (MCL), and provides a working memory manager, a production interpreter, and several utilities to make the creation of a tutor easier. While this is better than starting to program a tutor from scratch, a fair amount of non–domain knowledge is needed to create even the simplest tutor. The student's interface needs to be constructed, the appropriate representation of that interface needs to be encoded within the tutor's working memory, and the production rules for the task need to coded. All of this requires not only a good knowledge of the TDK's syntax, but also a decent familiarity with LISP and how to create graphic elements in MCL.

As an example of what it takes to program a tutor using the TDK, consider the code displayed in Table 1. This is part of the code one might write for a tutor whose domain is multi–column addition. The actual syntax is not important, but rather the general feel for the overall complexity involved in creating a tutor for a seemingly easy task like multi–column addition. The top four lines define the types of objects that can be in

working memory (WME is short for Working Memory Element). The first word after “defwme” is the name of the type of object, and then the following words are attributes, or slots, of that object. The next two statements, the “make–wme’s” actually create elements within the tutor’s working memory. A column is made, and then a cell. The value of the slots are assigned upon the WME’s creation, and they can change as the student interacts with the tutor. These lines are necessary, the defwmes and make–wmes, because the ACT Tutors must maintain within their working memory a representation of the current state of the problem—what appears onscreen to the student. As the student works on the problem, this representation is changed to reflect what the student has done. Help messages and hints are partially based upon this representation. Many more make–wme statements would be required in the complete definition of this tutor.

```
(defwme addition–goal column)
(defwme addition–column rows part–of english)
(defwme cell text row column parent part–of note)
(defwme write–answer column object)
(make–wme column1 isa addition–column rows (c03 c02 c01 c00)
          part–of root english (name tens))
(make–wme c00 isa cell text “2” part–of column1 row 0 column 3)

(defproduction process–column model (=goal)
   =goal>
      isa addition–goal
      column =column
   =column>
      isa addition–column
      rows ($ =cell1 =cell2 =cell3)
   =cell1>
      isa cell
      text =num1
   =cell2>
      isa cell
      text =num2
 ==>
    !eval! =sum (princ–to–string (+ (convert–to–digit =num1)
                                    (convert–to–digit =num2))
    =subgoal>
       isa write–answer
       column =column
       object =sum
    !chain! model (=subgoal)
    :help ‘(“Add “ ,=num1 “ and “ ,=num2 “.” ~n)
          ‘(,=num1 “ + “ ,=num2 “ = “
            ,(princ–to–string (+ (convert–to–digit =num1)
                                 (convert–to–digit =num2)) “.” ~n)
```

Table 1. TDK code segment for a multi–column addition tutor

The last several lines of Table 1 contain one of the actual TDK productions needed by the expert model. The full model for multi–column

addition would contain on the order of 10 such productions. This production is responsible for figuring out the sum of the two digits in a particular column. Other productions would figure out which column is important and what exactly to write down. The words with equal signs in front of them are variables. This production will match to the two numbers in the current column of interest. Once it finds the sum, it sets a subgoal to actually write the answer in the column, which will be handled by another production. The last couple of lines contain some help text (which will use the current numbers in the problem) which will be used when the student asks for help.

Table 1 contains much less than 10% of the code needed for an addition tutor. The author of such a tutor would need to type all of the code into a blank document, run it, and then debug it. No provision is made within the TDK for the automatic generation of working memory elements or productions. For these things, authors simply have a clean slate they must fill in. The TDK does do the knowledge tracing automatically, provided that the necessary hooks are in place. In general, a lot of programming "glue" is needed to get everything—model and knowledge tracing, and the student interface—running correctly. It would probably take a competent MCL and TDK programmer a half day or more to program a basic addition tutor.

GOING BEYOND THE TDK

Based on the short example above, one can easily see that the creation of a tutor that would be used throughout an entire course, like the ACT Tutors for programming and algebra that have been constructed, would require many, many hours worth of work. Additionally, it requires close collaboration between the educators, the research scientists, and the programmers. An educator or other domain expert could not sit down and create a model–tracing tutor, nor could they easily modify an existing tutor for their individual needs. Existing tutors do have ways for educators to create new problems for the tutors (in the geometry tutor, it's as easy as using drawing package), but this is a far cry from being able to create a tutor which teaches a new skill—something that would be advantageous for educators and curriculum designers to be able to do. What is needed is a system that drastically reduces or eliminates the non–domain knowledge needed to create a model–tracing tutor.

By allowing the author to simply manipulate the actual interface that the student is going to use for the creation of working memory elements and productions, it becomes much more feasible for non–cognitive scientists to create intelligent, model–tracing tutors. The main mechanism which allows the tool described in this paper, Demonstr8, to infer the

production rules, the heart of a model–tracing tutor, is similar to ACT–R's analogy mechanism (Anderson, 1993)[1] to be explained in the following paragraphs.

Inducing Productions

An important distinction within the ACT–R architecture is between declarative knowledge, one's knowledge of facts (e.g., "Washington DC is the capital of the United States") and procedural knowledge, one's knowledge of how to perform actions (e.g., adding numbers together). One of the claims of the ACT–R theory is that all knowledge has declarative origins. That is, the only way new procedural knowledge, in the form of production rules, enters the system is by the process of analogizing from the current goal to some previous declarative knowledge. This mechanism operates by forming an analogy from examples stored in declarative memory to the current goal. The current authoring tool uses this idea, since the items on the screen can be represented by declarative knowledge structures, and as the author interacts with them, the procedural knowledge of what to do with them (i.e., what will become the expert model of the tutor) can be inferred. Demonstr8 translates the actions of the domain expert (i.e., their interactions with the objects in the interface) into the knowledge needed by the tool to successfully tutor students.

To provide an example of ACT–R's analogy mechanism, assume that an ACT–R model of a person learning LISP has the declarative knowledge that the command to multiply 5 by 7 is (* 5 7). This model has a goal to add 132 and 254. The situation can be diagrammed as:

Current Goal:	*Previous Example:*
Add 132 and 254	Multiply 5 and 7
	⇓
	(* 5 7)

The arrow shown connecting the two pieces of declarative information in the Previous Example simply indicates that the goal of multiplying 5 by 7 can be realized by typing (* 5 7), as illustrated by the example. Two additional pieces of declarative information that the system has stored are that the symbol for multiplication is '*' and the symbol for addition is '+.' By analogizing from the Previous Example in order to solve the Current Goal, and using the additional knowledge of which operator is indicated by which English word, the model will construct the following production:

[1] The ACT Tutors do not use ACT–R, the most recent version of Anderson's ACT Theory. The TDK, and most of the ACT Tutors, were created before ACT–R. They do share a common heritage.

IF the current goal is a LISP operation with two arguments
AND *op* is the symbol corresponding to the operation
AND *arg1* and *arg2* are the two arguments
THEN the LISP call should be: *(op arg1 arg2)*

As can be seen, the analogy mechanism has generalized across the different numbers, and has linked the knowledge that '*' is the operator for multiplication with '+' is the operator for multiplication to construct the generalization that when the goal is to do some LISP operation, use the associated operator in the front position of the LISP call. The rest of this paper describes Demonstr8, a system which begins to address the ability of non–cognitive scientists being able to program a model–tracing tutor by using something like this analogy mechanism. Using Demonstr8, both the time and knowledge required to produce a model–tracing tutor is drastically reduced. With limited training, a non–cognitive scientist can produce a model–tracing tutor for arithmetic in less than 20 min (an estimate based on informal observations).

DEMONSTR8 DESCRIPTION

Using Demonstr8, an author can create a model–tracing tutor for some specific application within arithmetic—multi–column addition or subtraction, for example. While Demonstr8 has been implemented for the construction of a particular class of tutors, parts of it are general enough to apply to any domain, and the ideas behind the parts that are not can be modified to work within almost any setting. Throughout the description of Demonstr8, these generalizable features will be highlighted, and the General Discussion addresses this specific point of which features are readily generalized and which require additional effort.

Within Demonstr8 the author has available three things:

- a palette of MacDraw–like tools used to create the student interface
- a method for creating higher–order declarative representations for these student interface elements (e.g., making a column of numbers)
- a programming by demonstration method for creating productions.

There is also a way of inputting new abstract, declarative knowledge into the system, such as subtraction facts, which aids in the induction of the production rules. However, the system has a number of built–in facts for doing arithmetic, so an author may never have to use this feature. The above three items are all that are necessary to create a model–tracing tutor for some aspect of arithmetic.

When the system first starts, three windows, a menu titled Author, and the Author Palette appear. The three windows are:

- Student Interface: The author will create the student's interface within this window, using the available tools. (Shown in Figure 1)
- Working Memory Elements: This window contains a list of the current working memory elements of the tutor. At startup, the only item is a list of predefined memory elements, which are the numbers 0 through 18 (sufficient for arithmetic tasks through multi–column subtraction), a blank, and a slash character.
- Productions: This window contains a list of current productions, which is empty at startup. At the bottom of this window is a checkbox which indicates if the system is in recording mode. When the box is checked, the system is in recording mode, and is recording the author's actions and trying to induce the productions behind them (more on that in the Demonstrating Productions section below).

Both of the last two windows are scrolling, hierarchical lists from which items can be selected, edited, and deleted.

The Author menu contains options for creating working memory elements, new productions, a problem generator, and other authoring activities. These will be discussed as they arise in the creation of a tutor. The Author Palette will be described in the next section. The rest of this section discusses designing an interface, creating the working memory elements, and programming the productions within the context of creating a multi–column subtraction tutor.

DESIGNING THE STUDENT INTERFACE

The tools available to the author for creating the student's interface are basic, but complete enough to provide all the functionality needed for an arithmetic tutor. All the tools are contained in the Author Palette, as seen in Figure 1. The tools work like the tools in a drawing program. They are, from left–to–right:

- Selection Tool: Used to select any object within the student's interface. Multiple objects can be selected, which is useful when creating working memory elements (see next section).
- Cell Tool: Used to place cells within the student's interface. Cells are the place holders for numbers, and authors create them simply by clicking where they want the cell to be.
- Line Tool: The author can draw lines in the student interface, to make it look more like an arithmetic problem.

- Worksheet Tool: This is the grid onto which the cells get placed. While it is possible to have more than one worksheet per tutor, it probably is not necessary (though another class of tutors for a different domain could make use of this feature).

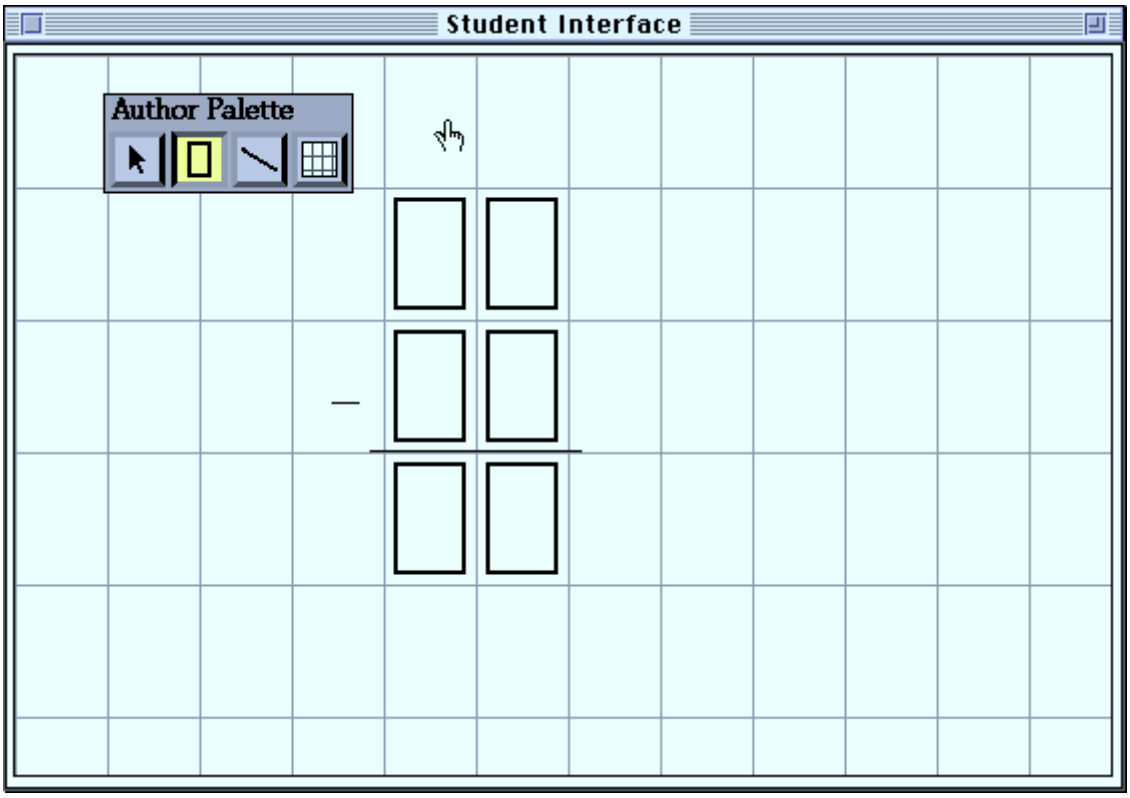


Figure 1. Designing the interface.

worksheet, the author has created six cells arranged in two columns of three, and has drawn two lines so that the interface resembles a simple subtraction problem. When cells are placed on the worksheet, they automatically become available as WMEs. Designing a student interface in such a way, using a set of drawing tools, can easily be applied to any domain. Domain specific tools, like the Cell Tool, could be supplied as plug–in packages available to the author. These domain specific tools, as will be seen with the Cell Tool, can contain specialized knowledge for operating correctly in the authored domain (e.g., cells know how to display numbers and that numbers can have slashes through them).

At this point, the author has created the interface that he or she will use to demonstrate the skills, and this interface will also be the one students will ultimately use. The actions the author has performed are:

- Selecting the Cell Tool from the Author Palette
- Placing six cells on the provided worksheet grid
- Selecting the Line Tool from the Author Palette

- Drawing two lines so that the interface looks like a subtraction problem

CREATING WORKING MEMORY ELEMENTS

After the author has created the basic layout of the student's interface, he or she must next define any higher–order working memory elements that are part of this interface. For the simple subtraction tutor, the six cells that have been placed correspond to two columns, and these two columns correspond to one problem. These two columns and one problem must be defined in order for the correct productions to be induced. One could imagine a system, particularly a system devoted to arithmetic, already knowing about columns (i.e., cells that are aligned vertically) and problems (i.e., columns that are aligned horizontally), but this feature of Demonstr8 was provided for two reasons. First, it allows the system to be more extensible by allowing the author to define novel arrangements of cells. Second, and more importantly, it demonstrates the generality of the way in which working memory elements based on primitives (like cells in an arithmetic domain, or any of the other items directly available from the Author Palette) could be authored, based on the student interface and employing end–user programming techniques (Smith et al., 1994; Nardi, 1993). The following description shows how the cells can be grouped into columns, and then the columns into a problem.

Grouping cells into columns

To group the three cells to the right into a column, the author selects all three cells using the Selection Tool. With all three cells selected, he or she must next select Name WMEs from the Author menu. A dialog box similar to what is depicted in Figure 2 appears, which has been filled out. Since this is the first time a column has been created, the author must leave the New radio button checked, and fill in what the classname (*defwme* in TDK parlance) should be, "Column." Next, the author must give a new instance name to this particular column, for example, "Column1."

The middle of the dialog contains a scrolling, two–column table which lists the properties (or "slots") of this classname on the left, and their current values for this instance on the right. Since three cells were selected in the student's interface, the system assumes there are three properties, one for each cell. That was the proper guess in this instance, since columns are composed of a Top, Bottom, and Answer cell. The author can rename the property names, as the author is currently doing in Figure 2. If the values of the properties are wrong, they can be edited by double–clicking them and

obtaining an editing box, or values can be dragged–and–dropped from the Working Memory Elements window.

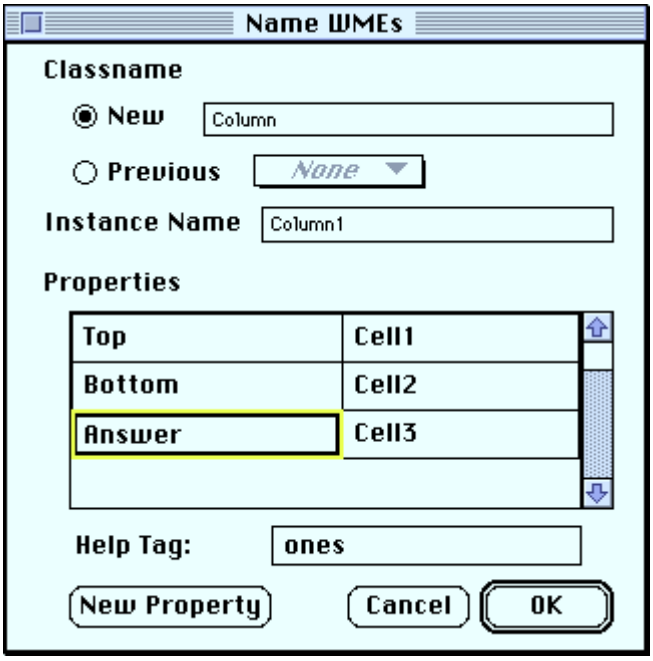


Figure 2. Creating working memory elements

Below the properties table is an edit box labeled “Help Tag.” The author can input here a word or short phrase that will be used by the help system when the system generates a help message. Since this is the ones column, the author can type “ones column” or “ones” in the box. When the author clicks the Okay button, the WME type Column appears in the Working Memory Elements Window, with its one instantiation, Column1.

Defining the second column, the tens columns, is considerably easier, since the column type has already been defined. After the author has selected the three cells in the other column and selected Name WMEs from the Author menu, the author can select “Column” from the pop–up menu under classname (this pop–up menu contains the name of the WME types the author has already created), which automatically fills in the correct property names. All that is left for the author to do is to give this particular column a name (“Column2”), provide a help tag (“tens”), and click the Okay button.

At this point, two columns have been created based on the six initial cells the author placed in the interface. The actions the author had to perform were:

- Selecting the Selection Tool from the Author Palette and selecting the three leftmost cells.
- Choosing “Name WMEs...” from the Author Menu

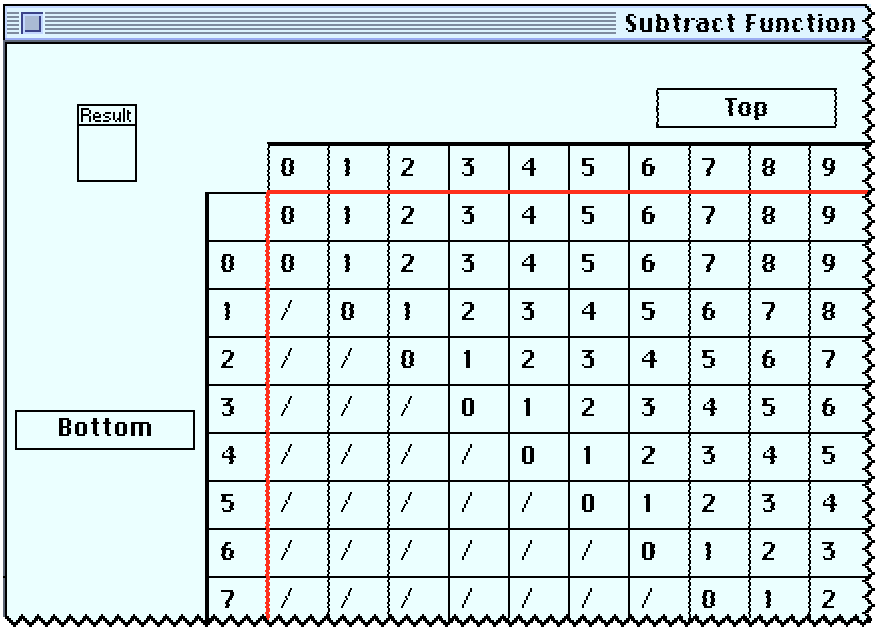
- Completing the resulting working memory dialog box:
  - Creating a new classname (“Column”) for the selected object, and naming this particular instance (“Column1”)
  - Providing names for the attributes of this class (“Top,” “Bottom,” and “Answer” cells)
  - Indicating a piece of help text to be associated with this instance
- Selecting the three rightmost cells and choosing “Name WMEs...” from the Author Menu
- Completing the dialog box for this column (“Column2”), which is greatly simplified since the class Column has already been defined

Grouping columns into a problem

The last WME the author defines is the problem. This demonstrates the way a WME can be created that has no direct objects on the screen that represent them. A problem is composed of the two columns, and whereas the cells that make up the columns are objects on screen that can be selected, the columns themselves cannot (though one could imagine an authoring interface which could make apparent some of these higher–order WMEs). Therefore, the author cannot have anything selected when the Name WMEs item is selected from the Author menu. Once the Name WMEs dialog appears, and the new classname of “Problem” and an instance name of “Problem1” has been entered, the author fills in the Properties table, which is currently empty, since nothing was selected. Using the New Property button, the author can either create two new properties, one for each column (“OnesColumn” and “TensColumn”), or one property (“Columns”) which has list of columns as its value. Based on the way the author has decided to create the production system, the latter option is selected since it will allow for the easier creation of the rest of the system. To create the list of values, the names of the columns are dragged over from the Working Memory Elements Window and dropped on the Properties table. Once more than one value has been dropped, a list enclosed in parentheses is automatically created for that property’s value. The author decided to use this method because Demonstr8 has special features to reason about lists (e.g., finding the rightmost element in a list, or an item which is to the immediate left of another item). This feature will be highlighted in the next section, which describes how the author demonstrates the procedural knowledge of a task.

The author has now defined all the necessary WMEs needed for this tutor. To define the Problem WME, the actions the author had to perform were:

- With nothing selected in the interface, choosing “Name WMEs...” from the Author Menu
- Completing the resulting working memory dialog box:
- Creating a new classname for the selected object (“Problem”, and naming this particular instance (“Problem1”)
- Providing a name for the attribute of this class (“Columns”)
- Dragging the values for this attribute from the WMEs window into the row named above



	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	/	0	1	2	3	4	5	6	7	8
2	/	/	0	1	2	3	4	5	6	7
3	/	/	/	0	1	2	3	4	5	6
4	/	/	/	/	0	1	2	3	4	5
5	/	/	/	/	/	0	1	2	3	4
6	/	/	/	/	/	/	0	1	2	3
7	/	/	/	/	/	/	/	0	1	2

Figure 3. The Subtract Knowledge Function

The structures that are perhaps most key to inducing the productions behind the actions of the author are Knowledge Functions, which can be thought of (and depicted as) ACT–R working memory elements which represent a table of values. These structures represent declarative knowledge not depicted directly in the interface. In the current implementation, Knowledge Functions take one or two values and return a single result. In this respect, they are similar to look–up tables, and as such enable many WMEs to be depicted in an easy–to–read, easy–to–enter manner. While they are limited to only two–dimensions in the current application, n–dimensional Knowledge Functions are possible (and indeed, the ACT–R analogy mechanism is meant to deal with such structures). Demonstr8 has built–in the Knowledge Functions for the standard arithmetic operations (e.g., addition, subtraction, and decrementing). Figure 3 shows part of the Subtraction Knowledge Function. An easy interface exists within Demonstr8 to create new, novel Knowledge Functions. After selecting New Knowledge Function... from the Author menu, and

indicating the size and name of the table, the author can simply drag and drop keys (the inputs to the table, which can be any WME in the system) and values (the result to be returned—again, these can be any WME) onto the table. In such a way, an author could create a subtraction function for octal arithmetic, and have it automatically available to the system from which to infer productions.

This method of creating new working memory elements—grouping items in the interface or creating tables of values—is general to any domain. Since almost all working memory elements correspond to things a person can point to (or at least has an easy–to–generate representation that could be pointed to), having a visual method of placing and creating WMEs should be natural and easy–to–do in any domain.

DEMONSTRATING PRODUCTIONS

With all the necessary working memory elements constructed, the author is now ready to demonstrate the skill to be tutored, and have Demonstr8 induce the underlying production rules. First, however, the author will want to create a problem within the student's interface to be solved. The author can drag the predefined numbers from the Working Memory list and drop them onto the cells which represent the problem. The cells will then display those numbers. Alternatively, the author can use one of the options under the Author menu, Define Problem Generator. Selecting this option displays a dialog which lists all the cells in the student interface. The author can indicate which cells should contain a number, and what range that number should fall. In such a way, the author can prohibit, if desired, the system from generating problems that would involve borrowing. While the description to follow is specific to demonstrating the productions needed for subtraction, the general approach (essentially presenting the system with before and after shots and having it infer the underlying action) can be applied to almost any domain, and is the basic way ACT–R's analogy mechanism works.

Once the problem is in place, the author can begin to solve the problem. To indicate to the system that it should now start recording what the author is doing, the author checks a box marked "Recording" at the bottom of the Production List. The system will now take note and will attempt to induce the productions behind the author's actions, recording the results within the currently highlighted production in the Production List.

Two ways exist for the author to define a production using Demonstr8. The first is to interact directly with the Knowledge Functions. With the proper Knowledge Function on the screen, the author can drag and drop the items of interest from the problem displayed in the student's interface onto the table, and then drag the function's result onto the proper cell in the

problem. Dragging and dropping a result from a Knowledge Function onto the student interface indicates a student action, and the system constructs a production based on the actions of the author from the creation of the last production (or the start of recording) to the use of some function's result. Having the author define the production in this way directly specifies to Demonstr8 the relevant WMEs and Knowledge Function needed to construct the production.

The other method that Demonstr8 has for creating productions is much more powerful and flexible, and is based on ACT–R's analogy mechanism. To define a production using this method, the author option–clicks on a cell. A dialog is displayed, asking what value should be entered into that cell. Once the author enters a value and clicks the dialog's Okay button, the system attempts to induce why the author entered that value into that cell. In trying to induce the production, Demonstr8 has available its Knowledge Functions and the Working Memory Elements which the author defined. For example, in the present case the author has arranged three cells into a column of numbers, and has created the appropriate WMEs. The top two cells on the right have numbers in them (say a "7" with a "2" below). The author indicates that the bottom–most cell should have a "5" in it. Since these three cells have been arranged in working memory as a column, the system has access to the "7" and "2" in order to figure out where the "5" comes from (i.e., this cell that should contain a "5" is part of this column, and so the other cells of this column may contain useful information; this column is also part of this problem, and if necessary, the system may reason about that as well). By using the "7" and "2" in various combinations with the available Knowledge Functions in order to figure out the appearance of the "5," it will find that a "7" and a "2" with the Subtract Function will produce a "5." In such a way, the system has induced a rule that the Top number of a column and the Bottom number of a column can be used to produce the Answer of that column. If the indicated number (the "5" in the above example) could have come about from multiple ways (e.g., by using different Knowledge Functions with the WMEs that are related in some way to the "5"), the system displays a Discrimination Dialog, asking the author to pick the proper interpretation of the action.

After the system has either gone through the above process, or the author has chosen to interact directly with the Knowledge Functions, a dialog box similar to the one displayed in Figure 4 appears. Using this dialog, the author can fine–tune the production that was just created. If the author used the second method of production creation, probably very little, if any, fine–tuning will be needed, since the system has had to infer much more about the production than it would if the first method had been used. Specifically, using the first method, the system would not know that the

representation of column needs to be invoked in order to properly use this rule (i.e., what was important about putting the “5” in that cell was because that cells containing a “7” and a “2” were also part of that column).

Condition and Action

The dialog box in Figure 4 has three parts, accessed by the radio buttons in the upper right. These parts appear in the thick rectangle that takes up most of the dialog, and parts that have not been accessed are italicized. The production’s name appears in the upper left, and can be edited easily. The first part that comes up, and the one displayed in the figure, is the part that actually shows the form of the production that has been induced. This is the way the dialog would look if the act of entering a “5” after the option–click had been done, as discussed above. Many of the pop–up menus are a consequence of how the working memory elements were initially created. While it may look complicated, the interpretation is easy, and as previously stated, the author will probably not have to make many, if any, changes. The dialog as shown in Figure 4 is the correctly specified production, and only two related changes had to be made from the default (which are mentioned at the end of this subsection).

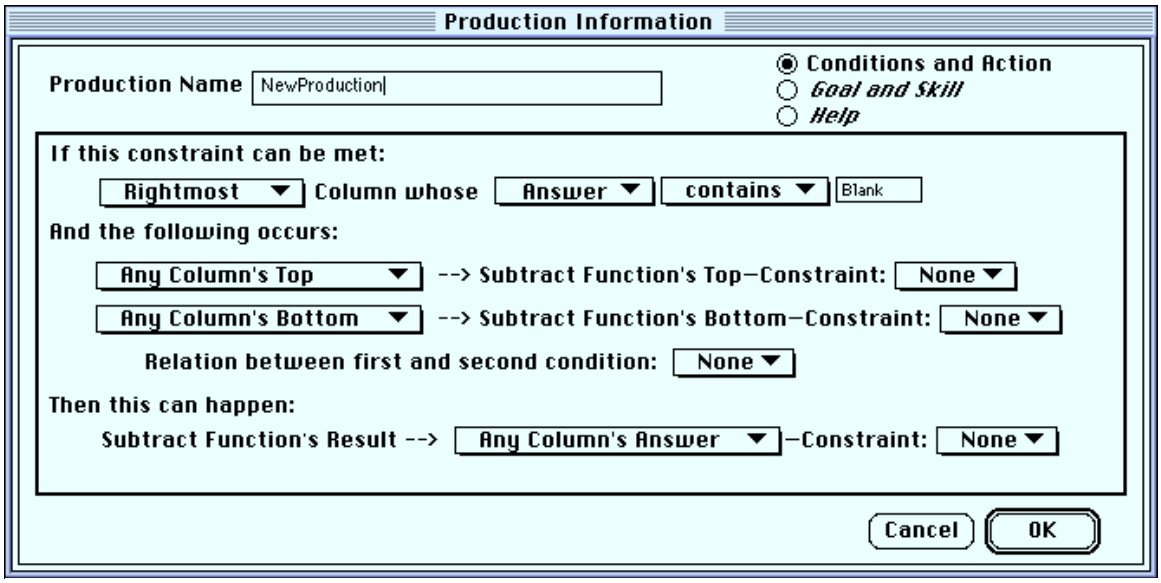


Figure 4. Fine-tuning a production

Let us first consider the lines which follow the statement “And the following occurs:” in the middle part of the dialog. This is the production’s condition statement. The way this condition should be thought of is, “If the top and bottom number of a column is used with the Subtract Function...” The menu options for the pop–up menu currently set to “Any Column’s Top” contains the full container hierarchy of how to refer to the top–most cell of that column (e.g., Cell1 —> Any Cell —> Column1’s Top —> Any

Column's Top), and one can generalize the production to any of those levels. This refers to the cell that contained the "7" in the problem. Since the Column1 working memory element needed to be invoked in order to access this "7" (because the "5" is in the same column), the pop–up menu is automatically set to a one higher level of generality (just as a rule of thumb, which turns out to be correct in most instances in the arithmetic domain). The arrow can be translated as "is placed on," and then after it appears the Knowledge Function which was used. The pop–up menu set to "None" contains a list of comparison functions (e.g., "<," "=," ">=," "contains," etc.). When one of these comparison functions is selected, a box appears to the right of the pop–up. A value can then be entered into the box in order to place a constraint on the possible values which whatever cell is indicated can have. For example, you may want one action to occur when borrowing from a zero occurs and another action to occur when borrowing from a non–zero. In the first case, you can set the pop–up to "=" and put a "0" in the box that appears.

The second line, which starts "Any Column's Bottom" has the same interpretation, and came about the same way as the first line. The line below it, which starts off "Relation between first and second condition:" and ends with a pop–up menu gives the author the opportunity to input a constraint (the pop–up contains a list of comparison operators) between the whatever the values the first and second conditions are instantiated as. For example, in multi–column subtraction with borrow, one rule may apply when the top number is equal to or greater than the bottom, and a different would apply when the bottom number is bigger. That relation can be specified with this pop–up.

Demonstr8 is currently limited to having only two of these conditions, primarily because the Knowledge Functions are limited to two dimensions. While this may seem a detriment (though it would scale up easily—the ACT–R analogy mechanism can create arbitrarily complex productions, if needed and correctly specified), there is a movement within the ACT–R community to create simpler productions, those in which only one, or at most two, memory retrievals are performed. Larger productions are harder to understand and debug, and given the purpose of this tool (allowing non–programmers and non–cognitive scientists to create ITSs), forcing an upper limit on the complexity of the authored productions is a desirable attribute.

The line after the phrase, "Then this can happen:" is the production's action, and is arrived at much the same way was as both of the conditions. In this particular instance, it can be thought of as "...then the result from the Subtract Function can be placed in the column's result cell." Constraints can also be placed on the values that this result can have. Again, Demonstr8 is limited to creating productions with only one action. This action corresponds to a change being made within the tutor's interface. If

multiple actions are possible at any one point in solving a problem (e.g., different strategies can be followed), then multiple productions which correspond to each of the individual actions will need to be authored.

Given just the above information (what is specified under “And the following occurs” and “Then this can happen”), the system does not know which column is being referenced. The information under “If this constraint can be met:” gives this information. Since the system has induced that columns are important for this production, it continues up the WME container hierarchy looking for a slot value which contains a list of columns. The Problem WME has such a list, and this is the list it will use. (If the author was not having Demonstr8 induce the rules, he or she could drag the WME that contains such a list and drop it in the box below where it says, “If this constraint can be met:” and the system will generate the line shown.) In the last section it was mentioned that Demonstr8 has knowledge about how to reason with lists—what it means to be rightmost or left of an item in a list. The first pop–up menu under “If this constraint can be met:” contains such relations. The word after the pop–up is the Classname of interest (in this case, Column). The second pop–up is a list of the slot names associate with the Classname of interest (Top, Bottom, and Answer for the present example). The third pop–up of comparison operators and its accompanying text box operate like they do in the other lines. The interpretation given to this line in Figure 4, which has been set to the right selections, would be, “This production will match to: The rightmost column whose answer property contains a blank.” (The default is to set the first pop–up to `rightmost,’ since arithmetic strategies usually go from right to left, the second pop–up to the first item in the list—Top, in this case—and to leave the box empty. This is where the two changes had to be made from the default assumptions—change Top to Answer, and put Blank in the box). Wherever Any Column appears in the following “If” and “Then” parts of the parts of the production, it will take on the value of that column.

Goal and Skill

When the Goal and Skill radio button is selected, the main part of the Production Dialog changes to two simple sections. The top section contains two checkboxes and one text box and states what the topmost goal of the system currently is (Demonstr8 defaults to “Do Arithmetic” as the topmost goal). As the author demonstrates the actions that make up a skill, subgoals may have to be set and satisfied. The top checkbox states: “I am now going to do this:” and the text box follows. When the author is going to start some specialized procedure, like borrowing in subtraction, after the just–specified action, a subgoal may need to be set. This indicates to the system to place the indicated subgoal on top of the goalstack, and subsequent productions will be in service to that subgoal. Once the author has

demonstrated an action that accomplishes this subgoal, the second checkbox can be checked, which states, “I have completed this subgoal.” This will remove the topmost goal from the goalstack. Subgoaling and keeping a goalstack is necessary for when the same action occurs in different contexts (e.g., crossing out a number to add ten to it versus crossing out a number to decrement it). A current research aim is the automatic detection of subgoals and subgoal completion.

The second part of the dialog allows the author to indicate which skill this production supports. As mentioned in the Introduction, one of the features of the ACT Tutors is the Skillometer, a list of skills that are being taught and the probabilities that the skills are in a learned state. This tool supports such a Skillometer. The author can indicate if the current production supports a skill that has already been identified (via a pop–up menu listing skills previous productions have supported), or if this production supports a new skill (by typing the skill’s name in a text box).

Help

Selecting the last radio button allows the author to enter help text specific to the current production. When selected, the main part of the dialog will display four text boxes, labeled Levels 1 through 4. Each level corresponds to a more specific hint that the tutor will provide when the student asks for help repeatedly at the same spot (the ACT Tutors generally provides three levels of help). The help text can contain variables in order to provide context–sensitive help. The values of the variables will either be a specific number (when the variable refers to a slot name whose value contains a cell) or the text of a WME’s help tag (when the variable refers to a specific WME). The variable names are preceded by an equal sign, and the actual name must refer to a name from the Conditions and Action part of the dialog (these names are listed at the bottom of the Help part to aid the author). In the current instance, =top, =bottom, =answer, and =column are valid variable names, generated by Demonstr8 based on the slot names of the appropriate classnames. The value of the first three would be the value of a cell, and the value of =column would be its help tag text. An example of explicit help message would be, “You must subtract =bottom from =top in the =column column,” which might be instantiated as “You must subtract 2 from 7 in the ones column,” if a student asked for help.

While the preceding section has been long, the author has had to perform very few actions in demonstrating this production. They were:

- Option–clicking the Answer cell of the ones column
- Indicating that a “5” should be placed there
- Completing the resulting production dialog box:

- In the Condition and Actions part, changing the second pop–up menu to read “Answer” and dragging “Blank” from the WMEs window to the box in the first line
- In the Goal and Skills part, typing a new skill name (“Subtract”) in the bottom box
- In the Help part, typing in the help text

The author continues demonstrating how the problem should be solved, specifying the productions as needed. For a subtraction tutor that does not handle borrowing, the production just specified is sufficient for the task. For subtraction with borrow, six productions are needed. An easy way exists to specify when the problem is solved. One could imagine writing a specific Done production (realizing when a problem is solved is a skill that students need to learn), but for most problems in the domain of arithmetic, problems are done with then leftmost cell is filled. The system has an implicit Done production, whose condition is satisfied when that cell is filled (the author indicates that cell by selecting it and then choosing a menu command). Once the productions and the Done Cell have been specified, the author can have the system Auto–Solve a problem by simply generating a new problem and selecting “Auto–Solve” from the Author Menu. The system will work through the problem, filling in the appropriate cells as it goes along. This provides the author an opportunity to see if the specified productions are sufficient for the desired curriculum. If a mistake is encountered, the author can fine–tune the faulty production (double–clicking on a production’s name will bring up the Production Dialog), or it may be necessary to record additional actions if they were not demonstrated. In general, knowing when enough example solutions have been demonstrated is difficult, particularly as the domain gets more complex. For those complex domains, the best recourse is to simply test the tutor in the real world and see where the deficiencies lie. When some are found, the missing productions can easily be demonstrated and added to the system.

The preceding description of how to create productions could also apply to the creation of what are termed “buggy” productions as well (Brown & Burton, 1978). In the course of learning a domain, a student may develop a misconception about a particular step in the problem solving process. These misconceptions are “buggy” pieces of knowledge that the student has. There is a set of common misconceptions for any domain, and the author may want to target instruction for when a student apparently possesses one of these misconceptions. A common misconception in subtraction is that one always takes the larger number from the smaller, even if the larger one is on bottom (thereby eliminating the need for borrowing). While Demonstr8 does not currently have this feature (though

having buggy productions with specific remediation is a feature of the ACT Tutors), one could easily imagine a menu toggle indicating that the next action that the author is going to demonstrate is a buggy production, and so should not be considered a correct step. When the student is using the tutor, if he or she performs one of the buggy productions the proper help text could be displayed.

USING THE TUTOR

Once the author is satisfied with the productions that have demonstrated, the system can be put into "Student Mode" by selecting that option from the Author Menu. The screen will then appear as in Figure 5.

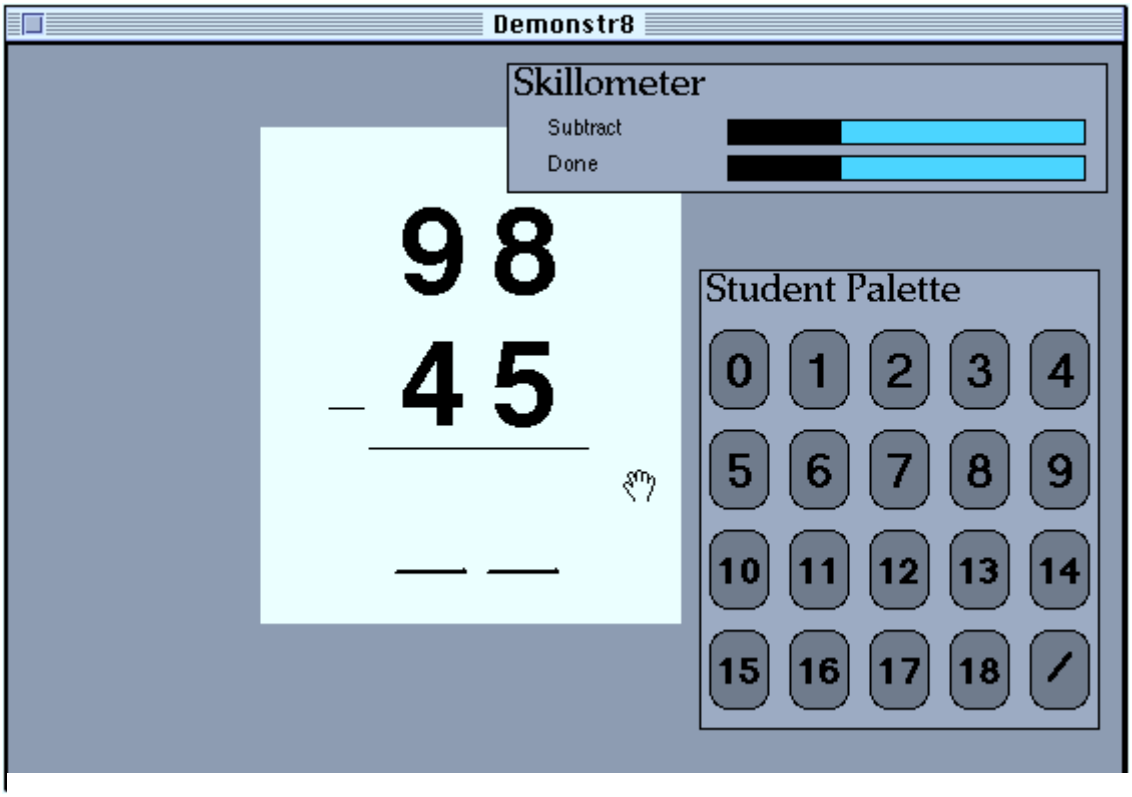


Figure 5. The finished tutor's interface

The authoring windows have disappeared, and the Author Menu is replaced with the Student Menu, which has three options: Help, Done, and New Problem. The Student Interface has been modified to be more presentable to a student, and two windows have been created: Skillometer, which contains a list of all the skills the author specified, and the Student Palette, which contains the numbers 0 – 18 and a slash. Next to each skill in the Skillometer is a bar graph indicating the probability that the skill is in the learned state. The numbers and slash of the Student Palette can be dragged from the palette and dropped onto the problem in the appropriate (or not appropriate) places. In operation, this tutor behaves essentially like

one of the ACT Tutors. The student can ask for help at any time, and the tutor will respond with an appropriate, context–sensitive help message. When the student performs an action, the tutor will check that action against the productions specified by the author. If the action is correct, the appropriate skill is incremented. If incorrect, the action is not allowed and the appropriate skill decremented. Once all of the skills have more than a 95% chance being in the learned state, a dialog box appears congratulating the student on mastering the tutored domain.

SUMMARY

This section has described how Demonstr8 can be used to create a simple model–tracing tutor for multi–column subtraction without borrowing. The total time the author spent creating the tutor was less than 10 minutes. To extend the tutor to problems with borrowing, the author would need to demonstrate 5 more productions, embodied in a single example, which would add no more than 10 minutes to the tutor creation time. Creation of the tutor involved using a set of MacDraw–like tools to design the student interface, defining working memory elements through a simple–to–use dialog, and finally demonstrating the underlying rules of the domain by working with the student's interface, with only minor additions by the author.

GENERAL DISCUSSION

The three parts of creating a tutor in Demonstr8, designing the student interface, creating working memory elements, and demonstrating productions, can each be evaluated with respect to two questions: 1) How easy is it to do in the current system, and 2) How general is the technique? The following sections consider each of these parts and the questions in turn, followed by a discussion of making these three parts separate tools.

GENERAL PRINCIPLES WITHIN DEMONSTR8

Designing the student interface

The tools in Demonstr8 to design the interface the student will use are extremely simple (essentially containing widgets only for drawing lines and cells), but they are sufficient to create any kind of arithmetic interface. Any person familiar with using a computer drawing package could create a student interface within Demonstr8. (Remember, the alternative in the TDK would be to code LISP statements describing lines and cells—a task doable only by programmers.) Being able to create a student interface using a

drawing–like package creates a situation in which designing the interface is not only easy to do, but it is also quite general. A sophisticated interface can be easily created using such methods. Several off–the–shelf, commercial packages (e.g., Apple's Hypercard or Macromedia's Director programs) allow designers to easily create interfaces with sophisticated widgets (pop–up menus, radio buttons, etc.) that could be used as student interfaces in tutors. Some even allow for plug–in widgets, so that a programmer could create something like the cell tool, which an educator could then have available in designing a tutor.

However, designing an interface that maximizes the time spent by the student in learning the domain, and not in learning the interface, is not trivial. There are human–computer interface design issues that need to be taken into consideration, but that the average educator could not be expected to have. There are a few courses of action that could be taken. First, packages of plain interfaces could be made available, and the domain experts could modify them to suit their needs. Lewis, Milson and Anderson (1987) described an algebra tutor that allowed teachers to slightly modify the interface and the tutoring method (though no provision was made for the teachers to actually augment the underlying production rules). Alternatively, the application used by the educator to create the interface could offer suggestions about the interface, or maybe even not allow some interface choices. Finally, either an interface designer could work with the educator in creating the interface, or perhaps the educator could be given some instruction beforehand in proper interface construction.

Creating working memory elements

This is arguably the most important step in creating a tutor using Demonstr8, since the working memory elements as defined by the author serve as a major determiner the format of the resulting productions. It is important to use the right representation for the WMEs, so that the induction method can successfully produce the right productions. As realized within Demonstr8, this process isn't much more difficult than using a drawing package like discussed above, but to do it correctly still requires more knowledge of cognitive science and production systems than a typical educator would have. To take the example of creating the subtraction tutor, why was it important that each column be identified and segmented into the three cells (Top, Bottom, and Answer)? Why did those columns need to be grouped into a problem? The answer to both questions is so that productions could be created that would sufficiently solve the problem. The realization that such WMEs would be needed comes from introspection as to how the problem is segmented in order to solve it, as well as experience in designing such systems. It may also come from listening to people solve such problems, and finding out what sorts of

things they mention in the course of the solving. To make this task doable by educators, some instruction in designing WMEs may inevitably have to be given. However, things could be done within the tool to make this process easier, such as having it ask questions or make intelligent guesses as to the correct way of parsing the interface as drawn. Also, the method itself could be made easier, perhaps by doing away with the dialog and having the author construct and group the WMEs on the interface itself.

In terms of the generality of this method of designing WMEs, that of visually defining them as opposed to typing out statements in a LISP document, it can be quite general. Most WMEs represent things that can be pointed to, like a column of numbers or the symbols that make up an algebraic equation. That being the case, they can be selected, grouped, and named, and their parts can be separably identified. For WMEs that have no on–screen counterpart, perhaps a representation can be created to allow them to be done so. In Demonstr8, the addition and subtraction facts are represented as tables, and these tables can be easily created and modified. A table format can perhaps represent many such abstract WMEs. For WMEs that represent goals and other declarative information without an easily represented form, the best representation may be something akin to the currently used list form in the TDK.

Demonstrating Productions

Once an adequate set of WMEs have been defined, actually generating the productions is relatively easy. The author merely has to perform the task, and for each intended student action, make sure that the created production is at the right level of specificity, assign it to a skill, and write help text for it. With Demonstr8 inducing the structure of the production (i.e., what WMEs are being referred to), little or no fine–tuning of these productions should be needed, though the author will need some training in understanding the induced production, in case a slight modification is needed. Creating a production without the induction method (as in the tool described in Blessing, 1995) is moderately difficult and requires an adequate understanding of production systems. Also, if the skill requires subgoaling, those have to be indicated at this point as well. Similar to WME creation, instruction on when to use subgoals may have to be given to the educator –author as well.

Given that the induction method employed by Demonstr8 is based on the analogy mechanism of ACT–R, it should be as general as that mechanism is. As stated in the Introduction, one of the strong claims of the ACT–R theory is that all productions are created by this mechanism, based on the contents of declarative memory (i.e., existing WMEs). By embodying such a mechanism within a tool like Demonstr8, a strict test of that claim can be made.

COMPONENT ARCHITECTURES

As described, Demonstr8 is an all–inclusive system. It contains the components necessary to design the student interface, create working memory elements, demonstrate the productions, and tutor the student. However, other researchers have argued for a component–system approach to tutor design (e.g., Ritter & Koedinger, 1995). The pieces of Demonstr8 are amenable to such a view.

In a component architecture system, the different parts are actually separate applications which communicate by sending messages back and forth (e.g., by using AppleEvents on a Macintosh). This allows for the student interface to actually be a piece of off–the–shelf software like Microsoft Excel, which would report the student's interface actions to a tutoring agent (a separate application, designed apart from Excel, and usable in the context of other spreadsheet applications). This agent then judges the correctness of the student's actions, which it would then report back to Excel.

Applying such a model to Demonstr8, the component that creates the student's interface could be anything—something simple like what exists now in Demonstr8, a program like Macromedia's Director, or some other off–the–shelf solution. The only restriction is that the component must be able to tell other applications what the student is doing (in AppleEvents nomenclature, be recordable), and also other applications must be able to tell it what to do (i.e., be scriptable) and be able to request information from it, so that when the student is using the interface, the interface can be updated correctly if the student makes a mistake.

A second tool, essentially the guts of Demonstr8, would be needed that aided the author in creating WMEs and productions. This tool would be able to parse the student interface in order to create WMEs, and also allow the author to create higher–order WMEs (e.g., columns for multi–column subtraction) by grouping the existing WMEs in the interface. Furthermore, this tool would also need to support the creation of more abstract WMEs, such as addition and subtraction facts, and any goal structures needed to solve the problems. Once those are in place, the author could create the productions by putting the interface application in "before" and "after" states (in the course of solving an actual problem), and the tool would induce the production needed to get from the "before" state to the "after" state. The output of the tool would be a set of productions usable by a tutoring agent in a component architecture system.

BEYOND DEMONSTR8

With the current tool only arithmetic tutors can be created. What about other domains? I have argued that the techniques embodied within Demonstr8 are usable to design other systems with which other tutors could be created. This section considers the form of those other systems could take in different domains.

The main requirement for a tool like Demonstr8 to work is that successive states in the problem solving process be concretely represented. Let us consider what this means in algebra—a domain similar to arithmetic, but different and complex enough to be worthwhile creating another authoring tool for. Within algebra, representing the successive states of the problem solution is quite natural. An author should be able to demonstrate the following problem:

$$3x + 5 = 14$$
$$3x = 9$$
$$x = 3$$

and have the system infer the productions needed to solve such problems (i.e., first subtracting from both sides and then dividing by the coefficient in order to solve for x). Neves (1978) actually developed a system, called Alex, for doing just this, and in a manner similar to Demonstr8. Neves' concern in constructing Alex was how a system could learn from textbook examples, similar in most respects to the main concern of Demonstr8. Alex implemented Newell and Simon's (1972) GPS approach to problem solving. It had available knowledge of simple arithmetic, and when given an example like the one above, could infer the rules behind it by seeing what had changed between pairs of lines in the example (i.e., what has been removed or transformed between the left and right sides of the equation).

Using manipulations like those described in creating the subtraction tutor, one could imagine the author indicating to the system the general form of equations by placing cells that could contain numbers, variables, or operators onto the worksheet, and then grouping those cells into terms, the terms into sides, and the sides into an equation. Having created the necessary working memory types and elements, the author could demonstrate how the problems should be solved by dragging and dropping those elements to the next line on the worksheet. In going from Line 1 to Line 2 in the example presented above, the 3x stays the same on the left–hand side, and the 5 and the 14 get combined (the result of applying a Knowledge Function) to generate the 9. Likewise, in getting the solution, the x stays on the left–hand side, but the 3 and 9 are used to generate the answer, 3.

However, not all domains are equally amenable to having tutors constructed in them using Demonstr8–like tools. Successful ACT Tutors

have been built for algebra word problem solving and algebra equation solving, and, as shown above, versions of these tutors could be authored using programming by demonstration methods. Even the tutors for programming languages could be demonstrable (Neves had Alex learn LISP by example). However, programming a geometry tutor (Koedinger & Anderson, 1993b; Anderson et al., 1985) by demonstration may not be straight–forward. These tutors taught how to do geometry proofs, and the student could work on the proof either forwards or backwards, by applying the requisite theorems and axioms necessary to do the proof. The problem in trying to induce the productions necessary lies in the fact that there is much knowledge between successive states of the problem solving process that is not directly observable in the interface—much of the reasoning behind why certain theorems were applied when is not represented in the interface, but rather is inside the solver's head (Koedinger & Anderson, 1989). This indicates that it may be necessary in some domains to create the production rule set in an interface different than that the student will use—perhaps one that has been augmented to indicate the overall plan that is being implemented to solve the current problem.

In general, the challenge in using the techniques discussed in this paper to create an authoring tool lies in finding the correct representations to use. For arithmetic the challenge was perhaps simpler than it would be for other tasks, but above I have argued that it would be just as straight–forward for some (algebra and even programming) and more difficult for others (geometry). For any task, the difficulty in applying these techniques arises when whatever is implicit in performing the task must be made explicit. In Demonstr8 this was satisfied by the use of Knowledge Functions, and I believe these would generalize to many domains. However, it will only be constructing authoring tools for other domains that we will be able to see what techniques truly are general and which are specific for a particular task.

CONCLUSION

This paper has described a tool which makes the creation of model–tracing tutors much easier than currently realized (e.g., by the Tutor Development Kit). The goal of the research is to empower any domain expert, even an educator in a classroom, to be able to create an intelligent tutor in the domain in which they are expert. The current tool, Demonstr8, falls somewhat short of that goal, since some training is still required, but contains some techniques, such as using ACT–R's analogy mechanism to induce the production rules, that places us closer to that ideal.

Author Notes

The work presented in this paper is based upon work supported by the National Science Foundation and the Advanced Research Projects Agency under Cooperative Agreement No. CDA–940860. I would like to thank Marsha Lovett, Steven Ritter, and three anonymous reviewers for their comments on earlier drafts of this paper, and also John Anderson and Jim Spohrer for their help and suggestions with this work.

References

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

Anderson, J. R. (1992). Intelligent tutoring and high school mathematics. In C. Fasson, G. Gauthier, & G. I. McCalla (Eds.). *Proceedings of the Second International Conference on Intelligent Tutoring Systems*. Spring–Verlag: Berlin, Germany.

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.

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

Anderson, J. R., Boyle, C. F., & Yost, G. (1985). The geometry tutor. In *Proceedings of the International Joint Conference on Artificial Intelligence—85*. Los Angeles: IJCAI.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). The cognitive tutors: lessons learned. Journal of the Learning Sciences, Vol. 4(2), 167–207.

Anderson, J. R., Boyle, C. F., Corbett, A. T., & Lewis, M. W. (1990). Cognitive modelling and intelligent tutoring. *Artificial Intelligence*, *42*, 7–49.

Blessing, S. B. (1995). ITS authoring tools: The next generation. In J. Greer (Ed.), *Proceedings of AI–ED 95–7th World Conference on Artificial Intelligence and Education* (p. 567). Charlottesville, VA: Association for the Advancement of Computing in Education.

Brown, J. S., & Burton, R. (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, *2*, 155–192.

Corbett, A. T., & Anderson, J. R. (1990). The effect of feedback control on learning to program with the LISP tutor. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Cypher, A. (1993). *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA.

Koedinger, K. R. & Anderson, J. R. (1989). Perceptual chunks in gemoetry problem solving: A challenge to theories of skill acquisition. In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Koedinger, K. R., & Anderson, J. R. (1993a). Effective use of intelligent software in high school math classrooms. In *Proceedings of the World Conference on Artificial Intelligence in Education*, 1993. Charlottesville, VA: AACE.

Koedinger, K. R. & Anderson, J. R. (1993b). Reifying implicit planning in geometry: Guidelines for model–based intelligent tutoring system design. In S.P. Lajoie and S.J. Derry (Eds.) *Computers as Cognitive Tools*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Lewis, M. W., Milson, R., & Anderson, J. R. (1987). The Teacher's Apprentice: Designing an intelligent authoring system for high school mathematics. In G.P. Kearsley (Ed.) *Artificial Intelligence and Instruction: Applications and Methods* (pp. 269–301). Addison–Wesley Publishing Company: Reading, MA.

Myers, B. A., McDaniel, R. G., & Kosbie, D. S. (1993). Marquise: Creating Complete User Interfaces by Demonstration. In *Proceedings of INTERCHI '93: Human Factors in Computing Systems*, April 24–29, 1993.

Nardi, B. A. (1993). *A small matter of programming: Perspectives on end–user computing*. Cambridge, MA: MIT Press.

Neves, D. M. (1978). A computer program that learns algebraic procedures by examining examples and by working test problems in a textbook. *Proceedings of the Second National Conference of the Canadian Society for Computational Studies of Intelligence*.

Newell, A., & Simon, H. A. (1972). *Human problem solving*. Englewood Cliffs, NJ: Prentice–Hall.

Ritter, S., & Koedinger, K. R. (1995). Towards lightweight tutoring agents. In J. Greer (Ed.), *Proceedings of AI–ED 95–7th World Conference on Artificial Intelligence and Education* (p. 567). Charlottesville, VA: Association for the Advancement of Computing in Education.

Smith, D. C., Cypher, A., & Spohrer, J. (1994). KidSim: Programming agents without a programming language. *Communications of the ACM, 37*(7), 55–67.