AN ABSTRACT OF THE THESIS OF

Brian Lantz for the degree of Master of Science

in Computer Science presented on the ninth day of July, 1982

Title: A Tutorial Interface for a Problem-Solving Production System

Redacted for Privacy

Abstract approved:

Dr. William S. Bregar

The system described is an interface between a student and a problem-solving production system that solves some class of problems. Its purpose is to help a student learn some part of the realm of problem solving. As a student attempts to solve a problem the system controls the firing of productions in the problem-solving production system in such a way as to parallel the solution proposed by the student. The problem-solving production system, paralleling the student's solution, serves both as a model of the student's partial solution and as a source of expert problem-solving knowledge for offering advice to the student. If the student asks for help, the system uses the history of the problem solution and the knowledge of the expert problem solver to provide assistance.

A Tutorial Interface for a Problem-Solving Production System

by

Brian Scott Lantz

A Thesis

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed July 9, 1982

Commencement June, 1983

APPROVED:

Redacted for Privacy

Associate Professor of Computer Science in Charge of Major

Redacted for Privacy

Chairman of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented The ninth day of July, 1982

Typed by Brian Lantz for Brian Lantz

Table of Contents

List of Figures

A Tutorial Interface for a Problem-Solving Production System

Chapter 1

Introduction

1.1. Overview of the System

The goal of this project is to develop a computer system that possesses some of the capabilities of a tutor. A tutor should be able to understand a solution proposed by a student, rather than imposing the tutor's own solution scheme on the student. A tutor should always be available to lend assistance when it is necessary, but should allow the student to learn by doing whenever possible. The tutorial system, to be described here, possesses these capabilities.

The system is implemented as an interface between a student, and a production system (PS) that solves some class of problems. The interface acts as a tutor whose area of expertise is defined by the problem-solving PS. The expert PS that is currently used with the interface solves elementary algebra problems. As the student attempts to solve a problem, the interface controls the firing of productions in the problem-solving PS in such a way as to parallel the solution proposed by the student. Thus the state of the problem-solving PS serves as the representation of the system's model of the student's problem-solving behavior. In the algebra problem domain the productions of the expert PS perform the operations required for solution of elementary algebra problems. A sequence of algebraic operations performed by a student can be modelled by a sequence of production firings in the expert PS.

The interface is designed to be easily matched with an arbitrary problem-solving production system. The Tutorial interface and the production system monitor are implemented in LISP. It is possible to link an arbitrary problem-solving PS with the

interface simply by loading them both into the same LISP system, though this does not yield the best results. The performance may be improved, however, by a certain amount of tailoring, as is described in section 5.3. Since the problem domain is determined by the associated problem-solving PS, rather than the interface itself, the tutorial interface can be adapted to tutor in any problem-solving domain for which a problem-soiving PS can be produced.

The system operates in a cycle where the student is asked to specify the next step in a problem solution, and the interface tries to find an applicable step in the problem-solving PS that parallels the student's step. If such a step is found it is performed, and the results are presented to the student. If no such step is found, or the student asks for help, the system uses the history of the problem solution and the power of the problem solver to provide assistance. The following text refers to the sample protocol in figure 1-1 to illustrate the operation of the interface.

The most interesting feature of the interface is its ability to form a model of a student's problem-solving behavior, and then to use this model for providing advice to the student. The modelling technique used allows the interface to compare the past behavior of the student to that of an expert computer system that solves problems in the domain of interest. The interface can also query its expert system to determine what problem-solving step it would apply in a situation where the student has not yet made a decision as to what to do next. This is obviously of great use for giving advice to the student.

In the sample protocol of figure 1-1, the system starts out by asking the student for information about the problem to be solved. The previously described cycle wherein the student directs the interface to perform some operation followed by the interface performing that operation, if possible, can be seen from viewing the figure. The exception to the rule occurs when, as in paragraph 7, the student indicates that he needs help. Here, the interface has noticed that something was done and then undone again, and so it backtracks to rectify the situation.

The incorrect operation in paragraph 6 effectively undid the operation of paragraph 5. This mistake occurred because the production in the expert PS that isolates the goal variable always assumes that there is only one occurrence of the goal variable on the left-hand side of the equation. This assumption is always true when the expert PS is running under its own control. This problem serves to illustrate the need for developing an expert PS that is specifically designed for operation with the tutorial interface.

The ability of a student to undo previous operations is demonstrated in paragraph 12. The step is repeated in paragraph 13 to show that the backtracking was successful.

Paragraph 14 illustrates the ability of the interface to give some additional information when it rejects a problem-solving operation as being inapplicable in the current problem step. Unfortunately, in this example the inappropriateness of the operation is a function of the expert problem solver that is being used, rather than any mathematical reason. The production in the expert PS that corresponds to a division operation is constrained to only function on a goal term on the left-hand side of an equation.

The student asks the system to provide information concerning what operations would be appropriate, first in paragraph 15, and again in 16. Notice that the first time that he asks, he is given only a hint. On the second time, the system gives the operation away by performing it for the student. After the intervening operation has been executed, the original directive can be honored.

Another odd behavior that stems from the use of an expert PS whose development was independent of the tutorial interface is illustrated in paragraph 18. The algebra problem solver uses a means-ends strategy to solve a set of equations for the value of the goal variable. For each non-goal variable in the equation the expert PS will generate a subproblem whose goal is to find a value for that non-goal variable. Paragraph 18 is the commentary from the expert PS while it is generating the subproblem whose goal is to find the value for the variable x.

The use of a directive that indicates more than one operation is shown in paragraph 21. This directive instructs the interface to first substitute the value 1 for the variable 'x' in the equation. It then causes the resulting constants to be added together, which yields the final answer for the problem. The comment about the need to simplify the problem was generated by the expert PS when it determined that all of its outstanding subproblems had been completed. When the algebra problem solver was developed as a stand alone system, the comment was meant to provide a trace of the problem solver's operation.

Figure 1-1: Sample protocol

1. *(tutor algebra)
   Type in the variables for your problem: x y

2. Ok, which variable must we solve for? y

3. So Y is our goal variable.
   Please input new information about Y
   Type in an equation containing the goal: 8y-3x=14-x+y+5

4. Ok! We have new information about Y.
   The equation is not simplified in terms
   of the goal, so you need to simplify it.
   What do you want to do? eat chocolate chip cookies
   The system is at a loss as to how to proceed.
   Please reword your command, or request help if you are stuck.

5. What do you want to do? move the y term to the left hand side
   Moving the goal term to the left side
   produces the equation 8Y-3X-Y=14-X+5

6. What do you want to do? isolate the goal term
   Isolating the goal variable on the left side
   produces the equation 8Y=14-X+5+3X+Y

7. What do you want to do? help
   It looks like we need to back up a bit.
   Lets go back to just after
   Moving the goal term to the left side
   produces the equation 8Y-3X-Y=14-X+5
   Try doing something involving Y

8. What do you want to do? add 3x to each side
   Combining terms from opposite sides involving X
   produces the equation 8Y-Y=2X+14+5

9. What do you want to do? isolate the goal
   Isolating the goal variable on the left side
   produces the equation 8Y=2X+14+5+Y

10. What do you want to do? combine the goal terms
    Combining terms from opposite sides involving Y
    produces the equation 0=-7Y+2X+14+5

Fig. 1-1, continued

11. What do you want to do? add the constants
    Combining constant terms on the right side
    produces the equation 0=19-7Y+2X

12. What do you want to do? undo 1 step
    Backtracking over 1 step.

13. What do you want to do? add the constants
    Combining constant terms on the right side
    produces the equation 0=19-7Y+2X

14. What do you want to do? divide everything by 7
    You need to do something else before you can do that.

15. What do you want to do? what should i do
    Try doing something involving Y

16. What do you want to do? help
    I'll do the next step for you.
    Moving the goal term to the left side
    produces the equation 7Y=19+2X

17. What do you want to do? divide everything by 7
    Making the coefficient of the goal term equal to one
    produces the equation Y=19/7+2/7X

18. We notice that the variable X appears
    in the equation Y=19/7+2/7X
    So we will consider X to be our new goal.
    We must solve for it in order to solve for Y.
    We will remember that goal and go back to it later.

19. Please input new information about X
    Type in an equation containing the goal: x = 1

20. Ok! We have new information about X.
    We see we have determined a result.
    X is equal to 1
    So we return to the parent goal variable, which is Y

Fig. 1-1, continued

```
21. What do you want to do? substitute for x and add the
    constants on the right
    We can substitute 1 for X
    The equation is now Y=19/7+2/7
    The equation is not simplified in terms
    of the goal, so you need to simplify it.
    Combining constant terms on the right side
    produces the equation Y=3

22. We see we have determined a result.
    Y is equal to 3
    Our final result is Y equals 3
```

Note: Keyboard input has been underlined for this example.

1.2. Impetus

1.2.1. Computer Aided Instruction

Computer Aided Instruction (CAI), as the name implies, is concerned with using computers as educational aids. The purpose of educational tools is to aid in the creation of a good learning environment. Factors involved in a good learning situation include a motivated student, instruction tailored to the needs of the individual, and frequent feedback (Smith, 1974, Mitzel, 1970). Some CAI systems provide simple drill and practice functions. These give frequent feedback, but little individualized help. Many follow the idea of *programmed texts*, providing alternate information and problems dependent upon previous success or failure to correctly respond to questioning. Programmed text type systems provide frequent feedback, and also allow alternate paths through the curriculum, dependent upon individual needs.

An educational opportunity unique to CAI is that provided by computerized simulations (Hoffer, Barnett, Farquhar, and Prather, 1975, Smith, 1974, Seltzer, 1971). A student can learn from his mistakes while studying such things as highly reactive chemical systems, without suffering the disastrous consequences.

Simulation is used heavily in such forms as nuclear reactor simulators, flight simulators, and medical patient simulations.

The introduction of Artificial Intelligence (AI) methods in CAI research has led to the expanded area of *Intelligent* CAI (ICAI). ICAI systems bring many new features to CAI. Expert systems allow ICAI systems to include expert knowledge about a domain. The decisions made by a student may be compared to those of an expert system in order to evaluate the correctness of a student's judgment. AI techniques can be used to maintain models of a student's knowledge or strategies. Models of student knowledge are useful for focusing instruction on the areas where it is most needed. Strategy models allow an ICAI system to follow a student's strategy rather than imposing a fixed strategy on the student. ICAI systems can be constructed in which knowledge concerning tutoring strategies is kept separate from the expert knowledge of the problem domain. This allows for a great deal of effort to be put into optimizing the tutorial expertise, knowing that it will be applicable over a range of problem domains (Clancey, 1979a). AI also brings methods of knowledge representation to ICAI. ICAI systems may include, for example, factual knowledge represented in semantic networks, or procedural knowledge represented in finite-state machines (Brown, Burton, and Zdybel, 1973). ICAI systems may also communicate with the user in quasi-natural or natural language.

As mentioned above, AI provides ICAI system designers with tools for gathering, manipulation, and representation of the data that allows for the modelling of a student's knowledge (Clancey, 1979a, Clancey, 1979b, Brown and Burton, 1978, Sleeman, 1977, Sleeman and Smith, 1981, Sleeman, 1982). A good model of a student allows the system to determine what the student does know, what he does not know, and what incorrect knowledge he may believe. The system is then able to concentrate on the correction of the student's misconceptions, and the presentation of knowledge the student does not already possess. The student is not bored by repetition of things that he already knows. Diagnosis of a student's misconceptions leads to early correction, and a great savings of time and effort for the student as well as the instructor (Mitzel, 1970).

To be acceptable to users, communication with an ICAI system must be simple, fast, and accurate. Keyword systems are accurate, but not generally easy for new users. Many natural language systems are not accurate, and loss of accuracy often makes usage laborious. Systems that require the designers to second guess all possible responses and errors are generally unacceptable. When using a *second guess* type of system the designer must either heavily restrict user responses or

accept the consequent loss of accuracy associated with unrestricted response to such a system (Charp, 1970). Such *second guess* designs are also expensive to produce lessons for, thus rendering them impractical (Seltzer, 1971).

Gains in efficiency and capability may be obtained by separation of the Tutorial model from the problem domain (Clancey, 1979a, Clancey, 1979b). In this way the effort put into a tutoring model need not be repeated when a new problem domain is introduced. Changes to the tutorial model may also be made without close attention to each problem-solving domain. A primary advantage of this technique is that this allows the designer much more freedom to experiment with various tutorial strategies. Much time and effort may be put into developing a good tutorial model, with the knowledge that it can be used for many problem domains.

Many ICAI systems for problem solving are only capable of judging the correctness of the final result of a problem-solving procedure. Others are able to monitor the individual steps of the problem solution, but only when the student is constrained to a single solution strategy. A more flexible system will allow the student to pursue his own problem-solving strategy rather than constraining him to a single solution scheme. It should be able to assist a student who is learning to make his own judgments and decisions rather than simply learning a fixed procedure by rote. The system should understand the student's problem-solving behavior and be able to lend assistance to the student based on his partial or erroneous solution. Such needs have been long recognized by researchers in ICAI (Koffman and Blount, 1977, McKeachie, 1974).

1.2.2. Instructional Capabilities

The student of problem solving must learn two things. He must learn the operators necessary to solve the problem and he must learn to select the particular operator, in a given state, that will further the solution. The student will inevitably make mistakes. Carry, Lewis, and Bernard (1980) have divided the possible mistakes into three categories.

The first of these three categories are called *operator errors*. Operator errors consist of those errors resulting from the application of invalid operators. The subject may be trying to apply an operator that was given to him by an instructor, but which he misunderstood. Another source of invalid operators is the tendency of students to alter operators to conform to their previous experience (Chapman

and Chapman, 1959, Mayer, 1977). For example, if a student notices that a new operator has many of the properties of another operator that he already knows then he may attribute some of the additional properties of the more familiar operator to the new operator.

It is possible for an operator to be invalid, yet yield correct results for many cases. If the special cases for which an operator does not work are few, it may be very difficult to determine that the student is using an operator that is invalid.

The second class of errors are *applicability errors*. These errors happen when the subject chooses a valid operator, but the operator does not further the solution of the problem. This is the case in which the student knows how to execute the required operators, but has problems deciding when a given operator is appropriate for solving a problem.

The third class of errors are the *execution errors*. These are simply those errors that occur when the correct operator is chosen at the correct time, but some error is made while performing the operation. An example of this would be an addition error while adding a term to both sides of an equation.

The implementation of the tutorial interface described in this thesis assumes that the student already has knowledge of a complete set of operators for solving the class of problems in the repertoire of the problem-solving PS. The operators possessed by the student need not all be valid, however. The student may have gained knowledge of the operators through conventional classroom study, or possibly through a more comprehensive CAI system into which the tutorial interface has been integrated.

The utility of the tutorial interface is in the learning of the selection of operators in multi-step problem-solving situations. The student can enter an interactive session with little or no knowledge of this facet of the solution of the given class of problems. Through a combination of the assistance rendered by the interface, and the student's own successful application of the operators as the session continues, the student can learn to select applicable operators in many situations. It bears repeating here, that the interface allows the student a large degree of freedom in his selection of operators, and follows whatever solution sequence is proposed by the student. The system can always determine whether the student's solution is progressing toward the goal. The flexibility of the interface allows the student to learn operator selection in a much more general environment than would be available if he were limited to a particular solution strategy.

The implemented system is intended to explore the detection and correction of the second type of error, the applicability errors. What of the first and third types? The third type, execution errors, will not be seen since the system performs the operations. The first type, the operator errors, can exhibit themselves in either of two ways. If the student's incorrect understanding of the operator leads him to think that it is applicable, when it is actually not, then this will show up as an applicability error. Otherwise, the case is that in which the student's invalid operator happens to be applicable at the same point as the valid operator known by the same description. If the student were to perform this invalid operation himself, it would lead to an error. Since the system performs the operation, the student sees only a correct application of the operator. The inability to detect some operator errors and all execution errors is a direct result of the decision to have the system perform the problem-solving operations.

If the student is required to supply the system with the results of his problem-solving steps, rather than simply directing the solution, then the system would have the ability to handle the full set of student errors. This change may be advisable for more practical versions of the tutorial interface. However, the current implementation is fully capable of demonstrating the use of the tutor's modelling technique.

1.3. Implementation Considerations

1.3.1. The Problem Domain

The problem-solving PS currently in use with the interface solves problems in algebra. Many algebra problems require several steps in their solution, and are thus good sample problems for this system. Algebra problems can be solved using a well-defined set of operators. Thus the set of operators that the prospective student is assumed to know when entering a tutorial session can easily be defined. Another advantage of choosing algebra problems is that constructing a problem-solving production system for algebra problems is fairly straight forward. Well-defined strategies exist for solving many classes of algebra problems (Bundy and Welham, 1981). The analysis of the algebra domain that was used in the development of the problem-solving PS used here (Bregar and Farley, 1978) has yielded a method that is similar to that proposed by Bundy.

The heart of Bundy's strategy is called the *Basic Method*. He defines three basic operators; collection, attraction, and isolation. Collection is the operator that combines two terms into a single term. Attraction repositions terms in the equation to where the collection operator can be applied to them. Isolation operates to produce an equation with the goal variable on one side of the equation, and everything else on the other side. The basic method is illustrated algorithmically in Figure 1-2. In the method of Bregar and Farley, the collection operators are designed in such a way that Bundy's explicit attraction is not necessary.

Figure 1-2: Bundy's Basic Method

```
While there is more than one occurrence of the unknown
      If collection is possible Then
         collect
      Else
         attract
Isolate the single unknown
```

In Bregar and Farley's algebra problem-solving system, control of processing is by a means-ends process. The implicit goal of the problem solver is to isolate some unknown variable which is the goal variable, and to determine its value with respect to the other terms in the equation. Once a single unknown has been isolated, the remainder of the equation is checked for additional unknown variables. If any is found, then a subgoal is generated whose purpose is to find a value for the unknown variable. The solution for a subgoal proceeds in the same manner as the isolation of the original goal variable. If additional information is required in order to satisfy a subgoal then a request for the necessary information is made to the terminal.

1.3.2. Production Systems

The Tutorial interface and the problem solver are both implemented as production systems. Production systems offer significant advantages for a system such as this. These advantages will be described in detail. Before that, however, will be a brief summary of production systems in general (Davis and King, 1975, McDermott and Forgey, 1978), and of the specifics of the particular production system that is used in the implemented system.

Production systems have three major parts, an interpreter, a database, and a set of production rules, or productions. A production is a rule of the form "If <condition list> then do <action list>." The LISP-based PS used by the interface is called PSMON (Rood and Bregar, 1980). In it, a production is written as "(if (condition) then (action))." A production system works by selecting production rules whose condition parts are satisfied by the current state of the system, and executing their action parts. The productions that are selected and whose actions are executed are said to have *fired*.

The production system interpreter, or monitor, of a production system works in a three-stage cycle. First it evaluates the condition parts of each production. The set of productions whose condition parts are satisfied by the current state of the database is called the conflict set. The second step is called conflict-resolution, and consists of choosing a subset of the productions in the conflict set that will be fired. For most conflict-resolution schemes, this subset has exactly one member. The final step is the firing, or the execution of the action parts, of the chosen productions.

One of the most common means of conflict-resolution treats the set of productions as an ordered list. Conflicts are resolved by choosing the production that is nearest to the top of the list. This is equivalent to looking down the list, checking condition parts, until finding the first production whose condition is satisfied. This is the default method of conflict-resolution used by PSMON.

The database for a production system may take many forms. The database serves the role of recording the current state of the system. One of the advantages of production systems is their state sensitivity. Given a complete production system, the next step will always be a function of the state information recorded in the database. A change in the state of the system is easily detected by way of the change in the conflict set, the set of productions which will be satisfied by the new state.

PSMON is a production system monitor intended for psychological modelling. The psychological modelling nature of PSMON is exhibited mainly in the structure and conventions of the database that is used. The database is structured to have four memories; short-term memory (STM), intermediate-term memory (ITM), long-term memory (LTM), and environmental memory (ENV).

The STM may be considered as an ordered list of separate pieces of information.

The length of STM is limited, usually to about seven items. Data that has been recently accessed may be found at the beginning of the STM. As new data is accessed the older data is pushed towards the end of the STM. If an item in STM is not accessed and pulled to the front of the list, it will eventually be pushed to the end of the STM. When it reaches the end, it is removed from STM and inserted into the ITM. This design helps reduce the memory that must be searched for information that is currently in use. The existence and length of STM in humans is supported by psychological evidence (for example, Miller (1956).)

The ITM is not limited like the STM. It will continue to grow as long as new information is added. The LTM contains information that remains constant throughout a given run of the system. In the tutorial interface, that information will be the knowledge necessary for problem solving, in the form of production rules. The ENV is an external buffer which holds data that is entering the system from the outside.

In PSMON, the condition part of a production not only matches data in the database, but it also *binds*, or associates, the matched data to specified labels. The bound data may be referred to at any point after it has been bound. This includes being able to refer to it in subsequent portions of the condition part as well as during execution of the action part. The scope of a binding is limited to the production in which is was bound.

Another interesting feature of PSMON is that it can be called recursively. That is, a production may specify in its action that a call is to be made to PSMON with an entirely separate database and/or set of productions. The use of this feature by the tutorial system is explained in section 2.1.2.

1.3.3. Modelling

In order to provide a system that can follow the student's problem-solving strategy it is necessary to build a model of the student's solution. There are various ways to go about doing this. One possible way would be to make a graph structure which is a complete network of all possible action sequences involving a given set of operators (Brown and Burton, 1978, Sleeman, 1982). Then the student's solution could be placed as an overlay on the net. Overlaying the student's solution would be done by simply marking off the nodes in the net that corresponded to the student's solution path (Clancey, 1979a). It can readily be

determined that this net could become quite large, even when limited to only valid solutions.

One way to limit the size of such a net would be to only include those nodes that are involved in a particular solution (Sleeman, 1977, Koffman and Blount, 1977). This would constrain the student to using the problem-solving strategy embodied in that net.

The use of production systems allows an alternative representation. Instead of building a net ahead of time, one or more productions in the problem-solving PS will be fired for each step in the student's problem solution. Each production firing is equivalent to the application of an operator to the current state of the problem solution. The sequence of production firings and the resulting state of the database is called a *trace,* and provides a model of the student's problem solving. This sequence of production firings can be viewed as the single path through the net that corresponds to the student's solution.

In a production system, the actions of the productions correspond to the primitive operations used in the solution scheme. Changing the level of detail of the actions in the productions can alter the *granularity* of the PS (Bregar and Farley, 1978, Davis and King, 1975). Experimenting with the granularity of the system is one way to match it to the needs of the intended students.

The set of productions chosen for modelling the student's solution is precisely that required for the PS to solve the problem on its own. Thus the problem-solving PS will act as an *expert system* in the domain of the problems it is designed to solve. An embedded expert system is, in fact, considered to be one of the central features of many ICAI systems (Clancey, 1979b). It seems intuitive that an intelligent system should be able to perform a task itself if it is to be used to instruct others in the performance of that task.

1.3.4. Remarks

This section has presented an overview of production systems and has tried to justify some of the design decisions made in implementing the tutorial interface. The implemented system utilizes a production system whose productions embody an expert system in the domain of algebra problem solving. The productions provide a means to model a student's problem solutions at any predetermined granularity.

The second chapter of this paper talks about the overall structure of the interface, while chapters three and four go into the details. Chapter five includes information about how to tailor a problem-solving PS so as to make the interface run as smoothly as possible. The final chapter discusses the inadequacies of, and possible improvements to, the system.

CHAPTER 2
STRUCTURE OF THE INTERFACE

2.1. The Static System

The tutor system consists of three components. The first of these is a problem-solving PS for the domain of interest. The second is the tutorial interface. The interface logically occupies the position between the problem-solving PS and the student. The third component is the production system monitor itself, PSMON.

2.1.1. The Tutorial Interface

The tutorial interface is implemented as a PSMON production system. Running both the tutorial interface and the problem-solving PS on the same monitor allows the interface to make full use of the specialized features of PSMON. The PS implementation of the interface also allows greater flexibility for experimentation with the tutorial model.

The interface can be considered as two functional parts, the *modeller* and the *model user*. The modeller is responsible for monitoring the student's solution and maintaining a model of that solution. The model user has the job of providing information to the student. The model user uses the model created by the modeller to help the student determine what the next step should be.

2.1.2. The Production System Monitor

The PSMON production system monitor provides several special features. One such feature is the ability to *push* and *pop* PSs. A push is equivalent to the current database and set of productions being pushed onto a stack, followed by a recursive call being made to the monitor with an arbitrary new database and set of productions. Conversely, a pop consists of an exit from a called PS and the

restoration of the original database and productions. The result of the push is deposited in the ENV of the database of the restored PS so that the state of the calling production system will be changed. The value returned by the push contains an ordered list of the productions fired during the push, and a copy of the database representing the final state of the pushed PS before it was popped.

Every invocation of the monitor, including those following a push to a new PS, also includes the specification of a flag indicating the mode in which the monitor is to interpret the specified PS. The normal calling mode causes the monitor to enter its primary loop where it generates a conflict set, resolves conflicts, and fires a production. This looping continues until the PS is explicitly deactivated, or a state is reached in which the generation of a conflict set yields an empty set. When called in normal mode, the monitor uses the default conflict-resolution scheme that considers the set of productions as an ordered list and choses the first encountered production whose condition part is satisfied.

Another mode in which the monitor may be called is *CS-generation mode* (Conflict Set generation mode). In CS-generation mode, the monitor does not enter its primary loop. Instead, a conflict set is generated and then returned, with all of its bindings, as the value of the call. Access to the conflict set provides the opportunity for the user to introduce his own conflict-resolution schemes into the system. The user determines which production, or set of productions, from the returned set that he would like to have fired. The specified production can then be fired by yet another push to the monitor. This call is made in *single firing* mode. The modified database is returned as the value of this push.

The interface makes use of these special features of PSMON to enable it to control the problem solver. The tutor can push to the problem solver to obtain conflict sets or to fire problem-solving productions. The problem-solving PS may, itself, include pushes to other subordinate PSs. When this occurs, the interface simply pushes to another copy of the interface with hooks to the subordinate PS to which the problem-solving PS was trying to push. In this way the interface maintains control of the problem solver, even when it involves pushes to other PSs during its execution.

2.1.3. The Problem-Solving PS

The productions of the problem-solving PS are divided into two categories, *foreground* and *background*. In a *pure* PS, each production is a representation of domain knowledge only. The *behavior* of the system is determined by the state of the domain at any particular time (Davis and King, 1975). Most working production systems fall short of this lofty goal, however, and resort to the introduction of productions that explicitly affect the flow of control in the system. Foreground productions are those that actually manipulate the problem components in order to move closer to a solution. These correspond to the steps of the problem solution which a student may be expected to specify in his directives to the system. Background productions are those concerned with maintaining the flow of control within the PS.

A simple method of maintaining the flow of control in a PS is to have productions place tags on the information that they place in the database. McDermott and Forgey (1978) note other methods of controlling flow within production systems that take advantage of inherent characteristics of the productions and data to construct more elegant control mechanisms. Tags are intended to be recognized only by a select group of productions in the PS. By this means a reliable, but inflexible, sequencing of production firings can be brought about. Similarly, a tag may be entered into the database with no other information attached to it. Such a tag may serve as a flag to other productions, either enabling or inhibiting them from firing.

Some of these flow of control actions may be associated with productions that also manipulate the problem state with solution operators. Others may be performed by productions whose only purpose is to set a flag when a particular condition becomes true. These flow of control operators are peculiar to the PS implementation of the problem solver. The foreground productions are used to model a student's solution. The flow of control (ie. background) productions are not problem-solving operators, and so are not used to model the student's solution.

Background productions could be used to model a student's flow of control, or problem-solving strategy. It is not necessary, however, for this tutorial system to place a name on the strategy used by the student. It needs only to be able to follow the problem-solving steps that a student takes. Further detail concerning the identification of a student's strategy can be found in Burton and Brown (1979).

For each directive from the student, the tutor performs the foreground problem-solving steps that are indicated, along with any background steps that are necessitated by the nature of the problem-solving PS. A detailed explanation of how this is done is given in the following section.

2.2. Flow of Control in the Interface

When the interface causes a conflict set to be generated from the problem-solving PS, that conflict set may contain any number of foreground and/or background productions. For purposes of modelling the student's solution scheme it is necessary to fire those foreground productions that correspond to the steps indicated by the directive from the student. Although the foreground productions are the focus of interest, the problem of when to fire the background productions must also be considered. The Interface implementation must address these two problems. It must provide for the firing of appropriate foreground productions, as indicated by the student's directive, while maintaining an appropriate mix of foreground and background production firings. The flow of control in the interface is illustrated algorithmically in figure 2-1.

Figure 2-1: The Major Cycle

```
Repeat
    Generate conflict set from problem-solving PS
    If All productions are background Then
       Choose one
       Fire it
    Else {* there is some foreground production in the CS *}
         If No text is held in the tutors DB Then
            Read the student's text
         Weigh the foreground productions in the CS
         If One can be chosen for firing Then
            Fire the selected foreground production
         Else {* no foreground production is indicated by the text *}
              If The directive has been executed previously Then
                 Purge the old directive from the tutors DB
              Else {* A new directive has just been read *}
                   Indicate failure to interpret the directive
Until The problem is solved
```

Once a conflict set has been generated, the algorithm of figure 2-1 has two major

branches. When the conflict set does not contain any foreground production, then one of the background productions in the conflict set is chosen and fired. The more interesting case is that in which there are foreground productions present in the conflict set.

The presence of foreground productions in the conflict set indicates that some problem-solving operations are applicable in the current state of the problem solution. The task is then to determine which possible operation the student wants to have performed, and to fire the foreground production that performs the same action as that directed by the student. If the student has not already entered a directive at the terminal then he is requested to do so at this time. The directive is used to assign a weight to each foreground production in the conflict set. If the most highly weighted foreground production in the conflict set receives a weight that is above the threshold required for selection, then that production is fired. The weighing functions are implemented so that no production is fired more than once by a given directive.

If no production can be selected for firing then there are two possible actions. If the directive used for weighing the productions has been used on previous cycles to weigh other conflict sets, then the directive is simply discarded. Since no production has been fired, the conflict set on the next cycle will be the same as that generated for the present cycle. The absence of a directive in the interface's memory will cause a new directive to be requested from the student. If the directive used in the unsuccessful weighing of the productions was read into the interface on the present cycle, then a failure to find an interpretation of the directive is reported. The directive is discarded, and another will be requested in the following cycle of the interface algorithm.

2.2.1. Foreground/Background Mix

Background productions provide for the flow of control within the problem-solving PS. Without the firing of the background productions, tags and flags that enable foreground productions would not appear in the database. Causing only foreground productions to be fired in the problem-solving PS may eventually starve the PS through a lack of the tags and flags normally made available by background production firing.

The problem of when to fire background productions must be considered

seriously. There are several straightforward and easily implementable approaches to this problem.

One approach is to fire foreground productions whenever it is possible to do so. This is the same as saying that background productions are to be fired only when no foreground productions can be fired. Such a condition occurs when the conflict set contains only background productions. This approach can produce the problems of starvation mentioned above. Foreground productions that would be applicable – save for the absence of the result of some background production – may not be enabled when they are needed for modelling the student's solution.

This solution may still be acceptable since many production systems handle their control problems by making explicit use of such starvation. They fire foreground productions whenever possible, and, when it is not possible, they fire a background production that enables the next phase of the processing. Such production systems are generally organized so that an uninterrupted series of foreground productions will fire during each phase. The PS will be suitable for student solution modelling when the processing phases correspond to necessary divisions within the problem solution.

A second approach is to fire background productions whenever it is possible to do so. This would have just the opposite of the starvation effect of the first proposed solution. The introduction of tags and flags into the database by the firing of the background productions would cause the enabling of as many foreground productions as possible. A greater number of enabled foreground productions helps to ensure that the conflict set will contain the foreground productions that are needed to model the student's solution. Unfortunately, this scheme fails for many cases, including the case illustrated in the example of the last paragraph. The example in the last paragraph describes a system that uses background productions to cycle to the next phase of the processing when there is no more foreground work to be done in the present phase. If background productions are always fired first, then the system may end up cycling through all phases of the control structure without ever having done any processing. Thus, this second solution would be acceptable only in a system where the firing of background productions is dependent on the contents of the database.

A third approach is to fire a mix of foreground and background productions in the same way that the problem-solving PS would do if left to its own resources. This stands as a middle ground between the first and second solutions, without the

dangerous fault of the second. When using a problem-solving PS that uses starvation as a control mechanism, the behavior of this scheme will be similar to that of the first proposed solution. However, the first solution will run up against the noted problems when used with production systems that are controlled by means other than starvation, while this third solution can not be any worse in that respect, and should perform better in many cases.

However, there is a major difficulty associated with the third approach. The problem-solving PS may be organized to move on to the next phase of processing while the student wants to remain in the current phase. That is, the student wants to perform actions that would be paralleled by foreground productions in the current phase while the problem-solving PS has already disabled those productions and enabled those of the next processing phase. Such a situation would occur if the system removes a flag that results in the disabling of some foreground production that would otherwise be applicable. Another possibility is that the system may have been solving a subproblem by having pushed to a specific set of productions. Some production then causes it to pop back to the calling PS while there are still foreground productions in the specific set that are applicable to the current state of the problem solution.

The algebra problem-solving PS that is being used with the interface conforms to the control flow by starvation model noted in the discussion of the first solution. The processing proceeds in phases that focus on a subproblem that must be solved before the next part of the problem may be attacked. This focusing is achieved by enabling only those foreground productions that are applicable to the current subproblem. Foreground productions are responsible for the actual problem solution. Foreground productions continue to be fired until there are no remaining applicable foreground productions. When this occurs, background productions that enable the processing of the next subproblem are allowed to fire.

It can be seen that the first method for selecting the time for firing of background productions closely matches the control flow of the algebra problem-solving PS (described above). In the first selection method, when a conflict set containing foreground productions is generated, all background productions in the conflict set are ignored. But if the conflict set contains only background productions, then one is chosen and fired. Conflict-resolution for background productions uses the default method of PSMON, i.e., the list of productions is treated as an ordered list and the conflict-set production which is closest to the beginning of the list is chosen for firing.

2.2.2. Foreground Conflict-Resolution

The heart of the modelling scheme comes into play when the interface has finished firing background productions. At this point the system will have generated a conflict set that includes foreground productions from the problem-solving PS. The next step in the modelling is to determine what the student's next solution step will be. The interface must then determine which production in the conflict set will correctly model the student's indicated solution step.

It was decided to monitor the student's solution by observing the natural language directives that the student types to the interface. The student does not perform the problem-solving operations himself. Instead, he types directives that indicate what operations he wants the interface to perform. The system performs the actions and returns the results to the student. The justification for the use of this method is discussed in section 3.1.

Choosing a production from the conflict set whose firing will model the student's solution step amounts to the same thing as conflict-resolution. In this case, conflicts are resolved in favor of those productions that best model the solution steps indicated by the student's directive. This method of conflict-resolution using natural language text is discussed in more detail in section 3.2.

Ultimately, a foreground production is selected from the conflict set and fired. A single directive from a student may trigger several problem-solving steps. For example, the student may direct the system to *"Move the goal term to the left hand side of the equation and divide through by 4"*. Here, the directive indicates that first the goal term is to be moved to the left-hand side of the equation and, second, that each term in the equation is to be divided by four. Because a single directive may indicate several actions, a directive is retained by the interface until it has been determined that all indicated problem-solving steps have been executed. The pseudo-code in Figure 2-1 is helpful in visualizing the interface control structures that implement this. After the initial foreground production is fired, the interface may determine that it is time for the firing of more background productions. Alternatively, there may be other enabled foreground productions that are indicated for firing by the student's directive. A directive is not discarded until a conflict set containing foreground productions is generated, and the directive does not indicate any of the foreground productions in the conflict set.

CHAPTER 3
MODELLING THE STUDENT'S SOLUTION

3.1. Monitoring the Student

This chapter discusses the creation and maintenance of an incremental model of a student's problem solution. There are two basic problems involved. The first is to decide upon a method of monitoring a student's solution steps so that the tutor can follow along. The second is to create and maintain a representation of that solution for use by the portion of the tutorial interface that lends assistance to a student.

The decision of how to monitor a student's solution steps is very important. It will determine the nature of the information that a student must provide to the interface. This will have a great effect on the way that the interface appears to a student. The main constraint is that of needing to know what operators a student has applied to arrive at the current problem-solution state. The interface should be easy for a student to use, and should not interfere with the student's performance on a problem-solving task.

One possible method of gaining knowledge about a student's solution does not directly monitor the solution steps. Instead, the steps of a student's solution are inferred from the starting and ending states. In this method, the student enters the state of the problem solution each time he wants the tutor to give him help. The interface would then have to infer the problem-solving steps that were needed to arrive in the current state. This is similar to the method used in BUGGY (Brown and Burton, 1978) for diagnosing problems in childrens' arithmetic. An advantage of this method is that it is not likely to inhibit or alter a student's performance in any way. Unfortunately, when this method is used with complex problems, it can require more computation time than is reasonable for interactive tutoring. Another limitation is the possibility for error in inferring a set of operators. Even if the system succeeds in inferring the correct set of operators, it may only be able to

infer a partial ordering over them. A partial ordering would occur when a problem state is attained through an order independent series of operator applications.

Some of the problems in this method may be overcome by having the student enter the state of the problem after each step that he performs. In the domain of algebra problem solving this would amount to retyping an equation anytime that it has been altered during a problem-solving operation. The system would then be required to infer only one problem-solving step at a time, provided that the student never skips any steps. But it is inevitable that steps will be inadvertently skipped. and results of the combined steps will be provided to the interface. Continually asking a student to go back and enter intermediate steps would tend to interfere with his problem-solving behavior. Students would also tend to resist having to both solve the problem and report each step to the system.

The problem of combined steps could be handled by doing a limited amount of multi-step inferencing. Seeing that the interface will do multi-step inferencing, a student may be tempted into performing more steps between entries to the system. He will still eventually come up against a cut off point and have to re-enter an equation that combines fewer operations.

Another method is to have the student explicitly state which operations he is applying, and to what data he is applying them. This method has successfully been used for other student modelling tasks (Clancey, 1979b), and has the advantage that it is much simpler than the previously mentioned inferencing techniques. A system using this method may be configured so that the student's entries to the system are treated as directives which are applied in the current problem state, and whose results are then displayed to the student. This may help to combat the student's feeling that providing the system with such detailed information about his problem-solving steps it is a waste of his time. The typed information serves the dual purpose of directing the system to perform a set of operations, and informing the tutor of the student's choice of operations.

The use of directives for student monitoring is most simply implemented by having the student enter keywords that specify the problem steps that are to be performed. The simplicity of a keyword system also makes it very fast, which is an advantage for an interactive system such as this. While the keyword scheme simplifies the implementation, it does so at the cost of increasing the burden on the student. The student must learn the set of keywords for the domain, and may have to learn a new set when the problem domain changes. The time spent learning

keywords could become a significant fraction of the total time spent learning in the domain.

An alternative to a keyword system is to allow the student to specify directives in natural language. Given that the natural language system performs well, it would be very easy for the student to use. A natural language understanding system is more complex than a keyword system, and so computation time must be considered as a factor.

The tutor discussed here has been implemented with a natural language style interface that uses both keywords and explicit data references to determine what actions are indicated by the natural language text. This task is made easier by the fact that the expert PS can supply a list of applicable actions from which to choose. Choosing an action from among the foreground productions supplied by the expert PS is equivalent to production conflict-resolution. The following section describes how the text is used to resolve conflicts among the foreground productions. Correct-resolution allows accurate modelling of the states of a student's problem solution.

3.2. Conflict-Resolution Using Natural Language Text

Conflict-resolution is the process of selecting from the conflict set a subset of productions that are to be fired. This section concerns itself only with conflict sets that contain foreground productions. The interface immediately removes any background productions from a conflict set that contains foreground productions. Before conflict-resolution commences, the interface asks the student to enter a directive at the terminal, thus, obtaining the line of text to be used in the conflict-resolution.

The production selection is done by assigning each production a weight and selecting the production with the highest weight. Certain restrictions which apply to the selection of the weighted productions will be described later in this section. The weights are assigned by two distinct checks. The first check is called the *common reference* check. The common reference check in the current implementation is totally independent of the particular problem-solving PS that is being used, and thus needs no special attention from the supplier of the problem solver. This independence from the particular domain has the advantage of reducing the work required to integrate a new domain into the system.

The common reference weighting method consists of comparing the variable bindings for the productions with the text of the directive, and looking to find common references. This system works tolerably well without the second weighing system. However, it is dependent on the student including direct references to the items being manipulated by the operation specified in the directive he typed. Substituting a more liberal matcher for the direct pattern matcher that is currently in use would further add to the usefulness of this method. The weight assigned to a production by this check is simply a sum of the number of common references between the production and the text.

The second check is a keyword check. The keyword check uses information, supplied by the domain designer, that is specific to the particular problem-solving domain. The information supplied is structured around a list of keywords. Associated with each keyword is a list containing production names and numbers. If a particular keyword occurs in the directive, then each production named in the associated list has its associated number added to its weight. A production may be associated with more than one keyword, and may have a different associated number for each keyword (see Appendix B). Neither of these two simple weighting methods alone would be adequate for the natural language task that has been defined. When both weighting methods are used together, however, the results are acceptable.

Given that the interface has a conflict set with foreground productions, and a directive entered by the student, there are three possible actions.

1. If the directive leads to the selection of a production, the production is fired, the text is retained in STM, and another conflict set is generated. This allows for the possibility of several actions being indicated in a single directive.

2. If no production in the current conflict set is indicated by the directive for firing, and this is the first conflict set to be checked against the directive, then help is solicited from the student. This is the case in which the student's text makes no sense to the system in the current state (details further on).

3. If no productions in the current conflict set are indicated for firing, but the directive has indicated productions to be fired from previous conflict sets, then all actions indicated by the directive have been executed. Some sort of reply may be made to the student, the student's text is discarded, and a new cycle begins. The reply may inform the student of the current state of the problem solution, etc.

The restrictions concerning production selection (above) are in the form of two

minimums, or thresholds, that a production must equal or exceed in weight in order to be selected for firing. These thresholds are given default values in the system, and may be altered when mating the tutor with various problem-solving PSs. The first, or *new*, threshold applies when productions are being compared to a newly entered directive. The second, or *old*, threshold applies when a directive has already been used to select productions from a previous conflict set (see section 5.3). Setting the new threshold lower than the old will allow the system to *grasp at straws* when trying to interpret a newly entered directive, while maintaining a certain hesitation to commit itself to an additional meaning.

In the case that a newly entered directive indicates no production in the conflict set, several measures may be taken. If the directive may indicate a production of the problem solver, but that production is not in the current conflict set, then the interface can allow the problem solver to run under its own control for several cycles to see if the student is indicating future steps of the solution. If so, the interface can tell the student that there are several intervening steps, and either perform them for the student, or ask the student to indicate them to the system. The latter is useful for teaching situations, and helps verify that the interface has accurately inferred the student's intentions. If the indicated production is not fired after the problem solver has advanced several steps, then the help processor can be called.

If absolutely no productions are indicated by the directive, it is appropriate to inform the student that the system is at a loss as to how to interpret the student's input. Provisions for assisting the student are detailed in the following chapter.

CHAPTER 4
PROVIDING ASSISTANCE TO THE STUDENT

4.1. The Benefits of Student Models in Tutoring Systems

Maintaining a model of a student provides a valuable knowledge source for use in providing tutorial aid. One of the simplest forms of a student model is the protocol of the student's performance on a task, or set of tasks. A student protocol allows a tutor to determine the current context in which a student is working, and what previous states he has visited. A more sophisticated student model can provide information concerning both what a student does and does not know, and what erroneous knowledge the student may hold. A very good student model can even tell something about the strategies that a student is using to perform the tasks given to him.

Given a model of the student that can supply these types of information, how can that information then be applied in a tutoring system? The protocol of a student's problem solving can be used for giving hints to students, wherein the past history of the student's work is taken into account. The protocol can also be used by a system to backtrack to previous states and investigate alternative approaches. A model that can represent what a student already knows can be used to avoid unnecessary instruction in those areas. A model that points out gaps in a student's knowledge will help a tutor to concentrate instructional resources where they are most needed. Incorrectly held knowledge that is identified by a student model will aid in correcting a student's misconceptions.

It would seem, then, that the presence of a student model would greatly aid efforts to tutor students. In order to decide what form of student model will meet the needs of a specific task, it is necessary to find which types of models provide the various types of information that are useful for tutoring. It will then be possible to determine how much effort will be required to obtain the types of information that are needed for the specified task.

Recording a protocol from an interactive session with a student is a relatively easy task. The issue that arises with protocols is that of representing the protocol in such a way that it is easily interpreted by the tutoring system. The desired representation varies with the nature of the tutoring system under consideration.

A central component of many student models is some method of marking off on a check list those rules, or bits of information, that a student is thought to know. This type of modelling is commonly called an *Overlay* model. The GUIDON system (Clancey, 1979b) represents knowledge in the form of production rules, and represents a student's knowledge of some operation by marking the production rule that the system uses to represent that same operation. The Leeds Modelling System (LMS) (Sleeman and Smith, 1981) takes a somewhat different approach. It creates a student model in the form of a production system whose action is intended to be similar to that of the student whom it models. Thus, the production rules present in the rule set represent the domain knowledge possessed by the student.

In the LMS, a model is created one rule at a time using what Sleeman and Smith call the *Selective Algorithm*. Each skill is individually tested by giving the student a task that requires that skill, and does not require any other untested skill. As each skill is tested, a production rule is selected that operates in a manner similar to the behavior demonstrated by the student. The modeller has a choice among many possible erroneous and correct productions as it tries to match the demonstrated behavior of the student. If the student demonstrates an incorrect knowledge of the tested skill, then a production is selected that also performs that particular operation incorrectly. By choosing a production that shows the same incorrect behavior as the student, the model correctly represents the student's actual performance. Skills are tested in a predetermined sequence in order to simplify the task of building the production rule set.

The productions are inserted into the LMS student model PS in such a way that they will be selected for firing under the same conditions as those in which the student has demonstrated that he would perform the operation represented by the production. In this way, the LMS not only represents the domain knowledge possessed by a student, but also represents his ability to apply that knowledge in different situations. This model also represents the gaps in a student's knowledge, and the student's incorrectly held knowledge. Thus, it would seem that the LMS provides a very powerful modelling capability. Its major disadvantage is that it requires a highly constrained interaction with a student in order to construct the student model PS.

The GUIDON system forms a *differential model* through the use of a domain expert system. In GUIDON, a differential model is constructed by comparing the behavior of a student with the behavior of a computer system that is an *expert* in some domain. When a student makes a choice that is the same as the choice of the expert system, that is taken as evidence that the student possesses the knowledge which the expert system deems necessary for making that decision. If a student makes a choice that differs from that of the expert system, that is taken as evidence that the student does not possess the knowledge which the expert system deems necessary for making that decision, or that the student possesses some knowledge that incorrectly leads him to another conclusion. Differential models have appeared explicitly in several recent ICAI projects (Burton and Brown, 1979, Goldstein, 1980). The number of systems using some sort of embedded expert system for comparison with a student's results is too large to mention. A differential model can be constructed from any student protocol produced on a given tutorial system, rather than requiring the constrained environment of LMS, and thus has a great advantage over the LMS.

The discussion here has introduced differential modelling techniques through the example of the GUIDON system, although they are necessary for the Leeds Modelling System as well. To permit the LMS student model PS and the expert PS to be compared, an expert system in the form of a PS must be available. This comparison will yield the desired information about the abilities of the student. Without some standard to compare the student model against, the LMS can not detect a student's incorrect operations, nor can it determine whether the student lacks knowledge of some operation.

It would seem, in fact, that differential modelling techniques are essential for obtaining information about a student's incorrect or missing knowledge. To be able to determine that it is unlikely that a particular skill is possessed by a student, the interface has to be able to determine that the student did not use the skill in a particular instance where that skill would have been appropriate. Some sort of expert knowledge is required in order to determine this. A differential modeller would compare the behavior of a domain expert to that of a student to determine this information. After collecting several instances that indicate that a certain skill is not possessed by a student, the likelihood that the student does not possesses that skill can be computed with greater certainty. By similar reasoning, expert knowledge is also required to determine whether a student possesses incorrect knowledge.

One explanation for the failure of a student to use an appropriate operator is that

the student may be using a different solution strategy than the expert system. Knowing the strategy that a student is using would allow a modelling system to be more certain in attributing a student's failure to use a specific operator to lack of knowledge about that operator. Burton and Brown (1979) have constructed a system that is able to determine what strategy a student may be using. This is done by having a domain expert, or experts, that can solve the same problems using a variety of strategies. To determine the strategy that a student is using, the system has only to determine which expert strategy best matches the performance of the student. The system still has to allow for missing or erroneous skills on the part of the student. This could be done in the LMS by providing a collection of expert PSs for comparison, choosing that which best matches the student model PS.

In summary, it has been stated that producing student protocols is a fairly straight-forward task. Knowledge of the history of a tutoring session should be reasonably easy to obtain. Information about what a student knows can be obtained by creating a simple overlay model from the student's demonstrated skills. This type of model can be constructed by straight-forward analysis of a student protocol. Obtaining most information about the operations that a student does or does not know requires the use of a differential modeller. A differential modeller requires a domain expert with which to compare a student's behavior. Constructing such an expert in a computer system is a formidable task for realistic domains. The construction of expert systems that has been going on in various AI labs provides a ready starting point for producing differential modellers for ICAI systems[1]. The determination of a student's strategy requires a great deal of effort. Not only must there be a domain expert, but the expert must be able to use a wide selection of strategies. For this reason, it is an even harder problem than differential modelling.

4.2. Tutoring Strategies

It has been shown that there can be many types of information available to an ICAI system. Obtaining these different types of information may require differing amounts of effort in the building of student models. Now that some information about the sources for these types of information has been presented, it is time to consider what can be done with this information.

---

[1] For instance, Clancey's use of the MYCIN medical expert system in GUIDON.

The tutorial interface that has been constructed is similar to several *coaching* systems (Burton and Brown, 1979, Goldstein, 1980). A coaching system is a teaching system that occasionally offers hints to a student while the student is performing some task. However, the tutorial interface is not simply a coaching system. This interface is intended to be included as a part of a more general tutoring system, rather than to be used by itself. A computer coach spends most of its time observing a student's actions. The coaches are designed to occasionally interrupt a student with suggestions on how to improve his performance. In their paper on computer coaching, Burton and Brown present a detailed argument concerning the issue of just when to offer advice to a student.

The tutorial interface that has been constructed differs from other systems that have been termed coaches in several respects. The most obvious is that the other systems have been designed for game playing environments. They try to help a student improve his own ability to play some game by offering suggestions about weaknesses in the student's play. One of the less obvious differences is that a game is usually forward moving, while backtracking is often useful when solving algebra problems. The coaching systems spontaneously offer advice when they deem it proper to do so, while the tutorial interface is currently programmed to wait for the student to request its help before it will intervene. The tutorial system's use of student directives to guide the operation of the expert problem solver differs from the coaching systems which have separate components for the gaming expert and the game itself. A further difference comes from the assumption that a student will already know the set of required operations, while the game coaching systems are specifically designed to teach the required game playing operations.

While the tutorial interface differs from the coaching systems in many ways, it also exhibits features that are similar to those of the coaching systems. The interface follows the student's progress as he solves some problem. If the student directs the expert to perform an operation that is not possible at the current stage of problem solving, the interface informs the student of the presence of intervening operations that must be executed first. When a student requests help from the tutor, that help may be given at two different levels. .The first level is a hint that points out some object in the problem that is to be manipulated by the operation that the expert feels is most appropriate. If the student is still stuck, the interface will execute the next step for him. It would be possible for the interface to suggest the operation to the student and have him request the expert to execute

that operation, but the more direct approach was chosen instead. The interface may also determine that the student has been working away from the goal, rather than toward it. In this case, it will backtrack to a previous state that is closer to the goal, and proceed from that point. The next paragraph will begin the discussion of how the tutorial interface implements these features.

Using the algebra problem solving PS, it is easy to detect when an operation is not appropriate in the current state of the problem solving. If the indicated operation corresponds to a production that is not currently enabled in the expert PS, then that operation is not appropriate. This method is useful for determining whether a student's directed operation would be appropriate after some number of intervening steps. To determine whether intervening steps will enable the indicated operator, the interface allows the expert to move forward a specified number of foreground steps and checks whether the indicated operation was executed. Background steps are also executed during this look-ahead, but they are not counted as forward steps because they do not correspond to problem-solving operations. Specification of the number of steps to look ahead is described in section 5.3.

When a student asks the system for help, the first hint is generated by looking at the data that is bound by the next production that the expert problem solver would fire in the current state. One of the referents from this bound data is given to the student as a hint about what parts of the problem he should be looking at. For the second level of help, when the system executes the next step for the student, the interface simply directs the expert to fire the production that it has chosen as most appropriate.

For the system to be able to do backtracking, it is necessary to define some measure of how close a given problem state is to the goal state. The definition used by the tutorial interface is that a problem state's distance from the goal is equal to the number of foreground productions that the expert PS must fire in order to achieve the goal from that state. If the student is making progress, then each of his problem states should be one step closer to the goal state. This is an approximation, of course, as there is no guarantee that the expert problem-solver always solves a problem with the smallest possible number of operations.

As a heuristic, the distance of the current state to the goal state can be computed and compared to the distances of successive previous states to the goal. If it is found that the successive previous states are not further from the goal than

the most recent states, then it can be assumed that the current solution path is not progressing toward the goal. When this is the case, the interface can backtrack along the solution path until reaching a point where further backtracking will increase the distance from the goal. If this is not the case, then the solution is progressing toward the goal and no backtracking is necessary. When backtracking is done, the interface always goes back to the last step that was seen to be progressing toward the goal, and then one more. Undoing the last progressive step allows the interface to then execute that step again, and the result of executing that step will tell the student how far the interface has backtracked.

Burton and Brown have proposed twelve principles for coaching (figure 4-1), some of which are applicable to the environment of the tutorial interface. They are included here to allow a comparison with the tutorial techniques used by the interface. While the current implementation of the tutorial interface does exploit the unique potentials of its experimental modelling technique, it has ignored other useful techniques that have been used elsewhere. This comparison will indicate some of the useful techniques that may be used to augment the ability of the interface to help students in a more complete implementation.

Burton and Brown's principle 2 can be adapted to the environment of the tutorial interface in the form of a rule that states:

> *Only undo a student's work by backtracking if the state to which we backtrack is a dramatic improvement over the current state.*

Principle 9 suggests a corollary:

> *When backtracking is done, always backtrack to the optimal point.*

The interface always backtracks to as near an optimal point as its heuristics will allow. The interface does not obey the first of these two rules, however. Backtracking is done regardless of the degree of improvement. The methods that are currently being used would be able to handle this improvement, and it may be included later.

Principles 3 and 8 suggest that the tutorial interface should have a provision for allowing a student to undo operations that he has previously performed. This feature is easily implemented in the interface, since a backtracking feature is already included. In some sense, this feature is also related to principle 6. It allows a student to correct his mistakes himself, rather than having the tutor do all the correction.

Principles 1, 5, and 6 are concerned with the offering of spontaneous advice.

Figure 4-1: Burton and Brown's twelve principles*

1. Before giving advice, be sure the Issue used is one in which the student is weak.

2. When illustrating an Issue, only use an Example (an alternative move) in which the result or outcome of that move is dramatically superior to the move made by the student.

3. After giving the student advice, permit him to incorporate the Issue immediately by allowing him to repeat his turn.

4. If a student is about to lose interrupt and tutor him only with moves that will keep him from losing.

5. Do not tutor on two consecutive moves, no matter what.

6. Do not tutor before the student has a chance to discover the game for himself.

7. Do not provide only criticism when the Tutor breaks in! If the student makes an exceptional move, identify why it is good and congratulate him.

8. After giving advice to the student, offer him a chance to retake his turn, but do not force him to.

9. Always have the Computer Expert play an optimal game.

10. If the student asks for help, provide several levels of hints.

11. If the student is losing consistently, adjust the level of play.

12. If the student makes a potentially careless error, be forgiving. But provide explicit commentary in case it was not just careless.

* Burton and Brown, 1979

These issues are side-stepped in the current implementation by providing assistance only when it is requested by the student.

It would be possible to congratulate a student's achievements, a la principle 7, by adding an overlay model to the system. It is currently assumed that the student already knows the operations required for the problem solutions, and the interface doesn't model the student's knowledge of operations. Introducing an overlay model to record which operator applications seem to be causing problems for a student allows the tutor to congratulate the student when the student seems to have overcome those problems.

The tutorial interface does provide multiple levels of hints, as suggested in principle 10. An expansion to more than two levels would improve the tutor's abilities. Principle 12 is concerned with providing sufficient explanation of corrections to errors that a student has made. The only situation in which the tutorial interface can detect that an operation has been specified when it should not have been is when it detects a step that is valid, but only after some intervening step has been performed. All other cases of invalid operators are eliminated by the assumptions made about the student's prior knowledge. Augmentation of the interface's differential modeller would allow it to support this principle more completely.

Principle 11, concerning the difficulty of the student's task, cannot be addressed directly within the interface. Rather, it would have to be included in a more general tutoring system that included the tutorial interface. When a student has too much trouble with the problems he is trying to solve, the general tutor could use an inductive strategy to find a sequence of problems of increasing difficulty that begins with a problem that the student can solve and ends with a problem of the sort with which the student is having difficulties.

Of Burton and Brown's twelve principles, only principles 3, 8, 9 and 10 are currently supported. The student model used in the tutorial interface can easily support principle 2 without changing or augmenting the model itself. Support for principle 11 would come from a larger tutorial system in which the interface would be embedded. So, without changing or augmenting the student model, the current interface can support principles 2, 3, 8, 9, 10, and 11, or six out of twelve. Principles 1, 5, and 6, are side-stepped by providing assistance only when it is requested by the student.

Principle 7 can also be supported without a great deal of additional effort. An overlay model, as has been noted in the last section, requires few resources to build and maintain. The expert system included in the interface allows for the augmentation of the current differential modeller so that it can provide additional information about what a student does and does not know. The models that support principle 7 would also be of use in supporting principle 12. However, supporting principle 12 would require the addition of explanatory textual material corresponding to the many situations in which each operation may be appropriate. Principle 4, the last remaining principle, is unique to the game-playing environment, and has no parallel in the tutorial interface.

CHAPTER 5
THE PROBLEM-SOLVING PS

5.1. Features of the PSMON production system monitor

One of the design goals for the Tutorial Interface is that it be easily mated with any problem-solving PS. The practical constraint is that the problem-solving PS must be implemented for the same production system monitor as the interface. To permit the reader to better understand the details of the problem-solving PS that is used here as an example, this chapter will begin with a discussion of the practical aspects of the PSMON production system monitor. A more complete description of PSMON is given in a paper by Rood and Bregar (1980).

Section 1.3.2 included a discussion of the nature of production systems in general. A complete system consists of an interpreter, a database, and a set of production rules. The interpreter, or monitor, is controlled in a three-step loop. First, it identifies that set of productions whose condition parts are satisfied by the current state of the database. Then the interpreter must chose one or several productions from the conflict set that are to be fired before iterating the loop. Finally, the chosen production rules are fired.

In section 2.1.2, some of the special features of PSMON were introduced. The push and pop mechanism allows the system to make recursive calls to itself. These recursive calls may involve production rules and/or database memories that differ from those in use by the calling PS. Also mentioned were the special calling modes of PSMON. These allow access to the facilities of the interpreter without having to enter the interpreter loop. Thus the interpreter can generate a conflict set and return it to the caller, rather than continuing with the interpreter's normal processing. After choosing one of the conflict-set productions for firing, the interpreter can be called in a different mode in which it will fire the chosen production in its correct context.

PSMON is implemented in LISP. Both the production rules and the system database are represented as LISP structures. Each production has the form outlined in figure 5-1(a). A set of these productions are represented as a list. The four memories of the database, STM, ITM, LTM, and ENV, are also represented as lists. A sample value of STM is shown in figure 5-1(b). The datum at the head of the STM is a flag which will enable the productions that initialize a new run of the system. This value is actually used as the initial value of STM for the algebra problem-solving domain. The NILs serve to *pad out* the STM to its constant length of six.

Figure 5-1: PSMON representations

(a) Form of a Production

```
(<production name>
 IF   <list of conditions>
THEN <list of actions> )
```

(b) Sample STM

```
((NEW START) NIL NIL NIL NIL NIL)
```

(c) Sample Production Rule

```
(GET-VAR-LIST
 IF   ((A NEW START))
 THEN ((SAY (MSG "Type in the variables for your problem: "))
       (LISTEN)
       (SAY (MSG T))))
```

5.1.1. Matching Production Conditions

A production system monitor includes a matcher with which it compares parts of production rule conditions to the various database memories. The matcher used by PSMON treats the list of conditions in a production rule as a conjunctive set. Every condition in the list must be satisfied or the match will fail.

The matcher needs to know which of the memories in the database should be matched against a given condition. In PSMON, this is handled by each condition having an explicit flag that identifies the memory that is to be used for matching. Figure 5-1(c) shows a production rule from the algebra problem solver. The rule GET-VAR-LIST is one of those which match the value of STM that is shown in 5-

1(b). The condition list has only a single condition. The letter 'A' as the first atom of that condition indicates that the condition is to be matched against STM. The letter 'A' stands for *Active* memory. The other two allowable flags are 'I' for *Intermediate*-term memory, or ITM, and 'E' for *Environmental* memory, ENV. No explicit provision was made for matches against LTM because it is assumed to be constant throughout the duration of PS execution.

A production condition, excluding the flag that specifies the memory to be matched, is simply a *template* that is to be matched by a memory element. Ignoring the memory flag of the condition in the GET-VAR-LIST production, what remains is the template (NEW START). Matching this to the first element of the STM shown in part B is a relatively simple task. Any atom in a template, except the atom '?', will be used for a literal match against atoms in the elements of the indicated memory. The atom '?' acts as a *wild card* by matching any single atom in a memory element.

Lists that appear in the template are given special treatment. One of the special uses of lists is for the binding of matched values to labels, and for the retrieval of those bound values. The binding and retrieval operators are always used in the form (<operator> <label>). There are two binding operators in PSMON, '>' and '*>'. The operator '>' is like the wild card '?', except that it causes the matched arbitrary value to be bound to the specified label. The operator '*>' is slightly different, in that it causes the list of all remaining data in the memory element to be bound to the variable. Figure 5-2(a,b) illustrates the use of PSMON's binding facility. The binding operators are complemented by the two retrieval operators '<' and '*<'. One use of the retrieval operators is shown in figure 5-2(c,d). Notice that '*<' causes the bound list to be *spliced* into the surrounding form, rather than substituted as a list. As was mentioned in section 1.3.2. the retrieval of bound values may take place anywhere in the production after the point at which the value is bound.

There are still two remaining interpretations of list forms that appear in templates. If the list form is a call to a LISP function, then that function will be evaluated. If the function returns the value NIL, then the match will fail. The matching process will continue if a non-NIL value is returned. The value of a function call will be treated as a boolean value which is used to determine whether the condition is satisfied or not. The matcher will otherwise ignore the function call. It will not be used to match items from the database. If the list form is not a PSMON binding or retrieval operator, and it is not a function call, then PSMON will try to literally match it against forms in the database.

Figure 5-2: Value Binding and Retrieval

(a) Binding Using the '>' Operator

```
       The Condition Template:  (GOAL (> GOAL-VARIABLE))
Will Match the Memory Element:  (GOAL A)
        And Cause the Binding:  GOAL-VARIABLE has the value 'A'
```

(b) Binding Using the '*>' Operator

```
       The Condition Template:  (VARIABLES (*> VARIABLE-LIST))
Will Match the Memory Element:  (VARIABLES A B C)
        And Cause the Binding:  VARIABLE-LIST has the value (A B C)
```

(c) Retrieval Using the '<' Operator

```
            With the Binding:  GOAL-VARIABLE has the value 'A'
       The Condition Template:  (GOAL (< GOAL-VARIABLE))
Will Match the Memory Element:  (GOAL A)
```

(d) Retrieval Using the '*<' Operator

```
            With the Binding:  VARIABLE-LIST has the value (A B C)
       The Condition Template:  (VARIABLES (*< VARIABLE-LIST))
Will Match the Memory Element:  (VARIABLES A B C)
```

When the condition part of a production rule is found to be satisfied, the memory elements used in matching it are brought to the front of the STM. As was mentioned in section 1.3.2, this feature insures that STM contains those memory elements that have been most recently accessed. It is assumed that for a given memory accesses, recently accessed memory elements are more likely to be accessed than memory elements that have not been accessed recently. Keeping the most recently accessed memory elements in STM allows for a short search of only the STM, rather than a longer search of all of memory, when the memory elements are needed again.

The example shown in figure 5-3 shows a complete production rule that uses a somewhat tricky binding and retrieval. If the operator '*<' had been used to splice the list of variables in the call to ISVARLIST, then ISVARLIST would have had to have been written to accept a variable number of arguments. Instead, the operator

'<' allowed the list of variables to be directly substituted into the call. The retrieval operators may be nested to any level in any part of the condition or action lists.

Figure 5-3: The Production GET-GOAL-VARIABLE

```
(GET-GOAL-VARIABLE
 IF ((A HEAR (*> VARS) (ISVARLIST (< VARS)))
     (A NEW START))
 THEN
    ((SAY (MSG "Ok , which variable must we solve for? "))
     (REPLACE (A NEW START) (OLD START))
     (REPLACE (A HEAR (*< VARS)) (VARS (*< VARS)))
     (ITMSAVE (VARS (*< VARS)))
     (LISTEN)
     (SAY (MSG T))))
```

```
       The Condition Template:  (A HEAR (*> VARS))
Will Match the Memory Element:  (HEAR A B C)
        And Cause the Binding:  VARS has the value (A B C)

            With the Binding:  VARS has the value (A B C)
      The Condition Template:  (A HEAR (*> VARS) (ISVARLIST (< VARS)))
     Will cause evaluation of:  (ISVARLIST (A B C))

      The Condition Template:  (A HEAR (*> VARS) (ISVARLIST (*< VARS)))
    Would cause evaluation of:  (ISVARLIST A B C)
```

5.1.2. Evaluating Production Actions

The elements of the action list of a production have the *form* of LISP function calls. The actions are processed as a sequential list, the first action in the list being the first action to be processed. Although the form of the elements are like those of function calls, they need not actually refer to any extant LISP functions. Of the actions in the production GET-GOAL-VARIABLE in figure 5-3, only ITMSAVE and LISTEN are calls to LISP functions. REPLACE and SAY are utilities provided by PSMON. The difference between utilities and functions will be explained shortly.

There are three ways in which PSMON may process an action. The action may be

a utility, a function call, or a literal. The first step in processing any of the three types of actions is to splice or substitute any indicated bindings into the action. The binding operators, '>' and '*>', have no meaning in the action part of a production, since no matching is taking place. Values may still be bound in actions by using the monitor utility named BIND.

Utilities may be considered as virtual functions that have rapid access to the internal data structures of the monitor. To the designer of a set of productions, the only difference between a utility and a function is that a utility can not be used other than as a production action. The processing for functions and utilities consists of making the indicated substitutions and then executing the function, or virtual function. If the first atom of an action does not correspond to any known function or utility, then the action is taken as a literal which is simply inserted at the head of the STM. A summary of the PSMON utilities appears in figure 5-4.

5.2. The Algebra Problem Domain

The algebra problem solving PS is divided into three parts. These parts are named PROB-SOLVER, SIMPLIFY, and COMBINE. The names actually refer to the atoms on whose values are stored the lists of production rules (See Appendix C.) The PROB-SOLVER production set is the entry point for the algebra system. The production sets SIMPLIFY and COMBINE are invoked by the PSMON push facility from within PROB-SOLVER. The production set SIMPLIFY manipulates an equation so that the goal variable is alone on the left hand side of the equation, and has a coefficient of one. The COMBINE rule set takes two equations which have shared variables and tries to eliminate one of the unknown variables by combining the two equations. The whole system is controlled by the productions of the PROB-SOLVER rule set.

Most all of the productions in SIMPLIFY and COMBINE perform algebraic operations on the problem state. This makes most of the rules in these sets foreground rules. PROB-SOLVER, on the other hand, contains mostly background productions. This rule set is responsible for obtaining equations and goals from the user, and creating subgoals for the other two rule sets.

The remainder of this discussion of the algebra problem solving domain will include references to specific production names. The casual reader may wish simply to refer to figure 5-5 to determine whether a production is classified as foreground or background. The references to production names will be most useful to those who wish to read the production rule sets shown in appendix C.

Figure 5-4: PSMON Utilities

(BIND <variable> <value form> ...)
If the value form is a function call, that function will be evaluated and the resultant value bound to the variable. Any subsequent value forms will be ignored. If the value form is not a function call, then the list of all of the value forms will be taken as a literal and bound to the variable.

(DELETE <literal>) If the literal appears in the STM then the value of STM will be changed so that it no longer contains that literal.

(MARK <literal>) The form derived by CONSing the atom MARKED to the literal is made to be the first element of STM.

(NOTICE <condition list>)
The condition list is processed by the matcher. Memory elements that are matched are subsequently brought into the STM, or *Noticed*.

(RECEIVE) The terminal is queried, and the result is made a part of the ENV memory.

(REPLACE <condition element> <literal>)
The condition element is processed by the matcher. If a match is found, the matched memory element will be replaced by the literal, and placed in STM. Bound variables are substituted into the literal before processing.

(SAY <value form>) The value form is taken as a literal, and printed to the user terminal. This utility has been *hacked* for this system, so that it will evaluate the functions MSG and MAPC if they appear in the value form.

(SEND <literal> ...) A list will be made of the literals, and that list will be made an element of ENV.

(SHOVE <literal>) The literal is made to be an element of STM.

In its usual operation, the production system first gathers data about the problem, via the rules GET-GOAL-VARIABLE, GET-VAR-LIST, and REQUEST-INFO-ABOUT-A-GOAL. From this point, the nature of the problem determines what will be done next. If the problem is not simplified in terms of the goal then the BEGIN-SIMPLIFICATION rule invokes the SIMPLIFY rule set to achieve that condition. If a simplified equation contains unknown variables on the right hand side, then one of the unknown variables is chosen as a subgoal. The productions NOTE-VAR and

Figure 5-5: Production Names from Algebra Rule Sets

```
PROB-SOLVER
   GET-GOAL-VARIABLE                          background
   GET-VAR-LIST                               background
   NOTE-MAIN-GOAL                             background
   BEGIN-SIMPLIFICATION                       background - invokes SIMPLIFY
   IS-SIMPLE                                  background
   NOTE-RESULT                                background
   RETURN-TO-SUPERGOAL                        background
   REPORT-RESULT                              background
   SUBSTITUTE                               FOREGROUND
   TWO-EQUATIONS-CAN-BE-COMBINED              background - invokes COMBINE
   RECALL-RESULT                              background
   MAKE-VAR-GOAL                              background
   NOTE-VAR                                   background
   NO-NEW-INFORMATION                         background
   GET-NEW-INFORMATION                        background
   REQUEST-INFO-ABOUT-A-GOAL                  background
SIMPLIFY
   COMBINE-TERMS-FROM-OPPOSITE-SIDES        FOREGROUND
   COMBINE-TERMS-ON-LEFT-SIDE               FOREGROUND
   COMBINE-TERMS-ON-RIGHT-SIDE              FOREGROUND
   MOVE-GOAL-TO-LEFT-SIDE                   FOREGROUND
   ISO-GOAL-ON-LHS                          FOREGROUND
   REDUCE-GOAL-TERM                         FOREGROUND
   RETURN-TO-SOLVE                            background
COMBINE
   ALIGN-EQUATIONS                          FOREGROUND
   FIND-VARIABLE-TO-ELIMINATE               FOREGROUND
   MAKE-COMPLEMENTARY-EQUATIONS             FOREGROUND
   RULE-TO-SUBTRACT-EQUATIONS               FOREGROUND
   EQUATIONS-THE-SAME                         background
   ELIMINATE-TERMS-WITH-ZERO-COEFFICIENT    FOREGROUND
```

MAKE-VAR-GOAL are responsible for the creation of new subgoals. If no information is available about the subgoal then REQUEST-INFO-ABOUT-A-GOAL requests that an equation containing the subgoal be given to the system. Information from a satisfied subgoal may be used either by substituting the value of the subgoal variable into the parent equation, via SUBSTITUTE, or by TWO-EQUATIONS-CAN-BE-COMBINED invoking the COMBINE rule set to combine two

equations with the same variables into a new equation with one less unknown variable.

The PROB-SOLVER PS performs a means-ends decomposition of problems into subproblems (Bregar, Farley, and Lantz, 1982). Although the productions of PROB-SOLVER are specific to the algebra domain, the strategy that they embody is independent of the domain. The background rules of the PROB-SOLVER PS include a set of problem-solving meta rules that execute a means-ends strategy. Since the tutor needs to model a student's solution steps, but and not his strategy, only the foreground productions are used for student modeling. If identifying a student's strategy was required, then it would be necessary to make use of the background meta rules as well.

The nature of the action of the SIMPLIFY production system is suggested by the names of the productions that are included in it (see figure 5-5). When the default conflict-resolution of PSMON is used, the productions are treated as an ordered list. The production that is fired is the first production from the list whose condition part is satisfied. In the SIMPLIFY production set, terms are combined within the equation whenever it is possible to do so. The first three productions in the rule set identify themselves as the productions responsible for this behavior. When no more combining can done, the goal term is brought to the left-hand side of the equation, all other terms are removed from the left side, and the coefficients are adjusted so that the goal term is left with a coefficient of one. When this state has been reached, control is returned to the invoking PS via the pop mechanism.

The action of the COMBINE rule set is also reflected in the names of its constituent productions. As is indicated by the first production name, the first step of the COMBINE rule set is to align the terms of the equations. In this step, the equations are rewritten so that each equation has all of its terms on the left hand side. The terms are sorted by the names of the variables in them so that the equations are easier to compare to each other. The next step is to find a variable to eliminate. The variable must be common to both equations, and must not be a goal variable. The equations are then modified so that the terms containing the variable to be eliminated each have the same coefficient. This is done so that the subsequent subtraction of one equation from the other will produce a result that does not contain the selected variable. Following the subtraction of the two equations, terms with coefficients of zero are removed from the result, and control is returned to the invoking PS.

The algebra problem solver uses a tagged data formalism. Each element in the database has a first atom which acts as a tag. Tags are recognized only by the specific productions that are designed to deal with the information in the tagged element. Referring to the productions in the appendix, the reader will notice that many of the production conditions specify an explicit match for such tags as SIMPLE, GOAL, INFO, and RESULT.

Input to the system is done with the LISP function LINEREAD, which reads a series of atoms terminated by an ASCII NEW-LINE character. Equations are interpreted by a recursive descent parser and converted to an internal format, as in figure 5-6.

Figure 5-6: Algebra Equation Representation

The equation

4/3x+8y-3=2z-y

is internally represented as the LISP expression

```
(((+ (4 / 3) X) (+ (8 / 1) Y) (- (3 / 1)))
 =
 ((+ (2 / 1) Z) (- (1 / 1) Y)))
```

5.3. Mating an Expert PS with the Tutorial Interface

When mating an expert PS with the tutorial interface, the PS designer is required to provide values for two variables; SPECIAL and FOREGROUND. The value for each of these variables will be a list of production names. The variable SPECIAL has a list of the names of those productions that use the PSMON push or pop facilities, or do input from terminal or file. These are all actions that require special treatment when the tutor is looking ahead to possible future states of the problem solution. If the tutor, while looking ahead, were to fire a production that queried the terminal, the student would then receive a request for information that had nothing to do with the current state of his problem solving.

The variable FOREGROUND has a list of the names of those productions that

correspond to operations that the student is expected to specify in his directives. This list is used by the tutor to differentiate between foreground and background productions. Examples of the values for SPECIAL and FOREGROUND for the algebra PS are shown in figure 5-7.

Figure 5-7: Required Variables With Sample Values

```
Variable SPECIAL

        Explanation     Production Name

                pop     RETURN-TO-SOLVE
              input     GET-GOAL-VARIABLE
              input     GET-VAR-LIST
               push     BEGIN-SIMPLIFICATION
                pop     REPORT-RESULT
              input     REQUEST-INFO-ABOUT-A-GOAL
               push     TWO-EQUATIONS-CAN-BE-COMBINED
              input     NO-NEW-INFORMATION
                pop     EQUATIONS-THE-SAME
                pop     ELIMINATE-TERMS-WITH-ZERO-COEFFICIENT

Variable FOREGROUND

                Production Name

                COMBINE-TERMS-FROM-OPPOSITE-SIDES
                COMBINE-TERMS-ON-LEFT-SIDE
                COMBINE-TERMS-ON-RIGHT-SIDE
                MOVE-GOAL-TO-LEFT-SIDE
                ISO-GOAL-ON-LHS
                REDUCE-GOAL-TERM
                ALIGN-EQUATIONS
                FIND-VARIABLE-TO-ELIMINATE
                MAKE-COMPLEMENTARY-EQUATIONS
                RULE-TO-SUBTRACT-EQUATIONS
                ELIMINATE-TERMS-WITH-ZERO-COEFFICIENT
                SUBSTITUTE
```

Other than these two required variables, all system variables have default values. The tutor is able to perform correctly, and somewhat usefully, with the defaults alone. As described in section 3.2, running in this simple mode causes interpretation

of directives to be done only through the recognition of common references between directives entered by the student and bindings made by the problem-solving PS. The performance will be greatly enhanced, however, when a keyword list is provided with which the remainder of the directive interpretation facility may be driven. The name of the variable whose property list is the keyword list is FUNCT:KEYWORDS-PROP-LIST. The form of the keyword list is illustrated in figure 5-8. Including background productions in a keyword list will not have any meaning since only foreground productions are considered for selection by directives. The keyword list used with the algebra PS is included in appendix B. The default keyword list is NIL.

Figure 5-8: Form for Keyword List

```
(<keyword>   ((production name> <numeric weight>)
              ... )

 <keyword>   ((production name> <numeric weight>)
              ... )

 ... )
```

To simplify the use of the keyword system, the student's directive is preprocessed before the keyword search is performed. During preprocessing, all space characters in the directive are preserved and new spaces may be introduced. The added spaces are used to separate mathematical symbols from other text. For example, APPLES+ORANGES would become APPLES + ORANGES after the preprocessing. This separation of mathematical symbols from object names simplifies later matching stages.

The second stage of the preprocessing is meant to reduce the number of keywords that need be defined. It works by converting various operator references into a common form. The symbol '+' and any atom beginning with the letters 'ADD' will be converted to the atom PLUS. Thus, the words ADD, ADDITION, ADDED, and PLUS are all detectable by the single keyword PLUS. Similarly, '-' and atoms beginning with 'SUBTRACT' become DIFF; '*' and atoms beginning with 'MULTIPL' become TIMES; and '/' and atoms beginning with 'DIVI' become QUOTIENT.

Another, much simpler, list of keywords is the list of those words that will cause the tutor to give aid to the student. This list is kept as the value of the variable FUNCT:ASSISTANCE-KEYWORDS, and has the default value (HELP WHAT WHY).

The two remaining system variables are used to determine the minimum total weights that must be assigned by the directive interpretation mechanism in order for productions to be selected for firing. Each of the values is simply a number corresponding to a minimum required weight. The difference between the use of the two minimum weights is that one is used when selecting a first production for firing from a directive, and the second is used when selecting a production for firing when the directive has already caused at least one previous production firing. The purpose of the two minimum weights is to allow the system to extract one operation from a directive fairly easily, but to be more hesitant about finding indications of more than one operator in a directive.

The first, and always smaller, weight is the number kept on the atom FUNCT:MIN-SCORE-TO-CHOOSE-ONE. The default value for the first minimum weight is one. The second minimum weight is kept as the value of the atom FUNCT:MIN-SCORE-TO-CHOOSE-MORE, and has the default value three. When assigning these minimum weights, the interaction between the minimum weights and the weights assigned by the keyword list must be taken into account. If the keyword list assigns weights on the order of ten, the default minimum weights of one and three will become useless since any weight assigned by a keyword will immediately satisfy both minimum weights.

A final consideration for producing an expert PS for use with the tutor is the granularity of the PS. Ideally, the PS should be written such that each production performs an operation with complexity similar to that which the student is expected to specify in his directives. For example, a production that performs an addition operation might be appropriate for many mathematical experts. For some systems, however, the designer may wish for the student to enter directives with complexity on the order of "*Now simplify for X.*" The algebra PS used in the examples for this thesis illustrates the use of different levels of granularity in the different parts of the PS. The subsystem COMBINE is written at a high level. For instance, RULE-TO-SUBTRACT-EQUATIONS performs the operations required to subtract two equations, all in one step. On the other hand, the SIMPLIFY subsystem is written at a relatively low level. The production COMBINE-TERMS-ON-LEFT-SIDE performs only a simple addition or subtraction when it is fired. This great variety is included in the sample PS for illustrative purposes. An ICAI system intended for classroom

use might also make use of systems of productions with various granularities. Fine grain productions could be used for skills that students are just learning while a courser grain would be appropriate for skills that have already been ascertained.

This chapter has presented an introduction to the PSMON production system monitor, an overview of the algebra problem-solving domain, and instructions for mating new domains with the tutorial interface. The PSMON features used by the algebra system were explained here so that the reader could understand the operation of the algebra PS. For further information about the features of PSMON, the reader is referred to Rood and Bregar (1980). This information should be sufficient for someone experienced in LISP programming to create a new domain and test it in conjunction with the tutor. Those who are not familiar with the writing of production systems are advised to first read and understand the operation of the algebra PS included in the appendix.

CHAPTER 6
CONCLUSIONS

6.1. Inadequacies and Extensions of the Current System

The tutorial interface has been implemented in order to demonstrate the unique abilities of its modelling system for tutoring applications. As has been discussed in section 4.1, there are many types of student models, and each provides its own peculiar set of information for use in tutoring. While the current implementation of the interface only demonstrates those features that are unique to its particular student modelling system, it can easily be extended to include other useful features as well.

A simple addition to the current interface is suggested by looking at the comparison, in section 4.2, of the current tutorial interface with Burton and Brown's twelve coaching principles. By changing the stopping condition for the backtracking facility it is possible to change the backtracker to undo a student's work only if it will lead to a significant improvement in the solution. This is as easy as changing the value of the constant that the backtracker uses when determining whether the distances of two states to the goal differ by enough to go on with the backtracking. One can imagine more sophisticated modifications to the backtracking function that would allow it to look beyond its current stopping point in order to get a more global picture of the progress of the solution. Even this more sophisticated modification would be little trouble to implement.

Although the interface does use an expert system for comparison with a student's solutions, it does not generate all of the data that a differential modeller is capable of. Augmentation of the modelling system to provide data about the strengths and weaknesses in a student's skills would eliminate the need for the assumption that the student already possesses all of the skills required to solve the problems in the problem-solving domain. It would then be possible to respond to Burton and Brown's principles 7 and 12, both of which need this additional data.

Principle 7 is concerned with using information about a student's weaknesses to allow the offering of congratulations when the student overcomes his problems. Principle 12, concerning the proper way to discuss a student's errors, suggests that the system might be able to use information about a student's strengths and weaknesses to determine when he has made a careless error. Information about a student's strengths and weaknesses may be used locally, within the interface, or passed on to a larger tutorial system in which the interface is included.

An improvement that might be made to the system is to have it spontaneously offer advice, rather than requiring a student to ask for it. Spontaneous advice might be appropriate when the interface has detected that the student is not making progress toward achieving the problem goal. An augmented differential model would allow the interface to determine when the student had a problem with some operation. The interface could then offer advice if a student seemed to be having trouble at a point in the problem solution where the problematic operation was required. Providing spontaneous advice would, of course, open the set of problems concerning when to offer advice. Burton and Brown's principles 1, 5, and 6 are related to this topic.

The simple natural language style interface that is used by the interface is one candidate for improvement. It was designed to be easy to implement, and to use few system resources. One way to improve it would be to completely replace it with a sophisticated natural language subsystem. Short of this, there is still room for improvement. A more sophisticated matcher can be used for finding common references between student directives and data bound by applicable foreground productions. This match is currently done by the LISP function EQUAL, which tests whether two LISP structures look exactly alike. A matcher that knows about the internal data representations used by the expert PS could improve the performance of the system. This was not included in the original implementation because the author was trying to remain as independent of the problem-solving domain as possible. The experience of the author in implementing the system has shown that there are many trade offs between independence from the problem domain and the quality of the performance of the system.

A problem that crops up from trying to use an expert problem solver that was not specifically written to run with the tutorial interface is that it is possible for the nature of the expert PS to limit the strategies that a student is able to use to achieve the goal. An example of this problem is shown in the protocol in figure 1-1, paragraphs 14 through 17. The preconditions for the REDUCE-GOAL-TERM

production of the SIMPLIFY production set require that the goal term be on the left side of the equation. Looking at the REDUCE-GOAL-TERM production in appendix C, it can be seen that the expression (ISTERM LEFT (< GOAL-VAR) (< EQN)) in the condition part is responsible for this behavior. Only after this condition is satisfied can the coefficient of the goal term be reduced to one. This type of problem can only be solved by ensuring that expert problem solvers are written in such a way that they avoid this kind of restriction.

In order to achieve the best results, the design of the problem-solving PS should take into account the nature of the tutorial interface. Many of the above mentioned problems concerning the use of the expert PS may be alleviated by proper concern for the design of the domain expert. Care must be taken in order to avoid constraining the student to the same solution strategy as that used by the expert PS. The expert PS must be designed so that the correct production will be enabled for firing at a particular stage of the problem solution. The design should also allow that all other productions that are applicable in the same problem state are also enabled for firing. When the expert PS is running without dependence on the interface, the conflict-resolution scheme will be responsible for determining the single correct production for firing. When the expert PS is being controlled by the interface, the enabling of all productions associated with applicable operators will allow the interface to chose the correct interpretation of a student's directive.

When constructing a problem-solving PS, it may be possible to solve problems in a particular domain without the use of certain common operators. The absence of these operators will limit the possible strategies open to students, however. It is advisable to include a large set of operators in the problem-solving PS so as not to limit the possible solutions open to the student. Any included operators not required by the problem-solving PS should be included in such a way that they are enabled in the appropriate situations, but are not chosen during conflict-resolution when the expert PS is running independently.

It is also important that the production rules that embody the domain operations should be able to apply the operators at a granularity appropriate to the level of prospective students. To provide a range of production granularity one can include several subsets of productions that all accomplish the same task, but at different levels of granularity. As above, conflict-resolution for the independently running expert PS will always choose one particular subset of these productions, but all appropriate productions will be enabled at each problem step. By this means many levels of granularity, and student ability, may be accommodated simultaneously.

One can see how it would be possible to include productions that embody erroneous operations, similar to the Leeds Modelling System or the BUGGY program, to model deficits in a student's repertoire. In order to detect subtle differences between correct and incorrect operator applications, it would probably be necessary to replace the current natural language style interface with a more exacting method. A possible replacement would be an interface that required the student to enter his results after each operation.

6.2. Summary and Conclusions

6.2.1. What was done

The implementation of the tutorial interface has served as a useful experiment. The interface includes three parts; a natural language style interface, a problem-solving PS, and an advising package. The expert PS serves both as a model of the student's solution and as a source of expert problem-solving knowledge for advising. The natural language interface is used to convey information from the student for use in maintaining a model of the student's solution. Information from the expert PS is used by the advising system to provide help to the student.

The natural language interface was designed to be fast, and is not always accurate. A student's directive is used to decide which applicable production from the problem-solving PS is to be fired on the next step. Using the conflict set to determine the intent of the directive, supplies an *expectation* of the student's intent for use by the natural language interface. The use of common references between a student's directives and the data bound by productions in the expert PS is another interesting feature of this part of the system.

The tutorial strategies employed by the advising component are intended to demonstrate the unusual capabilities afforded by the tutor's modelling scheme. The use of an expert system that can operate in the same context as the student allows the system to look ahead of the student's current problem-solving step. This is used in advising the student concerning his next operator application. It is also used to detect when a directive indicates an operation that is not currently applicable due to the absence of some intervening step, or steps. Generating hints using data that has been bound by the productions of the problem-solving PS is also demonstrated.

The algebra problem-solving PS fills the position of the computer expert system

in the interface. The system requires a problem-solving expert that is implemented as a production system. The tutorial interface derives its name from its role as a tutor that functions as an interface between a student and an expert PS.

6.2.2. What was learned

The tutorial interface has been successful in providing a source of information for advising students during the solution of multi-step problems. The single problem-solving PS serves both as the model of a student's solution, and as a source of expert problem-solving knowledge for tutoring. This configuration allows the expert PS to operate in the same context as a student who is solving a problem.

It is necessary to tailor an expert PS to avoid limiting the possible solution strategies open to a student. Care must be taken to include all operations that a prospective student may require to implement his solution strategy. The operations should be made available at several levels of granularity in order to accommodate a wide range of students.

The use of common references as a source of information for interpreting student directives seems to be a viable approach. When a foreground production in the expert problem solver refers to some part of the problem that is also referred to by the directive from the student, then the production and the directive have a reference in common. It is more likely that a directive indicates the operation of a production with which it shares a common reference than of one with which it does not have any references in common. This method is far from able to stand alone as an interpreter of student directives, but its use as one of several information sources seems to be warranted.

The experimental version of the tutorial interface is inadequate for classroom use, but there should be few problems encountered in augmenting the interface to make use of additional information sources. Most noteworthy is the ease with which additional differential modelling techniques may be applied. The interface already includes an expert knowledge source for use in a differential modeller. The representation of a student's strengths and weaknesses in a differential model, along with the present capabilities of the tutorial interface, would be most helpful in creating a useful tutoring system. These information sources would provide a solid base on which to implement a collection of tutoring and coaching strategies. It is recommended that an augmented version of the tutorial interface be included as a

component of a more complete educational computer system for purposes of testing its viability for classroom use.

BIBLIOGRAPHY

Bregar, William S. and Farley, Arthur M. *Interactive Problem Solving in Elementary Algebra.* Technical Report 78-3-1, Oregon State University, March 1978.

Bregar, William S., Farley, Arthur M., and Lantz, Brian S. *The Use of Production Rules in the Development of an Expert Tutor for Algebra.* Association for the Development of Computer-based Instructional Systems, June, 1982.

Brown, J.S. and Burton, Richard R. Diagnostic Models for Procedural Bugs in Basic Mathematical Skills. *Cognitive Science,* 1978, *2,* 155-192.

Brown, John S., Burton, Richard R., Zdybel, Frank. A Model-Driven Question-Answering System for Mixed-Initiative Computer-Assisted Construction. *IEEE Transactions on Systems, Man, and Cybernetics,* May 1973, *SMC-3(3),* 248-257.

Bundy, Alan and Welham, Bob. Using Meta-level Inference for Selective Application of Multiple Rewrite Rule Sets in Algebraic Manipulation. *Artificial Intelligence,* May 1981, *16(2),* 189-211.

Burton, Richard R. and Brown, John Seely. An investigation of computer coaching for informal learning activities. *Internation Journal of Man-Machine Studies,* January 1979, *11,* 5-24.

Carry, L. Ray, Lewis, Clayton and Bernard, John. *The Psychology of Equation Solving: An Information Processing Study.* NCTM, April, 1980.

Chapman, Loren J. and Chapman, Jean P. Atmosphere Effect Re-examined. *Journal of Experimental Psychology,* 1959, *58(3),* 220-226.

Charp, Sylvia. *Computer Technology in Education -- How to make it Viable,* pages 1-35. IFIP, August, 1970.

Clancey, William John. *Transfer of Rule-Based Expertise through a Tutorial Dialogue.* PhD thesis, Stanford University, September, 1979.

Clancey, William J. *Dialogue Management for Rule-based Tutorials,* pages 155-161. IJCAI, 1979.

Davis, Randall and King, Jonathan. *An Overview of Production Systems.* Technical Report AIM-271, Stanford Artificial Intelligence Laboratory, October 1975.

Goldstein, Ira. Developing a Computational Representation for Problem-Solving Skills. In *Problem Solving in Education: Issues in Teaching and Research,* Hillsdale, NJ: Lawrence Erlbaum Associates, 1980.

Hoffer, Edward P.,Barnett, G. Octo,Farquhar, Barbara B.,Prather, Penny A. Computer-Aided Instruction in Medicine. *Annual Review of Biophysics and Bioengineering,* 1975, *4,* 103-117.

Koffman, Elliot B. and Blount, Sumner E. *Artificial Intelligence and Automatic Programming in CAI,* pages 86-94. IJCAI, 1977.

Mayer, Richard E. *Thinking and Problem Solving: An Introduction to Human Cognition and Learning.* : Scott, Foresman and Company 1977.

Mcdermott, J. and Forgy, C. Production System Conflict Resolution Strategies. In Waterman, D.A. and Hayes-Roth, Frederick (Eds.), *Pattern-directed Inference Systems*, : Academic Press, Inc., 1978.

McKeachie, Wilbert J. Instructional Psychology. *Annual Review of Psychology*, 1974, *25*, 161-193.

Miller, G.A. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 1956, *63*, 81-97.

Mitzel, Harold E. The Impending Instruction Revolution. *Engineering Education*, 1970, *60*, 749-754.

Rood, Andrew Loyd and Bregar, William S. Production System Monitor User's Manual and Technical Report. Working Paper.

Seltzer, Robert A. Computer-Assisted Instruction -- What it can and cannot do. *American Psychologist*, 1971, *26*, 373-377.

Sleeman, D.H. *A System which Allows Students to Explore Algorithms*, pages 780-786. IJCAI, 1977.

Sleeman, D.H. A Rule-directed Modelling System. In R. Michalski, J.G. Carbonell, T.M. Mitchell (Eds.), *Machine Learning*, : Tioga Press, 1982. To be Published.

Sleeman, D.H. and Smith, M.J. Modelling Student's Problem Solving. *Artificial Intelligence*, May 1981, *16(2)*, 171-187.

Smith, Linda C. The Medical Librarian and Computer-Assisted Instruction. *Bulletin of the Medical Library Association*, January 1974, *62(1)*, 6-18.

APPENDICES

APPENDIX A
TUTORIAL INTERFACE PRODUCTION SYSTEM

```
(DEFV LTM (

(TUTOR:KEEPING-BOOKS
 IF ((A BACK-GROUND (*> CONFLICT-SET))
     (FAIL A PROD (*> PROD-TO-FIRE)))
    THEN
    ((SELECT-ONE (*< CONFLICT-SET))
     (DELETE (A BACK-GROUND (*< CONFLICT-SET)))))

(TUTOR:READ-USERS-TEXT
 IF ((A FORE-GROUND (*> CONFLICT-SET))
     (FAIL A TEXT (*> TEXT)))
    THEN
    ((READ-USERS-TEXT)
     (SHOVE (NEW-TEXT-FLAG))
     (CLEAR-UNIQUE-FIRING)))

(TUTOR:WEIGHT-PRODS
 IF ((A FORE-GROUND (*> CONFLICT-SET))
     (A TEXT (*> TEXT))
     (FAIL A DONE-WITH-TEXT))
    THEN
    ((WEIGHT-PRODS (*< CONFLICT-SET) (*< TEXT))
     (FREE-RUN--DECREMENT)
     (DELETE (A FORE-GROUND (*< CONFLICT-SET)))))

(TUTOR:RESOLVE-CONFLICTS1
 IF ((A PRODS (*> PRODS))
     (A NEW-TEXT-FLAG))
    THEN
    ((RESOLVE-CONFLICTS (*< PRODS) T)
     (DELETE (A PRODS (*< PRODS)))))
```

```
(TUTOR:RESOLVE-CONFLICTS+
 IF ((A PRODS (*> PRODS)))
    THEN
    ((RESOLVE-CONFLICTS (*< PRODS) NIL)
     (DELETE (A PRODS (*< PRODS)))))

(TUTOR:STACK-MEMORIES
 IF ((E A (*> MEMS)))
    THEN
    ((SETQ MEM-STACK (CONS (CONS 'A '(< MEMS)) MEM-STACK))
     (DELETE (E A (*< MEMS)))))

(TUTOR:ERROR
 IF ((A DONE-WITH-TEXT)
     (A NEW-TEXT-FLAG)
     (A TEXT (*> TEXT)))
    THEN
    ((DELETE (A NEW-TEXT-FLAG))
     (DELETE (A TEXT (*< TEXT)))
     (DELETE (A DONE-WITH-TEXT))
     (USER-ERROR (*< TEXT))))

(TUTOR:PUSH-AND-FIRE
 IF ((A PROD (*> PROD-TO-FIRE)))
    THEN
    ((DELETE (A PROD (*< PROD-TO-FIRE)))
     (DELETE (A NEW-TEXT-FLAG))
     (PUSH-AND-FIRE (*< PROD-TO-FIRE))
     (COND [FUNCT:POPPING
            (SETQ FUNCT:MASTER-STM STM)
            (SETQ FUNCT:MASTER-ITM ITM)
            (SETQ FUNCT:MASTER-ENV ENV)
            (SETQ STM (FUNCT:GET-STM))
            (SETQ ITM (FUNCT:GET-ITM))
            (SETQ ENV (FUNCT:GET-ENV))
            (SETQ PSMON:PSTRACE FUNCT:POPPING)
            (SETQ FUNCT:POPPING NIL)
            (DEACT)])))

(TUTOR:CLEANUP-AND-REPLY
 IF ((A DONE-WITH-TEXT)
     (A TEXT (*> TEXT))
     (FAIL A NEW-TEXT-FLAG))
    THEN
    ((DELETE (A TEXT (*< TEXT)))
     (DELETE (A DONE-WITH-TEXT))))
```

```
(TUTOR:GEN-CONFLICT-SET
 IF ((FAIL A (> FLAG) (NOT (EQUAL '(< FLAG) 'TEXT)) (*> DUMMY)))
    THEN
    ((GEN-CONFLICT-SET)))
))
```

APPENDIX B
KEYWORDS PROPERTY LIST

```
(RPLACPLIST 'FUNCT:KEYWORDS-PROP-LIST
            '(COMBINE ((COMBINE-TERMS-FROM-OPPOSITE-SIDES 1)
                       (COMBINE-TERMS-ON-LEFT-SIDE 1)
                       (COMBINE-TERMS-ON-RIGHT-SIDE 1))
              OPPOSITE
                       ((COMBINE-TERMS-FROM-OPPOSITE-SIDES 2))
              MOVE
                       ((MOVE-GOAL-TO-LEFT-SIDE 2))
              ISOLATE
                       ((ISO-GOAL-ON-LHS 2))
              REDUCE
                       ((REDUCE-GOAL-TERM 2))
              LEFT
                       ((COMBINE-TERMS-ON-LEFT-SIDE 1)
                        (MOVE-GOAL-TO-LEFT-SIDE 1))
              RIGHT
                       ((COMBINE-TERMS-ON-RIGHT-SIDE 2))
              PLUS
                       ((ISO-GOAL-ON-LHS 1)
                        (COMBINE-TERMS-FROM-OPPOSITE-SIDES 1)
                        (COMBINE-TERMS-ON-LEFT-SIDE 1)
                        (COMBINE-TERMS-ON-RIGHT-SIDE 1)
                        (MOVE-GOAL-TO-LEFT-SIDE 1))
              DIFF
                       ((ISO-GOAL-ON-LHS 1)
                        (COMBINE-TERMS-FROM-OPPOSITE-SIDES 1)
                        (COMBINE-TERMS-ON-LEFT-SIDE 1)
                        (COMBINE-TERMS-ON-RIGHT-SIDE 1)
                        (MOVE-GOAL-TO-LEFT-SIDE 1))
              TIMES
                       ((REDUCE-GOAL-TERM 2))
              QUOTIENT
                       ((REDUCE-GOAL-TERM 2))
              SUBSTITUTE
                       ((SUBSTITUTE 2))
```

```
REPLACE
        ((SUBSTITUTE 2))))
```

APPENDIX C
PROBLEM SOLVING PRODUCTION SYSTEM

```
(DEFV PROB-SOLVER (

(GET-GOAL-VARIABLE
 IF ((A HEAR (*> VARS) (ISVARLIST (< VARS)))
     (A NEW START))
    THEN
    ((SAY (MSG "Ok , which variable must we solve for? "))
     (REPLACE (A NEW START) (OLD START))
     (REPLACE (A HEAR (*< VARS)) (VARS (*< VARS)))
     (ITMSAVE (VARS (*< VARS)))
     (LISTEN)
     (SAY (MSG T))))

(GET-VAR-LIST
 IF ((A NEW START))
    THEN
    ((SAY (MSG "Type in the variables for your problem: "))
     (LISTEN)
     (SAY (MSG T))))

(NOTE-MAIN-GOAL
 IF ((A OLD START)
     (A VARS (*> VARS))
     (A HEAR (> VAR) (ISIN (< VAR) (< VARS))))
    THEN
    ((SAY (MSG "So " '(< VAR) " is our goal variable." T))
     (REPLACE (A HEAR (< VAR)) (GOAL (< VAR) (NIL)))))
```

```
(BEGIN-SIMPLIFICATION
 IF ((A SIMPLE (> EQN) (> ALIST))
     (A GOAL (> VAR) (> GALIST) (NOTSIMPLE (< EQN) (< VAR))))
    THEN
    ((SAY (MSG "The equation is not simplified in terms" T))
     (SAY (MSG "of the goal , so you need to simplify it."))
     (PS-PUSH SIMPLIFY TRACEFLAG (LIST 'A STM 'I ITM 'E ENV))
     (LIFT-MEMS)
     (SAY (MSG T))))

(IS-SIMPLE
 IF ((A SIMPLE (> EQN) (> ALIST)))
    THEN
    ((REPLACE (A SIMPLE (< EQN) (< ALIST))
              (INFO (< EQN) (< ALIST)))))

(NOTE-RESULT
 IF ((A GOAL (> VAR) (> GALIST))
     (A INFO
        (> EQN)
        (> ALIST)
        (ISABOUT (< VAR) (< ALIST))
        (ISRESULT (< VAR) (< EQN))))
    THEN
    ((BIND RESULT (RS (< EQN)))
     (REPLACE (A INFO (< EQN) (< ALIST))
              (RESULT (< RESULT) (< ALIST)))
     (SAY (MSG "We see we have determined a result." T))
     (SAY (MSG '(< VAR) " is equal to "))
     (SAY (MAPC 'PRINC (VALUEIN (< EQN))))
     (ITMSAVE (INFO (< EQN) (< ALIST)))
     (SAY (MSG T))))

(RETURN-TO-SUPERGOAL
 IF ((A GOAL (> VAR1) (> GALIST1) (HASPARENT (< GALIST1)))
     (A RESULT (> EQN) (> ALIST) (ISABOUT (< VAR1) (< ALIST)))
     (I GOAL (> VAR2) (> GALIST2) (ISPARENT (< VAR1) (< GALIST2)))
     (I VARS (*> VARS)))
    THEN
    ((REPLACE (A GOAL (< VAR1) (< GALIST1)) (OLD (< VAR1)))
     (SAY (MSG "So we return to the parent goal variable, which is "
               '(< VAR2)))
     (SAY (MSG T))))
```

```
(REPORT-RESULT
 IF ((A GOAL (> VAR) (> GALIST))
     (A RESULT (> SIDE) (> ALIST) (ISABOUT (< VAR) (< ALIST))))
    THEN
    ((SAY (MSG "Our final result is " '(< VAR) " equals "))
     (SAY (MAPC 'PRINC (VALUEIN (< SIDE))))
     (PS-POP)
     (SAY (MSG T))))

(SUBSTITUTE
 IF ((A GOAL (> VAR) (> GALIST))
     (A INFO (> EQN1) (> ALIST1) (ISABOUT (< VAR) (< ALIST1)))
     (A RESULT
        (> EXP)
        (> ALIST2)
        (ISABOUTVARIN (< ALIST2) (< EQN1) T1)))
    THEN
    ((SAY (MSG "We can substitute "))
     (SAY (MAPC 'PRINC (VALUEIN (< EXP))))
     (SAY (MSG " for " (ABOUT (< ALIST2)) T))
     (BIND NEWEQN (SUBSTITUTE (< T1) (< EXP) (< EQN1)))
     (SAY (MSG "The equation is now "))
     (SAY (MAPC 'PRINC (EQIN (< NEWEQN))))
     (REPLACE (A INFO (< EQN1) (< ALIST1))
              (SIMPLE (< NEWEQN) (< ALIST1)))
     (SAY (MSG T))))

(TWO-EQUATIONS-CAN-BE-COMBINED
 IF ((A GOAL (> VAR1) (> GALIST1) (HASPARENT (< GALIST1)))
     (A OLD (> VAR2) (> GALIST2) (ISPARENT (< VAR1) (< GALIST2)))
     (A INFO (> EQN1) (> ALIST1))
     (A INFO
        (> EQN2)
        (> ALIST2)
        (OR [NOT (EQUAL '(< EQN1) '(< EQN2))
            [NOT (EQUAL '(< ALIST1) '(< ALIST2))]])
        (SAMEVARS (< EQN1) (< EQN2))))
    THEN
    ((SAY (MSG "We remember that we have another equation "))
     (SAY (MAPC 'PRINC (EQIN (< EQN2))))
     (SAY (MSG T "that has the same variables." T))
     (SAY
      (MSG
       "If we combine them we can eliminate one of the non goal terms."
       T))
     (REPLACE (A GOAL (< VAR1) (< GALIST1))
```

```
              (OLD (< VAR1) (< GALIST1)))
      (REPLACE (A OLD (< VAR2) (< GALIST2))
              (GOAL (< VAR2) (< GALIST2)))
      (REPLACE (A INFO (< EQN2) (< ALIST2))
              (COMB1 (< EQN2) (< ALIST2)))
      (PS-PUSH COMBINE TRACEFLAG (LIST 'A STM 'I ITM 'E ENV))
      (LIFT-MEMS)
      (SAY (MSG T))))]

(RECALL-RESULT
 IF ((A GOAL (> VAR) (> GALIST))
     (A INFO (> EQN1) (> ALIST1) (ISABOUT (< VAR) (< ALIST1)))
     (I RESULT
        (> EQN2)
        (> ALIST2)
        (ISABOUTVARIN (< ALIST2) (< EQN1) T1)))
    THEN
    ((SAY (MSG "We remember that " (ABOUT (< ALIST2)) " is equal to "))
     (SAY (MAPC 'PRINC (VALUEIN (< EQN2))))))

(MAKE-VAR-GOAL
 IF ((A GOAL (> GVAR) (> GALIST))
     (A INFO (> EQN) (> ALIST1) (ISABOUT (< GVAR) (< ALIST1)))
     (A NEWVAR (> VAR) (> ALIST2)))
    THEN
    ((ITMSAVE (GOAL (< GVAR) (< GALIST)))
     (REPLACE (A GOAL (< GVAR) (< GALIST))
              (OLD (< GVAR) (< GALIST)))
     (REPLACE (A NEWVAR (< VAR) (< ALIST2))
              (GOAL (< VAR) (< ALIST2)))
     (SAY (MSG "So we will consider " '(< VAR) " to be our new goal."
               T))
     (SAY (MSG "We must solve for it in order to solve for " '(< GVAR)
               "." T))
     (SAY (MSG "We will remember that goal and go back to it later." T))
     (SAY (MSG T))))
```

```
(NOTE-VAR
 IF ((A GOAL (> VAR) (> GALIST))
     (A INFO
        (> EQN)
        (> ALIST)
        (ISABOUT (< VAR) (< ALIST))
        (NOTENEWVAR (< VAR) (< EQN) V1)))
    THEN
    ((SHOVE (NEWVAR (< V1) ((PARENT (< VAR)))))
     (BIND NEWGALIST (CONS '(PARENTOF (< V1)) '(< GALIST)))
     (REPLACE (A GOAL (< VAR) (< GALIST))
              (GOAL (< VAR) (< NEWGALIST)))
     (SAY (MSG "We notice that the variable " '(< V1) " appears" T))
     (SAY (MSG "in the equation "))
     (SAY (MAPC 'PRINC (EQIN (< EQN))))
     (SAY (MSG T))))

(NO-NEW-INFORMATION
 IF ((A GOAL (> VAR1) (> GALIST1) (HASPARENT (< GALIST1)))
     (A HEAR (> EQN1) (ISSAME (< EQN1) NONE))
     (I GOAL (> VAR2) (> GALIST2) (ISPARENT (< VAR1) (< GALIST2)))
     (A INFO (> EQN2) (> ALIST2) (ISABOUT (< VAR2) (< ALIST2))))
    THEN
    ((REPLACE (A GOAL (< VAR1) (< GALIST1))
              (OLD (< VAR1) (< GALIST1)))
     (REPLACE (A HEAR (< EQN1)) (OLD (< EQN1)))
     (REPLACE (A INFO (< EQN2) (< ALIST2))
              (OLD (< EQN2) (< ALIST2)))
     (SAY (MSG "Please type in a new equation about " '(< VAR2) T))
     (SAY (MSG "that does not contain the variable " '(< VAR1) ": "))
     (BIND STUFF (LIST 'HEAR (READEQU)))
     (SHOVE (< STUFF))))

(GET-NEW-INFORMATION
 IF ((A GOAL (> VAR) (> GALIST))
     (I VARS (*> VARS))
     (A HEAR (> EQN) (ISEQUATION (< EQN)) (HASVAR (< VAR) (< EQN))))
    THEN
    ((SAY (MSG "Ok! We have new information about " '(< VAR) "." T))
     (REPLACE (A HEAR (< EQN)) (SIMPLE (< EQN) ((ABOUT (< VAR))))))))
```

```
(REQUEST-INFO-ABOUT-A-GOAL
 IF ((A GOAL (> VAR) (> GALIST)))
    THEN
    ((SAY (MSG "Please input new information about " '(< VAR)))
     (SAY (MSG "Type in an equation containing the goal: "))
     (BIND STUFF (LIST 'HEAR (EQN-INCLUDING-GOAL (< VAR))))
     (SHOVE (< STUFF))
     (SAY (MSG T))))
))

(DEFV SIMPLIFY (

(COMBINE-TERMS-FROM-OPPOSITE-SIDES
 IF ((A SIMPLE
        (> EQN)
        (> ALIST)
        (SAMEVAR OPPOSITE TERM1 TERM2 (< EQN))))
    THEN
    ((BIND VAR (VARIN (< TERM1)))
     (BIND MESSAGE
           (COND ['(< VAR)
                  '(MSG "Combining terms from opposite sides involving "
                        '(< VAR)
                        T)
                 [T
                  '(MSG "Combining constant terms from opposite sides"
                        T)]))
     (SAY (< MESSAGE))
     (BIND MOD-EQN (MTERM (< TERM1) LEFT (< EQN)))
     (BIND TERM1 (NEGATE (< TERM1)))
     (BIND MOD-EQN
           (CTERMS (< TERM2) (< TERM1) RIGHT (< MOD-EQN)))
     (REPLACE (A SIMPLE (< EQN) (< ALIST))
              (SIMPLE (< MOD-EQN) (< ALIST)))
     (BIND EQN-OUT (EQIN (< MOD-EQN)))
     (SAY (MSG "produces the equation "))
     (SAY (MAPC 'PRINC '(< EQN-OUT)))
     (SAY (MSG T))))]
```

```
(COMBINE-TERMS-ON-LEFT-SIDE
 IF ((A SIMPLE
        (> EQN)
        (> ALIST)
        (SAMEVAR LEFT TERM1 TERM2 (< EQN))))
    THEN
    ((BIND VAR (VARIN (< TERM1)))
     (BIND MESSAGE
           (COND ['(< VAR)
                  '(MSG "Combining terms on the left side involving "
                        '(< VAR)
                        T)
                  [T '(MSG "Combining constant terms on the right side"
                           T)]))
     (SAY (< MESSAGE))
     (BIND MOD-EQN
           (CTERMS (< TERM1) (< TERM2) LEFT (< EQN)))
     (REPLACE (A SIMPLE (< EQN) (< ALIST))
              (SIMPLE (< MOD-EQN) (< ALIST)))
     (BIND EQN-OUT (EQIN (< MOD-EQN)))
     (SAY (MSG "produces the equation "))
     (SAY (MAPC 'PRINC '(< EQN-OUT)))
     (SAY (MSG T))))]

(COMBINE-TERMS-ON-RIGHT-SIDE
 IF ((A SIMPLE
        (> EQN)
        (> ALIST)
        (SAMEVAR RIGHT TERM1 TERM2 (< EQN))))
    THEN
    ((BIND VAR (VARIN (< TERM1)))
     (BIND MESSAGE
           (COND ['(< VAR)
                  '(MSG "Combining terms on the right side involving "
                        '(< VAR)
                        T)
                  [T '(MSG "Combining constant terms on the right side"
                           T)]))
     (SAY (< MESSAGE))
     (BIND MOD-EQN
           (CTERMS (< TERM1) (< TERM2) RIGHT (< EQN)))
     (REPLACE (A SIMPLE (< EQN) (< ALIST))
              (SIMPLE (< MOD-EQN) (< ALIST)))
     (BIND EQN-OUT (EQIN (< MOD-EQN)))
     (SAY (MSG "produces the equation "))
     (SAY (MAPC 'PRINC '(< EQN-OUT)))
```

```
     (SAY (MSG T))))]

(MOVE-GOAL-TO-LEFT-SIDE
 IF ((A GOAL (> GOAL-VAR) (> GALIST))
     (A SIMPLE
        (> EQN)
        (> ALIST)
        (BIND TERM1 (ISTERM RIGHT (< GOAL-VAR) (< EQN)))))
    THEN
    ((SAY (MSG "Moving the goal term to the left side" T))
     (BIND MOD-EQN (MTERM (< TERM1) RIGHT (< EQN)))
     (REPLACE (A SIMPLE (< EQN) (< ALIST))
              (SIMPLE (< MOD-EQN) (< ALIST)))
     (BIND EQN-OUT (EQIN (< MOD-EQN)))
     (SAY (MSG "produces the equation "))
     (SAY (MAPC 'PRINC '(< EQN-OUT)))
     (SAY (MSG T))))

(ISO-GOAL-ON-LHS
 IF ((A GOAL (> GOAL-VAR) (> GALIST))
     (A SIMPLE
        (> EQN)
        (> ALIST)
        (BIND TERM1 (ISTERM LEFT (< GOAL-VAR) (< EQN)))
        (NOTALONE LEFT (< EQN))))
    THEN
    ((SAY (MSG "Isolation the goal variable on the left side" T))
     (BIND MOD-EQN (ISOTERM (< TERM1) LEFT (< EQN)))
     (REPLACE (A SIMPLE (< EQN) (< ALIST))
              (SIMPLE (< MOD-EQN) (< ALIST)))
     (BIND PRINT-EQN (EQIN (< MOD-EQN)))
     (SAY (MSG "produces the equation "))
     (SAY (MAPC 'PRINC '(< PRINT-EQN)))
     (SAY (MSG T))))
```

```
(REDUCE-GOAL-TERM
 IF ((A GOAL (> GOAL-VAR) (> GALIST))
     (A SIMPLE
        (> EQN)
        (> ALIST)
        (BIND TERM1 (ISTERM LEFT (< GOAL-VAR) (< EQN)))
        (NOTRED (< TERM1))))
    THEN
    ((SAY (MSG "Making the coefficient of the goal term equal to one"
               T))
     (BIND MOD-EQN (REDUCE (< TERM1) (< EQN)))
     (REPLACE (A SIMPLE (< EQN) (< ALIST))
              (SIMPLE (< MOD-EQN) (< ALIST)))
     (BIND EQN-OUT (EQIN (< MOD-EQN)))
     (SAY (MSG "produces the equation "))
     (SAY (MAPC 'PRINC '(< EQN-OUT)))
     (SAY (MSG T))))

(RETURN-TO-SOLVE
 IF ((A SIMPLE (> EQN) (> ALIST)))
    THEN ((PS-POP)))
))

(DEFV COMBINE (

(ALIGN-EQUATIONS
 IF ((A INFO (> EQN1) (> ALIST1))
     (A COMB1 (> EQN2) (> ALIST2))
     (A GOAL (> VAR) (> GALIST)))
    THEN
    ((DOALIGNEQS (< EQN1) (< EQN2) (< ALIST2) (< VAR))))

(FIND-VARIABLE-TO-ELIMINATE
 IF ((A ORDERED (> EQN1) (> ALIST1))
     (A COMB2 (> EQN2) (> ALIST2))
     (FAIL A ELTERM1 (> TERM1))
     (FAIL A ELTERM2 (> TERM2)))
    THEN
    ((FNDVAREL (< EQN1) (< EQN2))))
```

```
(MAKE-COMPLEMENTARY-EQUATIONS
 IF ((A ORDERED (> EQN1) (> ALIST1))
     (A COMB2 (> EQN2) (> ALIST2))
     (A ELTERM1 (> TERM1))
     (A ELTERM2 (> TERM2)))
    THEN
    ((DOCCOMPT (< EQN1)
               (< ALIST1)
               (< EQN2)
               (< ALIST2)
               (< TERM1)
               (< TERM2))
     (SAY (MSG "We make two complementary equations by multiplying both"
               T))
     (SAY
      (MSG "equations by the coefficients of the term to be eliminated"
           T))
     (SAY (MSG "in the other equation." T))
     (SAY (MSG T))))

(RULE-TO-SUBTRACT-EQUATIONS
 IF ((A READY (> EQN1) (> ALIST1))
     (A READY
        (> EQN2)
        (> ALIST2)
        (NULL (AND [EQUAL '(< EQN1) '(< EQN2)
                   [EQUAL '(< ALIST1) '(< ALIST2)]])))
     THEN
     ((SAY (MSG "We now subtract all of the terms in "))
      (SAY (MAPC 'PRINC (EQIN (< EQN1))))
      (SAY (MSG T "from all the terms in "))
      (SAY (MAPC 'PRINC (EQIN (< EQN2))))
      (SAY (MSG T))
      (SUBEQS (< EQN1) (< ALIST1) (< EQN2) (< ALIST2))
      (SAY (MSG T)))]

(EQUATIONS-THE-SAME
 IF ((A REDUCED (> EQN) (> ALIST) (TERMSZERO (< EQN))))
    THEN
    ((SAY (MSG "Since all of the terms were zero then they were equal"
               T))
     (SAY (MSG "and it was not new information." T))
     (REPLACE (A REDUCED (< EQN) (< ALIST))
              (OLD (< EQN) (< ALIST)))
     (PS-POP)))
```

```
(ELIMINATE-TERMS-WITH-ZERO-COEFFICIENT
 IF ((A REDUCED (> EQN) (> ALIST)))
    THEN
    ((SAY (MSG "Our result after subtracting the two equations is "))
     (SAY (MAPC 'PRINC (EQIN (< EQN))))
     (SAY
      (MSG
       T
       "in which the term to be eliminated has a coefficient of zero."
       T))
     (BIND NEWEQN (ELZERTS (< EQN)))
     (SAY (MSG "So we eliminate the term to obtain the equation "))
     (SAY (MAPC 'PRINC (EQIN (< NEWEQN))))
     (SAY (MSG T T))
     (REPLACE (A REDUCED (< EQN) (< ALIST))
              (SIMPLE (< NEWEQN) (< ALIST)))
     (PS-POP)))
))
```