

Winter 1992

Traces: Modeling the teaching consultant in a problem-solving domain

Brian Leigh Johnson

University of New Hampshire, Durham

Follow this and additional works at: <https://scholars.unh.edu/dissertation>

Recommended Citation

Johnson, Brian Leigh, "Traces: Modeling the teaching consultant in a problem-solving domain" (1992). *Doctoral Dissertations*. 1710.
<https://scholars.unh.edu/dissertation/1710>

This Dissertation is brought to you for free and open access by the Student Scholarship at University of New Hampshire Scholars' Repository. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of University of New Hampshire Scholars' Repository. For more information, please contact nicole.hentz@unh.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9307351

Traces: Modeling the teaching consultant in a problem-solving domain

Johnson, Brian Leigh, Ph.D.
University of New Hampshire, 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

**TRACES: MODELING THE TEACHING CONSULTANT
IN A PROBLEM SOLVING DOMAIN**

by

Brian Leigh Johnson
B.S., University of New Hampshire, 1967
M.S., University of New Hampshire, 1979

DISSERTATION

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

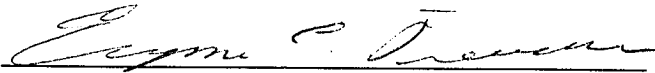
Doctor of Philosophy
in
Engineering

December, 1992

This dissertation has been examined and approved.



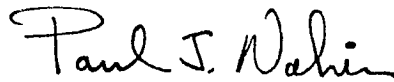
Dissertation Director, Dr. R. Daniel Bergeron
Professor of Computer Science



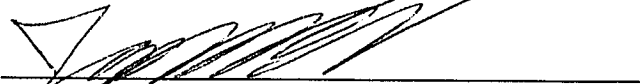
Dr. Eugene C. Freuder
Professor of Computer Science



Dr. David E. Limbert
Professor of Mechanical Engineering



Dr. Paul J. Nahin
Associate Professor of Electrical Engineering



Dr. James L. Weiner
Associate Professor of Computer Science

November 19, 1992

Date

ACKNOWLEDGEMENTS

I would especially like to thank Dr. R. Daniel Bergeron for his years of support and the many readings of this dissertation, Dr. David E. Limbert for his encouragement and insistence that I could and should finish, the rest of my committee for their efforts and encouragement, and Priscilla Malcolm for her work on the debugger portion of the system.

I would also like to thank my wife Nancy and our children for their patience over the years that it has taken me to complete this degree.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iii
LIST OF FIGURES	vii
ABSTRACT	viii
Chapter 1: INTRODUCTION	1
Chapter 2: Background	4
2.1. Consulting in Programming Courses	4
2.2. Conventional Teaching Model	5
2.2.1. Tutors	5
2.2.2. Coaches	6
Chapter 3: Overview: An Example	8
3.1. Student's problem	8
3.2. Generalized debugging process	8
3.3. Symptom acquisition	10
3.4. Student initiative	12
3.4.1. Debugging environments	12
3.4.2. Determining execution flow	15
3.4.3. Determining intermediate values	16
3.4.4. Viewing the output	17
3.4.5. Selecting the error line	18
3.5. Handling student error	19
3.5.1. Impossible input	19
3.5.2. Incorrect input	20
3.5.2.1. Impossible values	21
3.5.2.2. Incorrect symptoms	21
3.5.2.3. Incorrect student verification	21
3.5.3. Poor action	22
3.6. Consultant modes	24
Chapter 4: The Teaching Consultant	26
4.1. Teaching model	27
4.2. Student Model	29
4.3. Methodology of Consulting	34
4.4. Debugging Skills	34
4.5. Evaluation	35

4.6. Teaching	37
Chapter 5: Example of Understanding and Teaching	38
5.1. Sample prerequisite network	39
5.2. The teaching session	41
5.2.1. node relevancy	43
5.2.2. Student understanding	45
5.2.3. Topic teaching	47
5.2.4. Independent actions	49
Chapter 6: System design	51
6.1. Motivation	51
6.2. System overview	52
6.2.1. System scheduler	52
6.2.2. Input and Output	54
6.2.3. Problem environment module	54
6.2.4. Programmer debugger module	55
6.2.4.1. Parser	55
6.2.4.2. Variable comparator	56
6.2.4.3. Code comparator	57
6.2.4.4. Code emulator	58
6.2.4.5. Error localizer	58
6.2.5. Consultant module	59
6.2.5.1. Questioner	59
6.2.5.2. Goal sequencer	60
6.2.5.3. Goal consultant	61
6.2.5.4. Student Model and understanding	63
6.2.5.5. Task coach and teacher	64
6.3. Module interaction and design	64
6.3.1. Specialist invocation	64
6.3.2. Rule based shared databases	65
6.3.3. Rule based state machine behavior	66
6.3.4. Rule based function selection	67
6.3.5. Evaluation of design methodology	67
Chapter 7: Related work	69
7.1. Proust	69
7.2. Lisp Tutor	71
7.3. Scent-3	73
7.4. Apropos2	73
7.5. Spade-0	74

7.6. Talus	75
7.7. Teaching Algorithmics	75
7.8. Laura	76
7.9. Sophie	77
7.10. Hearsay-II	78
Chapter 8: Conclusion	79
8.1. Summary	79
8.2. Contributions	80
8.3. Future Research	82
Appendix A: Help provided by experts	83
Appendix B: Prerequisite network	84
Appendix C: Sample global database generation rules	89
Appendix D: State machine rules	94
D.1. Tutor task sequence rules	94
D.2. Coach state rules	99
Appendix E: Consulting function invocation rules	105
E.1. Student action rules	105
E.2. Supplementary information rules	107
E.3. Student response validation rules	111
E.4. Flow write placement validity rules	114
E.5. Debugging write correctness request rules	114
E.6. Student response correctness rules	115
E.7. Task completion rules	116
Appendix F: Teaching/coaching rules	118
F.1. Coach mode rules	118
F.2. Temporary coaching rules	119
F.3. Temporary teaching rules	120
F.4. Student understanding	120
Appendix G: Variable comparison rules	122
G.1. Input variable matching rules	122
G.2. Output variable machine rules	122
Appendix H: Rules governing input and output to the student	127
H.1. Reprompt rules	127
H.2. Requery rules	133
Appendix I: Dictionary	134
Bibliography	137

LIST OF FIGURES

1. Buggy program	9
2. Getting overall program state	10
3. Determining incorrect output	11
4. Student initiated actions	13
5. Viewing line numbers	14
6. Placing a <i>flow write</i>	15
7. Placing a <i>value write</i>	16
8. Viewing full output	17
9. Selecting error line	18
10. Handling bad input	20
11. Coaching an impossible response	20
12. Coaching an improper response	22
13. Coaching improper verification	23
14. Coaching mode	25
15. Multiple Prerequisites Combined with And Arcs	30
16. Alternate Prerequisites Combined with Or Arcs	31
17. A general prerequisite network	33
18. A misconception tree	36
19. Relevant portion of prerequisite network	40
20. Invoking the teacher	41
21. Action sequence when teaching	42
22. Relevant nodes in prerequisite network	46
23. Misconception tree	48
24. Topics to teach	49
25. Overview View	52
26. programmer-debugger	55
27. The Consultant	59
28. Goal sequencer states	60
29. Goal consultant states	61

ABSTRACT

TRACES: MODELING THE TEACHING CONSULTANT IN A PROBLEM SOLVING DOMAIN

Brian Leigh Johnson
University of New Hampshire, December, 1992

A model of a teaching consultant is presented. The teaching consultant, an extension of the concept of a computer coach, is concerned with filling gaps in a student's knowledge base relative to a specific application. The student model used by the consultant is stored in the form of a **prerequisite network**, which maintains nodes of information that are connected so that prerequisite information may be readily accessed. The Teaching Consultant Model is being used as the basis for an intelligent tutoring system serving as a programming consultant for novice programmers.

CHAPTER 1

INTRODUCTION

In many skill-oriented disciplines, a great deal of teaching takes place in the context of one-on-one *consulting*, in which a student requests help in discovering the error in a solution to a particular problem. Teaching within the context of consulting does not fit comfortably within traditional teaching or tutoring models. It provides both opportunities and constraints that demand a different perspective on the teaching process. A consultant is presented with an incorrect solution to a problem and must help the student diagnose and correct errors in that solution. The teaching context is based on *teaching by example*, but the process is driven by the student's own (incorrect) example, rather than ones provided by the consultant. In addition, a consultant should not just provide a corrected solution, nor should the consultant point out the errors to the student, but rather should *lead* a student to self-discovery of those errors and the needed corrections. The consultant must also use the student's actions to diagnose relevant gaps in the student's knowledge base. Consulting involves teaching problem solving at two levels, in the specific domain of the original problem and in the more general domain of diagnosis. The goal of the consultant is to improve the student's problem solving skills in both areas.

To accomplish its goals, the consultant must merge characteristics from both the coaching and the tutoring models. During the diagnostic and correcting phases, the

consultant must watch the student's actions, allowing the student flexibility in determining the actions to be performed, and must on occasion guide the student when the actions no longer are useful in the given situation. When the student performs incorrect actions, or does not respond to appropriate guidance hints provided, the consultant must switch to the teaching/tutoring mode, determine the reason for the student's difficulty, and teach the student the appropriate topics. Although the student's inability to diagnose the errors may stem from lack of knowledge or practice with diagnosis, it also may indicate gaps in the domain of the original problem. Thus, consulting during the diagnostic process may be used to tutor the student in the initial problem domain.

The student's knowledge base then must be comprised not only of informational, procedural, and heuristic knowledge within the specific problem domain, but also of procedural and heuristic knowledge in the diagnostic domain. This knowledge may be combined and represented in a *prerequisite network*, where the separate pieces of knowledge are associated with their prerequisites through *and/or* links. The consultant assumes that the student has been initially taught all the topics in the current ideal prerequisite network, and at the same time recognizes that there must be gaps in this knowledge relevant to the current problem. While tutoring the student, the consultant forms a *misconception tree* by discovering the problem relevant gaps or partial gaps in the student's knowledge base and repairs the student's knowledge by teaching, either fully or partially, the topics on the tree.

TRACES, an intelligent tutoring system which is fundamentally different from that of conventional Intelligent Tutoring Systems (ITS), is based on the paradigm of an

Intelligent Teaching Consultant (ITC) [JOHN90a] in the context of an introductory computer programming course. The goal of this system is help novice students find and correct logic and runtime errors in their programs, while teaching them conventional debugging techniques. Although this project is targeted as a programming consultant, the teaching model and the techniques are applicable to any discipline in which a teaching consultant can be used as a pedagogical tool.

The system is designed as a collection of independent expert systems, each with their own rule bases. Each expert specializes in a particular activity necessary to the consulting process. Taken as a whole, this collection of experts may be viewed as a team of specialists. Activation of the individual specialists is done by the team supervisor in response to specific requests for help from one of the other specialists. Although data is collected and utilized through the medium of several shared databases, the specialists are oriented more towards providing actions and services.

The focus of this dissertation is on the consulting model, the student model, and the specialist team concept, which comprise only a part of the entire system. Although it goes through the motions of a consulting session, the system itself is not complete. Considerable work and research are still required, especially in the areas of student action validation, student understanding, the actual teaching of topics, and the development of a more complete prerequisite network for understanding, debugging, and writing Pascal programs.

CHAPTER 2

Background

2.1. Consulting in Programming Courses

Novice programmers, especially those taking their first programming course, tend to need considerable help getting programming assignments to compile and execute properly. Supplying students with human tutors is one method for providing this aid. The help given by tutors can generally be classified into the following areas:

- a) understanding what problem to solve (understanding the assignment)
- b) understanding the solution to the problem
- c) understanding the necessary programming language constructs
- d) detecting and correcting errors in logic and code (debugging).

Experience has shown that students generally first *request* help in debugging, even if their true problems lie in one of the other areas. The tutor must determine the actual type of help needed, and then carry out the actions appropriate to the individual case.

Although it has been shown that students with private tutors perform better [BLOO84], tutors cannot generally be supplied on an individual basis. Instead, tutors must serve as programming consultants to all students in general. The time for any individual student is limited, sometimes to only a few minutes. While this may still be

superior to the classroom situation, the consultant often resorts to simply **showing** the students their errors, rather than **helping** them find the errors themselves. This does result in obtaining running programs for the students, but it does little to help them learn **how** to find their own errors. The lack of this ability increases the dependency of the student in later, more difficult, assignments and courses.

2.2. Conventional Teaching Model

In any teaching environment, the teacher is concerned with placing students in situations in which they may increase their knowledge. Conventionally, a teacher tries to present the student with information in a sequence of levels, each advancing or building on the knowledge at previous levels [HEIN85, SLEE81]. Thus the teacher forms a model of the student's knowledge and, with the use of a syllabus based on the progression of the material, places the student in different environments with the intent of advancing the student's knowledge one step at a time towards the ideal model.

2.2.1. Tutors

A tutor is conventionally viewed as a teacher working on a one to one basis with a student. Consequently, this teaching model also serves as the conventional tutoring model so that the conventional tutor

- 1) uses a syllabus representing the desired student progression through the material,
- 2) at each step in the progression, assumes that the previous knowledge has been acquired, and

- 3) attempts to add the next element of the syllabus to the student's knowledge base.

Many current tutoring systems are based on this conventional teaching model approach, and thus focus on the methods of organizing knowledge and of modeling the student to represent this progression. The combination of this information serves as the "syllabus" in these systems. BIP [BARR76] uses a tree of goals to store its curriculum. The WHY system [STEV82] uses "scripts" to represent reasoning sequences and attempts to find errors in student reasoning based on these sequences. WULSOR [GOLD82] uses a "genetic graph" to represent the association of procedural concepts and is used to determine topics for coaching the student. WEST [BURT79] builds a student model based on student performance, using it and differences between the student's and an expert's performance to determine topics to coach. BUGGY [BURT82] uses a "procedural network" to represent correct and incorrect methods of solving arithmetic problems, matching student's methods to correct methods through the use of example problems worked by the student. SCHOLAR [CARB70] uses a semantic network to represent information which is then used in a Socratic teaching environment.

2.2.2. Coaches

Coaches [BURT79, GOLD82] have been used to encourage new skill acquisition by discovery, particularly in *gaming environments*, in which it is acceptable that the student may lose. The *coach* compares the student's behavior to that of an expert, and occasionally interrupts the student with hints and suggestions as to how performance could be

improved.

Our system extends the concept of a *coach* into a *consultant*. The *consultant* deals with students who have already been presented with the necessary information in some other format. Their need is to improve their *problem solving* skills. Thus the *consultant*, although giving the student some freedom of choice, must insure that the students appropriately apply the techniques at their disposal. The *consultant* must act more as a guide to the students than the conventional *coach*, since the situation is one in which the students must *win* in order to find their errors.

CHAPTER 3

Overview: An Example

3.1. Student's problem

The following describes various interactions with TRACES during a debugging session. This chapter deals with student use of the system and the system's use of *coaching*. The *teaching* mode is discussed in the next chapter. For purposes of the discussion, assume the student has been given the following problem:

You are to find the average of a sequence of numbers. The number of values to sum will be the first value in the input, and this value will be followed by the specified number of values. You may assume that there is at least one value to average. The number of values, the total sum and the average are to be printed.

This problem requires that the student understand input/output, counting loops, the accumulation of a total, and an average. Figure 1 shows a possible solution by a student who did not understand the concept of keeping a count. The line in error is marked with a comment in the code. This error will cause the counting loop to be executed the wrong number of times, thus arriving at the wrong total and average.

3.2. Generalized debugging process

The debugging process might be viewed as consisting of the following steps:

- 1) let *set* be all lines in program
- 2) remove lines shown to be good from initial symptoms
- 3) if *set* contains one line - stop

```
program average(input,output);

var
  num: integer; {number of values to average}
  value: integer;{value to sum and average}
  sum: integer; {sum of values}
  ave: real;    {average of values}
  i: integer;   {loop counter}

begin

  sum := 0;

  i := 1;
  read(num);
  while i <= num do
    begin
      read(value);
      sum := sum + value;
      i := i + value  {ERROR}
    end;

  writeln('the sum of ',num,' values is ',sum);
  ave := sum / num;
  writeln('their average is ',ave);
end.
```

Figure 1. Buggy program

- 4) generate new symptom information
- 5) remove lines, which are shown to be correct by new symptoms, from *set*
- 6) repeat through step 3

Both experts and novices alike, to some degree, follow the above algorithm. They differ in how well they generate symptomatic information, and how well they can apply this information to localize the error. TRACES is designed to guide the student through this process.

3.3. Symptom acquisition

The system starts the session by asking the student for the problem number, so that it may access the appropriate problem database, and the file name of the student's program. In addition, the database containing the past history of the individual student is also loaded. As debugging relies heavily on symptom recognition and analysis, the student is required to determine and list the overt symptoms of the error. First, as shown in Figure 2[†], the overall status of the program is determined. Although the system generally uses *pick lists* when the student is provided with a choice of several items, the system as a whole is not dependent on this form of input. The form of questioning is determined by

```

what is the number of the programming assignment [value] ?> 10
please enter filename of your program [word] ?> student-10.p

adam, what is the state of your program

    1 - runs-to-completion
    2 - infinite-loop
    3 - eof-error
    4 - other-runtime-error

[ 1 - 4 ] ?> 1

```

Figure 2. Getting overall program state

[†]In some examples, the format of the output has been modified to enhance readability. Different fonts have been used as well: system output to the student is shown as "normal" text; student input is underlined; indented italic output represents actions requested of parts of the system which are currently stubs, those preceded by "<<<>>>" are requests for information not currently generated by the system.

the expert in charge of the actual input and output.

The system, through the program tracing and debugging experts, determines the correct symptom and matches its findings with the student's answer. Errors on the part of the student always result in some direct action by the coach or tutor. Such actions and the use of response verification information to direct general interaction philosophy are discussed later in this example.

After the overall program state has been determined, the student is questioned about the actual output of the program. Figure 3 shows the dialogue to specify the output items which were produced incorrectly in the program output. The pick list items in this case are problem specific, using phrases derived from the problem specific data base to identify the logical output, rather than having the student use actual variable names. In this way, the student is forced to think more in terms of **what** the program is required to do, rather than thinking of the actual code involved. As with the program state, errors invoke

adam, is there incorrect output [yes, no] ?> yes

adam, which are output which is bad

1 - number of values

2 - total of values

3 - average of values

[1 - 3] ?> 2 3

Figure 3. Determining incorrect output

the coach. Missing items, however, simply cause the system to ask the question again, thus indicating to the student that the answer given was not sufficient.

The student is questioned about missing and correct output in the same manner. In questioning about correct output, the system is stressing that knowledge of what is correct is as important as knowledge about what is incorrect.

3.4. Student initiative

After the directly accessible symptoms have been gathered, the initiative for interaction is turned over to the student. As shown by the earlier debugging algorithm, the student's goal during the debugging session is to generate sufficient symptomatic information describing the actions of the program to be able to deduce the locality of the error. The overall goal of the system is to teach the student: (1) how to generate this additional information; (2) how to determine what information is useful; and (3) how to utilize the information to isolate the problem.

3.4.1. Debugging environments

Although different environments provide a large variation in available debugging tools, they can generally be placed into two categories: those that provide information about the execution sequence of the code; and those which provide intermediate values generated during the execution. The main difference in the various debugging environments lays in the way that this information is requested and displayed, rather than in the actual information to be obtained.

Experience with novice students has shown that they often get "lost" when trying to manipulate these environments; they appear to spend more time trying to ascertain how to interface with the environment than they spend on the actual debugging process. Thus, TRACES is designed to simulate debugging in its simplest form, using solely the tools provided by the language itself. Thus the actions available to the student (see Figure 4) are not those available in a sophisticated interactive debugger, but rather simply simulate the placing of *debugging writes* into the program itself. Simulating these actions removes from the student the necessity of adding lines to the program and rerunning it; the debugging logic remains the same as if the student were actually to perform the steps.

A number of actions require the student to refer to specific lines in the program source. In this respect, TRACES is *line oriented*, and thus references are made relative to

what action would you like to perform

- 1 - done
- 2 - hint
- 3 - insert a flow write
- 4 - insert a value write
- 5 - remove a flow write
- 6 - remove a value write
- 7 - view output of program
- 8 - view lines of program
- 9 - found error

[1 - 9] ?>

Figure 4. Student initiated actions

the *line number* of the desired lines. The *view lines* action prints portions of the program along with the associated line numbers (see Figure 5).

Because there are two types of debugging information to glean from a program, there are two forms of debugging writes which may be used. The system provides the student with the ability to "place" these writes within the program, and to view the output which would result from such placement.

```

what action would you like to perform
    ...
    8 - view lines of program
    ...
[ 1 - 9 ] ?> 8

what is the line number for start of list [value] ?> 15

15 read(num);
16 while i <= num do
17   begin
18   read(value);
19   sum := sum + value;
20   i := i + value
21   end;
22
23 writeln('the sum of ',num,' values is ',sum);
24 ave := sum / num;

```

Figure 5. Viewing line numbers

3.4.2. Determining execution flow

A *flow write* is used to determine when and how many times a particular point in the program is reached. When the student picks this option (see Figure 6), the student is asked to pick a spot in the program to place the "write". The system responds with a count of the times the write is printed, which the student is asked to verify.

Student verification of the results of the debugging action is important, both to the student and to the consultant. The student's task during the debugging process is the collection and categorization of symptomatic information. Insuring that the student actively views each piece of data produced during the session forces the student to analyze the data as it is presented. Such "on the spot" analysis is important to the debugging process,

```

what action would you like to perform
...
3 - insert a flow write
...
[ 1 - 9 ] ?> 3

what is the line number of executable statement preceding flow write [value] ?> 19
      <<>> enter validity value [0..16] >> 16
      <<>> enter note [list] >> nil
printed 1 times

what is the number of times the line should have been executed [value] ?> 3

what action would you like to perform
...

```

Figure 6. Placing a *flow write*

as it helps the student to plan subsequent debugging steps. This should help limit any tendency to try to generate as much symptomatic information as possible, without thought of the implications of that data. The system uses correctness and appropriateness of the overall action as well as the correctness of the student's own validation of the debugging writes to determine the student's coaching/tutoring needs.

3.4.3. Determining intermediate values

The second form of debugging write is a *value write*, in which intermediate values are discovered. On picking this option (Figure 7), the student specifies both the desired line and the name of the variable to be viewed.

what action would you like to perform

...

4 - insert a value write

...

[1 - 9] ?> 4

what is the line number of executable statement preceding value write [value] ?> 16

please enter name of the variable to be written [word] ?> i

<<>> enter validity value [0..16] >> 16

<<>> enter note [list] >> nil

value(s) of variable at line is(are) (1 71)

are that the values for the variable correct [yes, no] ?> no

what action would you like to perform

...

Figure 7. Placing a *value write*

As with flow writes, the student is required to verify the correctness of the output and the student's success in the verification procedure as well as the appropriateness of the overall action are used to modify the system's view of the student's performance.

3.4.4. Viewing the output

Although it is often sufficient to simply know the number of times and/or values produced by the debugging writes alone, additional sequencing information may be obtained by viewing debugging write output in conjunction with the normal program output. Thus the system (Figure 8) provides the student with the option of viewing all the output, including that produced by debugging writes, which would be generated by the program in its current state.

```
adam, what action would you like to perform
...
7 - view output of program
...
[ 1 - 9 ] ?> 7

line 16 value write: i 1
line 19 flow write
line 16 value write: i 71
line 23 normal output: the sum of 3 values is 70
line 25 normal output: their average is 23.33333

what action would you like to perform
...
```

Figure 8. Viewing full output

3.4.5. Selecting the error line

Ultimately the student must produce sufficient symptoms to reduce the possible location of the error to a single line. Although able to determine when sufficient information has been collected, TRACES cannot by itself determine when the student has logically deduced this location. Since the student also may find the actual error visually, the student may have more information than can be attributed to symptomatic information alone. System action is restricted to simply verifying the correctness of the line when the student selects the *found error* action (Figure 9), rather than making any judgement on the appropriateness of the time of selection.

what action would you like to perform

...
9 - found error

[1 - 9] ?> 9

adam, what is the line number of executable statement in error [value] ?> 20

what action would you like to perform

...
1 - done

...
[1 - 9] ?> 1

do you desire an explanation of error and symptoms [yes, no] ?> no
normal system termination

Figure 9. Selecting error line

3.5. Handling student error

During the debugging process, the student may make three types of errors

- impossible response/request
- incorrect verification of result
- redundant/non-optimal action

These errors are handled at different levels of the system, as they generally imply different student states.

3.5.1. Impossible input

The modules responsible for the actual input/output handle most cases of impossible input. These routines, when invoked, are provided with a specification of what information to request of the student, and in many cases the range of possible values. How the question is asked, however, is determined by these routines themselves.

Since the questions indicate the type or possible range of answers, any failure on the part of the student to respond correctly indicates either a typographical error or a misunderstanding of the question itself. Thus, upon receipt of an impossible or unexpected response, the question is reprompted, and eventually may be phrased as a different type of question, depending on the situation and the difficulty that the student has in providing a reasonable response (see Figure 10).

what is the number of the programming assignment [value] ?> ten

You did not answer the question correctly, try again ?> ten

please enter an actual CONSTANT value or help ?> ten

The question is asking you to enter a value,
that is, a number, boolean, or character constant ?> 10

Figure 10. Handling bad input

3.5.2. Incorrect input

Student responses which pass initial input verification may still be incorrect. Although possibly due to typographical error, errors of this type more generally imply a

what action would you like to perform

...
3 - insert a flow write

...
[1 - 9] ?> 3

what is the line number of executable statement preceding flow write [value] ?> 5

*COACHING STUDENT for student goal localize-error
coach goal localize-error
student action insert-flow-write, supplementary info 5
validity (0)
notes (not-possible)
overall progress level 13.455216*

Figure 11. Coaching an impossible response

misunderstanding of the problem or of the language details. consequently, such errors always invoke some sort of immediate action by the system.

3.5.2.1. Impossible values

A request to insert debugging writes into non-executable code should be considered to be "impossible". However, assuming it is not a typographical error it implies a misunderstanding of program execution on the part of the student. Thus, the validity of the placement of such writes is determined by the *programmer module*, and errors handled by coaching/tutoring rather than simple reprompting (figure 11).

3.5.2.2. Incorrect symptoms

A similar approach is taken during the system initiated prompting for initial symptoms. For example, if the student chooses the wrong *overall state* of the program (Figure 12), the system coaches the student and then requests another answer. The decision to coach (or possibly tutor) immediately upon receipt of incorrect responses is based on the need for the student to be able to correctly recognize the symptoms of the error. Were the student to proceed using incorrect symptoms, the debugging session would result in failure, or at best with the student being reinforced for improper methods.

3.5.2.3. Incorrect student verification

For the same reason, whenever the student is presented with the results of a debugging write, verification of those results by the student is requested by the system. The student is asked to provide the **expected** results, which should be used by the student for

adam, what is the state of your program

...

4 - other-runtime-error

[1 - 4] ?> 4

*COACHING STUDENT for student goal get-program-state
 coach goal get-program-state
 student action other-runtime-error
 validity 0
 notes (wrong-answer)
 overall progress level of 14.4
 item desired is *program-state
 correct answer is runs-to-completion*

adam, what is the state of your program

...

Figure 12. Coaching an improper response

comparison against the actual results. Errors by the student while providing this information (1) indicate a lack of understanding of the expected execution of the program and (2) lead to the generation of incorrect symptoms. Thus, errors of this type are also coached/tutored immediately (figure 13), and the student is required to resupply the comparison values.

3.5.3. Poor action

Since the actions performed during the majority of the session are student initiated, it is possible for valid actions to be either redundant or non-optimal. Such actions correspond to valid but poor moves in a gaming situation [BADW91, BURT79, GOLD82], and

what action would you like to perform

...

3 - insert a flow write

...

[1 - 9] ?> 3

what is the line number of executable statement preceding flow write [value] ?> 19

<<>> enter validity value [0..16] >> 16

<<>> enter note [list] >> nil

printed 1 times

what is the number of times the line should have been executed [value] ?> 2

COACHING STUDENT for student goal localize-error

coach goal localize-error

student action insert-flow-write, supplementary info 19

validity (0 16)

overall progress level 14.4

what is the number of times the line should have been executed [value] ?> 1

COACHING STUDENT for student goal localize-error

coach goal localize-error

student action insert-flow-write, supplementary info 19

validity (0 0 16)

overall progress level 12.96

what is the number of times the line should have been executed [value] ?> 3

what action would you like to perform

...

Figure 13. Coaching improper verification

are handled in a somewhat similar fashion, by allowing the action to take place and providing the results of the writes without direct consultant intervention. However, these actions are *scored*, with the score being used to modify the overall *performance score* for the student. This performance indicator is used to determine the consultant's overall

interaction mode.

3.6. Consultant modes

During the session, the consultant may be in any of three possible modes: *watching*, *coaching*, or *teaching*. These modes, which may change during the session, are used to determine the overall method of interaction with the student. The current mode is derived from the student's overall performance during this and past sessions. It is an *aged* score in which the validity and appropriateness of all student actions during all sessions with the student are taken into account, with most current actions having more effect than those in the past. Actions, depending on their individual score, may have a positive or negative effect on the overall performance.

The consultant initially starts its first session with the student in *watching mode*, in which it interrupts the student, for the purpose of *coaching*, only on student error. As the performance score decreases, due to erroneous input or poor actions, the consultant shifts into the *coaching mode* (see figure 14), in which it gives the student hints between actions as to how to proceed, regardless of the correctness of the student's responses/actions. In this mode, when the consultant must intercede due to student error, it acts as a *tutor* rather than a *coach* (examples of tutoring are given in the next chapter).

If student performance improves, the consultant returns to *watching mode*; if it continues to deteriorate, it enters *teaching mode*, in which the student is essentially taught topics relevant to the next step.

what action would you like to perform

...

what is the number of times the line should have been executed [value] ?> 3

*COACHING STUDENT for student goal localize-error
coach goal localize-error
overall progress level 9.0*

what action would you like to perform

...

what is the number of times the line should have been executed [value] ?> 3

*COACHING STUDENT for student goal *localize-error
(coach goal *localize-error)
overall progress level 10.3*

what action would you like to perform

Figure 14. Coaching mode

CHAPTER 4

The Teaching Consultant

The activity of a consultant is generally centered around the solution to a specific problem; more specifically it involves detecting and locating errors in a student's solution to a specific problem. The consultant operates on two levels: overtly at the *debugging level* focusing on helping the student learn and use diagnostic techniques in order to find the error; and more covertly at the level of the original problem domain where the student understanding of the base problem is the issue.

In principle, the student who comes to the consultant for aid has already been taught all the material necessary for solving the problem. Additionally, if one were to consider just those students who have created a slightly buggy solution, the student has the knowledge, although possibly faulty, of how to apply this knowledge to solve the problem. The consultant is not directly concerned with finding and filling gaps in the student knowledge base, although this generally becomes necessary when evidence of such gaps appear naturally during the consulting process.

The goal of the consultant is to help the student find the errors in the problem at hand, and thus the consultant's actions are based on that context. The consultant does not actively search for areas in which the student's knowledge is flawed or missing, nor when gaps are found is the consultant concerned with fully filling the gap, but rather supplying

knowledge sufficient for the understanding of the current problem.

Thus, the goals and actions of the consultant are different from those of the conventional teacher or tutor, who are concerned with finding and filling all gaps in the student's knowledge base. Likewise they are different from the coach, in that relevant gaps must be found and filled, at least to the extent necessary for the current problem.

4.1. Teaching model

Since the consultant is driven by different goals, it is necessary to have a model which differs from the conventional teacher and tutor. In contrast to the more conventional teaching situations, consultation requires a highly flexible interaction between the student and the consultant. The consultant must determine the student's actual or perceived knowledge in the context of the specific problem at hand and must also deal with students who may have large unrelated gaps in their knowledge. The consultant is faced with finding and filling in gaps which relate to the specific problem, using the student's preliminary solution to the problem as the basis for the interaction.

This new model proposes that the teaching consultant

- 1) has a syllabus available which shows the ideal student knowledge state corresponding to the particular assignment;
- 2) assumes that the student should have the knowledge to solve the problem and to find the error in the solution;
- 3) assumes the student has not assimilated all the knowledge in this syllabus;
- 4) must be able to identify and fill only those missing gaps in the student's

knowledge which are required to find, understand and correct the current error;

- 5) must use the student's own debugging actions and responses during the consulting session to identify these gaps.
- 6) must identify students whose knowledge gaps are too extensive to be treated in this context, referring them to an instructor for a more conventional tutoring session.

The teaching consultant relies heavily on "teaching by example" [BOLD81, PALM75], using the student's program as the major source of examples. Although the examples used by the consultant are specific, it is hoped that the student will be able to generalize from these examples and thus gain more insight into the general concepts involved.

Rather than resorting to probing the student directly or through examples designed to reveal misconceptions, the consultant uses the debugging process itself to reveal the student's misconceptions and knowledge gaps. The student, during the debugging session, must be able to recognize the desired/expected results of debugging writes. Failures in this respect give evidence of misunderstandings and the lack of the necessary knowledge. Thus the student's debugging actions and symptom validation responses may be used as an indicator of understanding.

4.2. Student Model

As with all tutors, the teaching consultant must have a model, both of the ideal student and of the student currently being helped. The ideal model shows the knowledge that the student should possess in order to complete the current task. The model of the actual student is built by the consultant, both from past experience with this student, and from the current tutoring session.

Both models are represented as *prerequisite networks*. This representation shows the dependence of the different pieces of knowledge on other information in the network, although not necessarily the order in which the material was originally presented. The individual pieces of knowledge are stored in the nodes, while the arcs show the prerequisite information necessary to fully understand that knowledge. The direction of the arcs is from the more complex concepts towards the prerequisite knowledge nodes, giving an order which is conceptually the reverse of how the student should have learned the material. The importance of this ordering is that, from any node in the network, it is easy to find the prerequisites for that node.

The arcs in the network are of two types, corresponding to those in an AND/OR graph. AND arcs indicate that all the combined prerequisites are necessary to the understanding of the material in the node, while OR arcs show that there are several different ways of approaching the material. For instance, to understand a *conditional count* (a count which is updated only under certain conditions/constraints), the student must understand the concept of an **if** statement, the concept of counting and the problem specific constraint, and thus these nodes would be represented with AND arcs (see figure

15). On the other hand, the concept of determining evenness may be approached either through the understanding of the boolean function **odd**, or through a relational expression using the operator **mod**. Because either approach is acceptable, OR arcs would be used (see figure 16).

The ideal student model, which is specific to each assignment, is a representation of the knowledge which should have been attained by the student in order to complete that assignment. This includes programming, language, and debugging constructs and techniques that the student should have learned as well as any problem domain concepts necessary to understand the solution to the problem. For example, if the solution to the problem requires the understanding of how to determine evenness of an integer, this

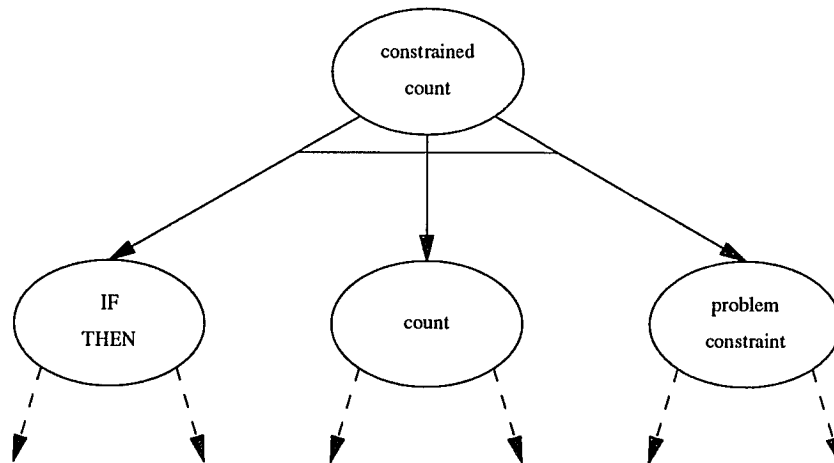


Figure 15. Multiple Prerequisites Combined with And Arcs

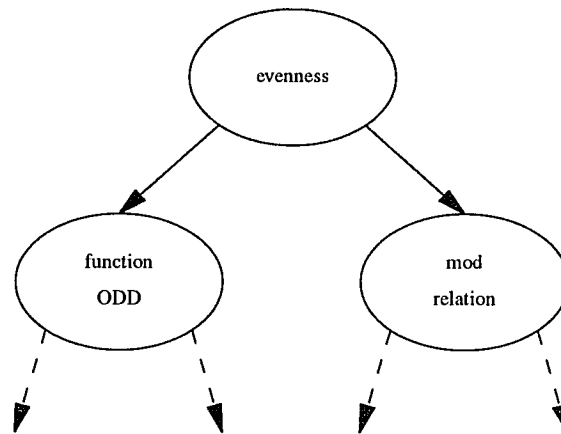


Figure 16. Alternate Prerequisites Combined with Or Arcs

information must be stored in the network, even though it may not have been specifically presented to the student through formal teaching sessions. The nodes of this ideal model are augmented with historical information pertaining to the degree of difficulty students at this level have had applying the knowledge, and with the types of misconceptions they have had. In addition to the usefulness of this information in predicting student difficulties, it also provides valuable feedback to the conventional instructor.

When teaching new material, a conventional teacher assumes that a given representation of the ideal current state of a student is the correct knowledge base and attempts to expand the base. The consultant initially makes the same assumption, based on the fact that the student has presumably already been taught the relevant material in the more conventional manner. The assumption, however, is not correct since the student's buggy

program clearly demonstrates that some nodes in the student's knowledge base are incorrect or missing. It is the consultant's job to identify these deficiencies and correct them.

The model for the current student initially looks like the ideal model. It is then augmented with information concerning past experiences with the student, marking nodes of the network which contain material that the student has either had difficulty with or has successfully used in previous sessions. This historical information is primarily used by the consultant during interaction with the student, as a means of relating the current session to past sessions. Because the student may have obtained additional tutoring or may have forgotten material since the previous session, this information does not necessarily reflect the student's present knowledge state accurately, and thus can serve only as an initial guide. during the current session.

The model is further modified during the consulting process, and at the end of the session it is preserved for use in future sessions with this student. It is also used to update the general student model to include this student's difficulties.

The differences between a teacher, a tutor, and a consultant may be shown using a sample prerequisite network (see figure 17). A teacher is responsible for teaching all the information in the network. The teacher assumes a certain level of previous knowledge and then starts by teaching the most basic knowledge (nodes at the bottom of the network), followed by the rest of the information, in hierarchical order, until the final knowledge (uppermost node) is imparted to the students. Although the teacher has some flexibility as to order, including *spiral* approaches to the material, the hierarchical order of the knowledge must be preserved. Thus, it is the function of the teacher to build and fill in

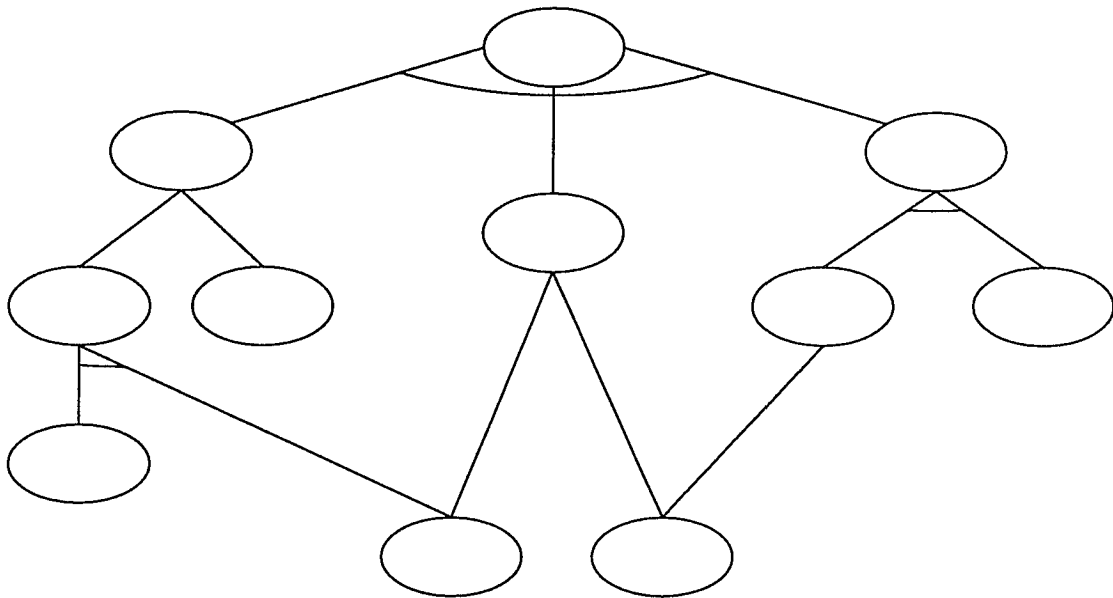


Figure 17. A general prerequisite network

the students' knowledge base.

The conventional tutor has the job of finding **all** the gaps in this knowledge base (nodes in the prerequisite network which are either partially or fully misunderstood or missing). Thus, the tutor uses questioning and sample problems to traverse all possible paths through the network with the student until all the required information is understood.

The consultant, however, is not concerned with the student's entire knowledge base, but only that knowledge which is relative to a particular problem. This means that the consultant need not be concerned with all nodes in the network, but only those which have impact on an actual problem. It is possible that partial knowledge of a node is

sufficient for the given instance. The consultant, then, only finds and corrects specific gaps in the knowledge base.

4.3. Methodology of Consulting

The programming consultant serves three very important roles:

- 1) to develop and exercise the student's *debugging skills*;
- 2) to *evaluate* the student's knowledge to determine what, if any, relevant conceptual weaknesses and gaps exist in the student's knowledge base; and
- 3) to *teach* the student by helping to fill in knowledge gaps.

4.4. Debugging Skills

Debugging skills are generally presented in the classroom, but as with other problem solving skills, they are only developed through practice. Students usually do not practice these methods until they encounter an actual error in a program. At this point, the student is faced, not only with a program which does not run correctly, but with having to utilize unpracticed techniques as well. One of the roles of the consultant is to show the students how to use this knowledge, and to guide them so that **they** find their own errors.

The relatively independent nature of the different debugging techniques allows the debugging knowledge to be maintained in the student model in **knowledge clusters** which consist of one or more interconnected nodes that are relatively independent of other information in the network. Thus, it is possible for the consultant to discuss the particular debugging techniques and tools that apply to the current situation, without

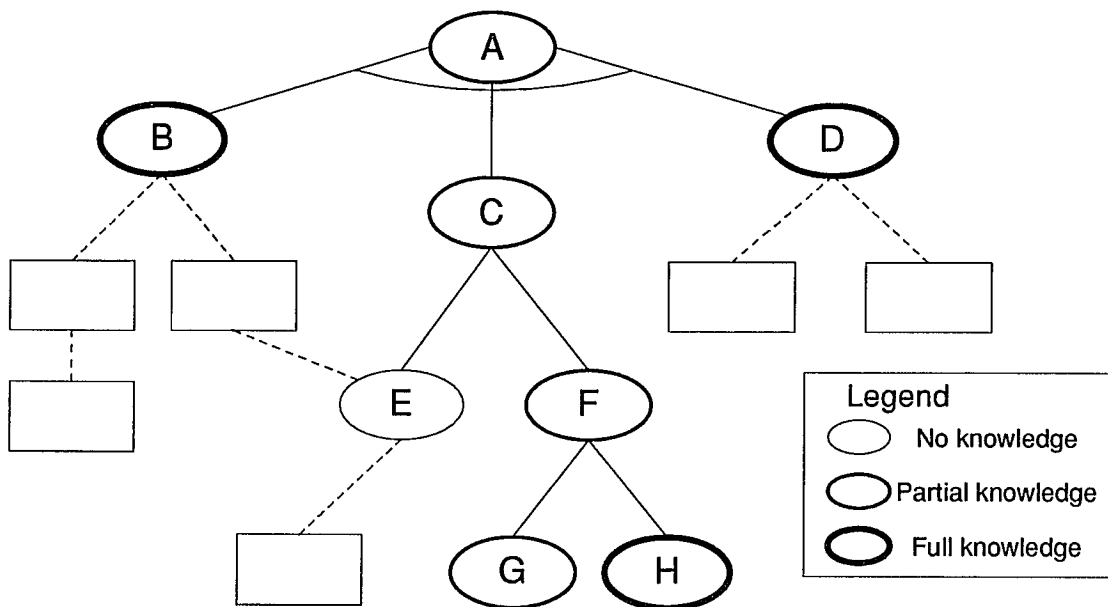
being concerned about the student's mastery of other techniques.

4.5. Evaluation

The second role of the consultant is activated during the debugging session. The gaps in the student's knowledge relevant to each error and misevaluation made while debugging must be found and repaired. When warranted by student performance, the consultant, using the student's program as a basis, questions the student about knowledge pertaining to the erroneous responses and actions. This probing starts at the most recently assumed knowledge in the student's prerequisite network and follows the arcs back through the prerequisite knowledge.

Since any particular assignment may not use all of the prerequisite knowledge for any specific node, the consultant need not question the student on all the items in the prerequisite chain. Instead, the consultant skips over nodes which do not contain knowledge which directly applies to the current problem and error. For example, even though the current knowledge state of the student includes compound boolean expressions, the student need not be questioned about this topic if the current error only involves simple relations. Likewise the consultant may omit any node which does relate to the actual error, but whose information the student correctly used. Once the student is shown to understand the information at a given node, and all of the facts relevant to the problem (pertinent to this branch of the tree) have been accounted for, its prerequisites need not be checked.

Thus the search through the student's knowledge takes the form of a traversal of a tree, called a *misconception tree*, which is formed from nodes from the *prerequisite network* (see figure 18). The ellipses in this diagram represent the nodes of the misconception tree, while the boxes are nodes of the prerequisite network which are not included in the tree. There are three types of nodes contained in a misconception tree: those which represent knowledge that the student appears to fully understand in the context of the given problem (nodes B, D, and H); those which represent knowledge which the student seems to partially understand (nodes A, C, F, and G); and those which represent information for which the student seems to have no understanding relative to the problem (node



E). If a leaf of the misconception tree is also a *sink node* of the prerequisite network, its prerequisites are based on *assumed external knowledge*. If such a node is not fully understood by the student, the system must either try alternate paths or assume, possibly incorrectly, that the student understands the assumed knowledge.

4.6. Teaching

As the consultant works down the prerequisite chain, each node in the student's model is annotated to indicate the student's level of success or failure in understanding the material. A post order traversal of this tree is then performed (actually this is done as the tree is being formed), presenting the student with the material in a "teaching" rather than "questioning" mode. This teaching process is directly applied to the specific problem, using the student's own program as its major example.

If the student is shown to have a large amount of unlearned information, either by having a great many unrelated errors in the program, or if the misconception tree has too many nodes representing incomplete knowledge, the consultant refers the student to a more conventional teaching session.

CHAPTER 5

Example of Understanding and Teaching

The teaching mode of the system is invoked only when coaching no longer appears profitable. The student's performance score, which is maintained throughout the debugging process, is used as a trigger to activate the transition between the different modes. This score rises and falls relative to the student's ability (or inability) to perform and verify debugging actions successfully. As long as the score remains above the initial trigger threshold, the system remains passive and responds to student error by simply giving hints as to the correct solution/response. As shown in the examples of the last chapter, the system enters the *consulting mode* when the student's score drops below this first threshold. In this mode, the system gives hints as to appropriate actions to take **prior** to the student picking an action, serving to focus the student's activity in a profitable direction. While the system is in the consulting mode, student errors no longer invoke the *consultant*, but rather a *topic teacher*. Should the performance continue to drop, the system activates its *teaching mode*, in which a topic teacher is activated prior to the student picking an action, in an attempt to teach the student the material needed for the next step in the debugging activity.

The topic teacher attempts to find and fill the gaps in the student's knowledge base which are relevant to the particular situation and the student's program. This chapter

explains the actions and sequencing of the teacher for a specific example.

5.1. Sample prerequisite network

As described in the last chapter, the prerequisite network is used to link knowledge in the form of a hierarchy based on necessary prior knowledge and contains no information relative to the actual order used when the student was initially taught the information. The teaching activity involves the use of this network. Figure 19 presents a visualization of the portion of the prerequisite network which is relevant to this sample teaching session. This represents only a portion of an entire prerequisite network and possible relationships to other parts of the complete network are not indicated in the diagram[†]. The letters outside the nodes are used to facilitate the following discussion, and are not actually part of the network.

It is important to note that the topics shown in the network in Figure 19 are broad representations of the sum total of the information the student has been currently taught about a particular topic, in essence the *concept*, rather than small individual facts. For example, to **fully** understand *boolean expressions* (node g5), the student must (fully) understand *boolean operators* (node h1), *boolean constants* (node h2), etc. Although not shown in this diagram, to fully understand boolean operators, the student would have to understand each of the individual operators: **and**, **or**, and **not**.

[†]As indicated in the previous chapter, an arc (*and*) connecting two or more children indicates all the connected children are required, while the absence of an arc (*or*) indicates that only one child is necessary. Situations involving both *and arcs* and *or arcs* are sometimes difficult to represent pictorially, as is the case of the children of node d1. In this case, shown by the use of dashed arcs, only the ends of the arcs indicate the *and* relationship, and thus the prerequisites of node d1 are (e1 and e2) or (e1 and e3) or (e1 and e4).

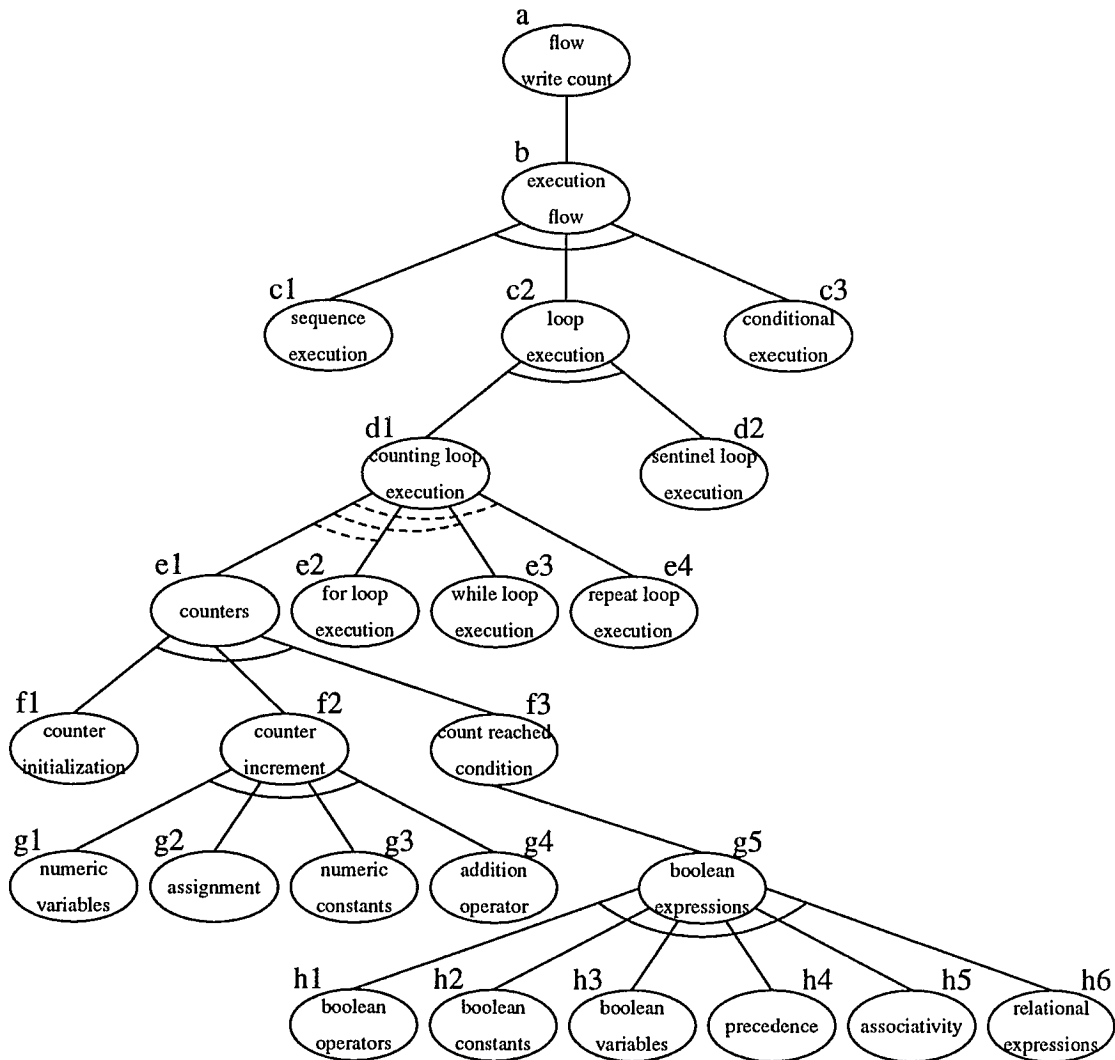


Figure 19. Relevant portion of prerequisite network

The network does not represent all the possible information on a topic, nor does it only contain that which the student needs for the current problem, but rather it contains all the information that the student should currently possess.

5.2. The teaching session

The teaching session portrayed in this example was invoked (see figure 20) by a student error when the system was in *coaching mode*.

The teaching session consists of two separate but associated tasks: finding the relevant gaps in the student's knowledge base and filling those gaps. The sequence of actions taken by the teacher is shown in Figure 21[†]. The prerequisite network is used to direct

what action would you like to perform

...

3 - insert a flow write

...

[1 - 9] ?> 3

what is the line number of executable statement preceding flow write [value] ?> 19

<<>> enter validity value [0..16] >> 16

<<>> enter note [list] >> nil

printed 1 times

adam, what is the number of times the line should have been executed [value] ?>

1

*TEACHING STUDENT for student goal *localize-error*

*(coach goal *localize-error)*

student action insert-flow-write, supplementary info 19

validity 0

overall progress level 6.9

Figure 20. Invoking the teacher

[†]Numbers were artificially placed on various lines. References in the form *line x* are used throughout the chapter without specification of figure number.

1 <<> enter root topic [topic] >> flow-write-count
 2 <<> enter relevancy for flow-write-count [t/nil/skip] >> t
 3 <<> enter understanding for flow-write-count [none/partial/full] >> partial
 4 <<> enter relevancy for execution-flow [t/nil/skip] >> skip
 5 <<> enter relevancy for sequence-execution [t/nil/skip] >> nil
 6 <<> enter relevancy for loop-execution [t/nil/skip] >> skip
 7 <<> enter relevancy for counting-loop-execution [t/nil/skip] >> t
 8 <<> enter understanding for counting-loop-execution [none/partial/full] >> partial
 9 <<> enter relevancy for for-loop-execution [t/nil/skip] >> nil
 10 <<> enter relevancy for while-loop-execution [t/nil/skip] >> t
 11 <<> enter understanding for while-loop-execution [none/partial/full] >> full
 12 <<> enter relevancy for counters [t/nil/skip] >> t
 13 <<> enter understanding for counters [none/partial/full] >> partial
 14 <<> enter relevancy for count-initialization [t/nil/skip] >> t
 15 <<> enter understanding for count-initialization [none/partial/full] >> full
 16 <<> enter relevancy for count-increment [t/nil/skip] >> t
 17 <<> enter understanding for count-increment [none/partial/full] >> none
 18 <<> enter relevancy for numeric-variables [t/nil/skip] >> nil
 19 <<> enter relevancy for assignments [t/nil/skip] >> nil
 20 <<> enter relevancy for numeric-constants [t/nil/skip] >> nil
 21 <<> enter relevancy for addition-operator [t/nil/skip] >> nil
 21a ||FULL TEACH of count-increment
 22 <<> enter relevancy for count-reached-condition [t/nil/skip] >> t
 23 <<> enter understanding for count-reached-condition [none/partial/full] >> partial
 24 <<> enter relevancy for boolean-expressions [t/nil/skip] >> skip
 25 <<> enter relevancy for boolean-operators [t/nil/skip] >> nil
 26 <<> enter relevancy for boolean-constants [t/nil/skip] >> nil
 27 <<> enter relevancy for boolean-variables [t/nil/skip] >> nil
 28 <<> enter relevancy for precedence [t/nil/skip] >> nil
 29 <<> enter relevancy for associativity [t/nil/skip] >> nil
 30 <<> enter relevancy for relational-expressions [t/nil/skip] >> t
 31 <<> enter understanding for relational-expressions [none/partial/full] >> full
 31a ||FILL TEACH of count-reached-condition
 31b ||FILL TEACH of counters
 31c ||FILL TEACH of counting-loop-execution
 32 <<> enter relevancy for sentinel-loop-execution [t/nil/skip] >> nil
 33 <<> enter relevancy for conditional-execution [t/nil/skip] >> nil
 33a ||FILL TEACH of flow-write-count

Figure 21. Action sequence when teaching

the search and to provide the topics for discussion[†].

The system picks a root topic for the search (line 1). This is a node (node a) within the prerequisite network and correlates to what becomes the root node in the misconception tree. The system then traverses the network following prerequisites to build the misconception tree. Although the various actions of the tutor at the nodes are intermixed during actual operation, it is simpler to describe them when considered separately.

5.2.1. node relevancy

Since tutoring the student is based on the issues arising from debugging a particular problem, the tutor need not be concerned with all the information within the prerequisite network. Only those paths which contain relevant information need be followed; the rest can be ignored. Thus, a full traversal of the network is not necessary, but may be truncated by eliminating nodes (and thus their prerequisites) which are not relevant to the current situation.

As a node is encountered during the traversal, its relevancy is determined. There are four possible situations: the node contains information which is not necessary to the current situation, and is not used by the student; the information is not necessary, but is used by the student; the information is necessary; and the topic at the node is too broad - that is, the topic is generally applicable to the situation, conglomerating other specifically applicable topics.

[†]The portions of the system pertaining to topic relevancy, student understanding and actual teaching of topics are topics for future research. Although they currently exist as rule based experts, the rules simply prompt for the desired values rather than deriving them.

In the example script of Figure 20, the student has incorrectly specified the number of iterations which *should* have been made by the program. This places the system in a temporary teaching mode and causes the sequence of actions shown in the script of Figure 21. Here, the root topic *flow write count* (node a) is relevant, and is included (line 2) in the possible topics for discussion. *Execution flow* (node b), however, is a broader topic than the tutor desires to discuss, serving primarily to group the different forms of control flow and thus is skipped (line 4), allowing its prerequisites to be considered, without becoming a tutoring topic itself.

Progressing down the network, although the code block under consideration does contain a sequence of statements, the issue is loop iterations, and thus *sequence execution* (node c1) is not relevant (line 5) and thus is eliminated, along with its prerequisites, from the search for topics, decreasing the overall search effort.

Since an *and arc* connects the prerequisites of *execution flow*, the system must check *conditional execution* (node c3, line 33) for relevancy, even though *loop execution* (node c2, line 6) is viable, albeit broad, in the context of the problem. In contrast, to understand *counting loop execution* (node d1), the student needs to understand *counters* and only *one* of the three possible loop constructions, indicated by the use of the *or arcs*. Thus, when it is found that *while loop execution* (node e3) is relevant (line 10), the relevancy of *repeat loop execution* (node e4) is never checked.

The search for nodes in any particular path within the network is terminated on one

of two conditions[†]. The traversal of a path naturally terminates when a *leaf* node is encountered, that is, a node with no prerequisites. Such a node is based on assumed knowledge, which the system also assumes and makes no attempt to tutor. The second case is exemplified by the search through the prerequisites (nodes g1, g2, g3, and g4) for *counter increment* (node f2). In each case, they are determined to be non-relevant to the problem (lines 18, 19, 20, and 21). Since all the prerequisites to a relevant node are not relevant, the traversal of that path terminates.

The system continues through the prerequisite network in this manner until the tree of relevant nodes is derived (figure 22)^{††}.

5.2.2. Student understanding

Once the relevancy of the topics is determined, the system must ascertain what, if any, understanding the student has of these topics. The tutor traverses this tree of relevant nodes and annotates the nodes with the degree of understanding the student shows for each topic^{†††}. This serves the dual purpose of determining which topics to teach the student and additionally serves to limit the topic search further.

Unlike conventional tutoring, understanding of any particular topic is determined relative to the specific instance which invoked the tutoring mode; hence only relevant nodes are investigated. Similarly, if a student only understands a small portion of the

[†]Actually there are three conditions, when one considers the intermixing of the actions during traversal. Only two of them, however, relate to topic relevancy.

^{††}Topics within light ellipses (in contrast to bold ellipses) in the figure indicate broad topic nodes which have been skipped.

^{†††}As shown in the example script, this is done concurrently with relevancy testing.

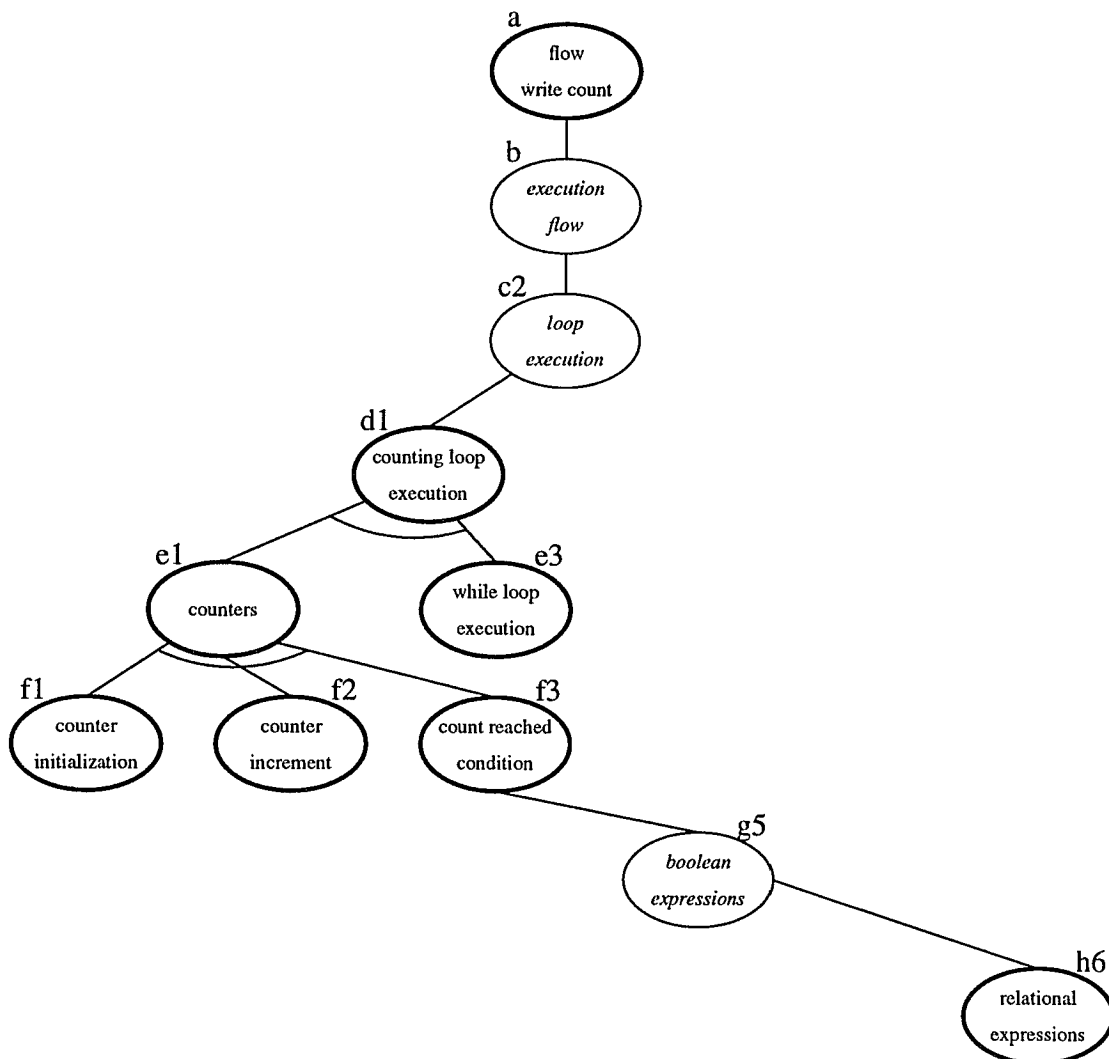


Figure 22. Relevant nodes in prerequisite network

information in a node, and that is sufficient in terms of the current situation, the knowledge would be considered to be *full*. Likewise, where a student understands the majority of the material represented by a node, but has no understanding of that which specifically relates to the the situation, the student is considered to understand *none*. Between these

two extremes the student can have *partial* knowledge.

Traversal of the tree branches continues as long as the student has no or partial knowledge of the topics at the nodes. However, when the student shows full understanding, for example *while loop execution* (node e3, line 11), the search along that path is terminated, providing the third method of limiting the search through the network, even though the node might contain relevant prerequisites. This termination is justified, as by the definition of prerequisite knowledge, the student could not have full understanding of the current node if any of its prerequisites were missing.

The result of this operation is the *misconception tree* (figure 23). It contains the relevant nodes of the network, annotated with the extent of the student's understanding.

As the example script shows (line 4 node b), the relevancy of a topic is checked prior to checking for student understanding, thus eliminating understanding checks on topics which are not relevant. Likewise, it is not necessary to check relevancy or understanding for topics which are prerequisites to fully understood items. In this way, the entire prerequisite network need not be traversed in order to construct the misconception tree.

5.2.3. Topic teaching

The topic teacher must teach the topics on the misconception tree which are not fully understood (Figure 24). Topics for which there is no understanding must be fully taught to the student (node f2, line 21a). Nodes which are partially understood need only have the missing bits and pieces filled in (nodes a, d1, e1, and f3; lines 33a, 31c, 31b, and

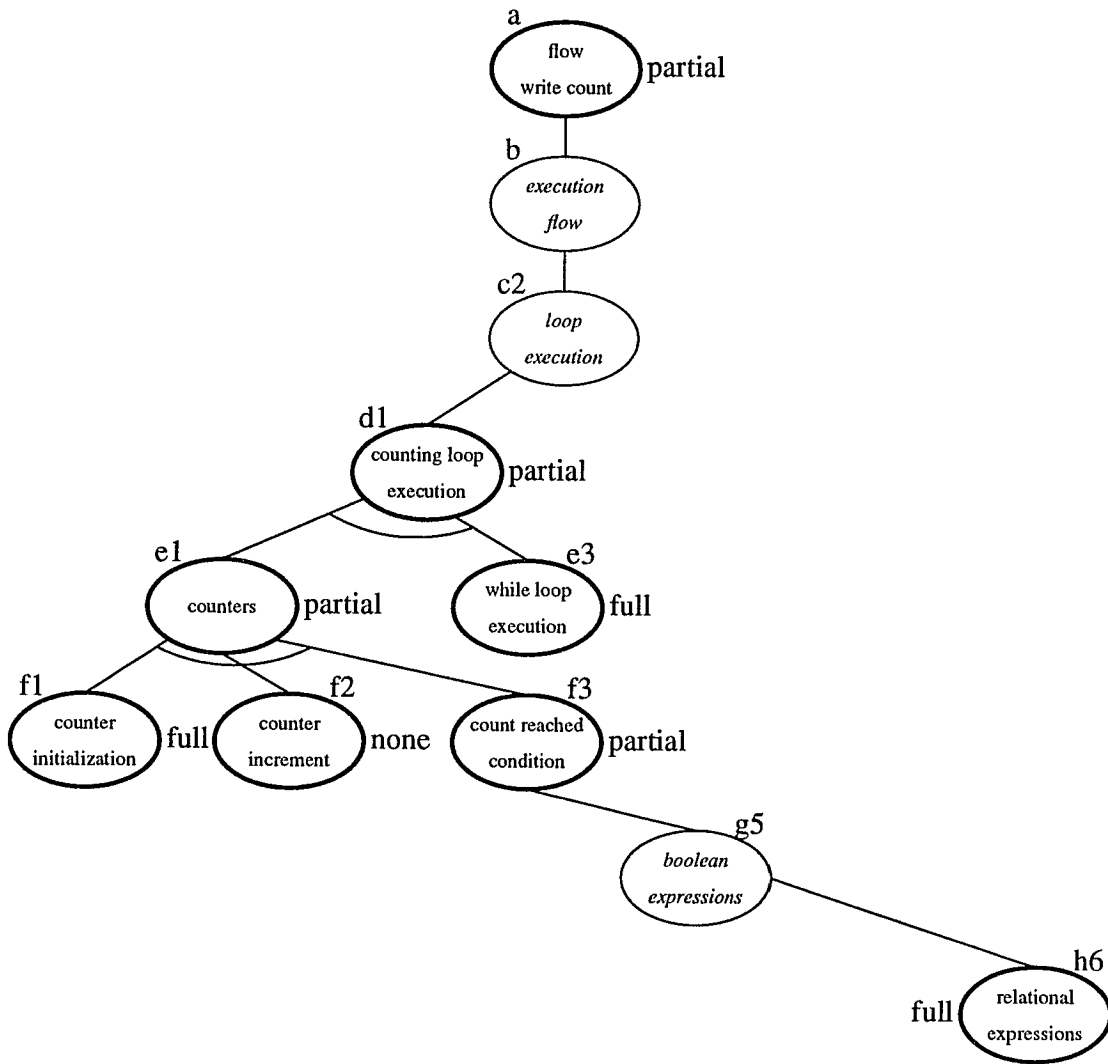


Figure 23. Misconception tree

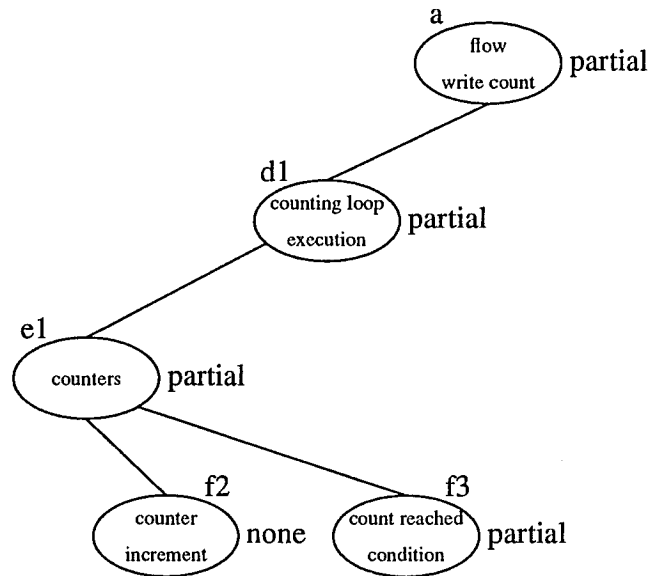


Figure 24. Topics to teach

31a).

If the student is lacking in understanding of both a node and one or more of its prerequisites, the prerequisites must be taught prior to the node (nodes f2, f3, and e1; lines 21a, 31a, 31b). Teaching may thus be accomplished with a *post order* traversal of the tree. Since the tree is being constructed with pre-order decisions, the information may be taught as the tree is being constructed, as is shown by the example script.

5.2.4. Independent actions

The tree is searched for student knowledge gaps concurrently with the teaching of the material. However, the two forms of decisions made by the system, topic relevancy

and student understanding, as well as the actual teaching of the topics are all accomplished separately. Each different action is controlled by a separate *expert* within the system.

CHAPTER 6

System design

6.1. Motivation

TRACES is intended to be used in conjunction with normal classroom activities to aid students in the successful completion of programming assignments at the introductory level. The intent is to reduce the load on human consultants associated with the introductory courses so that their time may be more effectively spent with students who require further aid. Thus, the system is not intended to be a general purpose debugger and tutoring system, but rather is focused on the specific assignments of these courses with an emphasis on teaching students how to debug their programs.

One of the goals of this project is to design and implement a knowledge based system using conventional *software engineering* principles, such as modularization, top down design, and data abstractions. The following correlations evolved:

modularization → independent experts (*specialists*)

data abstractions → rule based global databases

state machine design approach → rule based state machines

As a result, the overall system may be viewed as manifestations of these concepts.

6.2. System overview

In keeping with the desire to maintain a strict modularization and localization of tasks, the system is designed as a collection of relatively independent *expert systems* which emulate the different personalities embodied in a human consultant (see figure 25). Each of the "modules" shown in this figure are not individual expert systems themselves, but are rather collections of small individual experts who in combination are responsible for a particular aspect of the tutoring session. The system is also dependent on numerous local and global rule-based databases.

6.2.1. System scheduler

Each of the modules and submodules within the system act as individual specialists. As such, their expertise is limited and focused in a specific area. Whenever they encounter the need for information or actions which are outside their own speciality, they make a request for the specific help needed just as is done by their human counterparts.

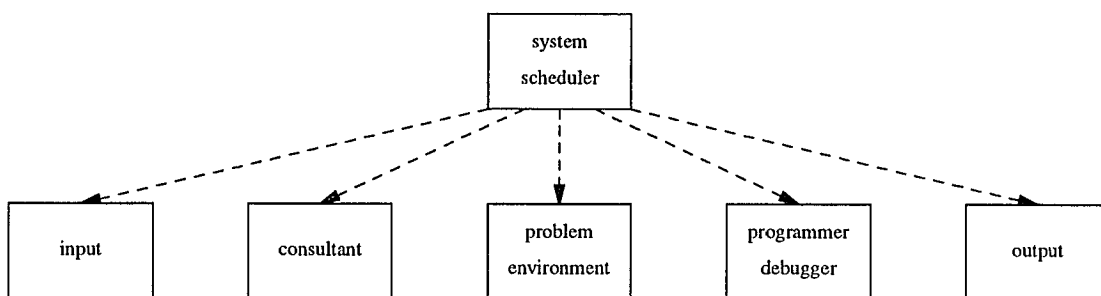


Figure 25. Overview View

As might be the case of human specialists working within a large organization, they do not know which specific specialist is most qualified to provide the necessary help. Thus the specialist in need simply makes a *help request* to the system, specifying the nature of the aid desired.

Upon system startup, each specialist informs the system about the aid it can provide. The *system scheduler* collects this information in the form of a dynamically generated rule base (see Appendix A). It then uses these rules to satisfy help requests during the consulting session by invoking the appropriate modules. Once the request for help is accepted by the specific module, it is translated into its own module specific calls and notation.

Upon successful completion of the task by the invoked module, the scheduler hands the results back to the original requester. If the invoked module is not able to complete the desired action successfully, and there are other modules which may provide similar aid, they in turn are invoked by the scheduler, until the desired help has been given, or the supply of participating specialists is exhausted.

In summary, although the individual modules must request help of other modules, they do not make specific calls to those modules, and in fact do not know which module actually provides the desired information. This allows for considerable flexibility in design and a high degree of modularization.

6.2.2. Input and Output

The *input* and *output* modules are responsible for the direct interaction with the student. These modules do not determine what is to be told to the student, nor the questions asked of the student, which is specified by other modules. The I/O modules do determine **how** the material is to be presented to the student[†].

Requests from other modules for student interaction, both for presenting information and questioning the student use an *internal vocabulary* in a formal form. The I/O modules use a *dictionary* (see Appendix I) which translates the internal form into an acceptable external form in the specified context.

The I/O modules are also responsible for rephrasing questions which are answered improperly by the student. This is accomplished through a set of rules (see Appendix H.1) which specify alternative forms for specific kinds of questions, based on the number of times the question has been rephrased.

6.2.3. Problem environment module

The *problem environment* module is responsible for supplying the appropriate problem specific databases required by the rest of the system^{††}. The primary function of this module is to supply a sample solution which closely matches the student's form of solution, along with a runtime trace of this solution and a correlation of the input and output

[†]The primitive form of interaction (pick lists, yes/no, etc) shown in the examples of the previous chapters are strictly a function of the input and output modules.

^{††}Currently this portion simply reads in previously generated files, based on problem number. A topic for future research is the generation of alternative solution forms from one or more samples.

variables in the solution to the terminology used in the assignment description. This information is primarily used by the programmer-debugger module.

6.2.4. Programmer debugger module

The *programmer-debugger* module (see figure 26) is responsible for debugging the student's solution and for providing static and runtime information about the student's solution and the sample solution. Each of the modules within this general expert are themselves expert systems, representing the different aspects of programming and debugging necessary for debugging a program.

6.2.4.1. Parser

The *parser* translates the student's program into an internal form. Although the system currently is geared to debug Pascal programs, it can be modified for similarly

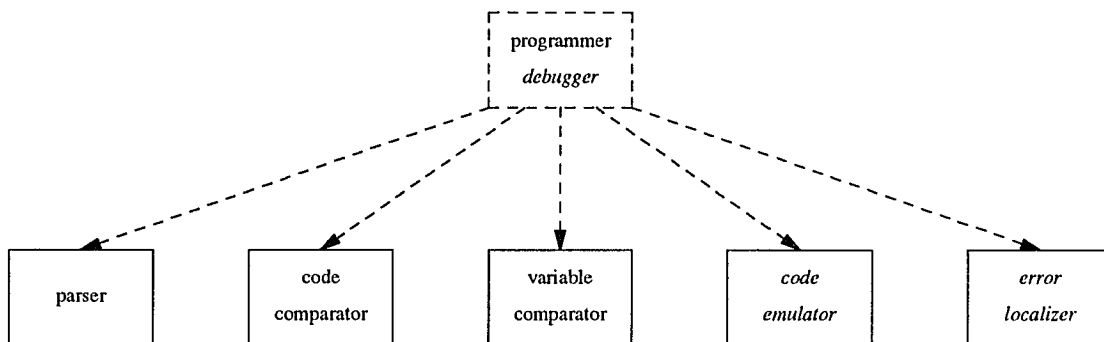


Figure 26. programmer-debugger

structured languages by rewriting the parser. The use of this internal form is restricted to the *programmer-debugger* portion of the system, as interaction with the student is based on actual source code.

6.2.4.2. Variable comparator

The *variable comparator* is used to match the input variables used in the student's program to the input items required by the problem description. It also provides matches for output variables, matching the statements in the student's program to output items specified by the problem description[†]. This is done by matching the variables to their counterparts in the sample solution provided by the environment module, and then using the correlation between the description and the sample provided along with the sample code. Both portions of the variable comparator are rule-based (see Appendix G), each rule specifying a different technique to be used to attempt to make a match. Possible variable matches are returned as triples, the paired variables and an indication of the confidence that the matcher has in the pairing.

Input variables are first matched by order of input. Other evidence, such as location of input within the program is used to strengthen confidence in the match. Since early programming assignments tend to contain few input variables, the matching rules may be correspondingly relatively simple.

[†]Currently only numeric output is considered. Any text output by either the student or sample solution is ignored. Expressions, if output, are internally reduced to temporary variables for ease in matching.

This is not the case for the matching of output variables, however, as they tend to be based on calculations within the program. For each output variable, a *derivation* tree is formed, which contains the other program variables used to form the ultimate result which is to be printed. Pairing of variables may then be accomplished by matching derivation trees developed for the student and sample solutions. Additional information is also provided in the form of the kind of operation (such as a sum, count, or initialization) which was done to form the value. As with the input variable match, the results are in the form of triples.

6.2.4.3. Code comparator

In addition to matching variables, blocks and/or individual lines of code in the student's solution must be matched to the sample solution, which is accomplished by the *code comparator* module. These matches are needed during the debugging process, both by the tutor when following and validating student actions and by the *debugger* module when initially finding the error in the student's code.

Code blocks and statements are matched by form, type, and position.[†] The program code is viewed statically as a hierarchy. Equivalent portions of the hierarchy are matched together. This match may not be exact. For example, were the student to use a decision (*if-then*) structure where the sample used a loop (*while*), a match would be made between the two structures, and would eventually indicate that the student used the wrong control

[†]Currently the code comparator is not generalized; it only provides matches to the specific items requested by the *debugger* module.

structure.

6.2.4.4. Code emulator

The *code emulator* is an interpreter which executes the student program in its internal form, using the input file specified by the *program environment* module. A complete trace of the program is saved, including each line executed, the values of each variable changed, and any output produced by the program. This information is kept in sequential form for analysis both during the automatic debugging stage and to provide trace information for the student.

6.2.4.5. Error localizer

Although it might be possible to find the error in the student's program simply by matching it line for line with the sample solution, debugging by the *error localizer* is done using *debugging rules* similar to those which would be used by the student. With the help of the *code comparator*, the **trace** of the sample solution is used to determine how the trace of the student's program **should** appear, were it a correct solution. For example, it could be used to determine how many times a particular statement should be executed, to determine if that portion of the student's program was executed the correct number of times. The sample solution, then, is used to represent the knowledge the human debugger has relative to how a particular solution should react to a particular situation.

The localizer uses the debugging rules to localize the error in the student's program. The debugger then provides the rest of the system with a *bug report* which gives the

debugging steps used to find the error in addition to the error location and the trace of the student's solution[†].

6.2.5. Consultant module

The *consultant* (see figure 27) is responsible for guiding the interaction with the student, as well as guiding the student through the debugging session.

6.2.5.1. Questioner

The *questioner* is activated when requests for student input are made. It determines the kind of question to ask (pick one of many, true/false, etc) based on a dictionary lookup (see Appendix I) of the item desired, and determines what the possible choices are, if necessary. It only provides the information needed to formulate the question. The

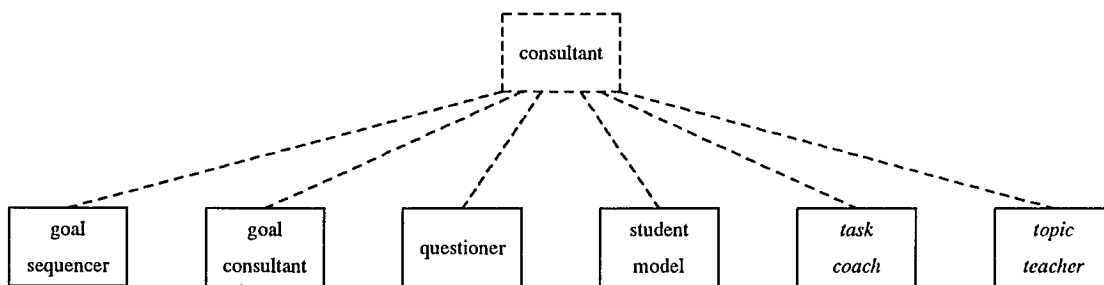


Figure 27. The Consultant

[†]Currently much of this information is unused by the rest of the system, as the associated modules are simply stubs.

actual formulation of the question, however, is left to the I/O modules. Should the student have trouble answering the question, the questioner uses a rule base to determine if a new form for the question is feasible (see Appendix H.2). For example, a *pick some of many* question might be converted into a sequence of *true/false* questions.

6.2.5.2. Goal sequencer

The consulting session is broken down into a number of goals to be achieved, both by the system and by the student. The *goal sequencer* defines these goals and causes them to be achieved by other modules within the system in a specified order. The actual goals and sequencing information are maintained as a state diagram (see Figure 28) which is encoded into a rule base, rather than being hard coded into the module itself (see Appendix D.1).

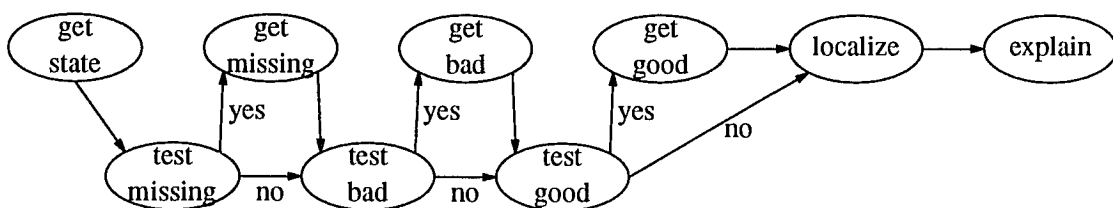


Figure 28. Goal sequencer states

6.2.5.3. Goal consultant

The *Goal consultant* is another rule-based state machine which provides the actual interaction sequencing with the student relative to a particular goal. Figure 29[†] shows an overview of the various states, which are designed to emulate the various goal related activities of the human consultant. As with the goal sequencer, the state machine

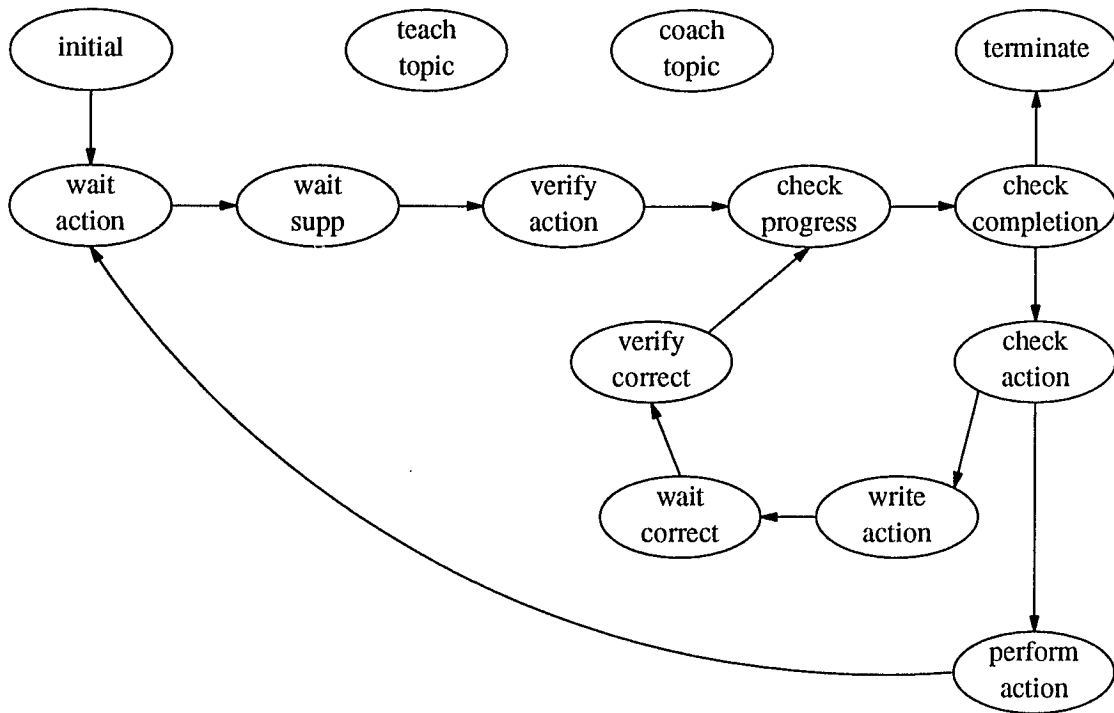


Figure 29. Goal consultant states

[†]The numerous transitions to and from the coaching and teaching states are not shown in order to make the diagram more readable.

description is in the form of a rule base (see Appendix D.2). The specific actions taken at most states are also chosen based on individual sets of rules.

The goal consultant first requests input from the student. The actual request depends on the specific task, and is determined using a rule base (see Appendix E.1). In some instances, the student is asked to supply information, such as when gathering explicit symptoms. In other situations student initiative is allowed such as while the student is attempting to localize the error. In some cases, supplementary information (see Appendix E.2), such as the line number for a debugging write, is also requested.

This information and/or action is then verified using rule base (see Appendix E.3) supplied verification functions[†]. Verification, which supplies the system numeric scores, takes into consideration both the correctness and the appropriateness of the answer and/or action. This score is used to initiate immediate coaching or teaching relative to this specific item. It is also used to update an overall student score^{††} which is used to determine the current mode (*watching*, *coaching*, *teaching*) of the consultant.

When involved with tasks which allow student initiative, the student may specify two different classes of actions: requests to the system to provide data, such as listing program lines; and requests to logically insert debugging writes into the program. Debugging write actions are more complex, as they require further interaction with the student. In this case, the student is required to determine whether the resulting output

[†]Although selected by rules, most of the verification functions in the current system are simply stubs, requiring the verification score to be input to the program.

^{††}Older values are *aged* so the effect of a single score on the overall score decreases with time.

corresponds to the anticipated values. As this is an important part of the debugging process, the system verifies the student's correctness determination, and accordingly adjusts the student's overall performance score. The student is not allowed to progress to another action until the new symptoms have been properly identified.

The coaching and teaching states are entered either on receipt of incorrect input from the student, or prior to prompting for the student action when the system has entered the coaching or teaching mode due to a decline in the student's overall performance score.

6.2.5.4. Student Model and understanding

Ideal student knowledge is represented as a prerequisite network, which may be viewed as a multiway tree except that children may have multiple parents. The children of a node represent the knowledge necessary to understand the concept represented by the node itself. Although the ideal student is maintained in this form, the actual student model is maintained in a more disjoint form, as historical information relating to the particular student's successes and failures during this and previous consulting sessions. This information is saved at the end of a session, and is restored at the beginning of the student's next session.

When the system determines that a student needs tutoring, either due to the current system mode or an individual failure on the part of the student, a root topic for discussion

is chosen from the network[†]. The system then proceeds to traverse this tree checking for relevancy of the topic within the context of the problem and the student's understanding of the relevant topics[†]. The topics found by this search are then taught by the task teacher.

6.2.5.5. Task coach and teacher

The *task coach* and *topic teacher*[†] coach or teach the student in a specific topic. The coach gives hints about the current task, while the teacher is assigned topics to teach by the student model.

6.3. Module interaction and design

In order to maintain modularity and flexibility throughout the system, module interaction is done by indirect means whenever possible. As a general rule, modules do not "know" each other by name, but rather by the services and information that they provide.

6.3.1. Specialist invocation

The primary mechanism for module invocation is the *help-request* interface provided by the system scheduler. Rather than directly calling another module, a module which desires aid issues a request for help on a particular item or task. The requesting module has the option of suspending its own actions immediately, waiting until the requested aid has been given, or it may continue on its own until the information is actu-

[†] These portions of the system are currently stubs, and are topics for further research.

ally necessary and enter the wait state at that point[†]

The major advantage to this approach is that it allows for the possibility that several different modules might be able to supply the requested aid. The scheduler may invoke possible modules sequentially, or all in parallel if such facilities are available, until one is found which is successful. An additional advantage is that modules may be added and/or removed with relative ease, as there is no direct tie to their names within the code. The system scheduler, using information provided by the modules themselves at startup time, maintains a list of possible aid which may be provided, along with the potential providers (see Appendix A). The disadvantage to this approach is that, since the invocation mechanism is relatively generic, information passing between modules is accomplished primarily through global databases.

6.3.2. Rule based shared databases

Global data within the system is distributed through a number of individual databases. Although the data is not physically combined, there is a single mechanism, used by the separate modules to access and/or generate data items. Individual data items are tagged with two keys, one being the "kind" of data it is and the other its internal name. The database search mechanism is given these keys and additionally it is provided with a rulebase. If the desired item is found within the global data bases, its value is returned to the caller, and the supplied rules are ignored. The rules within this rulebase

[†]The intent of this is to allow concurrent processing if such an environment were available. In a sequential environment actual invocation is withheld until the requesting module enters into the wait state.

are utilized only when the desired item is not to be found within the global databases. These rules (see Appendix C) provide the actions, specific to the module initiating the data search, necessary to generate the desired item. On search failure, the search mechanism locates the appropriate rule within the rule base and causes the current module to perform the actions needed to supply the missing item. The search/generation mechanism then inserts the desired value into the appropriate global database in addition to returning it to the calling module.

The advantage of this mechanism is that information may be shared, eliminating the need for different modules to continuously regenerate the same values, while at the same time each module may have its own method, depending on its own specific abilities, for generating a piece of data. For any particular lookup, the calling module does not know if the data returned was placed into the database by some other module, or whether they themselves generated the value and caused it to be placed in the global databases. The database search mechanism, through the supplied rule-bases, provides a generic search/generation mechanism for the various modules within the system.

6.3.3. Rule based state machine behavior

Modules which perform a sequence of actions are designed as state machines, translating the action sequence into a sequence of states. A single generic state machine, which provides state changes based on individually supplied rule bases (see Appendix D) and environments, is included in the overall system. State transition is accompanied by the execution of a block of code, which is used to provide the necessary actions specific

to the state as well as the manipulation of the environment.

The advantage to this approach is that modifications may be made to the sequences by modifying a rule base, rather than modifying actual system code. States and actions may be added, removed, or modified without modifying the system itself.

6.3.4. Rule based function selection

Since many of the tests and actions depend on the actual situation, they need to be designed in such a way that they are sensitive to the current goals and needs of the system. Rather than taking the traditional approach of *hardcoding* these differences with multiway selections structures, the rulebased approach was taken, using a generic rule selection mechanism which is provided with small local rule-bases (see Appendix E). Thus it is possible to change the actions of a function, or to account for new situations without actually modifying the function's code.

6.3.5. Evaluation of design methodology

The high degree of modularization within the system makes it possible to use a *top down* approach for both design and implementation at the overall system level and at the level of the individual specialists. Although this allowed each specialist to be designed and implemented separately, the rulebases were designed and implemented in more of a *spiral* approach. Initially specialists with rudimentary rulebases were designed and implemented. Independence of design implies that there is no requirement or even desire to insure that the internal vocabularies of the modules are consistent. This means that module specifics may be implemented in a way that is natural for that type of specialist.

The use of the help request system insures that calls are consistent within the system, as specific data is obtained through the rule based information retrieval mechanism from global databases.

Although the actual design and implementation details of the individual modules were strictly independent, the actions of the modules are very inter-dependent. Adding capabilities to one module has the side effect of requiring more capabilities of one or more other modules, which must be completed to be able to retest the system. In a multi-person project, this provides no serious difficulties, as each person would be responsible for one module or submodule, and would thus know and understand the internal vocabulary of that module well. Extreme module design and implementation independence means that the different individuals in the project have more flexibility, and less communication about inter-module protocol is required.

This approach is less than ideal, however, when project personnel are limited and one person is responsible for numerous specialists. In this case, one person is required to remember numerous different vocabularies and implementations, which become more and more varied as the system gains capability. This situation calls for far more detailed documentation during the design and implementation process than is required when each individual is responsible for a single specialist.

CHAPTER 7

Related work

There are a number of systems designed to tutor students in the programming domain. Most include the debugging of the students' code in some manner within the system. Those specifically directed towards teaching students to write code use automated debuggers to determine student errors, and hence misconceptions. Other automated debugging systems are used simply to find and explain student errors. The major difference between these systems and TRACES is in the motivation behind the debugging actions. In TRACES, the consulting actions are geared towards helping students find their own error and at the same time, using the results of the student actions to determine student misconceptions about their program in particular and programming and language concepts in general. That is, the student's own actions lead the consultant to the discovery of the student's weaknesses.

The following sections discuss a number of systems which have features and/or goals which are similar to TRACES. The systems and the major differences between them and TRACES are discussed.

7.1. Proust

Proust [JOHN84, JOHN85, JOHN90b] is designed to find and report multiple semantic and logic errors in novice level Pascal programs. Errors are found by matching

the student's code to *plans* developed by the system. Problem descriptions are stored, in the form of a set of goals which must be satisfied in order to solve the problem. Proust creates goal decompositions, breaking each of the goals into plans, which in turn may create further goals. Plans are stored in a knowledge base, comprised in part by templates which describe the Pascal code necessary to implement the plan.

The system develops a tree of possible goal/plan decompositions. The student's code is matched against the plan templates to determine the best fit of the various combinations. The process is iterative, as the various plans may in turn be subgoals which must then also be matched. The result is an *interpretation tree*, which best matches the student's code, and thus represents the student's *intentions* in creating the code. Mismatches between the system generated plan and the student's actual code reflect errors and/or misconceptions, which are reported in the form of a bug report, using the interpretation tree to provide necessary context.

Proust is designed to find and report errors, but unlike TRACES it is not intended to tutor the student relative to those errors, nor is it designed to help students find their own errors, but rather simply points them out to the student. It acts more like a *batch* system, and does not use student interaction to supply information about the student or program. Like TRACES, Proust requires knowledge of the solution to the problem being debugged. Proust uses a formal description in the form of goals, and a generic knowledge base to convert the goals into actual plans which may be matched against code. This is considerably more flexible than TRACES, which currently must rely on actual solution programs[†]

which match the student's methodology.

Proust's method of debugging is effectively by code comparison, using their generated solution. TRACES, on the other hand, uses the sample solution to provide information about expected values during a dynamic trace of the program's execution. Code matching is necessary to determine equivalent points within the student's program and the sample solution, rather than for determining mismatches in actual code. Proust's method does allow for the discovery of multiple errors, however, while TRACES is limited to programs with single errors. Both systems require the code to be a reasonably close match to the sample to be able to process the student's code.

TRACES forces the student to go through a dynamic debugging process, monitoring the student's progress and questioning the student about expected values within the solution. In addition to the actual error within their code, student misunderstandings become apparent through this interaction.

7.2. Lisp Tutor

The *Lisp Tutor* [ANDE90, CORB88, REIS85] is an interactive tutor that is designed to aid students in the writing of Lisp programs. The system is based on a set of pre-designed problems. The problem sequence is provided by the tutor, based on student performance. The system uses *model tracing*, which follows students step by step as they write their program. The tutor contains a "precompiled" set of possible solution paths the

[†]The method of plan/code generation provided by Proust would provide a possible enhancement to the *environment* module of TRACES, allowing the solutions to be generated rather than directly provided. Proust's code matching would then replace the *code comparator* within TRACES.

student may take to solve the problem, including possible erroneous paths. It then provides direction whenever the student provides a step which strays from a feasible solution. Since errors are immediately detected, a solution will never contain more than a single error at any one time.

There is very little in common between TRACES and the Lisp Tutor. Both are interactive, but at different extremes of the problem solving process. The Lisp Tutor's interaction is geared to helping the student write an initial solution to a problem, and to catch errors for the student as the solution is being written, while TRACES works with code which has already been written. Both systems require the use of a solution matching that used by the student. In the case of the Lisp Tutor, that solution is matched with the student's solution step by step, as the student is writing the program. In this way, the solution provided by the tutor will always, except possibly for the last step, match the student's solution. TRACES attempts to match complete solutions against the student's complete solution, but it also makes the assumption that there is a single error in the student's program. The sample is not, however, used to find the error by direct code matching, but rather it is used dynamically to determine what values should be present in the student's solution.

The Lisp Tutor detects and addresses students' errors based on the code that they write as they are writing it. TRACES, on the other hand, detects student misunderstandings through their attempts to debug their own program. While guiding the student in the finding of an error, the system requires the student to be able to specify what debugging write values should be, which the system matches against the sample solution.

Mismatches indicate gaps in understanding, which are then tutored.

7.3. *Scent-3*

The architecture of *Scent* [MCCA86] and *Scent-3* [MCCA88] is based on the concept of *blackboards* [ERMA80] as is the architecture of TRACES. Although major subsystem invocation is also provided by this mechanism, *Scent* is more traditional, in that it also allows the posting of requests of a smaller nature as well. TRACES utilizes a modified blackboard scheme which allows help requests to be made of full expert systems, and acts as the interface between them.

Both systems use a form of code matching while debugging the student's program. *Scent* uses the match itself, supported by trace information, to derive bugs, while TRACES uses this information to match trace values in student and sample solutions. *Scent-3* is designed more as a tutor, providing an instructional sequence focused on specific instructional goals, while TRACES is more opportunistic in that it uses the student's own debugging actions to determine the current instructional goals. Tutoring in both systems is based on an and/or prerequisite hierarchy; in *Scent* only tutoring tasks are stored in this fashion, while TRACES uses this mechanism to store knowledge in the student model, for which *Scent* uses a genetic graph.

7.4. *Apropos2*

Apropos2 [LOOI88] is the debugger component of a Prolog intelligent tutor, which is designed to help students learn to write programs in Prolog. The system defines predicates for the student to write. *Apropos2* is used to detect the errors in the student

programs and supplies bug fixes. The student model then uses the bugs to identify the student's misconceptions about the language, which are corrected by the tutor.

The student code is matched heuristically against a library of algorithms which have been supplied for the particular problem. Student intentions are developed by using the matched lines in the algorithms. Dynamic analysis is used to prove equivalence of portions of the code, and to detect errors.

Both TRACES and Apropos2 detect errors in program code. Apropos2 then explains and provides a tutorial on fixing the errors, while TRACES forces students to find their own errors. Misconceptions are linked to bugs in Apropos2, while TRACES uses the student interaction during the debugging session to identify the gaps in their knowledge.

7.5. Spade-0

The *Spade-0* [MILL82] system is more of a problem solving environment than a tutoring system. Students are lead through the various steps of problem solving by the system. Students are provided with multiple choice questions in which they may specify either problem solving specifics, or debugging. Students may cause the developed program to execute, and a trace may be provided for debugging purposes.

Both TRACES and Spade-0 provide a program trace during debugging. TRACES keeps this essentially hidden from the student, requiring the student to specify debugging write positioning to retrieve specific values. Spade-0, on the other hand, provides the full trace for the student to use at will. Spade-0 also places much of its emphasis on the

writing of the program.

7.6. Talus

Talus [MURR85, MURR87] is an automated debugger which is intended to function as part of a tutoring system to teach Lisp programming in an environment where the problems to be solved and their solutions are known. The system determines the basic algorithm being used by the student. The student's solution is then matched against this algorithm, pairing the student functions and variables with those of the reference solution. The pairs are checked to determine if the student function and reference function are functionally equivalent, using a form of induction theorem proving.

Talus's approach is very different from the debugger portion of TRACES in that it does not debug or reason in the same manner as would a novice programmer. Although it reliably provides information about bugs and their correction, the process is not one which the student would be expected to follow. Thus, although it is able to pass information about the bugs on to the tutor portion of the system, this information is not in a form which the tutor can use to help students locate their own errors. In this respect, although the debugging process itself is quite different, Talus resembles Proust. TRACES, by performing the dynamic trace and using debugging rules similar to those used by students, is able to help the student throughout the actual debugging process.

7.7. Teaching Algorithmics

Dion and Lelouche's [DION88] system teaches programming using the *Karel the Robot* [PATT81] programming system. The programming expert is able to solve

problems presented to it in an internal specification language, using goal decomposition to develop programs. It generates all possible solutions to the problem. The student's program is then matched to the generated solutions using rules to determine semantic equivalence between segments of the student's program and a generated program. Bug rules are used to detect common errors. Solutions are tried until a match is made or the solutions are exhausted. If no match can be found, the dynamic behavior of the program is checked using a number of test sets generated by the system. The student is then informed of the success/failure of the program, along with an explanation and/or counter example showing the failure.

Although the system does use dynamic testing of the student's code to be matched against sample solutions, the student is not involved in the actual debugging process. Thus, student misconceptions found are those directly related to the actual errors, rather than those which might also appear during the debugging process as is the case in TRACES.

7.8. Laura

Laura [ADAM80] is an expert system designed to detect or localize errors in Fortran programs. The system is provided with a correct program using the same algorithm as the buggy program. The programs are converted into a graph representation, which are then matched. Semantically equivalent portions are mapped together, using transformations if necessary. The unmatched portions then point to the location of the error.

Code matching is used in TRACES to identify equivalent spots in the sample and student solution, so that trace values from the sample solution may be used to verify values obtained in the associated position in the student code. TRACES uses techniques which would be available to the student programmer, as it must lead the student through the debugging process. Unlike TRACES, Laura makes no attempt to tutor the student.

7.9. Sophie

Sophie [BROW75, BROW82] provides an electronics laboratory to the student. The student is allowed to watch the system troubleshoot circuits, where the student is allowed to enter the fault. As the troubleshooting module in the system performs steps to find the error, they are explained to the student. A coach also exists to watch the student perform troubleshooting tasks. The coach looks for redundant checks and insures that the student hypothesis is supported by the evidence collected.

Although the domains of TRACES and Sophie are different, the process of troubleshooting an electronic circuit is very similar to debugging a program. Testing for redundant checks in Sophie resembles the determination of the quality of the placement of a debugging write in code. The systems differ in that Sophie is primarily designed to use learning by experimentation, while TRACES is attempting to find underlying gaps in student knowledge through responses during the debugging process.

7.10. Hearsay-II

Hearsay-II [ERMA80] uses a number of small independent programs called *knowledge sources* to manipulate data in a global database called a *blackboard*. The knowledge sources provide partial problem solutions which are stored on the blackboard. Each knowledge source may be invoked if the appropriate conditions exist on the blackboard, adding their contributions to the solution being constructed. The scheduler maintains a priority queue of triggered knowledge sources, which are executed in sequence until the overall problem is solved. Each new item entered into the blackboard may cause other knowledge sources to trigger and be placed on the queue.

TRACES uses a modified blackboard approach with its request-help mechanism. Requests for help are accepted by the scheduler, which uses a rule base to determine which expert(s) can provide the aid. The invoked experts leave the results of their activities in global databases, even if unsuccessful in providing the necessary aid, and they in turn may request help of other experts. The major differences are that the experts in TRACES are complete systems, and they request specific help on specific tasks.

CHAPTER 8

Conclusion

8.1. Summary

Teaching in the context of consulting requires a teaching model that differs from the conventional teaching model. Consulting involves guiding students through a diagnostic process in order that they discover their own errors in a problem solution. The consultant must use this process to determine what the student actually knows or thinks about the solution and its underlying concepts. While the tutor is continuously probing students to determine gaps in their knowledge base, the consultant monitors the debugging process, and observes variances in what should be correct and what the student thinks is correct. These differences point to relevant gaps in the student's knowledge, which is stored in a prerequisite network. The consultant must then fill these gaps, so that the student may understand the correct solution to the problem. Consulting also involves strengthening the debugging techniques used by the student.

TRACES is an intelligent teaching consultant which is based on this model. It operates as a programming consultant for students who are trying to find errors in their Pascal programs. The students initially use the system to aid them in debugging their own programs. Although TRACES helps guide the student through the debugging steps, and coaches the student on debugging, the system also uses the student actions and

verification of trace values to determine misconceptions relating directly to the program, rather than the debugging process itself.

The system is comprised of numerous complete expert systems, which are interfaced with a help-request mechanism. This mechanism allows help to be requested and given without either system knowing the actual name of the other, allowing for greater flexibility in design. Flexibility is also provided through the use of rule based state machines and rule-based global data base information entry and retrieval.

8.2. Contributions

The major contribution of this dissertation is the model of the teaching consultant. This model differs significantly from conventional teaching, coaching, and tutoring models, in that it operates at two separate levels. Unlike the teaching and tutoring models, the consultant model is based on monitoring students' problem solving processes while diagnosing errors in their own solutions to specific problems. The consultant acts like a coach in that the student is guided, using coaching methods, through the diagnostic steps. The second deeper level involves utilizing the student's verification of the diagnostic symptoms to discover, and thus fill, gaps in the students knowledge relevant to the current problem. Thus, unlike a teacher or tutor, the consultant does not actively search for knowledge gaps, but rather is opportunistic and uses the student's own actions during the diagnostic process to detect these gaps.

The overall help-request mechanism, which provides the basis for the system action, is a useful variation on the concept of a blackboard. Unlike conventional blackboard

based systems, which consist of numerous mini-experts which may provide partial solutions to a problem, the help-request mechanism is based on the concept of a number of cooperating complete expert systems, each with their own specialties. Requests are for specific aid, either in terms of information or tasks to be performed, which would be expected to be completed in total, rather than being accomplished piecemeal by numerous smaller experts. The mechanism provided allows for invocation of these experts without any direct knowledge of which expert can provide the necessary aid; the individual experts supply this knowledge to the system scheduler at system startup.

As with most expert systems, numerous global databases are provided for communication and information storage. The generic lookup/generation mechanism used within the overall system allows for a single mechanism to locate information within the numerous data bases and/or module specific generation of information into the databases. Module specific information generation rules are invoked whenever desired data is not found within the bases, generating and inserting the information automatically.

The use of rule-based state machines to provide sequencing for several of the experts integrates software engineering techniques into expert systems. Flexibility and predictability are provided through the rule-bases, without the necessity of hard coding the sequencing into the experts themselves.

Finally, the overall system itself contains numerous areas for significant future research, primarily in the realms of student understanding, the specific knowledge required for programming and debugging, and in the determination of the quality of actions taken during a diagnostic process.

8.3. Future Research

Currently, although the system has numerous gaps which are represented by rule-based stubs, it is able to find the bugs in a number of student programs. It is able to use the bug report and trace provide by the debugger to guide the student through the consulting session.

The majority of the gaps in the system represent areas for future research. The most significant are: the development of a complete prerequisite network containing the knowledge necessary to develop and debug novice programs; the determination of student understanding of topics within the network, using historical and current student actions, as well as interaction with the student; the development of topic coaching and tutoring modules for specific topics; the generation of alternative sample solutions by the environment module; and the introduction of a natural language interface to replace the current I/O modules. The determination of relevancy of the individual topics in the prerequisite network and the determination of the quality of the students' debugging actions are also potential areas for research.

APPENDIX A

Help provided by experts

The following is a list of the help provided by the various experts within the system. The first item is the internal name for the help desired and the second is the name of the expert which is stating that they can provide the help.

```
(add-request 'coach-student 'coach)
(add-request 'explain-symptoms 'coach)
(add-request 'match-invars 'comparator)
(add-request 'match-outvars 'comparator)
(add-request 'find-bug 'debugger)
(add-request 'initialize-environment 'environment)
(add-request 'sample-program-pathname 'environment)
(add-request 'dynamic-info 'environment)
(add-request 'ask-student 'student-interface)
(add-request 'tell-student 'student-interface)
(add-request 'check-student-understanding 'model)
(add-request 'init-student-model 'model)
(add-request 'save-student-model 'model)
(add-request 'parse-program 'parser)
(add-request 'parse-student-program 'parser)
(add-request 'parse-sample-program 'parser)
(add-request 'query-student 'questioner)
(add-request 'program-run-info 'run-info)
(add-request 'coach-task 'task-coach)
(add-request 'teach 'teacher)
(add-request 'teach-student 'topic-teacher)
(add-request 'topic-relevancy 'topics)
(add-request 'root-topic 'topics)
(add-request 'student-program-pathname 'tutor)
(add-request 'initial-student-help-request 'tutor)
```


APPENDIX B

Prerequisite network

The prerequisite network is built dynamically at system startup. It contains the programming, debugging, and problem specific topics which are needed to debug the current problem. Eventually the *environment* module should be given the capability to tailor this network by including only those prerequisites needed for the specific problem.

```
(set-prereq debug-program  
  (*or symptomatic-debugging formal-program-proofs hand-tracing inspection))
```

```
(set-prereq symptomatic-debugging  
  (*and symptom-generation symptom-analysis))
```

```
(set-prereq symptom-generation  
  (*or input-set-generation debugging-writes))
```

```
(set-prereq debugging-writes  
  (*and flow-writes value-writes))
```

```
(set-prereq flow-writes  
  (*and flow-write-placement output))
```

```
(set-prereq output  
  (*and output-statements expressions))
```

```
(set-prereq flow-write-placement  
  (*and execution-flow fault-localization-techniques))
```

```
(set-prereq fault-localization-techniques  
  (*and half-interval input-to-output output-to-input right-path))
```

```
(set-prereq value-writes  
  (*and value-write-placement output))
```

```
(set-prereq value-write-placement  
  (*and value-derivation-trees))
```

```
(set-prereq value-derivation-trees
```

```
(*and assignments execution-flow input))

(set-prereq input
  (*and input-statements variables))

(set-prereq symptom-analysis
  (*and flow-write-count value-write-output normal-output program-states))

(set-prereq program-states nil)

(set-prereq flow-write-count
  (*and execution-flow))

(set-prereq value-write-output
  (*and derivation-trees))

(set-prereq execution-flow
  (*and sequence-execution loop-execution conditional-execution))

(set-prereq sequence-execution nil)

(set-prereq loop-execution
  (*and counting-loop-execution sentinel-loop-execution))

(set-prereq counting-loop-execution
  (*and (*or for-loop-execution while-loop-execution repeat-loop-execution)
    counters))

(set-prereq counters
  (*and count-initialization count-increment count-reached-condition))

(set-prereq count-initialization
  (*and numeric-variables numeric-constants assignments))

(set-prereq count-increment
  (*and numeric-variables assignments numeric-constants addition-operator))

(set-prereq count-reached-condition
  (*and boolean-expressions))

(set-prereq sentinel-loop-execution
  (*and sentinels (*or while-loop-execution repeat-loop-execution)))

(set-prereq sentinels
```

```

(*and sentinel-initialization sentinel-update sentinel-found-condition))

(set-prereq sentinel-found-condition
  (*and boolean-expressions))

(set-prereq sentinel-initialization
  (*and input variables))

(set-prereq sentinel-update
  (*and input variables))

(set-prereq conditional-execution
  (*and if-then-execution if-then-else-execution case-execution
    boolean-expressions))

(set-prereq arithmetic-operators
  (*and addition-operator subtraction-operator multiplication-operator
    division-operator integer-quotient-operator
    integer-remainder-operator))

(set-prereq boolean-operators
  (*and or-operator and-operator not-operator))

(set-prereq relational-operators
  (*and less-operator less-equal-operator equal-operator
    greater-equal-operator greater-operator not-equal-operator))

(set-prereq relational-expressions
  (*and relational-operators arithmetic-expressions))

(set-prereq arithmetic-expressions
  (*and arithmetic-operators numeric-constants numeric-variables precedence
    associativity))

(set-prereq boolean-expressions
  (*and boolean-operators boolean-constants boolean-variables precedence
    associativity relational-expressions))

(set-prereq expressions
  (*and boolean-expressions numeric-expressions relational-expressions))

(set-prereq boolean-constants
  (*and literal-constants boolean-datatype))

```

```
(set-prereq numeric-constants
  (*and literal-constants numeric-datatypes))
```

```
(set-prereq boolean-variables
  (*and variables boolean-datatype))
```

```
(set-prereq numeric-variables
  (*and variables nunumeric-datatypes))
```

```
(set-prereq assignments
  (*and variables expressions))
```

```
(set-prereq variables
  (*and storage-locations data-values))
```

```
; -----
```

```
(set-prereq formal-program-proofs nil)
```

```
(set-prereq hand-tracing nil)
```

```
(set-prereq inspection nil)
```

```
(set-prereq input-set-generation nil)
```

```
(set-prereq half-interval nil)
```

```
(set-prereq input-to-output nil)
```

```
(set-prereq output-to-input nil)
```

```
(set-prereq right-path nil)
```

```
(set-prereq for-loop-execution nil)
```

```
(set-prereq while-loop-execution nil)
```

```
(set-prereq repeat-loop-execution nil)
```

```
(set-prereq addition-operator nil)
```

```
(set-prereq subtraction-operator nil)
```

```
(set-prereq multiplication-operator nil)
```

```
(set-prereq division-operator nil)
```

```
(set-prereq integer-quotient-operator nil)
```

```
(set-prereq integer-remainder-operator nil)
```

```
(set-prereq or-operator nil)
```

```
(set-prereq and-operator nil)
```

```
(set-prereq not-operator nil)
```

```
(set-prereq less-operator nil)
```

```
(set-prereq less-equal-operator nil)
```

```
(set-prereq equal-operator nil)
```

```
(set-prereq greater-equal-operator nil)
```

```
(set-prereq greater-operator nil)
```

```
(set-prereq not-equal-operator nil)
```

```
(set-prereq if-then-execution nil)
```

```
(set-prereq if-then-else-execution nil)
```

```
(set-prereq case-execution nil)
(set-prereq associativity nil)
(set-prereq precedence nil)
(set-prereq literal-constants nil)
(set-prereq boolean-datatype nil)
(set-prereq numeric-datatypes nil)
(set-prereq storage-locations nil)
(set-prereq data-values nil)
(set-prereq input-statements nil)
(set-prereq output-statements nil)
```

```
;-----
```

```
(set-prereq calculate-average
  (*and counters sums loop-execution))

(set-prereq sums
  (*and sum-initialization sum-update))

(set-prereq sum-initialization
  (*and additive-identity assignment))

(set-prereq sum-update
  (*and assignment))
```

APPENDIX C

Sample global database generation rules

Many of the separate experts use a global database for communication and the storage of information. Whenever information is needed, the database is first searched. Should that information not be present, a method specific to that expert is chosen to generate the desired item. The following is the rule set used by the variable comparator module.

```
; global ids
;
;   which-prog - *student *sample *combined
;   **temp** **temp1** **temp2**

;==> (generate what prog) -- program info

(setq **specific-generation-methods** '(

;==> *pathname -- program info
      ((*sample *pathname)
        (car (request-wait 'sample-program-pathname 'comparator)))

      ((*student *pathname)
        (car (request-wait 'student-program-pathname 'comparator)))

;==> *parsed-code -- program info
      ((*sample *parsed-code)
        (car (request-wait 'parse-sample-program 'comparator )))

      ((*student *parsed-code)
        (car (request-wait 'parse-student-program 'comparator )))

;==> *echo-only-outvars -- program info
      ((*student *echo-only-outvars)
        (progn
          (setq **temp** (generate '*input-vars which-prog))
          (setq **temp1** (generate '*output-vars which-prog))
          (setq **temp2** (list-intersection **temp** **temp1**))
```

```

        (check-possible-student-echoes **temp2**)))

    ((*sample *echo-only-outvars)
     (progn
      (setq **temp** (generate '*input-vars which-prog))
      (setq **temp1** (generate '*output-vars which-prog))
      (setq **temp2** (list-intersection **temp** **temp1**))
      (check-possible-sample-echoes **temp2**)))

;==> *invar-matches -- program info
    ((*combined *invar-matches)
     (produce-invar-matches))

;==> *outvar-matches -- program info
    ((*combined *outvar-matches)
     (produce-outvar-matches))
))

(setq **generic-generation-methods** '(

;==> *linear-code -- program info
    (*linear-code
     (progn
      (setq **temp** (generate '*parsed-code which-prog))
      (setq **temp** (make-linear-program **temp**))
      (put-global-proginfo which-prog (car **temp**)'*var-names)
      (number-linear-code (cadr **temp**))))

;==> *var-names -- program info
    (*var-names
     (progn
      (setq **temp** (generate '*parsed-code which-prog))
      (setq **temp** (make-linear-program **temp**))
      (put-global-proginfo which-prog (cadr **temp**)'*linear-code)
      (car **temp**)))

;==> *input-vars -- program info
    (*input-vars
     (collect-unique-ids (generate '*input-sequence which-prog)))

```

```

;==> *output-vars -- program info
      (*output-vars
        (collect-unique-ids (generate '*output-sequence which-prog)))

;==> *modified-vars -- program info
      (*modified-vars
        (collect-unique-ids (generate '*modification-sequence which-prog)))

;==> *input-sequence -- program info
      (*input-sequence
        (collect-vars-from-inlines (generate '*input-lines which-prog)))

;==> *modification-sequence -- program info
      (*modification-sequence
        (collect-modvars-from-modlines
          (generate '*modification-lines which-prog)))

;==> *output-sequence -- program info
      (*output-sequence
        (collect-vars-from-outlines (generate '*output-lines which-prog)
          (generate '*var-names which-prog)))

;==> *input-lines -- program info
      (*input-lines
        (collect-inlines-from-linear (generate '*linear-code which-prog)))

;==> *output-lines -- program info
      (*output-lines
        (collect-outlines-from-linear (generate '*linear-code which-prog)))

;==> *modification-lines -- program info
      (*modification-lines
        (collect-modlines-from-linear (generate '*linear-code which-prog)))

;==> *non-echo-outvars -- program info
      (*non-echo-outvars
        (list-difference (generate '*output-vars which-prog)

```



```

      (generate '*echo-only-outvars which-prog)))

;==> *sum-out-vars -- program info
      (*sum-out-vars
        (generate-fact-collection which-prog '(*is-true is-sum-var *)
          (generate '*output-vars which-prog)))

;==> *count-out-vars -- program info
      (*count-out-vars
        (generate-fact-collection which-prog '(*is-true is-count-var *)
          (generate '*output-vars which-prog)))

;==> *count-calculation-out-vars -- program info
      (*count-calculation-out-vars
        (generate-fact-collection which-prog
          '(*is-true is-count-calculation-var *)
          (generate '*output-vars which-prog)))

;==> *sum-calculation-out-vars -- program info
      (*sum-calculation-out-vars
        (generate-fact-collection which-prog
          '(*is-true is-sum-calculation-var *)
          (generate '*output-vars which-prog)))

;==> *loop-out-vars -- program info
      (*loop-out-vars
        (generate-fact-collection which-prog '(*is-true output-in-loop *)
          (generate '*non-echo-outvars which-prog)))

;==> *outvar-single-invar-dependency -- program info
      (*outvar-single-invar-dependency
        (generate-fact-collection which-prog
          '(*is-true depends-on-single-invar *)
          (generate '*non-echo-outvars which-prog)))

;==> *outvar-multiple-invar-dependency -- program info
      (*outvar-multiple-invar-dependency
        (generate-fact-collection which-prog
          '(*is-true depends-on-multiple-invars *)
          (generate '*non-echo-outvars which-prog)))

```

```

;==> *decision-out-vars -- program info
      (*decision-out-vars
        (generate-fact-collection which-prog
          '(*is-true output-in-decision *)
          (generate '*non-echo-outvars which-prog)))

;==> *outvars-depending-on-mult -- program info
      (*outvars-depending-on-mult
        (generate-fact-collection which-prog
          '(*is-true outvar-depends-on-mult *)
          (generate '*non-echo-outvars which-prog)))

;==> *outvars-depending-on-subt -- program info
      (*outvars-depending-on-subt
        (generate-fact-collection which-prog
          '(*is-true outvar-depends-on-subt *)
          (generate '*non-echo-outvars which-prog)))

;==> *outvars-depending-on-realdiv -- program info
      (*outvars-depending-on-realdiv
        (generate-fact-collection which-prog
          '(*is-true outvar-depends-on-realdiv *)
          (generate '*non-echo-outvars which-prog)))

;==> *outvars-depending-on-intdiv -- program info
      (*outvars-depending-on-intdiv
        (generate-fact-collection which-prog
          '(*is-true outvar-depends-on-intdiv *)
          (generate '*non-echo-outvars which-prog)))

;==> *outvars-depending-on-intmod -- program info
      (*outvars-depending-on-intmod
        (generate-fact-collection which-prog
          '(*is-true outvar-depends-on-intmod *)
          (generate '*non-echo-outvars which-prog)))

))

```

APPENDIX D

State machine rules

The following are the rule bases which control the actions of the two state machines within the system. Both sets of rules are used by the same state machine code. One governs the overall session sequencing, while the other controls the actions of the coach.

D.1. Tutor task sequence rules

```
(setq **tutor-task-rules** '(
  (*tutor-task-initial-goal-completed
    *if ((= state completed) (= goal current-goal))
    *action
      (state-machine-terminate)
  )
  (*tutor-task-current-completed
    *if ((= state completed))
    *action
      (progn
        (pop-var-environment 'state tutor-environment)
        (pop-var-environment 'current-goal tutor-environment)
      )
  )
  (*tutor-task-initial-call-debug-program
    *if ((= state initial-call) (= goal debug-program))
    *action
      (progn
        (put-var-environment 'state 'starting tutor-environment)
        (put-var-environment 'current-goal goal tutor-environment)
      )
  )
  (*tutor-task-starting-debug-program
    *if ((= state starting) (= current-goal debug-program))
    *action
      (exit-tut-state 'completed 'get-symptoms)
  )
)
```

```

)

(*tutor-task-starting-get-symptoms
  *if ((= state starting) (= current-goal get-symptoms))
  *action
    (progn
      (exit-tut-state 'completed 'get-program-state
        'get-missing-output
        'get-bad-output 'get-good-output)
    )
)

(*tutor-task-starting-get-program-state
  *if ((= state starting) (= current-goal get-program-state))
  *action
    (progn
      (exit-tut-state 'doing)
    )
)

(*tutor-task-doing-get-program-state
  *if ((= state doing) (= current-goal get-program-state))
  *action
    (progn
      (setq *temp*
        (request-wait 'coach-student 'tutor
          '*get-program-state))
      (cond
        ((car *temp*)
          (exit-tut-state 'completed))
        (t (state-machine-abort))
      )
    )
)

(*tutor-task-starting-get-missing-output
  *if ((= state starting) (= current-goal get-missing-output))
  *action
    (progn
      (setq *temp*
        (request-wait 'coach-student 'tutor
          '*check-missing-output))
      (cond
        ((car *temp*)

```

```

                                (cond
                                  ((get-program-run-info '*missing-output
                                                         '*student)
                                   (exit-tut-state 'doing))
                                  (t (exit-tut-state 'completed)))
                                ))
                                (t (state-machine-abort))
                              )
                            )
      )

(*tutor-task-doing-get-missing-output
  *if ((= state doing) (= current-goal get-missing-output))
  *action
    (progn
      (setq *temp*
            (request-wait 'coach-student 'tutor
                          '*get-missing-output))
      (cond
        ((car *temp*)
         (exit-tut-state 'completed))
        (t (state-machine-abort)))
      )
    )
  )

(*tutor-task-starting-get-bad-output
  *if ((= state starting) (= current-goal get-bad-output))
  *action
    (progn
      (setq *temp*
            (request-wait 'coach-student 'tutor
                          '*check-bad-output))
      (cond
        ((car *temp*)
         (cond
           ((get-program-run-info '*bad-output
                                  '*student)
            (exit-tut-state 'doing))
           (t (exit-tut-state 'completed)))
         )
        (t (state-machine-abort)))
      )
    )
  )

```

```

)

(*tutor-task-doing-get-bad-output
  *if ((= state doing) (= current-goal get-bad-output))
  *action
    (progn
      (setq *temp*
        (request-wait 'coach-student 'tutor
          '*get-bad-output))
      (cond
        ((car *temp*)
          (exit-tut-state 'completed))
        (t (state-machine-abort)))
      )
    )
)

(*tutor-task-starting-get-good-output
  *if ((= state starting) (= current-goal get-good-output))
  *action
    (progn
      (setq *temp*
        (request-wait 'coach-student 'tutor
          '*check-good-output))
      (cond
        ((car *temp*)
          (cond
            ((get-program-run-info '*good-output
              '*student)
              (exit-tut-state 'doing))
            (t (exit-tut-state 'completed 'localize)))
          ))
        (t (state-machine-abort)))
      )
    )
)

(*tutor-task-doing-get-good-output
  *if ((= state doing) (= current-goal get-good-output))
  *action
    (progn
      (setq *temp*
        (request-wait 'coach-student 'tutor '*get-good-output))
      (cond

```

```

                ((car *temp*)
                 (exit-tut-state 'completed 'localize))
                (t (state-machine-abort))
            )
        )
    )

(*tutor-task-starting-localize
  *if ((= state starting) (= current-goal localize))
  *action
    (progn
      (exit-tut-state 'doing)
    )
)

(*tutor-task-doing-localize
  *if ((= state doing) (= current-goal localize))
  *action
    (progn
      (setq *temp*
             (request-wait 'coach-student 'tutor '*localize-error))
      (cond
        ((car *temp*)
         (exit-tut-state 'completed 'explain-symptoms))
        (t (state-machine-abort))
      )
    )
)

(*tutor-task-starting-explain-symptoms
  *if ((= state starting) (= current-goal explain-symptoms))
  *action
    (progn
      (exit-tut-state 'doing)
    )
)

(*tutor-task-doing-explain-symptoms
  *if ((= state doing) (= current-goal explain-symptoms))
  *action
    (progn
      (setq *temp*
             (request-wait 'query-student 'tutor 'explain-symptoms))
      (cond

```

```

                ((car *temp*)
                 (request-wait 'explain-symptoms 'tutor)))
            (exit-tut-state 'completed)
        )
    )
)

```

D.2. Coach state rules

; note - order of rules IS important, as is one pass. could be made
; order insensitive, but conditions would be more complex.

```

(setq **coach-state-rules** '(
  (*coach-terminating
   *if (terminate)
   *action
     (progn
      (state-machine-terminate)
     )
  )

  (*coach-aborting
   *if (abort)
   *action
     (progn
      (state-machine-abort)
     )
  )

  (*coach-turning-off-temp-coaching
   *if (temp-coaching (= state coaching))
   *action
     (put-var-environment 'temp-coaching nil coach-environment)
  )

  (*coach-turning-off-temp-teaching
   *if (temp-teaching (= state teaching))
   *action
     (put-var-environment 'temp-teaching nil coach-environment)
  )

  (*coach-teaching
   *if ((*or (= state perform-action)
             (*and got-debug-write-results

```



```

                (= state check-student-progress)
                (*not (= action-quality *bad)))
            (*and (= state check-student-progress)
                (= action-quality *bad)))
        (*or temp-teaching (= coach-mode teaching)))
    *action
    (progn
      (put-var-environment 'state 'teaching coach-environment)
      (teach-specific-task coach-environment)
      (request-wait 'teach 'coach coach-environment)
    )
  )
)

(*coach-coaching
  *if ((*or (= state perform-action)
            (*and got-debug-write-results
                (= state check-student-progress)
                (*not (= action-quality *bad)))
            (*and (= state check-student-progress)
                (= action-quality *bad))))
        (*or temp-coaching (= coach-mode coaching)))
    *action
    (progn
      (put-var-environment 'state 'coaching coach-environment)
      (coach-specific-task coach-environment)
      (request-wait 'coach-task 'coach coach-environment)
    )
  )
)

(*coach-waiting-student-action
  *if ((*or (= state initial)
            (*and (= state coaching)
                  (*not (*and (= action-quality *bad)
                               got-debug-write-results)))
            (*and (= state teaching)
                  (*not (*and (= action-quality *bad)
                               got-debug-write-results)))
            (*and (= state perform-action)
                  (*not got-debug-write-results))
            (*and (= state check-student-progress)
                  (= action-quality *bad)
                  (*not got-debug-write-results))
            (*and got-debug-write-results
                  (= state check-student-progress)

```

```

(*not (= action-quality *bad))))
*action
  (progn
    (put-var-environment 'state 'waiting-action
      coach-environment)
    (setq *student-action* (wait-for-student-action
      coach-environment))
    (put-var-environment 'got-debug-write-results nil
      coach-environment)
    (push-var-environment 'student-action *student-action*
      coach-environment)
  )
)

(*coach-wait-supplementary-info
  *if ((= state waiting-action))
  *action
    (progn
      (put-var-environment 'state 'waiting-supplementary
        coach-environment)
      (setq *sup-info* (coach-wait-for-supplementary-info
        coach-environment))
      (push-var-environment 'supplementary-info *sup-info*
        coach-environment)
    )
)

(*coach-verify-action
  *if ((= state waiting-supplementary))
  *action
    (progn
      (put-var-environment 'state 'verify-action coach-environment)
      (push-var-environment 'action-quality
        (coach-verify-student-action coach-environment)
        coach-environment)
      (save-performance 'action coach-environment)
    )
)

(*coach-check-student-progress
  *if ((*or (= state verify-action)
    (= state verify-correctness)))
  *action
    (progn

```

```

(put-var-environment 'state 'check-student-progress
  coach-environment)
(let* (mode (past-progress (get-current-progress-level))
      (validity (get-var-value-environment 'action-quality
        coach-environment))
      (current-progress
        (update-progress
          (get-var-value-environment 'student-goal
            coach-environment)
          (get-var-value-environment 'student-action
            coach-environment)
          validity)))
      (put-var-environment 'past-progress past-progress
        coach-environment)
      (put-var-environment 'current-progress current-progress
        coach-environment)
      (cond
        ((setq mode (new-coach-mode coach-environment))
          (put-var-environment 'coach-mode mode
            coach-environment))
        ((need-temp-teach coach-environment)
          (put-var-environment 'temp-teaching t
            coach-environment))
        ((need-temp-coach coach-environment)
          (put-var-environment 'temp-coaching t
            coach-environment)))
      )
  ))
)

(*coach-test-task-completion
  *if ((= state check-student-progress)
    (*not got-debug-write-results))
  *action
  (progn
    (put-var-environment 'state 'check-task-completion
      coach-environment)
    (cond
      ((check-task-completion coach-environment)
        (put-var-environment 'terminate t
          coach-environment)))
    )
  )
)

```

```

)

(*coach-check-action
  *if ((= state check-task-completion))
  *action
    (progn
      (put-var-environment 'state 'check-action coach-environment)

      (cond
        ((or (equal *student-action* 'insert-value-write)
              (equal *student-action* 'insert-flow-write))
          (put-var-environment 'got-debug-write-results t
                               coach-environment)))
      )
    )
)

(*coach-perform-write-action
  *if (got-debug-write-results (= state check-action))
  *action
    (progn
      (put-var-environment 'state 'perform-write-action
                           coach-environment)
      (perform-student-action coach-environment)
      )
    )
)

(*coach-perform-action
  *if ((= state check-action))
  *action
    (progn
      (put-var-environment 'state 'perform-action
                           coach-environment)
      (perform-student-action coach-environment)
      )
    )
)

(*coach-wait-question-correctness
  *if (got-debug-write-results (*or (= state perform-write-action)
                                     (*and (= state check-student-progress)
                                           (= action-quality *bad))
                                     (*and (= state coaching)
                                           (= action-quality *bad))
                                     (*and (= state teaching)
                                           (= action-quality *bad))))

```

```

(= action-quality *bad)))
*action
  (progn
    (put-var-environment 'state 'waiting-question-correctness
      coach-environment)
    (setq *correctness-info* (coach-wait-for-correctness-info
      coach-environment))
    (push-var-environment 'correctness-info *correctness-info*
      coach-environment)
  )
)

(*coach-verify-correctness
  *if ((= state waiting-question-correctness))
  *action
    (progn
      (put-var-environment 'state 'verify-correctness
        coach-environment)
      (push-var-environment 'action-quality
        (coach-verify-student-correctness coach-environment)
        coach-environment)
      (save-performance 'correctness coach-environment)
    )
  )
)
))

```

APPENDIX E

Consulting function invocation rules

For versatility, many of the various states during the consulting process use individualized rule-based selection to determine the actual actions to take depending on the specific consulting circumstances. Since this technique is used, even when the state represents a stub in the system, changing the stub into the actual code will only require the replacement of the rule-base with one containing the appropriate rules.

E.1. Student action rules

These rules determine what the student is expected to be doing at any particular time in the consulting process, and provides the appropriate prompting for the specified action.

; *student-action* is a global variable with the most recent student action

```
(setq **student-input-rules** '(
  (**input-student-program-state
    *if ((= student-goal *get-program-state))
    *action
      (let (action)
        (cond
          ((setq action (request-wait 'query-student 'coach
            'program-state))
            (car action)))
        )
      )
  )
  (**input-student-check-missing-output
    *if ((= student-goal *check-missing-output))
    *action
      (let (action)
        (cond
          ((setq action (request-wait 'query-student 'coach
            'is-there-missing-output))
            (car action)))
        )
      )
  )
)
```

```

(**input-student-get-missing-output
  *if ((= student-goal *get-missing-output))
  *action
    (let (action)
      (cond
        ((setq action (request-wait 'query-student 'coach
          'missing-output))
          (car action)))
      )
    )
)

```

```

(**input-student-check-bad-output
  *if ((= student-goal *check-bad-output))
  *action
    (let (action)
      (cond
        ((setq action (request-wait 'query-student 'coach
          'is-there-bad-output))
          (car action)))
      )
    )
)

```

```

(**input-student-get-bad-output
  *if ((= student-goal *get-bad-output))
  *action
    (let (action)
      (cond
        ((setq action (request-wait 'query-student 'coach
          'bad-output))
          (car action)))
      )
    )
)

```

```

(**input-student-check-good-output
  *if ((= student-goal *check-good-output))
  *action
    (let (action)
      (cond
        ((setq action (request-wait 'query-student 'coach
          'is-there-good-output))
          (car action)))
      )
    )
)

```

```

(**input-student-get-good-output
  *if ((= student-goal *get-good-output))
  *action
    (let (action)
      (cond
        ((setq action (request-wait 'query-student 'coach
          'good-output))
          (car action)))
      )
    )
)

(**input-student-localize-error
  *if ((= student-goal *localize-error))
  *action
    (let (action)
      (cond
        ((setq action (request-wait 'query-student 'coach
          'localize-error))
          (car action)))
      )
    )
)
)

```

E.2. Supplementary information rules

These rules determine if the particular action being performed needs more information to be provided by the student.

; *student-action* is a global variable with the most recent student action

```

(defun null-info () nil)

(setq **student-sup-input-rules** '(
  (**sup-input-student-program-state
    *if ((= student-goal *get-program-state))
    *action
      (null-info)
    )
  (**sup-input-student-check-missing-output
    *if ((= student-goal *check-missing-output))
    *action
      (null-info)
  )
)
)

```



```

)

(**sup-input-student-get-missing-output
  *if ((= student-goal *get-missing-output))
  *action
      (null-info)
)

(**sup-input-student-check-bad-output
  *if ((= student-goal *check-bad-output))
  *action
      (null-info)
)

(**sup-input-student-get-bad-output
  *if ((= student-goal *get-bad-output))
  *action
      (null-info)
)

(**sup-input-student-check-good-output
  *if ((= student-goal *check-good-output))
  *action
      (null-info)
)

(**sup-input-student-get-good-output
  *if ((= student-goal *get-good-output))
  *action
      (null-info)
)

(**sup-input-student-localize--error-done
  *if ((= student-goal *localize-error) (= student-action done))
  *action
      (null-info)
)

(**sup-input-student-localize--error-hint
  *if ((= student-goal *localize-error) (= student-action hint))
  *action
      (null-info)
)

```

```

(**sup-input-student-localize--error-flow-write
  *if ((= student-goal *localize-error)
        (= student-action insert-flow-write))
    *action
      (let (supl)
        (cond
          ((setq supl (request-wait 'query-student 'coach
                                   'flow-write-location))
           (car supl)))
        )
      )
)

(**sup-input-student-localize--error-value-write
  *if ((= student-goal *localize-error)
        (= student-action insert-value-write))
    *action
      (let (item1 item2 supl)
        (cond
          ((setq item1 (request-wait 'query-student 'coach
                                   'value-write-location))
           (cond
             ((setq item2 (request-wait
                           'query-student
                           'coach 'value-write-variable))
              (setq supl (list (car item1)
                               (car item2))))
             )))
        )
      )
)

(**sup-input-student-localize--error-remove-flow-write
  *if ((= student-goal *localize-error)
        (= student-action remove-flow-write))
    *action
      (let (supl)
        (cond
          ((setq supl (request-wait 'query-student 'coach
                                   'flow-write-location 'removal))
           (car supl)))
        )
      )
)

(**sup-input-student-localize--error-remove-value-write
  *if ((= student-goal *localize-error)

```

```

                                (= student-action remove-value-write))
*action
  (let (item1 item2 supl)
    (cond
      ((setq item1 (request-wait 'query-student 'coach
                                'value-write-location 'removal))
        (cond
          ((setq item2 (request-wait
                        'query-student 'coach
                        'value-write-variable
                        'removal))
            (setq supl (list (car item1)
                              (car item2))))
          )))
    )
)

(**sup-input-student-localize--error-list-start
 *if ((= student-goal *localize-error)
      (= student-action list-lines))
*action
  (let (supl)
    (cond
      ((setq supl (request-wait 'query-student 'coach
                                'list-start-location))
        (car supl)))
    )
)

(**sup-input-student-localize--error-show-output
 *if ((= student-goal *localize-error)
      (= student-action show-output))
*action
  (null-info)
)

(**sup-input-student-localize--error-good-lines
 *if ((= student-goal *localize-error)
      (= student-action mark-good-lines))
*action
  (let (supl)
    (cond
      ((setq supl (request-wait 'query-student 'coach

```

```

                                'good-line-numbers))
                                (car supl)))
                            )
                    )
    (**sup-input-student-localize--error-err-line
     *if ((= student-goal *localize-error)
          (= student-action select-error-line))
     *action
       (let (supl)
         (cond
          ((setq supl (request-wait 'query-student 'coach
                                   'error-line))
           (car supl)))
        )
     )
  ))

```

E.3. Student response validation rules

These rules pick the function to determine if the student's action is valid or not.

; *student-action* is a global variable with the most recent student action

```

(setq **validation-rules** '(
  (*validate-program-state
   *if ((= student-goal *get-program-state))
   *action
     (validate-program-state *student-action* )
  )
  (*validate-check-good-output
   *if ((= student-goal *check-good-output))
   *action
     (validate-check-good-output *student-action* )
  )
  (*validate-get-good-output
   *if ((= student-goal *get-good-output))
   *action
     (validate-get-good-output *student-action* )
  )
)

```

```
(*validate-check-missing-output
  *if ((= student-goal *check-missing-output))
  *action
      (validate-check-missing-output *student-action* )
)
```

```
(*validate-get-missing-output
  *if ((= student-goal *get-missing-output))
  *action
      (validate-get-missing-output *student-action* )
)
```

```
(*validate-check-bad-output
  *if ((= student-goal *check-bad-output))
  *action
      (validate-check-bad-output *student-action* )
)
```

```
(*validate-get-bad-output
  *if ((= student-goal *get-bad-output))
  *action
      (validate-get-bad-output *student-action* )
)
```

```
(*validate-localize-error-done
  *if ((= student-goal *localize-error)
      (= student-action done))
  *action
      (excellent-action)
)
```

```
(*validate-localize-error-list-lines
  *if ((= student-goal *localize-error)
      (= student-action list-lines))
  *action
      (excellent-action)
)
```

```
(*validate-localize-error-show-output
  *if ((= student-goal *localize-error)
      (= student-action show-output))
  *action
```

```

        (excellent-action)
    )

(*validate-localize-error-hint
  *if ((= student-goal *localize-error)
      (= student-action hint))
  *action
      (excellent-action)
)

(*validate-localize-error-insert-value-write
  *if ((= student-goal *localize-error)
      (= student-action insert-value-write))
  *action
      (validate-localize-error-insert-value-write *sup-info*)
)

(*validate-localize-error-insert-flow-write
  *if ((= student-goal *localize-error)
      (= student-action insert-flow-write))
  *action
      (validate-localize-error-insert-flow-write *sup-info*)
)

(*validate-localize-error-remove-flow-write
  *if ((= student-goal *localize-error)
      (= student-action remove-flow-write))
  *action
      (validate-localize-error-remove-flow-write *sup-info*)
)

(*validate-localize-error-remove-value-write
  *if ((= student-goal *localize-error)
      (= student-action remove-value-write))
  *action
      (validate-localize-error-remove-value-write *sup-info*)
)

(*validate-localize-error-mark-good-lines
  *if ((= student-goal *localize-error)
      (= student-action mark-good-lines))
  *action
      (validate-localize-error-mark-good-lines *sup-info*)
)

```

```

(*validate-localize-error-select-error-line
  *if ((= student-goal *localize-error)
        (= student-action select-error-line))
  *action
    (validate-localize-error-select-error-line *sup-info*)
  )
))

```

E.4. Flow write placement validity rules

These rules determine the validity of the placement of a flow write. Currently it does allow for automated checking for impossible placing of the write, but other cases require further input into the system. This is representative of rules which bases which can automatically handle some situations, but have to be given values in others. The *value write placement* rules are similar.

; *sup-info* contains line number for flow trace

```

(setq **flow-write-rules** '(
  (*impossible-flow-write-placement
    *if ((*not (possible-trace-line supplementary-info)))
    *info
      (list (bad-action) '(not-possible))
  )

  (*default-flow-write-placement
    *if ((= a a))
    *info
      (let (value note)
        (princ " <<>> enter validity value [0..16] >> ")
        (setq value (read))
        (princ " <<>> enter note [list] >> ")
        (setq note (read))
        (list value note)
      )
  )
))

```

E.5. Debugging write correctness request rules

These rules cause the student to receive the appropriate prompting relative to the correctness of the debugging write results just produced by the system.

; *student-action* is a global variable with the most recent student action

```
(setq **student-correctness-input-rules** '(
  (**correctness-check-flow-write
    *if ((= student-action insert-flow-write))
    *action
      (let (info)
        (cond
          ((setq info (request-wait 'query-student 'coach
            'flow-write-correctness))
            (car info)))
        )
      )
  )
  (**correctness-check-value-write
    *if ((= student-action insert-value-write))
    *action
      (let (info)
        (cond
          ((setq info (request-wait 'query-student 'coach
            'value-write-correctness))
            (car info)))
        )
      )
  )
))
```

E.6. Student response correctness rules

These rules choose the appropriate method for determining correctness of the student's response when queried about the correctness of the results of a debugging write.

; *student-answer* is a global variable with the most recent student answer

```
(setq **correctness-rules** '(
  (*validate-correctness-value-write
    *if ((= student-action insert-value-write))
    *action
      (validate-correctness-value-write *correctness-info* *sup-info*)
  )
  (*validate-correctness-flow-write
    *if ((= student-action insert-flow-write))
```



```

      *action
      (validate-correctness-flow-write *correctness-info* *sup-info*)
    )
  ))

```

E.7. Task completion rules

These rules are used to determine if the current coaching task has been completed.

; *student-action* is a global variable with the most recent student action

```

(setq **completion-rules** '(
  (*test-completion-program-state
    *if ((= student-goal *get-program-state))
    *action
    (is-excellent-action (get-var-value-environment 'action-quality
      coach-environment))
  )
  (*test-completion-check-missing-output
    *if ((= student-goal *check-missing-output))
    *action
    (is-excellent-action (get-var-value-environment 'action-quality
      coach-environment))
  )
  (*test-completion-missing-output
    *if ((= student-goal *get-missing-output))
    *action
    (found-all-missing-output)
  )
  (*test-completion-check-bad-output
    *if ((= student-goal *check-bad-output))
    *action
    (is-excellent-action (get-var-value-environment 'action-quality
      coach-environment))
  )
  (*test-completion-bad-output
    *if ((= student-goal *get-bad-output))
    *action
  )
)

```

```
        (found-all-bad-output)
    )
  (*test-completion-check-good-output
    *if ((= student-goal *check-good-output))
    *action
      (is-excellent-action (get-var-value-environment 'action-quality
        coach-environment))
  )
  (*test-completion-good-output
    *if ((= student-goal *get-good-output))
    *action
      (found-all-good-output)
  )
  (*test-completion-localize-error
    *if ((= student-goal *localize-error))
    *action
      (equal (get-var-value-environment 'student-action
        coach-environment) 'done)
  )
))
```

APPENDIX F

Teaching/coaching rules

These rule bases relate to the teaching and coaching of the student.

F.1. Coach mode rules

These rules are used to determine the overall coaching mode: *watching*, *coaching*, or *teaching*.

; **student-action** is a global variable with the most recent student action

```
(setq **coaching-threshold-on** (mediocre-action))
(setq **coaching-threshold-off** (+ (mediocre-action) 3))
(setq **teaching-threshold-on** (poor-action))
(setq **teaching-threshold-off** (+ (poor-action) 2))

(setq **mode-rules** '(
  (*watching-to-coaching
    *if ((< current-progress *coaching-on) (> past-progress *coaching-on)
      (= coach-mode watching))
    *info
      coaching
  )
  (*coaching-to-watching
    *if ((> current-progress *coaching-off) (< past-progress *coaching-off)
      (= coach-mode coaching))
    *info
      watching
  )
  (*coaching-to-teaching
    *if ((< current-progress *teaching-on) (> past-progress *teaching-on)
      (= coach-mode coaching))
    *info
      teaching
  )
)
```

```

(*teaching-to-coaching
  *if ((> current-progress *teaching-off) (< past-progress *teaching-off)
      (= coach-mode teaching))
  *info
    coaching
)
))

```

F.2. Temporary coaching rules

These rules determine if a temporary coaching action, rather than switching the overall coaching mode, needs to occur.

; *student-action* is a global variable with the most recent student action

```

(setq **temp-coach-rules** '(
  (*strictly-correct-input
    *if ((>= action-quality *excellent))
    *info
      nil
  )
  (*already-coaching-teaching
    *if ((*or (= coach-mode teaching) (= coach-mode coaching)))
    *info
      nil
  )
  (*strictly-incorrect-input
    *if ((<= action-quality *bad))
    *info
      t
  )
  (*less-than-good--poor-action
    *if ((<= action-quality *poor) (< current-progress *good))
    *info
      t
  )
  (*less-than-mediocre--mediocre-action
    *if ((<= action-quality *mediocre) (<= current-progress *mediocre))
    *info

```

```

        t
    )
))

```

F.3. Temporary teaching rules

Similar to the temporary coaching rules, these rules determine if the coach needs to teach a specific topic while not in overall teaching mode.

; *student-action* is a global variable with the most recent student action

```

(setq **temp-teach-rules** '(
    (*already-teaching
      *if ((= coach-mode teaching))
      *info
      nil
    )
    (*already-coaching--poor-action
      *if ((= coach-mode coaching) (<= action-quality *poor))
      *info
      t
    )
))

```

F.4. Student understanding

This *stub* rule-base causes the system to prompt for the understanding that a student has about a particular topic in the prerequisite tree. This is representative of rule-bases which are not yet automated in any respect and thus are entirely dependent on external input. Similar are the bases to find the *root topic*, and to determine *relevancy* of a topic.

; *topic* contains topic from net to check

```

(setq **find-understanding-rules** '(
    (*default-topic-understanding
      *if ((= a a))
      *info
      (progn
        (princ " <<>> enter understanding for ")
        (princ *topic*)
        (princ " [none/partial/full] >> ")
      )
    )
))

```

(read)

)

)

)

APPENDIX G

Variable comparison rules

The following rule bases are used to select the possible methods for matching input and output variables in the student's program to those in the sample program.

G.1. Input variable matching rules

```
(setq invar-match-rules '(
  (*invar-3 *if ((= state initial))
    *action (invar-set-initial-environment))
  (*invar-1 *if ((*or (= u-student 0) (= u-sample 0)))
    *action (invar-make-satisfied))
  (*invar-2 *if ((= u-student u-sample) (= m-student m-sample)
    (= m-student 0) (*not simple-unsafe))
    *action (invar-try-simple-match))
  (*invar-4
    *if
      ((*exists-var *student (*is-true read-in-loop *))
       (*exists-var *sample (*is-true read-in-loop *))
       (*not try-loop-unsafe))
    *action (invar-try-loop-match))
))
```

G.2. Output variable matching rules

```
(setq outvar-match-rules '(
  (*outvar-1 *if ((*or (= u-student 0) (= u-sample 0)))
    *action (outvar-make-satisfied))
  (*outvar-2 *if ((= state initial))
    *action (outvar-set-initial-environment))
))
```

```

(*outvar-4a *if (> u-student 0) (*not echo-tried))
  *action (outvar-echo-input))

(*outvar-4b *if (> u-sample 0) (*not echo-tried))
  *action (outvar-echo-input))

(*outvar-3 *if ((= u-student 1) (= u-sample 1)
                (= m-student 0) (= m-sample 0) (*not single-tried))
  *action (outvar-single-output))

(*outvar-5
 *if
   ((*exists-var-list *student (*is-true is-sum-var *) unmatched-student)
    (*exists-var-list *sample (*is-true is-sum-var *) unmatched-sample)
    (*not sum-var-tried))
 *action
   (outvar-generic-collection-match
    '*sum-out-vars 'sum-var-matches 'sum-var-tried))

(*outvar-6
 *if
   ((*exists-var-list *student (*is-true is-count-var *) unmatched-student)
    (*exists-var-list *sample (*is-true is-count-var *) unmatched-sample)
    (*not count-var-tried))
 *action
   (outvar-generic-collection-match
    '*count-out-vars 'count-var-matches 'count-var-tried))

(*outvar-7
 *if
   ((*exists-var-list *student (*is-true is-count-calculation-var *)
                          unmatched-student)
    (*exists-var-list *sample (*is-true is-count-calculation-var *)
                          unmatched-sample)
    (*not count-calculation-var-tried))
 *action
   (outvar-generic-collection-match
    '*count-calculation-out-vars 'count-calculation-var-matches
    'count-calculation-var-tried))

(*outvar-8
 *if
   ((*exists-var-list *student (*is-true is-sum-calculation-var *)
                          unmatched-student)

```



```

(*exists-var-list *sample (*is-true is-sum-calculation-var *)
                        unmatched-sample)
(*not sum-calculation-var-tried))
*action
(outvar-generic-collection-match
 '*sum-calculation-out-vars 'sum-calculation-var-matches
 'sum-calculation-var-tried))

(*outvar-9
 *if
 ((*exists-var-list *student (*is-true output-in-loop *)
                        unmatched-student)
 (*exists-var-list *sample (*is-true output-in-loop *)
                        unmatched-sample)
 (*not loop-output-tried))
*action
(outvar-generic-collection-match
 '*loop-out-vars 'loop-output-var-matches
 'loop-output-tried))

(*outvar-10
 *if
 ((*exists-var-list *student (*is-true depends-on-single-invar *)
                        unmatched-student)
 (*exists-var-list *sample (*is-true depends-on-single-invar *)
                        unmatched-sample)
 (*not single-invar-dependency-tried))
*action
(outvar-single-invar-match
 'single-invar-dependency-tried))

(*outvar-11
 *if
 ((*exists-var-list *student (*is-true depends-on-multiple-invars *)
                        unmatched-student)
 (*exists-var-list *sample (*is-true depends-on-multiple-invars *)
                        unmatched-sample)
 (*not multiple-invar-dependency-tried))
*action
(outvar-multiple-invar-match
 'multiple-invar-dependency-tried))

(*outvar-12
 *if

```

```

      ((*exists-var-list *student (*is-true output-in-decision *)
        unmatched-student)
      (*exists-var-list *sample (*is-true output-in-decision *)
        unmatched-sample)
      (*not decision-output-tried))
    *action
      (outvar-generic-collection-match
        '*decision-out-vars 'decision-output-var-matches
        'decision-output-tried))

(*outvar-13
  *if
    ((*exists-var-list *student (*is-true outvar-depends-on-mult *)
      unmatched-student)
    (*exists-var-list *sample (*is-true outvar-depends-on-mult *)
      unmatched-sample)
    (*not mult-in-calc-tried))
  *action
    (outvar-generic-collection-match
      '*outvars-dependending-on-mult
      'multiplication-output-var-matches
      'mult-in-calc-tried))

(*outvar-14
  *if
    ((*exists-var-list *student (*is-true outvar-depends-on-subt *)
      unmatched-student)
    (*exists-var-list *sample (*is-true outvar-depends-on-subt *)
      unmatched-sample)
    (*not subt-in-calc-tried))
  *action
    (outvar-generic-collection-match
      '*outvars-dependending-on-subt 'subtraction-output-var-matches
      'subt-in-calc-tried))

(*outvar-15
  *if
    ((*exists-var-list *student (*is-true outvar-depends-on-realdiv *)
      unmatched-student)
    (*exists-var-list *sample (*is-true outvar-depends-on-realdiv *)
      unmatched-sample)
    (*not realdiv-in-calc-tried))
  *action
    (outvar-generic-collection-match

```

```

      '*outvars-depending-on-realdiv
      'realdivision-output-var-matches
      'realdiv-in-calc-ried))

(*outvar-16
  *if
    ((*exists-var-list *student (*is-true outvar-depends-on-intdiv *)
      unmatched-student)
     (*exists-var-list *sample (*is-true outvar-depends-on-intdiv *)
      unmatched-sample)
     (*not intdiv-in-calc-ried))
  *action
    (outvar-generic-collection-match
     '*outvars-depending-on-intdiv
     'integerdivision-output-var-matches
     'intdiv-in-calc-ried))

(*outvar-17
  *if
    ((*exists-var-list *student (*is-true outvar-depends-on-intmod *)
      unmatched-student)
     (*exists-var-list *sample (*is-true outvar-depends-on-intmod *)
      unmatched-sample)
     (*not intmod-in-calc-ried))
  *action
    (outvar-generic-collection-match
     '*outvars-depending-on-intmod
     'integermodulus-output-var-matches
     'intmod-in-calc-ried))

))

```

APPENDIX H

Rules governing input and output to the student

Student interaction with the system is localized, and governed by rule-bases.

H.1. Reprompt rules

These rules are used when the student need be reprompted for an answer.

```
(setq **reprompt-rules** '(
  (*reprompt-initial-multiple-response
    *if (multiple-response (= multiple-responses 0))
    *action
      (progn
        (incr-var-environment 'multiple-responses int-environment)
        (put-var-environment 'multiple-response nil int-environment)
        (list 'nl "you must enter a single answer or help")
      )
  )
  (*reprompt-initial-no-response
    *if (no-response (= no-responses 0))
    *action
      (progn
        (incr-var-environment 'no-responses int-environment)
        (put-var-environment 'no-response nil int-environment)
        (list 'nl "You did not answer the question"))
  )
  (*reprompt-t/f-multiple-response
    *if (true/false multiple-response (= multiple-responses 1))
    *action
      (progn
        (incr-var-environment 'multiple-responses int-environment)
        (put-var-environment 'multiple-response nil int-environment)
        (list 'nl "This is a true/false type of question, you must enter"
              'nl "a SINGLE answer of EITHER yes OR no OR help")
      )
  )
)
```

```

)
(*reprompt-t/f-initial-no-response
  *if (true/false no-response (= no-responses 1))
  *action
    (progn
      (incr-var-environment 'no-responses int-environment)
      (put-var-environment 'no-response nil int-environment)
      (list 'nl "Please enter yes, no, or help"))
)
(*reprompt-t/f-no-response
  *if (true/false no-response (= no-responses 2))
  *action
    (progn
      (incr-var-environment 'no-responses int-environment)
      (put-var-environment 'no-response nil int-environment)
      (list 'nl "for a true false question, you must either answer" 'nl
        "yes or no. If you don't understand the question enter" 'nl
        "the word help"))
)
(*reprompt-initial-word-multiple-response
  *if (word multiple-response (= multiple-responses 1))
  *action
    (progn
      (incr-var-environment 'multiple-responses int-environment)
      (put-var-environment 'multiple-response nil int-environment)
      (list 'nl "please enter a one word answer or help"))
)
(*reprompt-word-multiple-response
  *if (word multiple-response (= multiple-responses 2))
  *action
    (progn
      (incr-var-environment 'multiple-responses int-environment)
      (put-var-environment 'multiple-response nil int-environment)
      (append
        (list 'nl "please enter the SINGLE word which specifies")
        (map-to-words what)
        (map-to-words qualifiers)
        (list 'nl
          "or the word help if you do not understand the question"))))
)

```

```

(*reprompt-initial-word-numeric-response
  *if (word numeric-response (= numeric-responses 0))
  *action
    (progn
      (incr-var-environment 'numeric-responses int-environment)
      (put-var-environment 'numeric-response nil int-environment)
      (list 'nl "please enter a WORD answer or help"))
)

(*reprompt-word-numeric-response
  *if (word numeric-response (= numeric-responses 1))
  *action
    (progn
      (incr-var-environment 'numeric-responses int-environment)
      (put-var-environment 'numeric-response nil int-environment)
      (list 'nl "you entered a number when a word was desired"
            'nl "please answer with a WORD or the word help"))
)

(*reprompt-initial-value-multiple-response
  *if (value multiple-response (= multiple-responses 1))
  *action
    (progn
      (incr-var-environment 'multiple-responses int-environment)
      (put-var-environment 'multiple-response nil int-environment)
      (list 'nl "please enter a SINGLE CONSTANT value or help"))
)

(*reprompt-value-multiple-response
  *if (value multiple-response (= multiple-responses 2))
  *action
    (progn
      (incr-var-environment 'multiple-responses int-environment)
      (put-var-environment 'multiple-response nil int-environment)
      (list 'nl "You are entering more than one value, please"
            'nl "enter just one constant value"))
)

(*reprompt-initial-value-no-response
  *if (value no-response (= no-responses 1))
  *action
    (progn
      (incr-var-environment 'no-responses int-environment)

```

```

        (put-var-environment 'no-response nil int-environment)
        (list 'nl "please enter an actual CONSTANT value or help"))
    )

(*reprompt-value-no-response
  *if (value no-response (= no-responses 2))
  *action
    (progn
      (incr-var-environment 'no-responses int-environment)
      (put-var-environment 'no-response nil int-environment)
      (list 'nl "The question is asking you to enter a value,"
            'nl "that is, a number, boolean, or character constant"))
    )

(*reprompt-initial-pick-one-multiple-response
  *if (pick-one multiple-response (= multiple-responses 1))
  *action
    (progn
      (incr-var-environment 'multiple-responses int-environment)
      (put-var-environment 'multiple-response nil int-environment)
      (list 'nl "please enter a SINGLE choice from the list"))
    )

(*reprompt-pick-one-multiple-response
  *if (pick-one multiple-response (= multiple-responses 2))
  *action
    (progn
      (incr-var-environment 'multiple-responses int-environment)
      (put-var-environment 'multiple-response nil int-environment)
      (list 'nl "Enter the integer (between 1 and" maxvalue ") which"
            'nl "corresponds to the appropriate choice from the"
            'nl "above list. You may only pick ONE"))
    )

(*reprompt-initial-pick-one-no-response
  *if (pick-one no-response (= no-responses 1))
  *action
    (progn
      (incr-var-environment 'no-responses int-environment)
      (put-var-environment 'no-response nil int-environment)
      (list 'nl "please enter a choice (integer) from the list"))
    )

(*reprompt-pick-one-no-response

```

```

*if (pick-one no-response (= no-responses 2))
*action
  (progn
   (incr-var-environment 'no-responses int-environment)
   (put-var-environment 'no-response nil int-environment)
   (list 'nl "Enter the integer (between 1 and" maxvalue ") which"
         'nl "corresponds to the appropriate choice from the"
         'nl "above list"))
)

(*reprompt-initial-pick-some-no-response
 *if (pick-some no-response (= no-responses 1))
 *action
  (progn
   (incr-var-environment 'no-responses int-environment)
   (put-var-environment 'no-response nil int-environment)
   (list 'nl "please enter one or more choices (integers) from the list"))
)

(*reprompt-pick-some-no-response
 *if (pick-some no-response (= no-responses 2))
 *action
  (progn
   (incr-var-environment 'no-responses int-environment)
   (put-var-environment 'no-response nil int-environment)
   (list 'nl "Enter integers (between 1 and" maxvalue ") which"
         'nl "correspond to the appropriate choice(s) from the"
         'nl "above list"))
)

(*reprompt-initial-pick-one-outofrange-response
 *if (pick-one outofrange-response (= outofrange-responses 0))
 *action
  (progn
   (incr-var-environment 'outofrange-responses int-environment)
   (put-var-environment 'outofrange-response nil int-environment)
   (list 'nl "you entered a value which is not possible"
         'nl "You must only enter an item from the list given"))
)

(*reprompt-pick-one-outofrange-response
 *if (pick-one outofrange-response (= outofrange-responses 1))
 *action
  (progn

```



```

(incr-var-environment 'outofrange-responses int-environment)
(put-var-environment 'outofrange-response nil int-environment)
(list 'nl "Enter ONLY an integer (between 1 and" maxvalue ") which"
      'nl "correspond to the appropriate choice from the"
      'nl "above list"))
)

(*reprompt-initial-pick-some-outofrange-response
 *if (pick-some outofrange-response (= outofrange-responses 0))
 *action
      (progn
        (incr-var-environment 'outofrange-responses int-environment)
        (put-var-environment 'outofrange-response nil int-environment)
        (list 'nl "you entered a value which is not possible"
              'nl "You must only enter item(s) from the list given."
              'nl "You must reenter all choices again"))
      )
)

(*reprompt-pick-some-outofrange-response
 *if (pick-some outofrange-response (= outofrange-responses 1))
 *action
      (progn
        (incr-var-environment 'outofrange-responses int-environment)
        (put-var-environment 'outofrange-response nil int-environment)
        (list 'nl "Enter ONLY integers (between 1 and" maxvalue ") which"
              'nl "correspond to the appropriate choice(s) from the"
              'nl "above list" "You must reenter ALL choices again"))
      )
)

(*reprompt-int-list-outofrange-response
 *if (int-list outofrange-response)
 *action
      (progn
        (incr-var-environment 'outofrange-responses int-environment)
        (put-var-environment 'outofrange-response nil int-environment)
        (list 'nl "Enter ONLY integers which correspond to line numbers"
              'nl "in your program"
              "you must reenter ALL choices again"))
      )
)
))

```

H.2. Requery rules

These rules provide the mechanism to convert one kind of query to another in the event that the student appears not to be able to answer a question in one form.

```
(setq **requery-rules** '(
  (*requery-initial-t/f
    *if ((= kind t/f) (*eval (= tries 1)))
    *action
      (boolean-to-pick-one name what qualifiers)
  )
  (*requery-initial-pick-one
    *if ((= kind pick-one) (*eval (= tries 1)))
    *action
      (pick-one-to-boolean name what qualifiers possibles)
  )
  (*requery-initial-pick-some
    *if ((= kind pick-some) (*eval (= tries 1)))
    *action
      (pick-some-to-boolean name what qualifiers possibles)
  )
))
```

APPENDIX I

Dictionary

The following is the dictionary used by the system which correlates internal items to kinds of queries.

```
(dict-insert 'program-filename
  '(*as-question-kind word)
  '(*as-question-object "filename of your program"))

(dict-insert 'problem-number
  '(*as-question-kind value)
  '(*as-question-object "number of the programming assignment"))

(dict-insert 'first-name
  '(*as-question-kind word)
  '(*as-question-object "your first name"))

(dict-insert 'program-state
  '(*as-question-kind pick-one)
  '(*as-question-object "state of your program")
  '(*possible-answers (runs-to-completion infinite-loop eof-error
    other-runtime-error)))

(dict-insert 'is-there-missing-output
  '(*as-question-kind t/f)
  '(*as-question-object "there is missing output"))

(dict-insert 'value-write-correctness
  '(*as-question-kind t/f)
  '(*as-question-object "the values for variable are correct"))

(dict-insert 'flow-write-correctness
  '(*as-question-kind value)
  '(*as-question-object "number of times the line should have been executed"))

(dict-insert 'is-there-bad-output
  '(*as-question-kind t/f)
  '(*as-question-object "there is bad output"))
```

```
(dict-insert 'is-there-good-output
  '(*as-question-kind t/f)
  '(*as-question-object "there is good output"))
```

```
(dict-insert 'missing-output
  '(*as-question-kind pick-some)
  '(*as-question-object "output which is missing")
  '(*possible-answers *dynamic*))
```

```
(dict-insert 'bad-output
  '(*as-question-kind pick-some)
  '(*as-question-object "output which is bad")
  '(*possible-answers *dynamic*))
```

```
(dict-insert 'good-output
  '(*as-question-kind pick-some)
  '(*as-question-object "output which is good")
  '(*possible-answers *dynamic*))
```

```
(dict-insert 'answer-t-f
  '(*as-statement-verb "answer true or false"))
```

```
(dict-insert 'localize-error
  '(*as-question-kind pick-one)
  '(*as-question-object "the action you would like to perform")
  '(*possible-answers (done hint insert-flow-write insert-value-write
    remove-flow-write remove-value-write show-output list-lines
;   mark-good-lines select-error-line)))
  select-error-line)))
```

```
(dict-insert 'insert-flow-write
  '(*as-pick-list-item "insert a flow write"))
```

```
(dict-insert 'insert-value-write
  '(*as-pick-list-item "insert a value write"))
```

```
(dict-insert 'remove-flow-write
  '(*as-pick-list-item "remove a flow write"))
```

```
(dict-insert 'remove-value-write
  '(*as-pick-list-item "remove a value write"))
```

```
(dict-insert 'show-output
  '(*as-pick-list-item "view output of program"))
```

```
(dict-insert 'list-lines
  '(*as-pick-list-item "view lines of program"))
```

```
(dict-insert 'mark-good-lines
  '(*as-pick-list-item "mark lines as good"))
```

```
(dict-insert 'select-error-line
  '(*as-pick-list-item "found error"))
```

```
(dict-insert 'explain-symptoms
  '(*as-question-kind t/f)
  '(*as-question-object "you desire an explanation of error and symptoms"))
```

```
(dict-insert 'good-line-numbers
  '(*as-question-kind int-list)
  '(*as-question-object "line numbers can you consider good")
  '(*possible-answers *dynamic*))
```

```
(dict-insert 'flow-write-location
  '(*as-question-kind value)
  '(*as-question-object "line number of executable statement preceding flow write"))
```

```
(dict-insert 'value-write-location
  '(*as-question-kind value)
  '(*as-question-object "line number of executable statement preceding value write"))
```

```
(dict-insert 'value-write-variable
  '(*as-question-kind word)
  '(*as-question-object "name of the variable to be written"))
```

```
(dict-insert 'error-line
  '(*as-question-kind value)
  '(*as-question-object "line number of executable statement in error"))
```

```
(dict-insert 'list-start-location
  '(*as-question-kind value)
  '(*as-question-object "line number for start of list"))
```

Bibliography

- [ADAM80] Adam, A. and Laurent, J. P., LAURA, A System to Debug Student Programs, *Artificial Intelligence* 15(1980), 75-122.
- [ANDE90] Anderson, J. R., Boyle, C. F., Corbett, A. T. and Lewis, M. W., Cognitive Modeling and Intelligent tutoring, *Artificial Intelligence* 42, 1 (February 1990), 7-49.
- [BADW91] Badwal, D., Greer, J. E. and McCalla, G. I., UMRAO: A Chess Endgame Tutor, *Proc. IJCAI-12*, Sidney, August 24-30 1991, 1081-1086.
- [BARR76] Barr, A., Beard, M. and Atkinson, R. C., The computer as a tutorial laboratory: the Stanford BIP project, *International Journal of Man-Machine Studies* 8, 5 (Sept. 1976), 567-596.
- [BLOO84] Bloom, B. S., The 2 Sigma Problem: The search for methods of group instruction as effective as one-to-one tutoring, *Educational Researcher* 13(1984), 3-16.
- [BOLD81] Boldyreff, C., Generating a Programming Environment for Learners, in *Computing Skills and the User Interface*, M. J. Coombs and J. L. Alty (ed.), Academic Press, London, 1981, 315-330.
- [BROW75] Brown, J. S., Burton, R. R. and Bell, A. G., SOPHIE: A Step Toward Creating a Reactive Learning Environment, *International Journal of Man Machine Studies* 7, 5 (September 1975), 675-696.
- [BROW82] Brown, J. S., Burton, R. R. and Kleer, J., Pedagogical, Natural Language, and Knowledge engineering Techniques in Sophie I, II, and III, in *Intelligent Tutoring Systems*, D. Sleeman and J. S. Brown (ed.), Academic Press, New York, 1982, 227-282.
- [BURT79] Burton, R. R. and Brown, J. S., An Investigation of Computer Coaching for Informal Learning Activities, *International Journal of Man Machine Studies* 11, 1 (January 1979), 5-24.

- [BURT82] Burton, R. R., Diagnosing bugs in a simple procedural skill, in *Intelligent Tutoring Systems*, D. Sleeman and J. S. Brown (ed.), Academic Press, London, 1982, 157-183.
- [CARB70] Carbonell, J. R., AI in CAI: An Artificial Intelligence Approach to Computer-assisted Instruction, *IEEE Transactions on Man-Machine Systems* 11, 4 (December 1970), 190-202.
- [CORB88] Corbett, A. T., Anderson, J. R. and Patterson, E. J., Problem compilation and tutoring flexibility in the Lisp tutor, *Proc. ITS-88*, Montreal, June 1-3, 1988, 423-429.
- [DION88] Dion, P. and Lelouche, R., Architecture of an intelligent system to teach introductory programming, *Proc. ITS-88*, Montreal, June 1-3, 1988, 387-394.
- [ERMA80] Erman, L. D., Hayes-Roth, F., Lesser, V. R. and Heddy, D. R., Hearsay II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty, *Computing Surveys* 12, 2 (June 1980), 213-253.
- [GOLD82] Goldstein, I. P., The genetic graph: a representation for the evolution of procedural knowledge, in *Intelligent Tutoring Systems*, D. Sleeman and J. S. Brown (ed.), Academic Press, London, 1982, 51-77.
- [HEIN85] Heines, J. M. and O'Shea, T., The design of a rule-based CAI tutorial, *International Journal of Man-Machine Studies* 23(1985), 1-25.
- [JOHN84] Johnson, L. and Soloway, E. M., Intention-based diagnosis of programming errors, *Proceedings of the National Conference on Artificial Intelligence*, Austin, Texas, August 1984, 162-168.
- [JOHN85] Johnson, L. and Soloway, E., PROUST: Knowledge-Based Program Understanding, *IEEE Transactions on Software Engineering* 11, 3 (March 1985), 267-275.
- [JOHN90a] Johnson, B. L., Bergeron, R. D. and Malcolm, P., Modeling the Teaching Consultant, *Computers and Education* 14, 2 (1990), 125-136.
- [JOHN90b] Johnson, W. L., Understanding and debugging Novice Programs, *Artificial Intelligence* 42, 1 (February 1990), 51-97.

- [LOOI88] Looi, C., Apropos2: a program analyser for a prolog intelligent teaching system, *ITS-88*, Montreal, June 1-3, 1988, 379-386.
- [MCCA86] McCalla, G. I., Bunt, R. B. and Harms, J. J., The design of the SCENT automated advisor, *Computational Intelligence* 2, 2 (May 1986), 76-92.
- [MCCA88] McCalla, G. I. and Greer, J. E., Intelligent Advising in Problem Solving Domains: the SCENT-3 architecture, *Proc. ITS-88*, Montreal, June 1-3, 1988, 124-131.
- [MILL82] Miller, M. L., A structured planning and debugging environment for elementary programming, in *Intelligent Tutoring Systems*, D. Sleeman and J. S. Brown (ed.), Academic Press, London, 1982, 119-135.
- [MURR85] Murray, W. R., Heuristic and Formal Methods in Automatic Program Debugging, *Proc. IJCAI-85*, Los Angeles, California, August 1985, 15-19.
- [MURR87] Murray, W. R., Automatic program debugging for intelligent tutoring systems, *Computational Intelligence* 3, 1 (Feb 1987), 1-16.
- [PALM75] Palmer, B. G. and Oldehoeft, A. E., The design of an instructional system based on problem generators, *International Journal of Man-Machine Systems* 7, 2 (March 1975), 249-271.
- [PATT81] Pattis, R. E., *Karel the Robot - A Gentle Introduction to the Art of Programming*, John Wiley & Sons, New York, 1981.
- [REIS85] Reiser, B. J., Anderson, J. R. and Farrell, R. G., Dynamic student modelling in an intelligent tutor for Lisp programming, *Proc. IJCAI-85*, Los Angeles, California, August 1985, 8-14.
- [SLEE81] Sleeman, D. H. and Smith, M. J., Modeling student's problem solving, *Artificial Intelligence* 16, 2 (May 1981), 171-187.
- [STEV82] Stevens, A., Collins, A. and Goldin, S. E., Misconceptions in students' understanding, in *Intelligent Tutoring Systems*, D. Sleeman and J. S. Brown (ed.), Academic Press, London, 1982, 13-24.