Simulating Student Programmers

James C. Spohrer

Computer Science Department
Yale University
51 Prospect Street
New Haven, CT 06520
e-mail: spohrer@cs.yale.edu

Elliot Soloway

Dept of EE and CS
University of Michigan
1101 Beal Avenue
Ann Arbor, MI 48109
e-mail: soloway@csmil.umich.edu

Abstract

A cognitive model of student programmers is presented. The model is based on protocol studies of students writing Pascal programs, and is implemented in a computer simulation program. The claim of this paper is that a computational cognitive model of student program generation fits within a generate-test-and-debug (GTD) problem solving architecture in which impasse/repair knowledge plays a key role. The claim is supported by showing how the model provides a useful descriptive account of the way students write alternative programs.

1 Introduction: Motivation, Goals, and Overview

Our motivation for studying student programmers derives from three beliefs: (1) it is important to teach students design skills (i.e., planning, constructing, and debugging artifacts), (2) programming is an excellent vehicle for teaching design skills, and (3) computers by virtue of their ability to help students visualize and manipulate artifacts can play a unique role in supporting design activities. Our short term goal has been to develop a simulation model of the way students write programs.

Since design tasks can be solved in many different ways, any attempt to understand the way students write programs runs head-long into the *variability problem.* Unlike tasks such as subtracting numbers (i.e., non-design tasks) that have only a single correct answer, there are an enormous number of programs that solve any given programming task (just ask someone who has graded a couple hundred student programs!). When one considers alternative buggy programs as well as correct programs, the variability problem takes on truly staggering proportions, and bug diagnosis for tutorial purposes becomes quite difficult [JS85]. In addition, tracking student program generation behavior for the purpose of providing tutorial assistance is complicated by student variability [ABR85]. A student model for programming should provide an account of the program generation process and the *individual differences* between students that cause vari ibility.

Acknowledgements: This research was funded in part by the National Science Foundation, under NSF Grant MDR-88-96240. We wish to thank David Littman for his comments on drafts of this paper.

To develop a cognitive model of student programmers, we began an in depth study of students as they sat in front of computer terminals and wrote programs. A substantial amount of data -- both on-line protocols (the end-product programs) and thinking-aloud protocols (complete problem solving *behavior traces* of the verbally reported planning, implemention, and debugging steps involved in writing a program) - were collected. Based on an analysis of these data, a *generate-test-and-debug (GTD) problem solving architecture* (see [Su75], [Ham86], [Si88]) was adopted as an overall framework.

During the generate phase, students use different generation mechanisms to write code to achieve the goals of the task specification. The students either (1) used previously acquired programming knowledge to write the code, or (2) created new programming knowledge by translating relevant non-programming knowledge into code. Non-programming knowledge (see [BS85]) corresponds intuitively to knowledge that would allow a student to easily do a *calculation-by-hand.* For instance, a student may be able to calculate the average of an arbitrary set of numbers by hand, but have a great deal of difficulty writing a program to do the same. During the test phase, students use different program testing mechanisms to detect one of a few types of problems, or *impasses.* The students either (1) compared a simulation of their programs to an internal model, or (2) checked for specific commonly occuring bugs. During the debug phase, impasses are fixed using one of small set of *repairs.*

Variability can arise in several ways in a GTD impasse/repair student model. One way variability can arise is when different students choose different repairs for the same impasse [BV80] [BS85]. For instance, when we asked students to write a program that handled both valid and invalid input data, 42% of the students generated a program with an "output-after-error" bug. Figure 1 shows a pseudo-code program with the bug and two repairs. In the buggy program, *if the* input is invalid, after printing the error message the output will be attempted. Since the program should stop after printing the error message, a student might detect an impasse. The impasse is caused by an expectation violation — after the "error" goal, "stop" was expected, but

```
BUGGY PROGRAM            CORRECT PROGRAM 1           CORRECT PROGRAM 2
input                    input                       input
if invalid               if invalid                  if invalid
  then error               then error                  then error
  else calculate          else begin                   else calculate
output                          calculate            if valid
                                output                 then output
                              end

IMPASSE: BAD-NEXT-GOAL   REPAIR: MOVE-ENCOUNTERED    REPAIR: INSERT-SPLIT
after: error             before-encountered: calculate  context-encountered: valid
expected: stop
encountered: output
```

Figure 1: Variability resulting from different repairs.

"output" was found (i.e., BAD-NEXT-GOAL). Some students chose to repair the impasse by moving the the output (i.e., MOVE-ENCOUNTERED), while other students put a guard around the output (i.e., INSERT-SPLIT).

MARCEL is a simulation program that embodies the GTD impasse/repair model descibed in this paper. As shown in Figure 2, MARCEL's inputs are a specification of a programming task and a description of a particular student. MARCEL's outputs are a Pascal program and a behavior trace. A behavior trace corresponds to the main planning and debugging steps taken by a student writing a program. Inside the process box in Figure 2, three main components are identified: (1) model of individual differences, (2) model of program generation, and (3) the contents of working memory.

GRAPES [AFS84] is another system that simulates students writing both correct and buggy programs for moderately complex introductory tasks. Unlike MARCEL's GTD impasse/repair architecture, GRAPES uses a goal-restricted production system architecture and focuses on the way students learn to write short Lisp functions, rather than on accounting for inter-subject variability. In GRAPES, individual differences are modelled by using different production rules and manually changing the contents of working memory during processing.

The remainder of this paper summarizes the four main results of this research: (1) categorization of student programming data, (2) taxonomy of student programmer plan knowledge, (3) model of student program generation, and (4) preliminary model of individual differences and variability. Examples of student program generation and individual differences will be presented.

2 Student Programming Data

The students under study were Yale undergraduates taking their first Pascal programming course. The stimulus material included three programming tasks. Summaries of the tasks along with sample pseudo-code programs are shown in Figure 3 (familiarity with the tasks and programs is essential foi

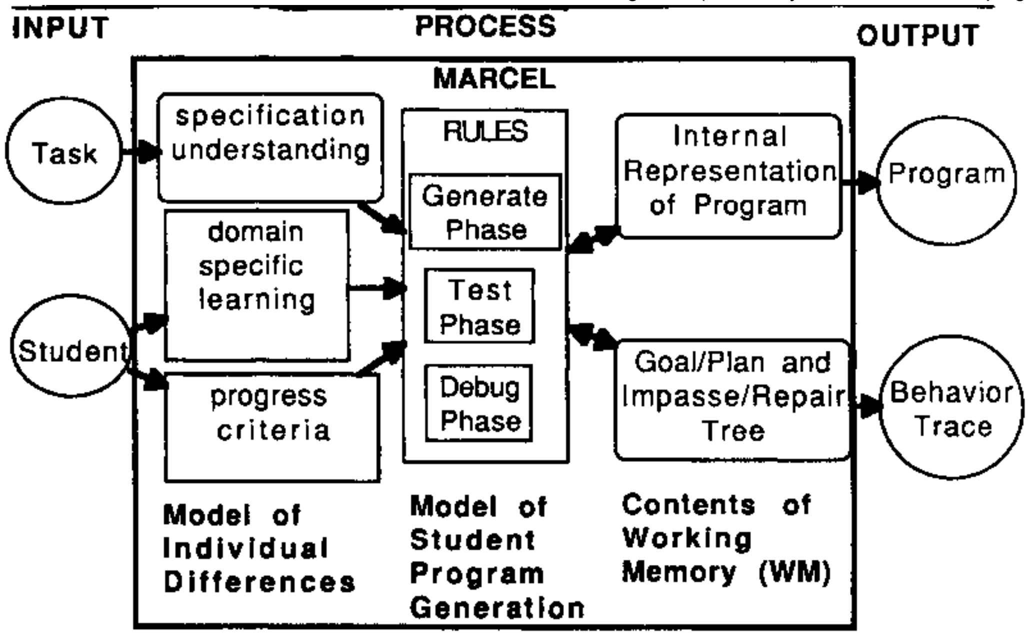

INPUT
PROCESS
OUTPUT
MARCEL
Task
Student
specification understanding
domain specific learning
progress criteria
RULES
Generate Phase
Test Phase
Debug Phase
Internal Representation of Program
Goal/Plan and Impasse/Repair Tree
Program
Behavior Trace
Model of Individual Differences
Model of Student Program Generation
Contents of Working Memory (WM)

Figure 2: An overview of MARCEL.

understanding this paper). The tasks are called arithmetic word programming tasks (see [KG85] [MPS87] for related non-programming tasks).

The data (programs and behavior traces) were broken down into four categories of variability: (1) goals and plans - the goals (e.g., input, validation, calculate, output, loop, etc.) and the alternative plans to achieve the goals with Pascal code were catalogued [SPL*85], (2) bugs - nearly all of the programs contained bugs, and different student made different bugs [SSP85], (3) goal orders - the order in which different students worked on the main goals in the program varied and so the different orders were catalogued, and (4) impasse/repair episodes — on average about once every three minutes while writing the programs, students would encounter difficulties or impasses that they would fix or repair, and so these episodes were catalogued. A coding scheme was developed, and the data were classified with 85% inter-coder reliability into over twenty different subcategories (see [S89] for details).

3 Student Programmer Plan Knowledge

Before considering the way MARCEL simulates inexperienced students, it is instructive to consider the knowledge that MARCEL requires to simulate experienced programmers. When MARCEL simulates an experienced programmer, it performs as an automatic programming system with a knowledge-base that is appropriate for arithmetic word tasks. Programming plans (i.e., multiple lines of code that work together in specific ways) are a key type of knowledge that programmers use when they write and read programs [Sh76][MRRH81][SE84]. Evidence for plans can be found in protocol snippets (e.g., "...same as [last] problem. I'm gonna have... two prompts to enter the value, one outside the loop... and then one inside the loop...).

Two categories of plans have been identified: (1) information transfer or communication plans (e.g., input/validation/output), and (2) information transmutation or calculation plans. One motivation for making this division is that a small set of very modular programming plans result, in which different calculation plans can be "plugged into" standard positions in communication plans. Also, students have well developed non-programming knowledge in these two areas: communication between agents $[P86]_t$ and "calculations-by-hand" [BS85].

The communication plans are broken down into two categories: (1) four input/output plans (i.e., "transform", "alternate", "compress", and "expand"), and (2) four validation plans (i.e., "no-check", "error-stop", "one-retry", and "multiple-retry"). The input/output plans correspond to the four possibilities of the input/output being non-stream/stream, where a stream is a series of values. A non-stream is either an individual (a single value) or an

1. The Electric Bill Task: Calculate the electric bill for a customer {id) based on how many kilowatt hours of electricity the customer used (kwh). The charge is 9 cents for the first 350 kwh used, 5 cents for the next 275 kwh used, 4 cents for the next 225 kwh used, and 3 cents for all usage over 850 kwh. [Implicit requirement from classroom lecture: The program should print an error message and stop if the kwh amount input is invalid.]

```
input(id.kwh)
if invalid
   then error
   else begin
           calculation
           output(id,kwh,cost)
        end
```

2. The Reformatting Task: Read in raw data collected during an experiment and print out the reformatted data. The input is the subject number, problem type ('a', 'b' or 'c'), the start and end times of the experiment (in hours, minutes and seconds), and the subject's accuracy (V or '-'). The output should be the subject number, the problem type, the elapsed time in seconds, and the accuracy. Perform valid-data-entry: If any of the input values are invalid, give the user a second chance to enter them and assume valid data will be entered the second time. The program should input and print out data for a series of subjects as long as the user has more data to process; stop when a sentinel value is entered.

```
valid-data-entry (more-data)
while  not-sentinel(more-data)
  do begin
         valid-data-entry(raw-data)
         calculation
         output(reformatted-data)
         valid-data-entry(more-data)
     end
```

3. The Rainfall Task: Read in a series of rainfall values stopping when the sentinel value (999999) is entered. The sentinel value is a special value, indicating the end of the input data, and should not be included in the calculation. If the input is invalid, prompt the user again and again until the input is valid (vde). The program should print out the number of valid rainfall amounts entered, the number of rainy days, the maximum rainfall amount, and the average rainfall amount.

```
initialization;   vde(rain)
if  sentinel(rain)
    then  no-valid-input-error
   else begin
           while  not-sentinel(rain)
             do begin
                   update; vde(rain)
                end
           calculation; output
        end
```

Figure 3: Three tasks and pseudo-code programs.

```
TRANSFORM: non-stream -> non-stream        EXPAND: non-stream -> stream
others            "error-stop"             others                "error-stop"
vde               input                    vde                   input
calculate         if invalid               while not-sentinel    if invalid
output              then error             do begin                then error
                    else begin                 calculate           else while not-s.
                         calculate             output                       do begin
                         output              end                                calc.
                        end                                                     output
                                                                               end

ALTERNATE: stream -> stream                COMPRESS: stream -> non-stream
others                "error-stop"         others                "error-stop"
vde(more?)            input(more?)         initialization        initialization
while not-sentinel    while not-sentinel   vde                   input
  do begin             do begin            if sentinel           while valid&not-s.
     vde                    input            then error            do begin
     calculate              if invalid     else begin                   update
     output                  then begin          while not-s.           input
      vde(more?)                 error           do begin              end
    end                          more?:=sent.         update     if invalid
                                end                   vde          then error
                              else begin            end            else if no-valid
                                 calculate        calculate                then error
                                 output           output                   else begin
                                 input(more?)   end                              calculate
                                end                                              output
                            end                                                 end
```

Figure 4: Eight info. transfer plans. VDE stands for valid-data-entry

aggregate (a small fixed set of values). Figure 4 shows pseudo-code programs for the possible information transfer plans ("error-stop" validation on right). Referring back to Figure 3, note that the Electric Bill task is of type "transform" "error-stop", the Reformatting Task is "alternate" "one-retry", and the Rainfall Task is "compress" "multiple-retry".

The calculation plans are indexed by a set of goals and objects. The goals consume objects and produce new objects. Each calculation plan is composed of a small set of goals (e.g., CONVERT, COMBINE, etc.) and objects (AMOUNT, COUNT, etc.). The complete set of goals and objects along with supporting protocol evidence is provided in |S89j.

4 Model of Program Generation

The MARCEL model of student program generation is an example of a generate-test-and-debug (GTD) problem solver. A GTD problem solver has three main phases: (1) a generate phase in which plans for goals are generated and implemented, (2) a test phase in which impasses are detected, and (3) a debug phase in which impasses are repaired. Fleshing out the model requires identifying specific mechanisms that underlie each phase and that could give rise to the observed behavior traces. In addition, it requires specifying the impasse/repair knowledge.

During the generate phase, MARCEL uses two mechanisms to write programs: (1) *plan instantiation,* if a student knows a programming plan for achieving a goal, and (2) *plan translation,* if a student does not know a programming plan, but does know a non-programming plan to achieve the goal by hand calculation. Unlike programming plans which organize subgoals using Pascal language constructs, non-programming plans organize subgoals in a domain general manner. Non-programming plans order goals and specify the circumstances (or cases) in which a goal should be achieved. Non-programming plans of this type are called *goal-case-network (GCN) plans. A* GCN plan is a directed and labeled graph (possibly with cycles) that represents information similar to that contained in a programming plan, but without using programming language constructs. A GCN plan for the Electric Bill Task is shown in Figure 5 (note the black square box in the GCN plan is a stop goal, terminating the plan).

A student solving the Electric Bill Task starts by retrieving a template for the top-level program goal (see top of Figure 6). The goal for the body of the program is the next substantitive goal to be achieved. To achieve the body goal, a student may decide to instantiate a "transform" plan learned from solving a previous task, or translate the GCN plan from the specification. If a student instantiates a "transform" "no-check" plan (i.e., input, calculate, output), the calculation will be done on invalid input data (i.e., BAD-KIND-TO-CONSUMER impasse). If the impasse is detected, it can be repaired by inserting a validation guard around the calculation goal (i.e., INSERT-SPLIT repair). However, the repair results in the "output-after-error" bug (the bug described previously in Figure 1; in Figure 6 the second box down from the top on the left).

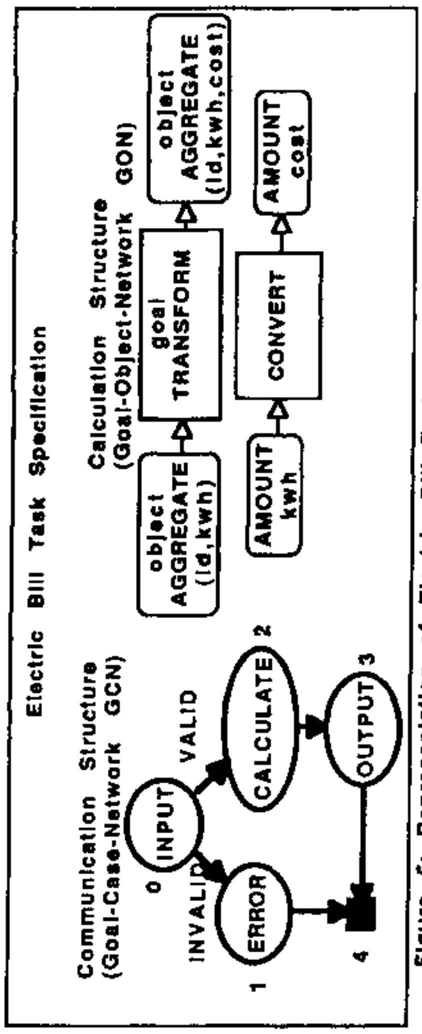


Figure 5: Representation of Electric Bill Task.

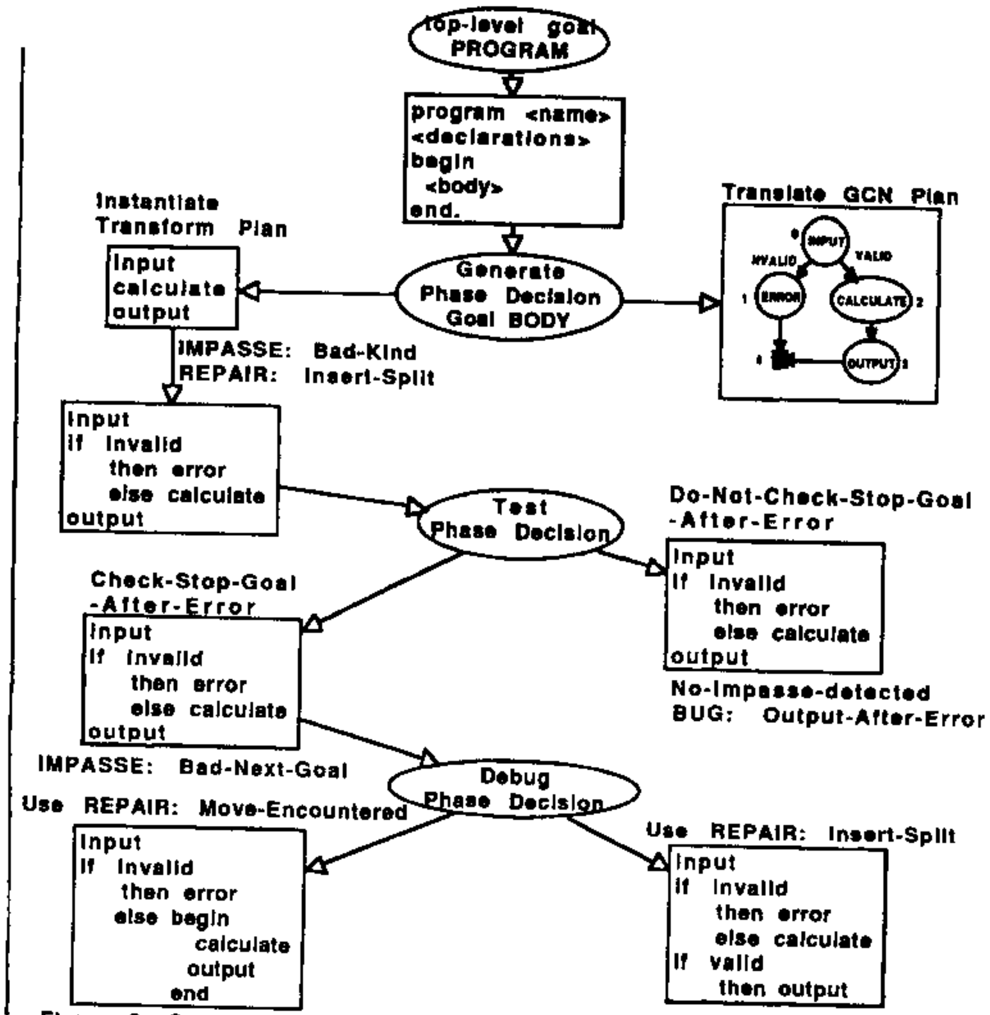


Figure 6: Generating pseudo-code programs for the Electric Bill Task.

During the test phase, MARCEL employs two main mechanisms to test the program and detect impasses: (1) *isonwrphism test,* a student may use expectations derived from non-programming plan knowledge to simulate and match against the program to detect impasses (e.g., bad-next-goal, object-used-but-not-produced, double-use, bad-source, over-write, and bad-kind-to-consumer) (2) *critics,* a student may use a set of critics, or special purpose heuristics learned from textbooks or classroom instruction, to evaluate a program (e.g., variable-used-but-not-declared, missing-begin-end, etc.). The iso-morphism test employs a simulate-and-compare mechanism to ensure that not only are goals being achieved in the correct order, but that appropriate object values are being produced and consumed by the goals. Sometimes students use isomorphism tests that do not detect all impasses under all circumstances. For instance, Figure 6 shows a pseudo-code program with an "output-after-error". The bug is not detected if the stop goal is not checked for after the error goal. However, if the stop goal is checked for then, after the error goal the student would expect to find a stop goal, but instead encounters an output goal, thereby detects an impasse (i.e., BAD-NEXT-GOAL impasse; see third box down on left in Figure 6, also see Figure. 1).

During the debug phase, a repair is selected for an impasse. All of the repairs involve simple editing operations (insert, delete, move, change, duplicate) on a few basic types of program elements (producer, consumer, expected, encountered, object, test, split), defined when a particular impasse is detected in a particular program context. For instance, in Figure 6, a BAD-KIND impasse is repaired with an INSERT-SPLIT repair. At the bottom of Figure 6 (as previously seen in Figure 1), a BAD-NEXT-GOAL impasse is repaired with, in one case, a MOVE-ENCOUNTERED repair, and in a second case, with an INSERT-SPLIT repair. After attempting a repair, the test phase is re-entered to see if the repair succeeds or gives rise to a new impasse. If the repair succeeds the generate phase is returned to, but if the repair gives rise to a new impasse, the debug phase is returned to and a new round of repairing will begin. By applying different repairs to an impasse, programs can be generated in different ways.

5. Model of Individual Differences

Three types of individual differences are considered in the current model: (1) *specification understanding* -- students interpret the same specification differently, (2) *domain-specific learning* -- students learn different programming plans and then re-use them later, and (3) *progress criteria*

[NS72] - students may use different criteria for evaluating what course of action will lead to progress. For example, consider writing a program that repeatedly processes different input values, stopping when a sentinel value is entered (e.g., like the Reformatting Task). Figure 7 shows correct and buggy pseudo-code programs based on the programs that students typically generate.

Specification Understanding: The "missing loop" bug can be generated by assuming an alternative interpretation of the programming task. For instance, one student who left out the loop said, *7 didn't [SEE] this the first time..."* , upon looking more closely at the problem speicification. MARCEL does not simulate the understanding process, but can be given "buggy" representations of the specification to work from.

Domain-Specific Learning: One way to simulate different students using different plans is to simply give different student models different plans. For instance, for dealing with repeately processing input, some students may have learned the "duplicate input" plan, others the "dummy inii' plan, and still others the "more data" plan. This assumes that the different ~~students somehow learned different plans (see below).~~

Progress Criteria - Relative Impasse Difficulty: Another way in which the programs in Figure 7 can be generated is based on different progress criteria. Note that all but the "missing loop" program of Figure 7, occur in the impasse/repair tree in Figure 8 ("duplicate input" third column and first row, "dummy init" second column and fifth row, "more data" first column and fourth row, "missing re-input" second column and first row, and "missing guard" second column and third row). Given a tree of possible impasse/repair episodes, each program in die tree can be obtained by (1) a proper setting of the relative impasse difficulty parameters (controlling backtracking and pushing on), and (2) a properly designed isomorphism test that overlooks certain types of impasses (stops before reaching a correct program at a leaf in the tree).

6 Concluding Remarks

This paper describes a cognitive model of student programmers. Parts of the model are implemented in a simulation program called MARCEL. This

CORRECT (DUPL. INPUT)
```
input
while not-sentinel
  do begin
        calculate
        output
        input
     end
```

BUGGY (MISSING RE-INPUT)
```
input
while not-sentinel
  do begin
        calculate
        output
     end
```

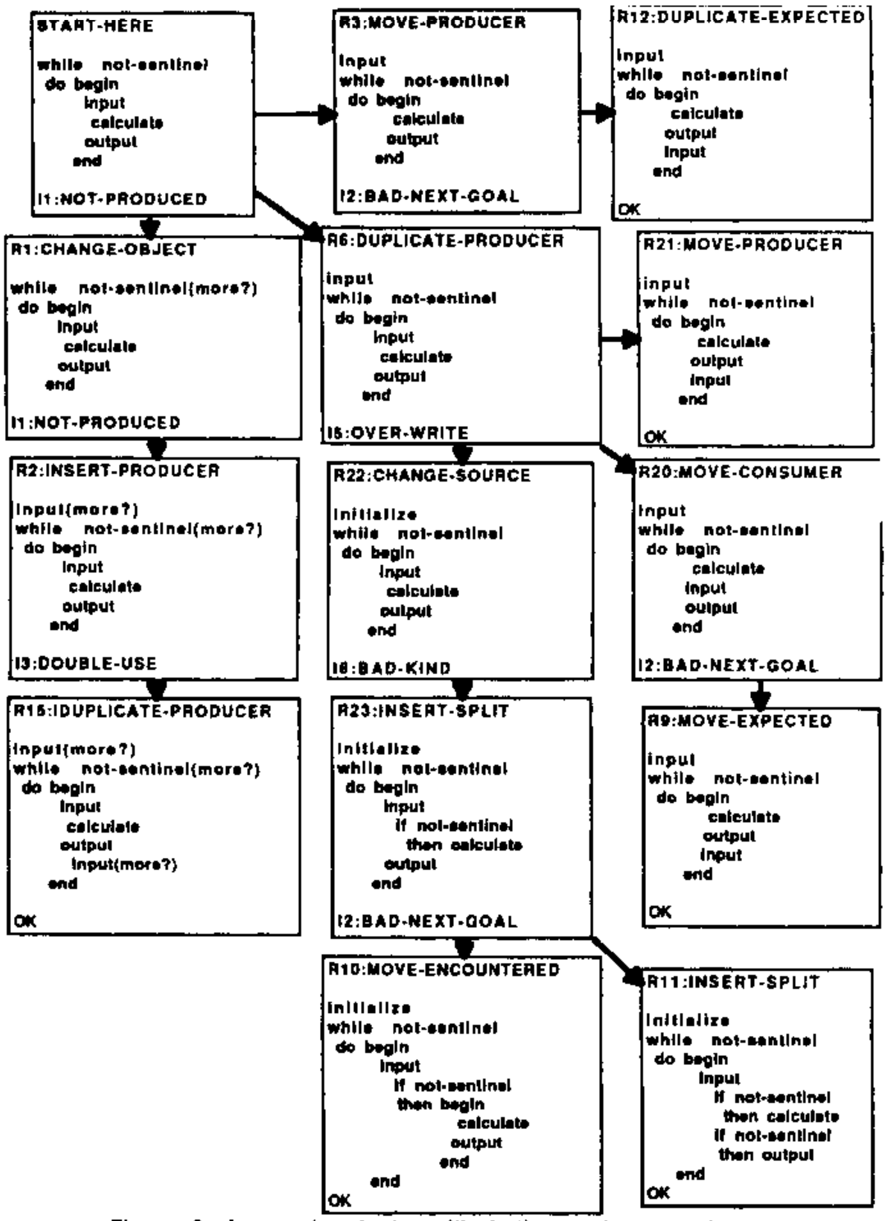
CORRECT (DUMMY INIT)
```
init-to-not-sentinel
while not-sentinel
  do begin
        input
         if not-sentinel
          then begin
                  calculate
                  output
               end
     end
```

BUGGY (MISSING GUARD)
```
init-to-not-sentinel
while not-sentinel
  do begin
        input
         calculate
        output
     end
```

CORRECT (MORE-DATA)
```
input(more-data)
while not-sent.(more-data)
  do begin
        input
         calculate
        output
         input(more-data)
     end
```

BUGGY (MISSING LOOP)
```
input
calculate
output
```

Figure 7: Example variability for an "alternate" type task.



```
START-HERE

while   not-sentinel
 do begin
       input
        calculate
       output
     end

I1:NOT-PRODUCED
```

```
R3:MOVE-PRODUCER

input
while   not-sentinel
 do begin
        calculate
       output
     end

I2:BAD-NEXT-GOAL
```

```
R12:DUPLICATE-EXPECTED

input
while   not-sentinel
 do begin
        calculate
       output
       input
     end

OK
```

```
R1:CHANGE-OBJECT

while   not-sentinel(more?)
 do begin
       input
        calculate
       output
     end

I1:NOT-PRODUCED
```

```
R6:DUPLICATE-PRODUCER

input
while   not-sentinel
 do begin
       input
        calculate
       output
     end

I5:OVER-WRITE
```

```
R21:MOVE-PRODUCER

input
while   not-sentinel
 do begin
        calculate
       output
       input
     end

OK
```

```
R2:INSERT-PRODUCER

input(more?)
while   not-sentinel(more?)
 do begin
       input
        calculate
       output
     end

I3:DOUBLE-USE
```

```
R22:CHANGE-SOURCE

initialize
while   not-sentinel
 do begin
       input
        calculate
       output
     end

I6:BAD-KIND
```

```
R20:MOVE-CONSUMER

input
while   not-sentinel
 do begin
        calculate
       input
       output
     end

I2:BAD-NEXT-GOAL
```

```
R15:IDUPLICATE-PRODUCER

input(more?)
while   not-sentinel(more?)
 do begin
       input
        calculate
       output
        input(more?)
     end

OK
```

```
R23:INSERT-SPLIT

initialize
while   not-sentinel
 do begin
       input
        if not-sentinel
          then calculate
       output
     end

I2:BAD-NEXT-GOAL
```

```
R9:MOVE-EXPECTED

input
while   not-sentinel
 do begin
        calculate
       output
       input
     end

OK
```

```
R10:MOVE-ENCOUNTERED

initialize
while   not-sentinel
 do begin
       input
        if not-sentinel
         then begin
                 calculate
                output
              end
     end
OK
```

```
R11:INSERT-SPLIT

initialize
while   not-sentinel
 do begin
       input
        if not-sentinel
         then calculate
        if not-sentinel
         then output
     end
OK
```

Figure 8: Impasse/repair tree illustrating various repairs.

research is important theoretically because it explores the use of a GTD impasse/repair problem solving architecture in a new domain, and important practically because of its educational implications for programming instruction.

Although the preliminary model for simulating variability is computational, it is still quite descriptive in nature (e.g., it does not provide an answer to why different impasses are more or less difficult for different student, where dispositions come from, by what process students understand the task specification, how students learn different plans, and what the space of incomplete isomorphism tests is, etc.). Another limitations is that the cognitive plausibility claim is supported by using qualitative protocol evidence. Stronger evidence (a future goal) might require that the model generate "all and only" the observed bugs that students make with limits on the tailorability of the model [BV80], or accurately predict the relative difficulty and bug-proneness of specific tasks [HW84], or predict error rates and reaction times for tasks [CMN 80], or account in detail for a larger percentage of the protocol data [NS74].

References

[AFS841 J.R. Anderson, R. Farrell, R. Sauers. Learning to program in LISP. *Cog. Sci.,* 8:87-129, 1984.

[ABR85] J.R. Anderson, C.F. Boyle, and B. J. Reiser. Intelligent tutoring systems. *Science,* 228(4698):456^62, 1985.

[BS85J J. Bonar and E. Soloway. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human Computer Interactions,* 1(2): 133-161, 1985. (Reprinted in [SS89]).

[BV80] J. S. Brown and K. VanLehn. Repair theory: a generative theory of bugs in procedural skills. *Cognitive Science,* 4:379-426, 1980.

(CMN83] S.K. Card, T.P. Moran, and A. Newell. *The psychology of human computer interactions.* Hillsdale, NJ: Lawrence Erlbaum Associates, Inc..

[I lam86] K J. Hammond. *Case-Based Planning. An Integrated Theory of Planning, Learning, and Memory.* PhD Diss. *CS* TR 488, Yale, New Haven CT, Oct 1986.

[HW85] J. Hiebcrt and D. Wearne. A model of students decimal computation procedures. *Cognition and Instruction.* 2(3&4), pp. 175-205. 1985.

[JSCD83] W.L. Johnson, E. Soloway, B. Cutler, and S. Draper. *BUG CATALOGUE: I.* CS TR 286, Yale, New Haven CT, Oct. 1983.

[JS85] W.L. Johnson and E. Soloway. PROUST. *Byte Magai.* 10(4) 179-192, 1985.

[KG85] W. Kintsch and J. G. Greeno. Understanding and solving arithmetic word problems. *Psychology Review.* 92, 109-129, 1985.

[MPS87] S.P. Marshall, C.A. Pribe, and J.D. Smith. *Schema knowledge structures for representing arithmetic story problems.* TR, Dept of Psych., SDSU, 87.

[MRRH81] K.B. McKeithen, J.S. Reitman, H.H. Rueter, and *S.C.* H rtle Knowledge organization and skill differences, in computer programmers. *Cog.Psych.* 13, 307-325, 1981.

[NS72] A. Newell and H.A. Simon. *Human Problem Solving.* Prentice Hall.NJ, 1972.

[P86] R. Pea. Language independent conceptual "bugs" in novice programs. *Journal of Educational Computing Research.* 2(1):25-36,1986.

[Sh76] B. Shneiderman. Exploratory experiments in programmer behavior. *Int. J. Comput. Inf. Sci..* 5, 2, 123-143, 1976.

lSi88] R. Simmons. A theory of debugging plans and interpretations. In *Proceedings ofAAAI-88.* Saint Paul, MN. pp 94-99, Aug. 21-26, 1988.

[SE84] E. Soloway, and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering.* 5:595-609, 1984.

[SS89] E. Soloway and J.C. Spohrer, Editors. *Studying the Novice Programmer.* Lawrence Erlbaum Publishers, Hillsdale NJ, 1989.

[SPL*85] J.C. Spohrer, E.P ope, M. Lipman, W. Sack, S. Freiman, D. Littman, W.LJohnson, and E. Soloway. *BUG CATALOGUE: II,III, IV. CS* TR 386, Yale New Haven *CT,* May 1985.

[SSP85] J.C. Spohrer, E. Soloway, and E. Pope. A goal/plan analysis of buggy Pascal programs. *Hum-Corn. Inter.,* 1(2): 163-207, 1985. (in [SS89]).

[SS86] J.C. Spohrer and E.Soloway. Novice mistakes: are the folk wisdoms correct? *Comm. of the ACM,* 29(7):624-632, July 1986.(Reprinted in [SS89]).

[S89] J.C. Spohrer. MARCEL: A GTD impasse/repair model of student program generation and individual differences. Ph.D. Diss, (in prep.).

[Su75] G. Sussman. *A Computer Model of Skill Acquisition.* Elsevier: NY, 1975.