An Approach to Constructing Student Models: Status Report for the Programming Domain

James C. Spohrer

Yale University
Computer Science Department
(As of Sept. '89, Apple Computer)

ABSTRACT

Student models are important for guiding the development of instructional systems. An approach to constructing student models is reviewed. The approach advocates constructing student models in two steps: (1) develop a descriptive theory of correct and buggy student responses, then (2) develop a process theory of the way students actually generate those responses. The approach has been used in the domain of introductory programming. A status report is provided: (1) Goal-And-Plan (GAP) trees have been developed to describe student program variations, and (2) a Generate-Test-and-Debug (GTD) impasse/repair architecture has been developed to model the process of student program generation.

INTRODUCTION: MOTIVATIONS AND GOALS

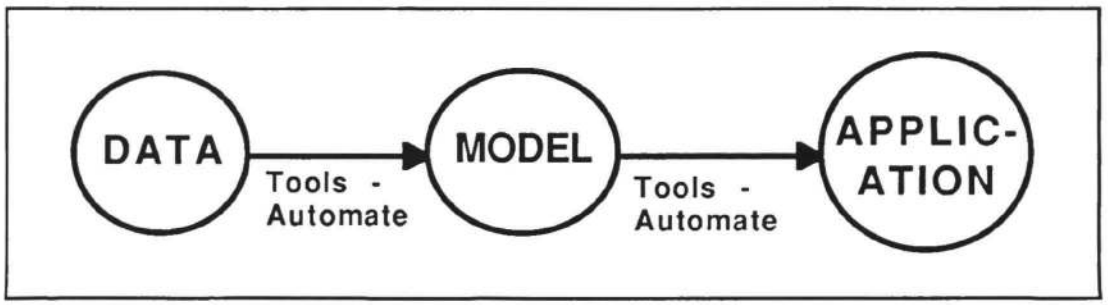
The long-term goal of the research reported here is to build computer-based applications that help students learn design skills (i.e., planning, constructing, evaluating, and debugging artifacts such as computer programs). Therefore, one purpose of this paper is to present a brief overview of a long-term research plan for building instructional systems. A key part of these systems is a student model. This paper provides a status report on the development of a student model for the domain of introductory programming.

Previously [SSP85], we have advocated an approach to constructing student models that decomposes the problem into two steps: (1) develop a descriptive theory of the alternative correct and buggy responses students generate, and (2) develop a process theory of the way students actually generate those responses. In [SSP85], we present a descriptive theory of correct and buggy student generated Pascal programs that is based on the notion of a Goal-And-Plan (GAP) tree. However, a descriptive theory provides only a systematic enumeration of what the alternatives are, and does not address the question of how the alternatives originate in the first place. In this paper, we present a model of the way different students write alternative programs. The problem solving behavior of students writing programs will be described in terms of a generate-test-and-debug (GTD) problem solving architecture in which impasse/repair knowledge plays a key role.

The paper will: (1) describe a long-term research plan for building instructional systems, (2) illustrate the important role of student models in the plan by arguing that limitations in existing instructional systems can be traced back to weaknesses in those systems' student models, (3) describe some alternative programs that real students generate, and finally (4) explain the process by which our student model generates these programs.

WHAT'S DMATA?

DMATA is an acronym for a long-term research plan aimed at developing and studying the development of computer-based applications, especially instructional systems.



The DMATA plan advocates using Data from empirical studies to first construct Models of people performing problem solving in some domain, and then to use the models to build Applications that support human problem solving. However, we are also interested in developing Tools that help researchers construct models from data and that help researchers build applications from models. Eventually, we would like to Automate the data-model-application development path.

Some application builders would argue that it is unnecessary to construct separate models of human problem solving before building applications. In fact, constructing a separate model is no guarantee that a successful applications can then be built. Nevertheless, weaknesses in existing instructional systems for the programming domain can be traced back to limitations in their underlying student models, as described in the next section.

COGNITIVE REVERSE-ENGINEERING: APPLICATIONS -> MODELS

In general, computer-based applications for design domains incorporate, either explicitly or implicitly, a model of the way humans solve tasks in those domains. We term the process of extracting a model from an application "cognitive reverse-engineering" (see [CK89] for a related concept). For instance, the student model underlying the PROUST system [JS85] might be termed an *enumeration student model*, because to find bugs in student programs PROUST requires a large knowledge-base that enumerates alternative correct solutions (i.e., plan library) and incorrect solutions (i.e., bug library) for a programming task. Alternatively, the student model underlying the GREATERP system [ABR85] might be termed a *restriction student model*, because students are forced to follow in the "foot-steps" of an ideal student and tutorial advice is provided as soon as a student deviates from the restricted ideal solution path.

In sum, PROUST and GREATERP deal with the variability problem -- the problem of coping with alternative correct and buggy programs -- by an enumeration and restriction approach, respectively. Unfortunately, enumerating plans and bugs is very time consuming and can never be totally complete. However, restricting the possible solutions does not give students a chance to acquire skills for exploring and evaluating alternative designs. Thus, neither the enumeration or restriction student models are entirely satisfactory. For design domains, we argue that computational *generative models* [BV80] are the preferred type of model to use to guide the development of computer-based applications because they parsimoniously account for variability.

DESCRIPTIVE THEORY OF PROGRAMS: GAP TREES

Because real students generate so many different correct and buggy programs when asked to solve even simple introductory programming tasks, some way must be found to systematically organize and describe all the variations before attempting to build a generative student model. For instance, consider a programming task that must process a series of input values, stopping when a sentinel value is entered. Figure 1 shows some alternative pseudo-code solutions. The solutions are based on actual Pascal programs.

CORRECT (DUPLICATE INPUT)

```
input
while not-sentinel
 do begin
      calculate
      output
      input
    end
```

CORRECT (DUMMY INIT)

```
init-to-not-sentinel
while not-sentinel
 do begin
      input
       if not-sentinel
        then begin
               calculate
               output
             end
   end
```

CORRECT (MORE-DATA)

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

BUGGY (MISSING RE-INPUT1)

```
input
while not-sentinel
 do begin
      calculate
      output
    end
```

BUGGY (MISSING GUARD)

```
init-to-not-sentinel
while not-sentinel
 do begin
      input
      calculate
      output
    end
```

BUGGY (MISSING RE-INPUT2)

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

Figure 1: Example variability for an "alternate" type task.

These examples illustrate a few of the many alternative correct and buggy programs we have catalogued in the GAP tree for this particular type of programming task [SPL*85]. The alternatives illustrate three correct plans for the sentinel-controlled-input goal, as well as a single buggy version of each plan. Because a programming task is composed of several goals, and each goal has several plans, and each plan may have several bugs, a GAP tree with bugs indexed off plans provides a concise description of the programs that students generate. In the next section, we will present a generative student model that can account for some of the alternative programs that a GAP tree only enumerates.

PROCESS THEORY OF PROGRAMMERS: GTD IMPASSE/REPAIR MODEL

The development of the generative model (i.e., a process theory) could only occur after a systematic organization of the the alternative correct and buggy programs had been developed (i.e. a descriptive theory). In addition, thinking-aloud protocol data -- complete problem solving *behavior traces* of the verbally reported planning, implementation, and debugging steps involved in writing a program -- had to be collected and analyzed. Based on the previously developed descriptive theory and the additional thinking-aloud protocol

data, a process theory has been developed that employs a *generate-test-and-debug (GTD) impasse/repair problem solving architecture* (see [Su75], [Ham86], [Si88], and also [BV80] [BS85] [NS72]). The three problem solving phases of the architecture are:

Generate Phase: 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* (i.e., "commonsense" plans) into code. Non-programming knowledge (see [BS85] for a related concept) is a key part of the model and corresponds intuitively to knowledge that would allow a student to easily do a *hand calculation*. 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.

Test Phase: 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 a simulation of an internal representation (i.e., mental model) of the expected solution or (2) checked for specific commonly occurring bugs. Impasses in MARCEL might more appropriately be called expectations violations, because they are unlike the impasses caused by lack of domain knowledge as in [BV80] and [BS85]. However, because repairing the impasses in all these model can lead to bugs, we prefer the term impasse to expectation violation.

Debug Phase: During the debug phase, impasses are fixed using one of small set of *repairs*. Associated with each impasse are between two and six repairs that might be applied to fix the impasse. One way variability can arise in the model is when different repairs are used to fix the same impasse.

A simulation program, called MARCEL, implements the model, and can be used to simulate students writing both correct and buggy Pascal programs [S89]. In the remainder of this paper, we will focus on accounting for the generation of alternative correct and buggy programs in terms of impasse/repair knowledge.

PROGRAM VARIABILITY IN TERMS OF IMPASSE/REPAIR TREES

In this section, a small set of impasses and repairs which students appear to use will be described. As in the subtraction domain [BV80], we will see that a small set of impasses and repairs can give rise to a great deal of variability. However, because the amount of variability in the programming domain (a design domain) is enormous compared to variability in the subtraction domain (a procedural skill domain), we do not yet make specific quantitative claims about the coverage of our model, but instead support the model with short quotes or *snippets* from thinking aloud protocol data.

To understand the impasses and repairs used in the model one should begin by viewing a program as a consumer-producer system in which certain goals *consume* some objects and *produce* others (e.g., the calculation goal *consumes* the input objects and *produces* the output objects). Most of the impasses and repairs used in the model are domain-independent, and can be applied to a variety of consumer-producer systems (e.g., biological systems, economic systems, etc.). For instance, a coal company *produces* coal that is *consumed* by an electric company to *produce* smoke and electricity. If the coal contains impurities (i.e., BAD-KIND) that lead to too much pollution, then a number of repairs can tried: separate the impurity before the coal is burned (i.e., INSERT-SPLIT), or scrub the smoke (i.e., INSERT-CONSUMER).

In programs (or any other consumer-producer system) six types of impasses can occur:

I1. NOT-PRODUCED: An object is consumed before it has a value (before produced).
"If I just start with a WHILE-DO statement, the variable is not gonna have any value yet." (AAS7.15).

I2. BAD-NEXT-GOAL: The next goal is not the expected goal.
"I'm thinking what will happen if they put in less than zero [invalid], *and I still try to print out the answer* [on invalid should stop, not output]." *(AVM5.126).*

I3. DOUBLE-USE: A goal uses (consumes) the same value twice.
"...I had to put it in a place so it wouldn't be affected [by the same value again]..." *(JBH7.232).*

I4. BAD-SOURCE: An object's value is not from the user.
"I'd have to rewrite the first line instead of... automatically [dummy initialization]... *assigning to the value... I could prompt* [get value from user]" *(AVM6.415).*

I5. OVER-WRITE: A value is destroyed before it is used (consumed by a goal).
"Every time it goes through its gonna see sum gets zero...so maybe I'll put that outside the whole loop [sum initialization over-writes the update]." *(JBH7.111).*

I6. BAD-KIND: A value is inappropriate for the case.
"Everything is gonna have to deal with... wait it did say something about impossible values [the calculation almost got run on invalid values]." *(JBH5.62).*

The protocol data contain examples of impasse/repair episodes on average once every 2.5 minutes. Protocol snippets guided the development of the impasse/repair component of the model, and provide support for the cognitive plausibility of the model.

During the debug phase, a repair is selected for an impasse. All of the repairs involve simple 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 (see [S89] for details). For example, consider the small set of repairs that apply to the impasse NOT-PRODUCED:

Repairs to I1:NOT-PRODUCED: When a consumer goal tries to consume an object whose value has not yet been produced, try one of the following repairs:

R1:CHANGE-OBJECT - If the consumer of the not-produced object is a split, then try changing the object to an auxiliary object.
R2:INSERT-PRODUCER - If the producer is not in the program yet, then insert the producer directly before the consumer.
R3:MOVE-PRODUCER If the producer is somewhere else in the program, then move the producer directly before the consumer.
R4:MOVE-CONSUMER If the producer is somewhere else in the program, then move the consumer to directly after the producer.
R5:DELETE-CONSUMER - Delete the offending consumer.
R6:DUPLICATE-PRODUCER If the producer is somewhere else in the program, then make a duplicate of the producer and insert it before the consumer.

Like the impasses, all of the repairs used in the model are derived from protocol snippets, and supported by additional snippets (see example below).

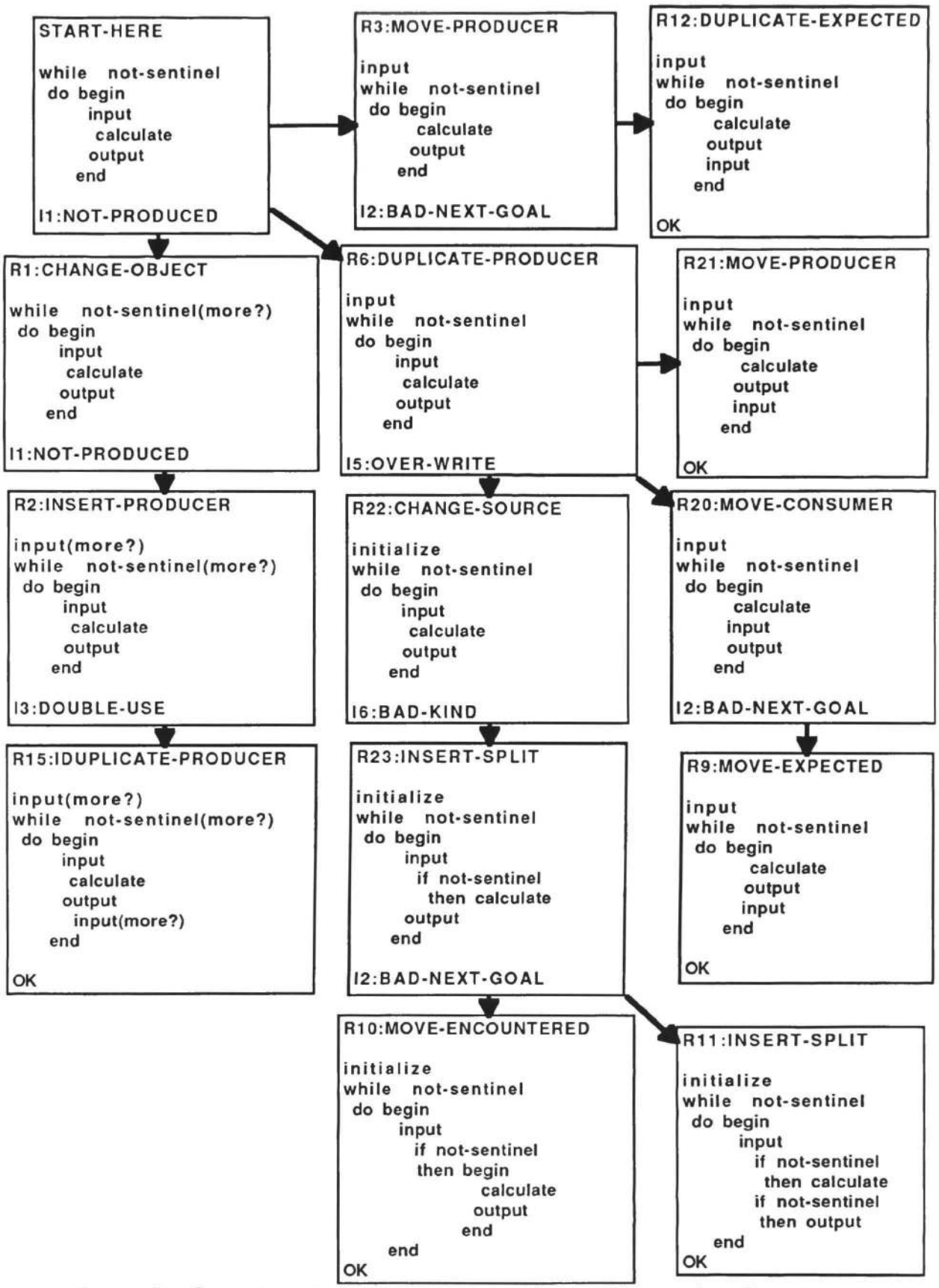


Figure 2: Correct and buggy programs in an impasse/repair tree.

Often when students tried to write a program to process a series of input values, they would start by saying something like -- what I need to do is a standard input-calculate-output, but with a loop wrapped around it. This would result in a program whose pseudo-code structure was like that in the upper left corner of Figure 2 (i.e., START-HERE). Since the "not-sentinel" test in the WHILE is testing a variable that has not yet been produced (the input is inside the loop), some students detect a NOT-PRODUCED impasse (e.g., *If I just start out with WHILE-DO statement, the variable is not gonna have any value yet." (AAS7.15).)* After detecting the impasse, some student may decide to repair the impasse by testing a different variable instead of the one which is input inside the loop, so they use a CHANGE-OBJECT repair (e.g.,*"I could create another variable and just say WHILE..."* (AVM6.59).) The CHANGE-OBJECT repair (repair R1 above), after being applied would result in a program whose pseudo-code structure was like that shown in the first box in the second row of Figure 2. Other students might decide to move the input from inside the loop to above the loop, so they use a MOVE-PRODUCER repair (e.g., *"that's gonna be outside of the loop... because its got to prompt before the loop." (AAS7.15)*). The MOVE-PRODUCER repair (repair R3 above) would result in a program like that shown in the second box in the first row in Figure 2.

Sometimes students add new bugs to a program when they are trying to fix a bug [GO86], and in a related phenomena repairing an impasse often leads to a new impasse. Because impasses can give rise to repairs that can give rise to new impasses, an impasse/repair tree is a convenient representational device for describing a large set of programs that students might conceivably generate. For instance, all of the correct and buggy programs of Figure 1 occur in the impasse/repair tree shown in Figure 2 ("duplicate input" third column and first row, "dummy init" second column and fifth row, "more data" first column and fourth row, "missing re-input1" second column and first row, "missing guard" second column and third row, and "missing re-input2" first column and third row). Three questions that remain are: How do students generate the initial program hypothesis? Why do some students detect an impasse, while others do not? Why do different students select different repairs for the same impasse? (see [S89] for some preliminary answers).

CONCLUDING REMARKS: MODEL -> APPLICATION

We have claimed that computational generative student models are to be preferred over enumeration or restriction student models when building computer-based applications to help students perform design activities and acquire design skills. Admittedly, this claim lacks convincing support until we complete the next stage of our research effort and build such an application. Nevertheless, to the extent that design domains are characterized by variability, and to the extent that generative models capture important aspects of that variability in a parsimonious model, we feel we are on the right track. If students can be explicitly and effectively taught how to detect the six impasses and apply the necessary repairs, then a programming environment that supports exploring and evaluating alternative programs via impasse/repair trees should help the students exercise and develop important design skills.

Acknowledgements: Elliot Soloway provided support and direction for this research. This work was supported in part by National Science Foundation Grant MDR-88-96240. I would like to thank David Littman for his comments on drafts of this paper.

REFERENCES

[ABR85] J.R. Anderson, C.F. Boyle, and B. J. Reiser. Intelligent tutoring systems. *Science*, 228(4698):456-462, 1985.

[BS85] 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.

[CK89] J.M. Carroll and W.A. Kellogg. Artifacts as theory-nexus: Hermeneutics meets theory-based design. *IBM Research Paper*. Yorktown Heights, NY.

[GO86] Leo Gugerty and Gary M. Olson. *Comprehension differences in debugging by skilled and novice programmers*. In Empirical Studies of Programmers, Soloway and Iyengar (Eds). Ablex: Norwood, NJ. 1986.

[Ham86] 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.

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

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

[Si88] R. Simmons. A theory of debugging plans and interpretations. In *Proceedings of AAAI-88*. Saint Paul, MN. pp 94-99, Aug. 21-26, 1988.

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

[SPL*85] J.C. Spohrer, E.Pope, M. Lipman, W. Sack, S. Freiman, D. Littman, W.L.Johnson, 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.-Com. Inter.*, 1(2):163-207, 1985. (in [SS89]).

[S89] J.C. Spohrer. MARCEL: A GTD impasse/repair model of student program generation and individual differences. Ph.D. Diss. CS TR 687, Yale New Haven CT, 1989.

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