AUTOMATIC STUDENT MODELING AND BUG LIBRARY CONSTRUCTION USING THEORY REFINEMENT

by

PAUL THOMAS BAFFES, B.A., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1994

For Kathryn.

Acknowledgements

I will always be indebted to NASA, whose generous support introduced me to the field of artificial intelligence, inspired my first work in intelligent tutoring systems over 8 years ago, and paid for my Ph.D. research under the NASA Graduate Researchers Program, grant number NGT-50732. Perhaps one day they will give me the additional opportunity to explore space, which has been my childhood dream. I am also indebted to the many people who contributed to the ideas presented here, most notably my advisor Ray Mooney who provided a constant source of guidance and inspiration, Marilyn Murphy who provided me with test data, Hamilton Richards who trusted me with the students of his C^{++} class, and the members of my research committee: Bruce Porter, Ben Kuipers, Risto Miikkulainen and Derek Sleeman. Others who contributed both ideas and those all-important distractions that maintained my sanity include Brian West, John Zelle, Jeff Mahoney, Cindi Thompson, Tara Estlin, Dan Clancy, James Lester, Je-Nien Ernst and Mark Johnstone.

And of course, none of this would have been possible without the constant support of my family who have always believed in my abilities even when it seemed no one else did. This is especially true of my wife, Kathryn, who put up with a 30% drop in income and two and a half years of long distance commuting. For her, no praise is too high.

AUTOMATIC STUDENT MODELING AND BUG LIBRARY CONSTRUCTION USING THEORY REFINEMENT

Publication No. _____

Paul Thomas Baffes, Ph. D.

The University of Texas at Austin, 1994

Supervisor: Raymond J. Mooney

The history of computers in education can be characterized by a continuing effort to construct intelligent tutorial programs which can adapt to the individual needs of a student in a one-on-one setting. A critical component of these intelligent tutorials is a mechanism for modeling the conceptual state of the student so that the system is able to tailor its feedback to suit individual strengths and weaknesses. The primary contribution of this research is a new student modeling technique which can automatically capture novel student errors using only correct domain knowledge, and can automatically compile trends across multiple student models into bug libraries. This approach has been implemented as a computer program, ASSERT, using a machine learning technique called *theory refinement* which is a method for automatically revising a knowledge base to be consistent with a set of examples. Using a knowledge base that correctly defines a domain and examples of a student's behavior in that domain,

ASSERT models student errors by collecting any refinements to the correct knowledge base which are necessary to account for the student's behavior. The efficacy of the approach has been demonstrated by evaluating ASSERT using 100 students tested on a classification task covering concepts from an introductory course on the C^{++} programming language. Students who received feedback based on the models automatically generated by ASSERT performed significantly better on a post test than students who received simple reteaching.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

In the middle of this century when computers burst onto the scene and transformed our lives, educators were quick to recognize the computer's enormous potential as an instructional aid. Just as the wide availability of books forever changed the way we teach and learn, it was felt that cheap access to computation would herald a renaissance in education that would profoundly impact the classroom. With the current trends towards ever smaller, faster and more powerful computers, this vision seems more realistic now than ever before. In our lifetime, it is not unreasonable to expect that powerful portable computers will become a standard school supply, just as paper, pencil and books are today.

There are undoubtedly countless potential applications of computer technology to education, but one approach in particular has dominated research in the field to date. This is the view of the computer as an individualized, one-on-one tutor. The computer tutor model is an extremely rich domain, presenting problems that span a wide array of research interests from cognitive science to instructional design to issues involving human-computer communication such as graphical interface design and natural language generation and understanding. But perhaps the most compelling motivation for the approach is illustrated by studies like the one performed by Bloom [Bloom, 1984] which show that students receiving one-on-one tutoring consistently experience a performance improvement two standards of deviation above students receiving traditional lecture-style instruction. By capturing and mass producing such individualized attention through the use of computer technology,

researchers have hoped to provide the benefits of one-on-one tutoring on a scale which was never before possible.

Early efforts to use computers as educational tools resulted in a paradigm now generally referred to either as *computer-based training* (CBT) or *computer-aided instruction* (CAI). Such programs are used to automate the presentation of well-prepared material to a student. In a typical CAI program, this presentation takes on a form similar to a book, with the additional ability to present different sections of material or different levels of detail based upon choices made by the user. The construction of an effective CAI program requires the author to anticipate student reactions so that appropriate choices can be encoded in the program. Knowledge of the educational task is thus *external* to a CAI program; the author uses his or her expertise to predetermine the choices and explanations which will be seen by the student.

In the 1970s criticisms of the CAI approach began to emerge. Several researchers, most notably Carbonell [Carbonell, 1970a; Carbonell, 1970b] pointed out that CAI could not be truly responsive to individual student needs until a knowledge of the domain similar to that possessed by the educator was encoded *within* the system. In short, a reactive system needed to be able to draw conclusions based upon interactions with the student similar to those made by a teacher. Since Carbonell's work, multiple efforts to construct *intelligent* CAI (ICAI), also called *intelligent tutoring systems* (ITS), have produced a variety of new techniques. A complete survey of the various approaches is beyond the scope of this work; for an excellent introduction to the field the reader is referred to [Wenger, 1987]. In general, however, research efforts have tended to focus on a few components seen as critical to the design of an ITS. One of these is the method for representing the knowledge of the student. Such representations are generally referred to as *user models* or *student models*. The goal of student modeling is to produce a representation that accounts for the differences

between the correct knowledge of the domain and the behavior of the student. Ideally, a unique model is built for every student who interacts with the system, including capturing misconceptions specific to each student which are not preprogrammed into the tutor. Using the student model, an ITS can modify its feedback to suit specific strengths and weaknesses, enabling it to be truly adaptive to the individual.

Student modeling has a long and interesting history, dating back well into the early CAI days. The best method for constructing and using a student model is still the subject of much debate. Unfortunately, the difficulty of constructing and testing student models has discouraged many researchers from pursuing further investigations into the field. Even though a body of truly exceptional research has resulted in a particularly functional set of student modeling concepts, the practical task of incorporating these techniques into a functioning tutoring system has proved to be a major roadblock. This is yet another instance of the infamous *knowledge-acquisition bottleneck*; the fruits of student modeling research are simply too labor-intensive to use.

Coincidently, recent machine learning methods have been developed which can be used to automate the techniques developed for student modeling. It is this confluence of student modeling and machine learning which is the subject of this research. The major contribution of this work is the ASSERT algorithm (Acquiring Stereotypical Student Errors by Refining Theories), which collects the ideas developed over the last two decades of student modeling research under one general machine learning framework for automatic student modeling in concept learning domains. With ASSERT, student performance increases significantly; in fact, by nearly one-and-one-half letter grades (for example, a typical “C” performance on a test would increase to a “B^+” performance on a post-test after exposure to an ASSERT tutorial). To understand how this is possible, one must have a basic familiarity with the current state-

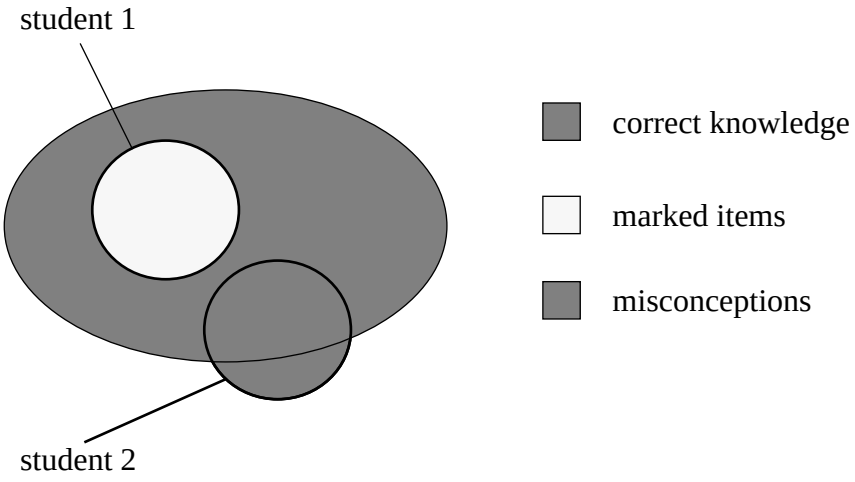


FIGURE 1 Example of an overlay model. Misconceptions occurring outside of the correct knowledge, such as are shown for student 2, cannot be modeled.

of-the-art in student modeling. Explaining ASSERT, then, begins with a history of the ideas developed for student modeling.

1.1 A Brief History of Student Modeling

The application of artificial intelligence to student modeling has yielded a steady progression of techniques for collecting information both about what a student knows and does not know, and about explaining the sources of misconceptions. The earliest AI-based student models, embodied in systems such as SCHOLAR [Carbonell, 1970a], WEST [Burton and Brown, 1976] and WUSOR [Carr and Goldstein, 1977b], used a form of modeling which is now generally referred to as *overlay modeling*. An overlay model relies on the assumption that a student's knowledge is always a subset of the correct domain knowledge. As the student performs actions which illustrate that he or she understands particular elements of the domain knowledge, these are marked in the overlay model. More sophisticated overlay models can express a range of values indicating the extent to which the system believes a student understands a given topic. Typically, the unmarked elements of the model are used to

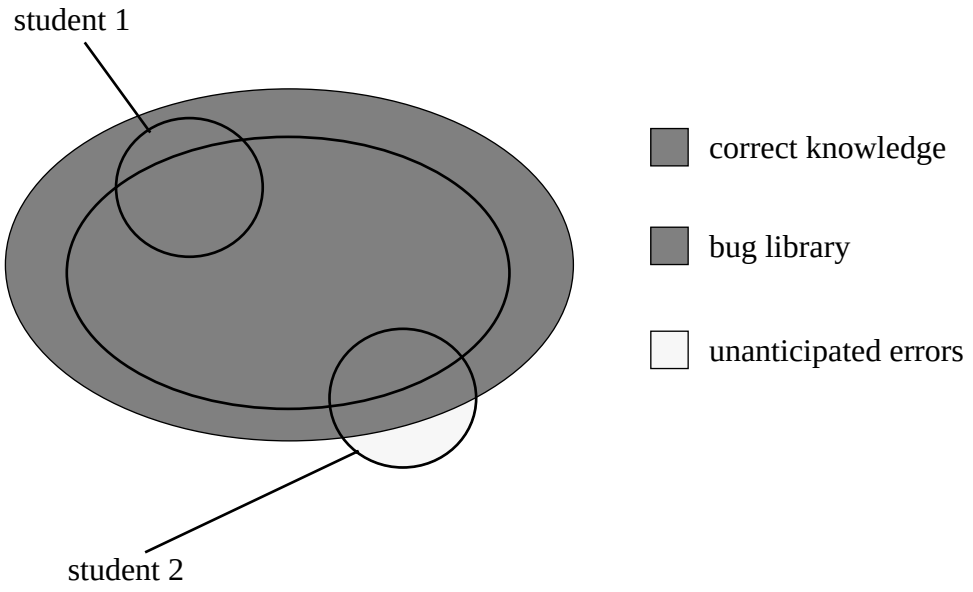


FIGURE 2 Example of bug library. Misconceptions are correctly modeled only if already present in the bug library.

focus tutoring on problem areas for the student, or to ensure full coverage of the domain. A diagram of an overlay model is shown in Figure 1.

While simple to implement, this method is incapable of capturing misconceptions, also called *bugs*, that represent faulty student knowledge. Said another way, overlay models can only capture the notion of a student's lack of knowledge; they cannot be used to model the student who knows of a topic but misunderstands it. To address this shortcoming, other researchers [Brown and Burton, 1978; Burton, 1982; Brown and VanLehn, 1980; Sleeman and Smith, 1981] focused on constructing databases of student misconceptions typically termed *bug libraries*. With a bug library, models are built by matching student behavior against a catalog of expected bugs which are preconstructed by hand through an analysis of student errors. Figure 2 depicts bug libraries in general, illustrating how they expand the space of discrepancies which the modeler can address. Though an important extension to the notion of an overlay, two problems remain with the bug-library approach. First, the construc-

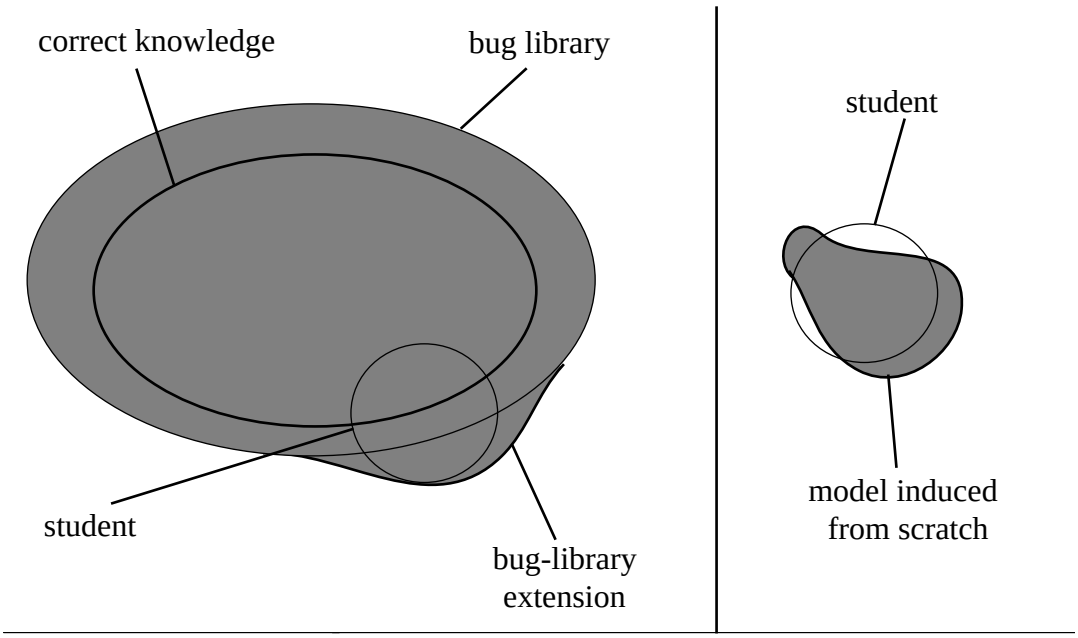


FIGURE 3 Dynamic modeling of bugs. The left half of the diagram depicts bug-library extension, the right half shows induction.

tion of such catalogs is a difficult and time-consuming task. Second, even if great care is taken the resulting library may still fail to cover a wide enough range of behaviors to function successfully. As with overlay models, the static nature of bug libraries renders them incapable of modeling unanticipated student behaviors.

A third approach to student modeling, shown in Figure 3, attempts to overcome this limitation either by patching the bug library dynamically [Sleeman et al., 1990] or by modeling student misconceptions from scratch [Langley and Ohlsson, 1984; Langley et al., 1984; Ohlsson and Langley, 1985] using a machine learning technique called *induction*. In both methods, novel misconceptions are modeled by constructing new buggy information dynamically. Patching methods use an analytical approach based on an underlying mechanism of how misconceptions can be formed to generate candidate extensions for the library. These candidates are then presented to the author of the tutoring system to determine which new bugs, if any, should be

kept. Induction, by contrast, is an empirical method used to fashion unique elements of the student model from examples of the student's behavior. Starting with no preconceived notions of a student's behavior, the system observes how the student solves problems and builds a complete model for the student, including both correct and buggy knowledge.

The advantage of the patching methods is that knowledge of the domain can be brought to bear on the construction of new bugs. The disadvantage is that the system must still rely upon the author or upon domain-dependent hand-built filters to determine which bugs to add to the library. Induction has the opposite problem; while it is a domain-independent technique that can function without human guidance, induction typically requires a large number of examples to produce accurate results. More importantly, neither of these approaches takes advantage of any trends that occur across multiple students which is another important source of knowledge for constructing a bug library. To address these issues, one needs a different learning mechanism which works with fewer examples than induction, can make use of any preexisting knowledge about student behavior, and can use the results of previous modeling efforts to update the bug library and enhance the modeling process. These capabilities are precisely what ASSERT provides.

1.2 ASSERT as a Framework for Automatic Student Modeling

ASSERT automates the student modeling ideas presented in the previous section by using a relatively new machine learning technique called *theory refinement*. Theory refinement is a method for automatically revising a knowledge base to be consistent with a set of examples. Typically, the knowledge base is considered incorrect or incomplete, and the examples represent behavior which the knowledge base should be able to emulate. However, the refinement procedure itself is blind to whether or not the input knowledge base is "correct" in any absolute sense; the theory-refine-

ment process merely modifies the knowledge base until it is consistent with the examples. Thus, one can use theory refinement for diagnosis by inputting a correct knowledge base and examples of erroneous student actions, and theory refinement will produce the modifications necessary to cause the knowledge base to simulate the student.

Theory refinement gives ASSERT other advantages as well. A theory-refinement learner combines the power of both analytic and empirical learning methods in an integrated, domain-independent way. Theory refinement can take advantage of any preexisting knowledge provided to system and resort to induction when necessary to extend the model using examples of the student's behavior. With theory refinement as its centerpiece, ASSERT can model any misconception consistent within the primitives used to define the domain. ASSERT can bring as much knowledge to bear on the modeling process as the author of the tutoring system is willing to encode. ASSERT can even work with no input knowledge, resorting to induction as previous methods have done. And most importantly, ASSERT provides an extension to theory refinement that combines the results of multiple student models to *automatically* construct new bugs for the bug library, without the necessity of intervention on the part of the author. Thus ASSERT encompasses three important student modeling principles: (1) it can model the differences between the system's correct domain knowledge and the student's behavior (2) it can make use of a bug library to catch known misconceptions and (3) it can extend its capabilities by modeling novel student misconceptions and by automatically constructing new bug-library entries.

Unlike other student modeling efforts however, which focus on procedural tasks, ASSERT is designed for use in classification domains. For example, several previous student modeling efforts have focused on the domain of writing computer programs, whereas this research is tested using a classification task where students

are asked to judge the correctness of program segments. This tie to classification domains is largely due to the fact that the most mature theory-refinement algorithms developed thus far are designed for classification and is not a limitation of ASSERT per se. However, shifting the focus of modeling from a procedural to a concept learning emphasis is not unprecedented. Other researchers, most notably Gilmore and Self [Gilmore and Self, 1988], have recognized the potential of using machine learning for tutoring conceptual knowledge. Additionally, a recent trade journal survey of applications for computer-based training indicated that over 80 commercial products are currently available for constructing CBT applications [IDS, 1990] in which concept learning is a primary task. Concept learning also has a fairly well understood pedagogy [Dick and Carey, 1990] and effective techniques for responding to incorrect student classifications are already extant in the literature [Tennyson and Park, 1980]. Thus a general technique for modeling in concept domains has a wide applicability, a potential for commercial impact, and can be coupled with instructional techniques shown to be effective in the presentation of conceptual material [Tennyson, 1971].

1.3 To Model or not to Model

To place this work in its proper historical context, it is important to point out that neither the utility nor the necessity of student modeling as a component of an ITS is a universally accepted fact. Quite to the contrary, an interview of ten well-known ITS researchers which appears in the March 1993 issue of AI Communications came to the conclusion that "most of the researchers no longer believe in online student modelling." [Sandberg and Barnard, 1993]. The article went on to conclude that "instead of becoming more integrated, the field has become more diverged in the last few years. It appears that scientists in the field of education and technology no longer share a research paradigm."

And yet, personally I do not feel that this spells the end of student modeling, though surely research efforts in the area will drop off dramatically. The history of artificial intelligence reveals a recurring pattern, which seems to have found its way into the field of AI and education. After starting with a good deal of excitement (some might say "hype") a period of ground-breaking, proof-of-concept systems are constructed which highlight the great potential of the research. Inevitably, however, the problems are more difficult than they seem at first, and excitement gives way to despair and, eventually, to disillusionment and calls for a new direction. This phenomenon is compounded for AI and education by the recent merging of technologies which heretofore were difficult to integrate. Nowadays it is a relatively simple matter to combine video, text, sound, graphical interfaces, database technology and access to a world-wide network into a hypermedia presentation with impressive potential. It is only natural for educators to explore these technologies; indeed, we would be remiss as a community if we did not do so.

None of this, however, changes the intrinsic value of an effective student model, as long as one can define what "an effective student model" actually is. The simplest definition is one based on performance; using a student model should help improve a student's proficiency. Furthermore, it is important to point out that very little conclusive evidence either for or against the utility of student models has actually been collected. One of the principal results of this research, in fact, is the presentation of empirical evidence showing that the use of ASSERT student models to generate feedback leads to improved student performance.

In a wonderfully eloquent article [Self, 1990], John Self address the generally negative perception of student modeling and makes suggestions for future student modeling research. He also points out the practical objections typically leveled at student modeling techniques; namely, that methods for modeling are too difficult to

employ and have little impact on the learning process. Both of these concerns have been taken into account in the design of ASSERT.

Objection 1: Practical student models are too difficult to construct. The argument made here is that effective models, defined as ones which can have a discernibly positive impact on the learner, are simply too complex to build. This is an effective argument and the history of student modeling does support this claim since most bug libraries have required many man-hours to construct. ASSERT was designed in direct response to this problem, and its solution bears emphasizing even at the risk of being redundant. ASSERT requires *only* the correct knowledge of the domain to construct its models. All bug-library information, as well as any novel bug modeling, is performed completely automatically, without necessitating feedback from the author of the tutorial. ASSERT responds quickly, operating at execution speeds which are linear in the size of the input knowledge base as will be discussed in Chapter 3. For any intelligent tutor, the correct knowledge of the domain has to be encoded in any event, so there is no added expense to construct an ASSERT-style tutorial.

Objection 2: Student modeling doesn't work. Here the complaint is that even the most accurate, most autonomous modeling algorithm is useless because modeling in general does not contribute to significantly improved student performance. As indicated above, the scientific evidence supporting this position is thin at best; there are relatively few complete studies of the effects of student modeling [Sleeman, 1987; Nicolson, 1992]. Moreover, recent results [Nicolson, 1992] indicate a positive benefit from the use of student models. Furthermore, results presented in Chapter 6 support the claim that student models are useful, showing that students who received the benefits of ASSERT modeling *significantly* improved their performance over students who were not modeled. In a study involving the classification of C^{++} program segments, 100 students from an introductory C^{++} class were given a test followed by

feedback and a post-test. One group of students received feedback based on a model generated using their pre-test answers, whereas a control group received simple reteaching of the correct concepts. Though both groups were given the same amount and type of feedback, students who received targeted feedback based on their student models increased their performance on the post-test an average of 15 percentage points more than students in the control group.

The remaining chapters are organized as follows. Chapter 2 presents an overview of ASSERT. The next three chapters describe the details of the ASSERT algorithm: Chapter 3 covers the theory refinement algorithm used to simulate the student, Chapter 4 describes ASSERT's method for automatically constructing bug libraries to enhance its modeling capabilities and Chapter 5 describes how feedback is generated for the student. In Chapter 6, the empirical results are presented, and Chapters 7, 8 and 9 discuss related work, future work and conclusions.

1.4 Summary

ASSERT is a shell for constructing student modeling tutors which operate in concept learning domains. It is able to construct student models efficiently and automatically, catching both expected and novel student misconceptions. It is the first modeling system which can construct bug libraries automatically using the interactions of multiple students, without requiring input from the author, and integrate the results so as to improve future modeling efforts. In this sense, ASSERT is a self-improving student modeler which unifies the ideas of two decades of student modeling research under a single paradigm. Finally, ASSERT can be used to significantly improve student performance, as will be shown in the chapters which follow.

CHAPTER 2

Overview

As discussed in Chapter 1, ASSERT is a generic program for building tutoring systems using the idea of refinement-based modeling and refinement-based remediation. Internal to the system is a knowledge base, provided by the author of the tutoring system, which can be refined so as to simulate the behavior of the student. These refinements are then used as the basis for generating feedback. In essence, ASSERT views tutoring as a process of communicating knowledge to a student, where the contribution of the modeling subsystem is to pinpoint elements of the internal knowledge base to be communicated.

The purpose of this chapter is to provide an overview of the ASSERT algorithm. Before diving into the details, however, it is important to describe the philosophical approach taken by ASSERT. This will provide the right context for understanding the decisions made in designing the algorithm, and will help the reader to understand how ASSERT compares with other intelligent tutoring systems research.

2.1 Tutoring as a Dialog

To begin, the left half of Figure 4 shows what might be the simplest possible view of a tutoring system. At its most abstract level, a tutorial can be seen as a dialog between student and system. From a research perspective a great many details are left out of this diagram and different researchers have chosen to emphasize different parts of the dialog. The first design decision then is one of emphasis: ASSERT focuses on the question of how to construct a useful interpretation of the student's actions. This decision is depicted as a new component inserted into the diagram as shown in

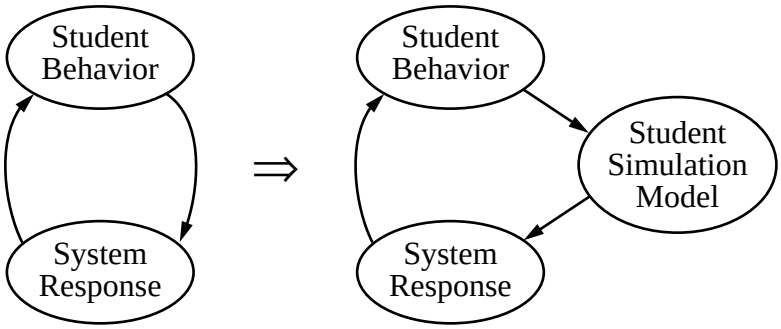


FIGURE 4 Abstract view of student-tutor interaction.

the right half of Figure 4. The implication is that ASSERT will translate the actions taken by the student into some internal format useful for determining a response.

There are other implications as well. First, it is assumed that building an interpretation of the student is a useful thing to do. Strong arguments in favor of this approach have already been stated in Chapter 1. Second, the name of the new component, a *student simulation model*, implies that the system contains a knowledge base that can be used to solve problems in the same context as the student must, and that this knowledge base can be modified to replicate the solutions furnished by the student. This is a fairly standard set of assumptions for modeling systems. However, ASSERT purports to be a generic system which relies only on correct domain knowledge. This means it will be essential to show how a generic system can construct misconceptions tailored to an individual student without the benefit of any predefined information about likely student errors.

2.2 The Student as a Classifier

Figure 5 depicts how ASSERT views student behavior. It is assumed that all actions taken by a student can be broken down to a set of *classification decisions*. That is, given a set of inputs, which for lack of a better term we shall call *problems*, the student will produce a set of *labeled examples* which classify each of the prob-

FIGURE 5 Student behavior diagram.

lems into a set of *categories*. Each problem consists of one or more *feature vectors* describing some aspect of the problem. The task of the student is to produce a label for each feature vector, selected from among some predetermined set of legal labels given to the student. The resulting set of labeled examples pairs each feature vector with the label selected by the student.

In its simplest form, a problem consists of a single feature vector presented to the student in a multiple-choice format, where the answers available to the student are taken from among a list of possible categories. Thus, for example, the classic diagnosis problem would present a patient's symptoms (the feature vector) and ask the student to select a diagnosis from a list of diseases (the label). This allows ASSERT to be used in concept learning domains which are common applications for tutoring systems. In fact, Chapter 6 describes a test using ASSERT in a concept learning domain.

2.3 Modeling by Theory Refinement

The labeled examples generated by the student are passed to the student simulation component of ASSERT depicted in Figure 6. The goal of this component is to produce a model of the student which accurately reproduces the behavior found in the labeled examples. As mentioned above, this component is generic and relies only upon correct domain knowledge to produce an individualized model of the student. ASSERT accomplishes this using a general machine learning technique called *theory refinement*. When given two inputs, one in the form of labeled examples and the other in the form of a propositional Horn-clause rule base, theory refinement will

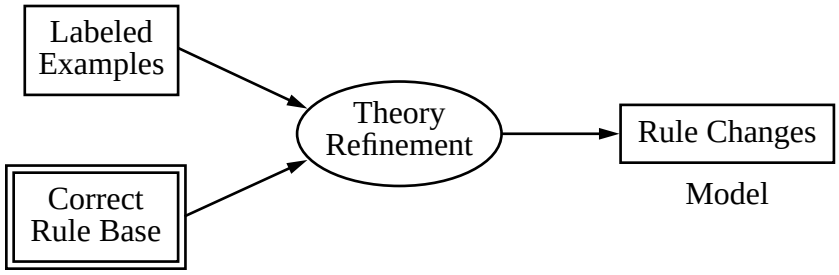


FIGURE 6 The student simulation model.

modify the rule base until it can reproduce the labels found in the examples. This is done by refining rules, removing rules, or adding new rules to the correct rule base. ASSERT uses the NEITHER theory refinement system, which is explained in more detail in Chapter 3.

The use of a propositional theory refinement algorithm for modeling carries with it the assumption that the author of the tutoring system will be able to provide a correct representation of the domain using propositional Horn-clauses. The resulting rule base is the "theory" which NEITHER refines to produce a model of the student. Note that this places a good deal of emphasis on how the correct rule base is constructed since it becomes the language through which ASSERT interprets the student's actions. If the correct rules are expressed at too high or too low a level of detail, the ability of the system to form accurate models will be diminished. Of course, this type of knowledge representation problem exists for all tutoring systems. However, ASSERT gains an advantage by purposely isolating the correct domain knowledge as a separate component: the author can easily change the focus of the tutor by altering the correct rule base. Moreover, if students possessing different levels of understanding will use the tutor, multiple rule bases can be written to give the system more flexibility.

Again, it must be emphasized that both the correct rule base and the model are functional, in that both can be used directly to solve the same problems given to the student. Specifically, by applying the rule changes of the model to the correct theory, one creates a rule base that replicates the answers furnished by the student. This goes to the core of the philosophy used to design ASSERT. Since the goal is to communicate its internal knowledge, there must be some mechanism for ASSERT to decide what is to be communicated. Theory refinement is this mechanism, acting like a search over the correct rule base, guided by the labels taken from the student. By refining the correct rule base until it mimics the student, NEITHER focuses attention on the applicable portions of the knowledge. Furthermore, note that ASSERT incorporates any assumptions implicit to NEITHER about how such a search for refinements should be conducted. Specifically, when faced with a choice, NEITHER will select the smallest refinement that accounts for the largest portion of the student's behavior.

Finally, it should be noted that theory refinement is purposely treated as a black box by ASSERT. Such modularity allows for incremental improvement as new refinement algorithms are developed. So, for instance, as first order logic refinement methods are enhanced, ASSERT can be updated accordingly, enabling it to address a wider range of applications than is currently possible using NEITHER's propositional Horn-clause representation. Or, if refinement algorithms are designed using entirely different knowledge representations, the approach taken by ASSERT can be applied to those domains as well.

2.4 Refinement-Based Remediation

The last component of ASSERT, the system response, is outlined in Figure 7. Using the refinements produced by NEITHER, ASSERT generates explanations and examples to reinforce the correct form of the rule or rules modified. The underlying

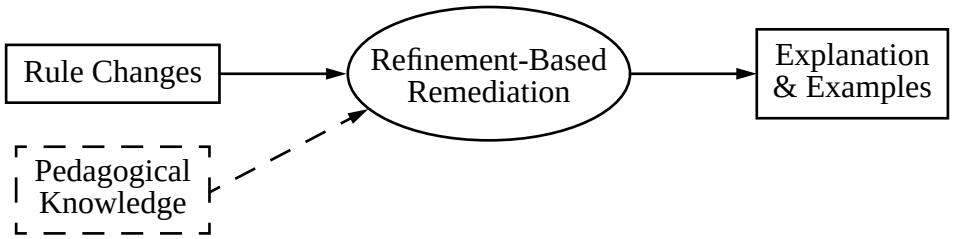


FIGURE 7 System response diagram.

approach, called *refinement-based remediation*, is again based upon the notion that the model highlights areas of the correct knowledge to be addressed. The output from NEITHER points out which rules were modified to account for the student's behavior, and the system remediates each refinement in turn.

Note, however, that rather than implementing any particular pedagogy ASSERT instead produces fundamental units of explanation called *units of remediation*. There are many possible ways to give feedback to the student, and determining which method is most appropriate is itself an open research question. Additionally, different domains are likely to require different instructional methods. In an attempt to support any number of these different approaches, ASSERT supplies the most elementary information required: an *explanation* with one or more *examples*. For each refinement detected by NEITHER, ASSERT provides two capabilities: the ability to explain a correct use of the rule which was changed, and the ability to generate an example which uses the rule. The designer of a tutoring system using ASSERT certainly has the option to generate multiple explanations or examples, to determine the circumstances when such feedback is given, and to decide whether the system or the student controls which explanations and examples are generated. By providing such explanation-example units, ASSERT supplies the raw materials for a variety of remediation techniques.

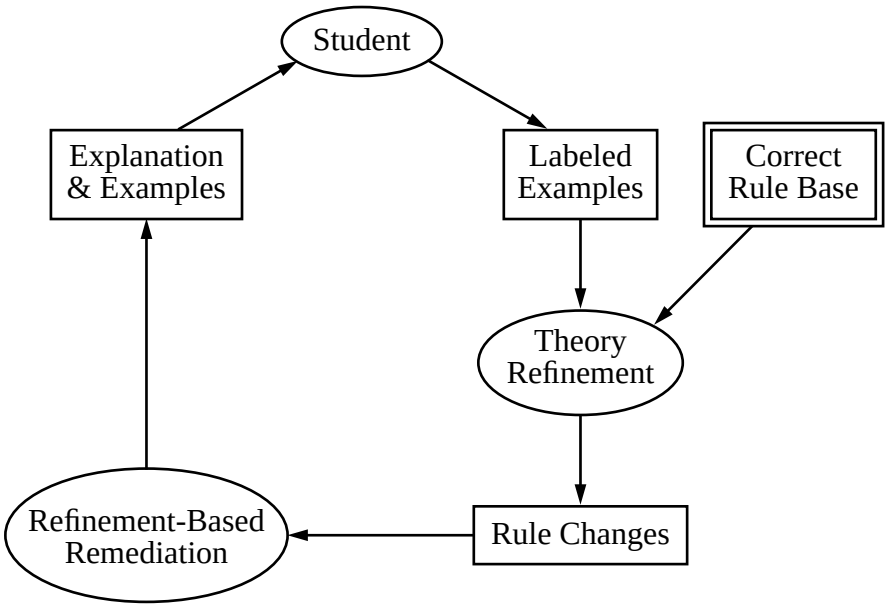


FIGURE 8 Basic design of the ASSERT algorithm.

Figure 8 combines the previous three diagrams, showing how the dialog flows between the student and the system. Problems given to the student are translated into labeled examples, which are passed to NEITHER. NEITHER uses these to refine a rule base representing correct knowledge of the domain to produce a modified rule base that simulates the student. The refinements are then used to generate explanations and examples for remediation which gets passed back to the student.

2.5 Extending ASSERT's Modeler

The previous sections have described the core of the ASSERT algorithm, showing how the flow of information between student and system can be implemented as a search for refinements that highlight the differences between how the system and the student evaluate the same set of problems. One of the benefits of using a theory refinement learner such as NEITHER is that it can modify the correct rule base to capture any labeling generated by the student. Thus one might wonder why any further

extensions to ASSERT are necessary. There are two basic reasons. First, it may be the case that the author of a tutoring system built with ASSERT has additional information about difficulties students are likely to have. This is often true for teachers possessing a great deal of experience in a given domain. The ability to incorporate such information can enhance the modeling process, especially when only a small amount of student behavior is observable. Second, after modeling multiple students ASSERT has accumulated a good deal of information about those elements of the correct rule base that are problematic. Such information can be vital to the author of the system, who may want to redesign the rules or enhance the remediation to emphasize the difficult concepts.

ASSERT can be extended to address both of these concerns. By taking advantage of the by-products of the dialog between student and system, one can *automatically* construct a *bug library* listing the various student errors detected by NEITHER. This library can then be used during modeling by searching it for any bugs applicable to the current student.

2.5.1 Building a Bug Library

The bug library represents a collection of data gleaned from the interaction of multiple students with the tutoring system. Referring back to Figure 8 on page 19, note that there are three by-products of the dialog between the student and the system. The students actions are translated into labeled examples from which NEITHER builds a set of rule changes to the correct theory. These are then handed to the remediation algorithm which selects explanations and examples to feed back to the student. Each of these by-products is essentially equivalent since the cycle between student and system translates one into another. The question is to select the form most appropriate for constructing the bug library.

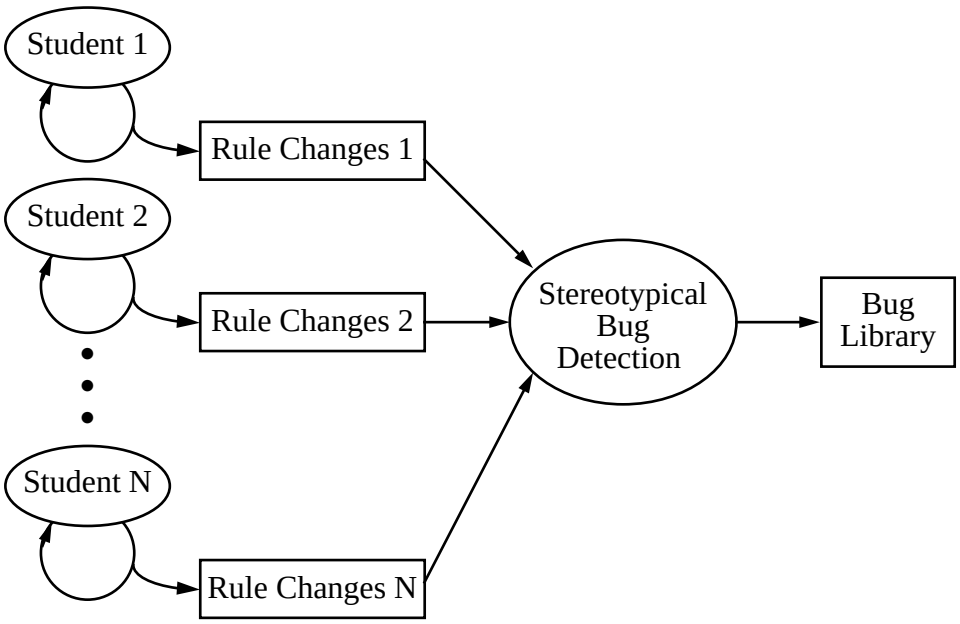


FIGURE 9 Bug library construction diagram.

ASSERT uses the rule changes, as depicted in Figure 9, for two reasons. First, the rule changes are most closely related to the type of input generated by the author of the tutoring system. Since a rule base must be supplied as input, expressing the bug library in terms of changes to that rule base is an effective way to communicate buggy information back to the author. Second, the rule change format is precisely what NEITHER uses to simulate the behavior of the student. A bug library built of rule changes is thus already in a form which can be incorporated into the modeling process. Rule changes, then, satisfy two important criteria: they can be easily communicated to the author and they can be used directly to enhance modeling.

Chapter 4 describes in detail how ASSERT automatically constructs and uses its bug library. Without getting into specifics, there are two general points which can be made about this process. First, after collecting the rule refinements from all the student models together, ASSERT will search for *subsets* of the refinements which can

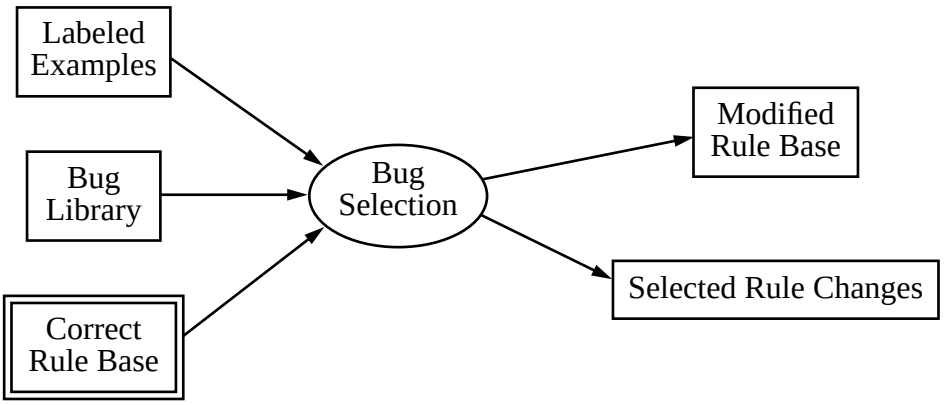


FIGURE 10 The bug selection module. Note that the correct rule base combined with the selected rule changes is equivalent to the modified rule base.

also be added to the bug library. These subsets capture information common to refinements which are similar but not identical. Second, all the refinements or subsets of refinements which end up in the bug library are ranked by their *stereotypicality*, which is essentially a measure of how frequently the refinement occurs. The final bug library is thus a list of refinements to the correct rule base ordered by how prevalent each refinement is in the student population.

2.5.2 Using the Bug Library for Modeling

Once the library is built the question becomes how to use its information to enhance the modeling process. Because ASSERT treats theory refinement as a separate module, the only way to influence modeling is to alter the inputs given to NEITHER. Recall from Figure 6 on page 16 that there are two inputs to theory refinement: labeled examples from the student and the correct rule base written by the author. Since the labeled examples come directly from the student, the only option left is to modify the rule base. Specifically, the input rules must incorporate the elements of the bug library that are relevant to the current student.

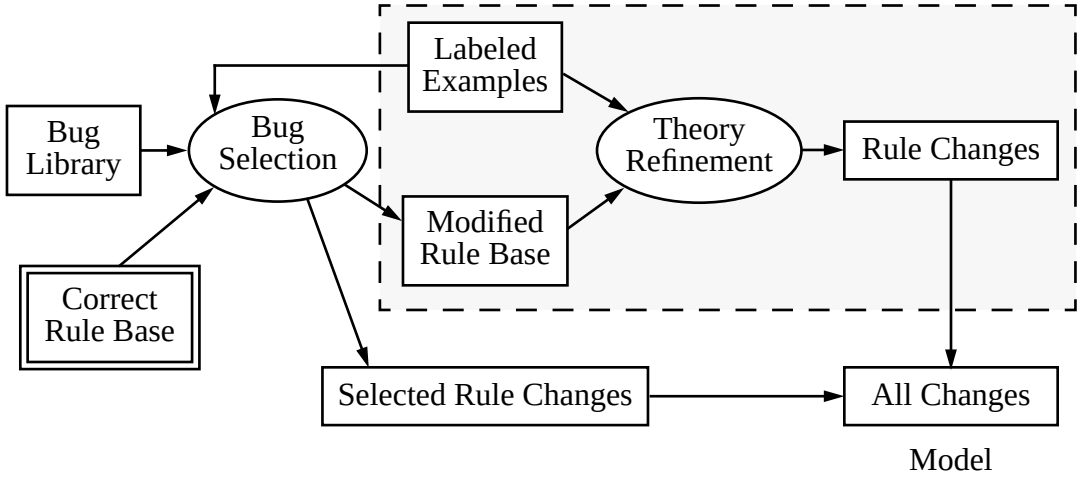


FIGURE 11 Extended modeling. Bug selection combined with theory refinement.

Figure 10 shows a schematic for how this is accomplished. The bug library, the correct rule base, and the student's labeled examples are input to a process which selects bugs from the library to be added to the rule base. Any bug which improves upon the predictive accuracy of the correct rule base is kept. The result is a modified rule base which resembles the student's behavior more closely than the correct rules but may still be incomplete. Note that the bugs which were selected are also returned as a separate item since they must be included with the final model of the student as shown in Figure 11. Once the modified rule base in constructed, it is passed to theory refinement along with the labeled examples to determine any additional refinements necessary for reproducing the behavior of the student. All rule changes, whether selected from the library or constructed by theory refinement, are returned as the final student model.

There are several advantages to this method of using a bug library. First, a wide variety of refinements including rare bugs or even parts of bugs can be present in the library. If a bug does not apply to a given student, then it will not influence the modeling of that student. Second, the bug library is incremental in nature. Thus, as more

data is collected on students, better libraries can be built to continuously improve the modeling process. Third, because any subset of a bug library can be applied in this way, one can imagine constructing a hierarchy of bug libraries, each of which might have a particular instructional value. This could be useful, for instance, in forming an initial impression of a student's ability by determining where he or she lies in a hierarchy of likely mistakes. But perhaps the most important feature of ASSERT's bug library algorithm lies in its ability to model both common and unique misconceptions. As with other bug-library based modeling methods, the ability to use a cache of expected errors gives the modeler an edge, especially in domains where a large amount of data would otherwise be required for an accurate diagnosis. And because the bug library is used as a precursor to theory refinement, ASSERT is not restricted to using only those bugs present in the library. Any specific student problem not in the bug library can still be captured by the theory refinement component of the algorithm.

2.6 An Example: The C++ Tutor

Figure 12 shows the entire ASSERT algorithm. As a concrete illustration, the remainder of the chapter will present the C++ Tutor, which is an implementation of the ASSERT paradigm for tutoring college-level students taking an introductory C++ course at the University of Texas at Austin. The tutorial was built in conjunction with the instructor for the class, and tests concepts which are present on the exams for the course. To make the tutor realistic, the same format used for exam questions was used for the questions posed to students in the tutorial, and the subjects covered were the subjects which students encounter on the final exam for the course. Unlike previous tutoring systems which have focused on programming languages, the C++ Tutor tests a student's ability to classify faulty program segments, rather than his or her ability to actually compose a program.

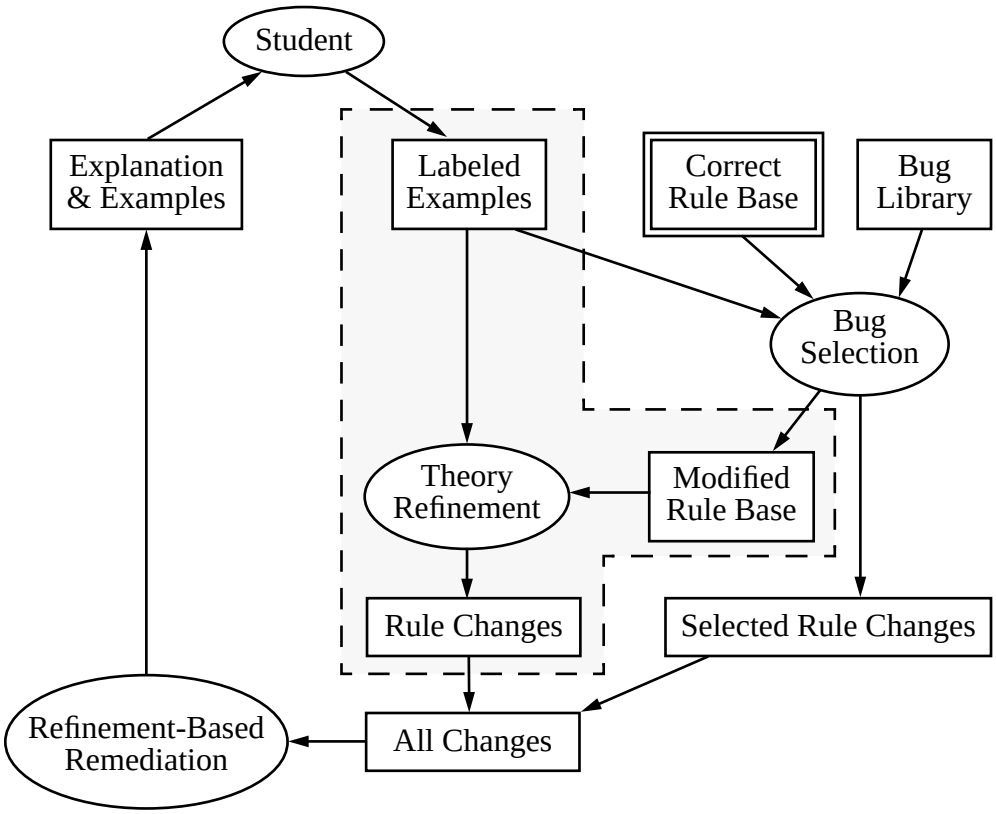


FIGURE 12 Overview of extended ASSERT algorithm. The shaded area represents the theory refinement component.

The tutorial consists of two problem sessions, each of which may be followed by remediation designed to address errors detected during the problem session. This two problem-session format has the advantage of giving students a chance to solve problems before getting feedback, and then test their new skills immediately in a second problem-solving session after the feedback. Unlike other tutorial programs which present material to the student and then test the student's understanding, the C^{++} Tutor instead assumes its users have received outside instruction. The focus is to give students practice answering questions similar to what they will see on exams, and provide feedback when they make mistakes. In this sense, the C^{++} Tutor resembles the "exercise" section typically found at the end of textbook chapters, aug-

mented with tutorial feedback. Thus the C++ Tutor is more like an intelligent *testing* system than a full-fledged intelligent tutoring system, since the tutorial information is presented only in response to the student's answers on an exam.

One advantage of this approach is that it illustrates how ASSERT can be used to supplement standard teaching methods. Rather than trying to reproduce an entire curriculum, one can focus on building tutorials for a few difficult concepts. This was the technique used to construct the C++ Tutor. Analysis of past final exams, plus consultations with the course professor, revealed two concepts historically problematic across a range of students. Specifically, these are the notions of ambiguity in C++ statements involving lazy operators and the proper declaration and use of integer and pointer constants. Of course, there is no reason why a larger ASSERT-style tutorial could not be built to cover the rest of the curriculum. However, narrowing the scope allowed for the construction of a complete tutorial which was then tested using the students actually enrolled in the course.

Figure 13 shows an example question taken from a trace of one student's interaction with the C++ Tutor. The question is presented with the corresponding feature vector which is the internal representation for that question. Thus note how the pointer "h" in the code segment is set to the address of the integer "j" in the expression "h = &j" which corresponds to the feature-value pair "(pointer-set true)" in the second line of the feature vector. After the code segment, the student is given a choice between three alternatives which represent three different labels for the problem. By selecting choice "A", the student labels this example as belonging to the "compiler error" category. That category is paired with the feature vector, and the pair becomes one of the labeled examples passed to the NEITHER theory refinement algorithm.

C++ Test Question

```
+---------------+
| Question 1 |
+---------------+

void main()
{
  const int j = 3, *h;
  int i, k;
  h = &j;
  cin >> k >> i;

  cout << (k % j);  cout << (i %= j);
}
```

Is the above (A) a compile error,
(B) ambiguous (i.e., different outputs from different compilers) or
(C) neither A nor B ?

answer: A

Feature Vector

((pointer non-constant) (integer constant) (pointer-init false) (integer-init true)
(pointer-set true) (integer-set no) (multiple-operands false) (position-a normal)
(operator-a-lazy ?) (lazy-a-left-value ?) (on-operator-a-side right)
(on-operator-b-side right) (operator-a modify-assign) (operator-b mathematical))

FIGURE 13 Example C++ problem with corresponding feature vector.

Recall that NEITHER takes a second input in the form of a propositional Horn-clause rule base. The first seven rules for the correct domain knowledge in the C++ Tutor are shown in Figure 14 (the complete rule base is shown in Appendix B). Taken together, they represent the knowledge for detecting whether or not a compile error will result from an illegal declaration or use of a constant integer or pointer. The first two rules indicate that an error will occur if a constant is not initialized or if it is assigned after its initial declaration. Following these are two rules to detect a constant which is not initialized and then three rules for catching an assignment to a constant.

compile-error ← constant-not-init
compile-error ← constant-assigned
constant-not-init ← (pointer constant) ∧ (pointer-init false)
constant-not-init ← (integer constant) ∧ (integer-init false)
constant-assigned ← (integer constant) ∧ integer-init ∧ (integer-set yes)
constant-assigned ← (integer constant) ∧ integer-init ∧
(integer-set through-pointer)
constant-assigned ← (pointer constant) ∧ pointer-init ∧ pointer-set

FIGURE 14 NEITHER rule base. Propositions without values are intermediate concepts or are shorthand for binary features requiring "true" as a value.

Looking at the feature vector of Figure 13 reveals that the student made a mistake labeling the question as a compile error. No constant is uninitialized since the only constant in the example is the integer "j" which is properly initialized to a value of 3. Also, while the pointer "h" is assigned, it is not a constant, and though the constant "j" is used in the expression "i %= j," it is never assigned. Consequently, nowhere in the code is there a constant which is uninitalized or assigned, making "A" an incorrect answer for this problem. In fact, it turns out there are no errors in this question which makes "C" the correct choice.

Given the labeled examples from the student, and the correct rule base, the goal of modeling is to produce a revised rule base that will simulate the student's behavior. Specifically, the final rule base should produce the same labels for the examples that the student does. For the current example, this means the rules of Figure 14 must be modified to conclude "compile-error" for the question in Figure 13 as the student does. Recall from Section 2.5 that ASSERT models the student in two steps, first by adding useful bugs from the bug library to the correct rules, and then by passing the modified rule base to NEITHER for further refinements. The top of Figure 15 shows three bugs selected for the student from a bug library which was automatically constructed from the models of 45 students who used the C^{++} Tutor (the complete bug library is shown in Appendix C). Each bug consists of the rule changed, the type of

Bugs Selected from Bug Library

bug type:	del-ante
rule:	constant-assigned ← (pointer constant) pointer-init pointer-set
antecedents:	(pointer constant)
stereotypicality:	-32
bug type:	add-ante
rule:	compile-error ← constant-assigned
antecedents:	(integer-set no)
stereotypicality:	-38
bug type:	del-rule
rule:	operator-b-sets ← (operator-b auto-increment)
stereotypicality:	-72

Additional NEITHER Refinement

refinement type:	del-ante
rule:	constant-assigned <- pointer-init pointer-set
antecedents:	pointer-init

FIGURE 15 C^{++} bug library and modeling example.

change made, and the stereotypicality value used to rank the bugs in the library. The first two bugs apply to the rules shown in Figure 14. In general, bugs selected from the library or changes made by NEITHER can apply to any number of mislabeled student examples. However, note specifically how the first bug in Figure 15 applies directly to the student's mislabeling of the question in Figure 13. The pointer in that question is not initialized nor is it constant, both of which are conditions for the last rule of Figure 14. The first bug from the library removes one condition from this rule which the feature vector for the question cannot satisfy. After the additional changes made by NEITHER, shown in the bottom half of Figure 15, note that only one condition remains in the rule. This condition can be met by the feature vector of Figure 13. Thus the final rule base, shown in Figure 16, can now successfully classify the question as a compile error.

compile-error ← constant-not-init
compile-error ← constant-assigned ∧ (integer-set no)
constant-not-init ← (pointer constant) ∧ (pointer-init false)
constant-not-init ← (integer constant) ∧ (integer-init false)
constant-assigned ← (integer constant) ∧ integer-init ∧ (integer-set yes)
constant-assigned ← (integer constant) ∧ integer-init ∧
(integer-set through-pointer)
constant-assigned ← pointer-set

FIGURE 16 Refined NEITHER rule base. Boldface represents modified rules.

Finally, Figure 17 shows a segment of the remediation taken from the trace for this same student. Note that the text consists of an initial explanation followed by an example, as discussed in Section 2.4. The text represents the C^{++} Tutor's remediation for the first bug of Figure 15 which deleted the "(pointer constant)" antecedent from the last rule of Figure 14. In the example portion of the text of Figure 17, note how the pointer "z" is both initialized and assigned, thus satisfying the last two conditions of the rule. The only condition of the rule not satisfied is the "(pointer constant)" antecedent, and it is this alone which keeps the example from being a compile error. In this fashion, the remediation highlights the exact details of the refinement, showing how the condition is essential to the rule.

2.7 Summary

Obviously, a great many details as to how ASSERT builds the bug library, selects the appropriate bugs, refines the rules, and generates the remediation have been left unstated. Hopefully, however, this early look at the C^{++} Tutor helps to frame the discussion for the algorithms presented in the chapters which follow. Before wrapping up the overview of ASSERT, several key features of its design bear repeating. First, note that the correct rule base is the *only* input which need be generated by an author who builds an ASSERT tutor. Everything else is constructed automatically from information drawn from student interactions with the system.This greatly reduces the

EXPLANATION

One way to detect a compilation error is to look for an identifier which is declared constant and initialized, then later assigned a new value.

A constant identifier is erroneously assigned when it is declared as a constant pointer to an integer, initialized to the address of some integer, and later set to the address of another integer. It does not matter if the identifier is a pointer declared to point to an constant integer or a non-constant integer; once a constant pointer is initialized it cannot be reset to the address of another integer.

Specifically, note the following which contribute to this type of error:
* There must be a pointer declared to be constant (but not necessarily pointing to a constant object).
* A pointer declared to be constant must be initialized.
* A pointer declared a constant and initialized must be set after its initialization.

Here is an example to illustrate these points:

Example

Here is an example which might appear to be a compile error but is actually CORRECT:

```
void main()
{
  const int x = 5, y, w, *z = &x;
  z = &w;
  cin >> w >> y;

  cout << ((y *= x) || (y > w));  cout << (w -= x);
}
```

This example is NOT a compile error because:
* The pointer 'z' is declared as a NON-CONSTANT pointer to a constant integer, so it does not have to be initialized and it can be reset.

IGURE 17 Example remediation given to a student.

time required to construct the tutoring system, especially when it comes to building a bug library. Instead of performing a laborious protocol analysis to uncover bugs one can let ASSERT find them gradually by analyzing student models over time. Second, ASSERT is purposely kept modular. Of particular note is the theory refinement module, shown in the shaded area of Figure 12, which can be replaced with improved refinement algorithms as they become available. And lastly, we have tried to keep ASSERT as generic as possible. Thus, for instance, the number and kinds of bug libraries used is not limited, nor does ASSERT dictate how explanations and examples are generated to remediate the student. And while ASSERT currently requires its input in the form of propositional Horn-clause rules, such a representation still provides the author with a fair amount of representational power. The result is a system which allows authors to experiment with different instructional methods and various levels of representational detail.

CHAPTER 3 The NEITHER Theory-Refinement Algorithm

The NEITHER theory-refinement algorithm forms the backbone of ASSERT's modeling capabilities. It is the means by which individual student misconceptions are modeled, and its outputs are the raw materials for bug-library construction. Furthermore, the choice of theory-refinement algorithm has a profound impact on the implementation of any ASSERT-type tutoring system because the knowledge representation used by the theory-refinement module becomes the de facto representation of the tutoring system. It is important, then, to defend the choice of NEITHER as ASSERT's theory-refinement component. To that end, this chapter begins with a brief history motivating the genesis of theory-refinement algorithms, followed by an overview of how theory refinement is used for student modeling and a description of the NEITHER algorithm.

3.1 Machine Learning and Theory Refinement

Over the past decade, a number of machine learning algorithms have been developed which can induce a classification system from a set of *training examples* [Quinlan, 1986; Michalski, 1983; Mitchell, 1982; Rumelhart et al., 1986]. These examples are labeled feature vectors presented in the form of input-output pairs, where the input is a collection of values for features of the domain and the output represents the category corresponding to the input. In the typical classification problem, the task of the induction algorithm is to learn, from the examples, some function which will produce the correct output category for any given set of inputs. The learned system can be thought of as a function which maps inputs to outputs corre-

sponding to the pattern implicit in the training examples. Unfortunately, induction techniques typically require a large number of examples to produce accurate results.

Recently, a new generation of more effective machine learning algorithms have been developed which combine induction with other machine learning techniques. These methods use two inputs, labeled examples plus an initial domain theory [Ginsberg, 1990; Ourston and Mooney, 1990; Craw and Sleeman, 1991; Towell and Shavlik, 1991]. Such learners are termed *theory-refinement* systems since they take an input knowledge base (called the domain *theory*) and produce a revised version which is consistent with the examples. The idea is one of incremental change; the learner starts with an initial theory that is imperfect and modifies it to fit a set of data. For example, the EITHER system [Ourston and Mooney, 1990] alters an initially incorrect or incomplete rule base by modifying or deleting existing rules or by adding new rules until the rule base is consistent with the input examples. Like other theory-refinement systems, EITHER has been shown to be more effective at learning concepts than induction alone when given an approximately correct theory. In general, this is because theory refinement need only infer the *differences* between its input theory and the correct theory, whereas induction must learn everything from scratch. The closer the original theory is to the desired target, the easier it is for systems like EITHER to produce accurate results. Said another way, the information stored in the initial theory allows theory refinement to be more accurate with fewer examples [Mooney, 1994].

3.2 Student Modeling by Theory Refinement

Casting student modeling as a general machine learning problem is straightforward enough. Given examples of student behavior, the goal is to produce a system which can simulate the student. From an induction perspective, this amounts to translating the student's behavior into labeled examples. For instance, examples can

be constructed by linking inputs taken as test problem specifications with the corresponding label provided by the student's solution. When given to an inductive learner, the resulting system produced is a simulation of the student's problem solving ability. Given new test problems, the induced system will tend to produce the same answers as the student.

Using theory refinement to model a student is only slightly more complicated. Normally, theory refinement is presented as a technique for *fixing* errors present in an *incorrect* theory by evaluating example data. In principle, however, there is no reason why one cannot use theory refinement "backwards;" i.e., to purposely *introduce* errors into a *correct* theory. This is precisely how theory refinement can be used to model students as shown in Figure 18. Starting with a theory representing correct knowledge of a domain, one models the student by introducing errors until the system matches the examples illustrating the student's performance. If the initial theory is a model of ideal student behavior (i.e., the knowledge the tutoring system wants to impart to the student), then the transformations made in creating the student model show where the student has misconceptions that caused him or her to deviate from the correct knowledge. Using this strategy, the initial theory provides a focus for the modeling task which becomes a search for refinements that will transform the input theory to a theory which mimics the student's behavior.

NEITHER (New EITHER) is a modification of the EITHER propositional Horn-clause theory-refinement algorithm [Ourston and Mooney, 1990; Ourston, 1991]. The EITHER system was chosen as a starting point for two reasons: its was essential that a symbolic refinement system be used and EITHER was the most complete symbolic theory-refinement system available. Non-symbolic refinement algorithms based upon distributed representations can be difficult to use as student modelers because the refinements are not localized. Without distinct knowledge of a bug it can be difficult to generate precise feedback. EITHER is a symbolic refiner that can gener-

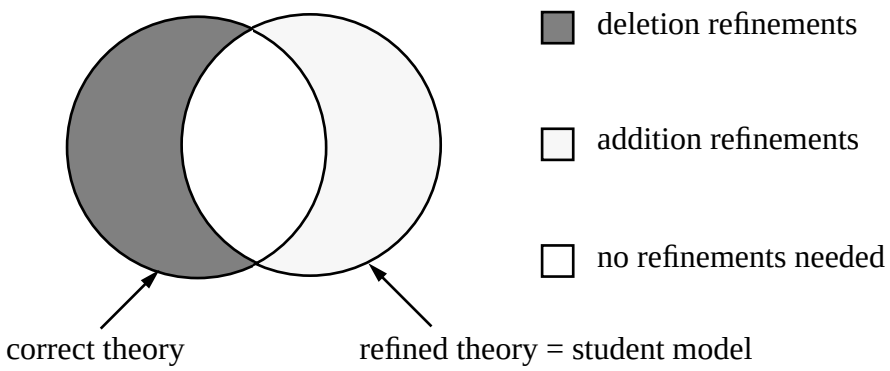


FIGURE 18 Overview of theory-refinement modeling.

alize or specialize a theory, and is guaranteed to produce a set of refinements which are consistent with the input examples. Unfortunately, EITHER's worst case run-time is exponential in the size of the theory, making it too slow for use as an interactive modeler.

The remainder of this chapter, then, focuses on the EITHER algorithm and the changes made in NEITHER which speed up the refinement process. After an introduction to EITHER, NEITHER is described followed by a brief empirical comparison of the two systems.

3.3 The EITHER Algorithm

EITHER was designed to repair propositional Horn-clause theories that are either overly-general or overly-specific or both using a set of input examples. The examples are assumed to be lists of feature-value pairs chosen from a set of *observable* domain features. Each example has an associated label or *category* which should be provable using the theory with the feature values in the example. Imperfect propositional Horn-clause theories can have four types of errors as shown in Figure 19. An overly-general theory is one that causes an example to be classified in categories other than its own (i.e., a false positive, also called a *failing negative*). EITHER spe-

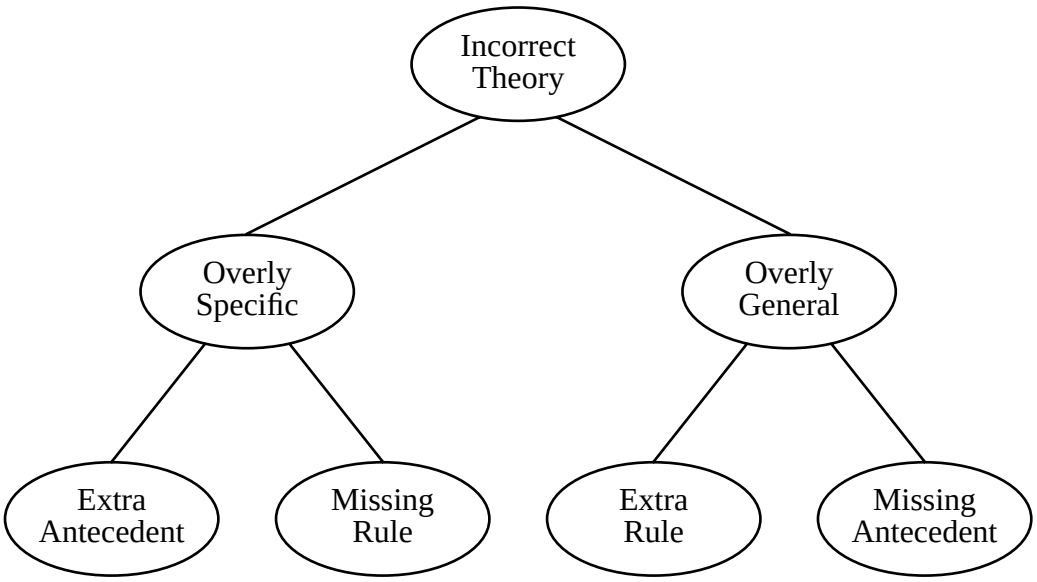


FIGURE 19 Theory error taxonomy for propositional Horn-clause theories.

cializes existing antecedents, adds new antecedents, and deletes rules to fix such problems. An overly-specific theory causes an example not to be classified in its own category (i.e., a false negative, also called a *failing positive*). EITHER retracts and generalizes existing antecedents and learns new rules to fix these problems. Unlike other theory-refinement systems that are subject to local maxima, EITHER is guaranteed to fix any arbitrarily incorrect propositional Horn-clause theory [Ourston, 1991].

EITHER uses a combination of techniques including deduction, abduction and induction to revise a theory. Deduction is used to determine which examples are the failing positives and failing negatives needing attention, and to identify potential changes for each example that will repair the theory for that example. Abduction is used to find a set of assumptions, corresponding to rule antecedents, which, if generalized or deleted, will fix a failing positive. A rule retraction method is used to find a set of rules which, if deleted or specialized, will repair a failing negative. When these simple techniques do not work, induction is used as a last resort to learn new

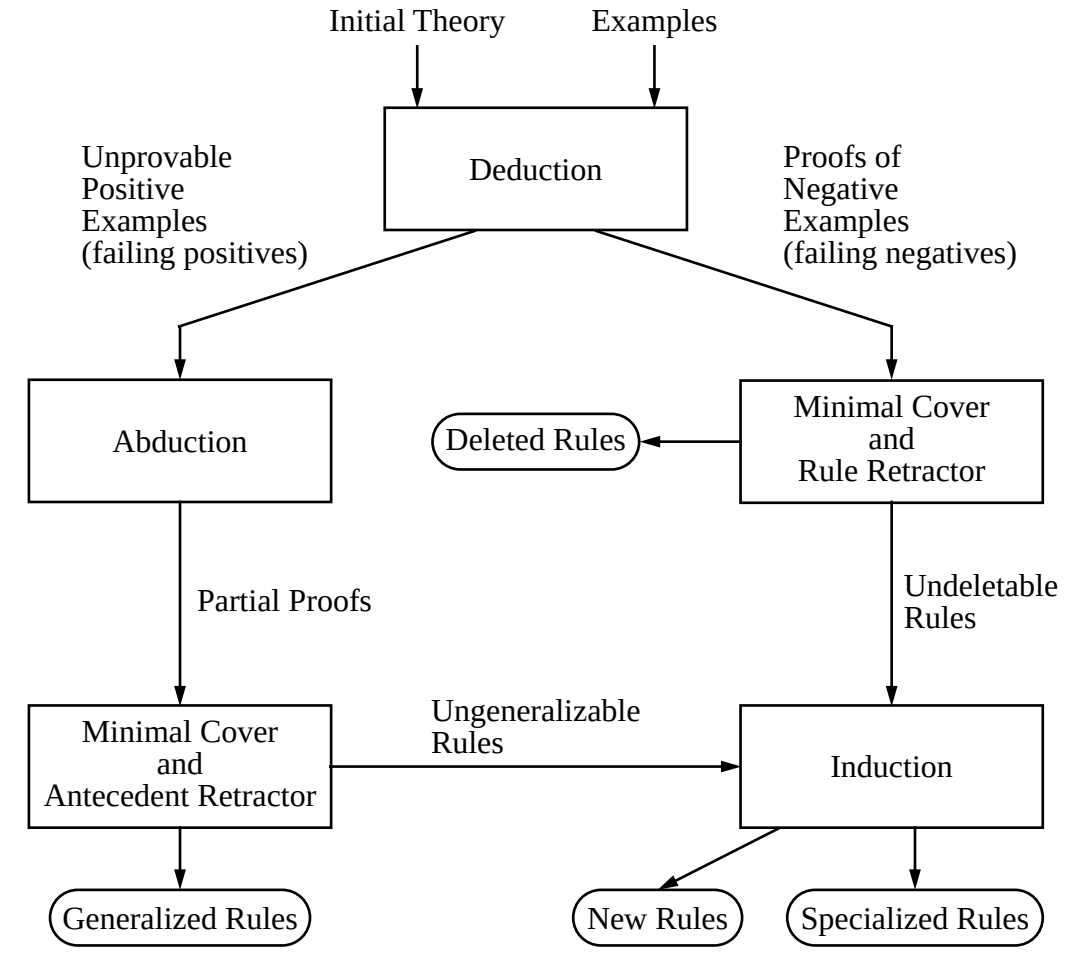


FIGURE 20 EITHER architecture.

rules to repair failing positives or to add antecedents to existing rules to repair failing negatives. Figure 20 shows an overview of the architecture of EITHER.

EITHER works by finding *covers* to fix the failing examples. A cover is a set of deletions which will fix one or more failing examples. Two types of covers are computed: an *antecedent cover* and a *rule cover.* An antecedent cover is a set of antecedents which, if deleted from the theory, will allow all the failing positives to prove their correct category. A rule cover is a set of rules which, if deleted from the theory, will prevent all the failing negatives from proving an incorrect category. The prob-

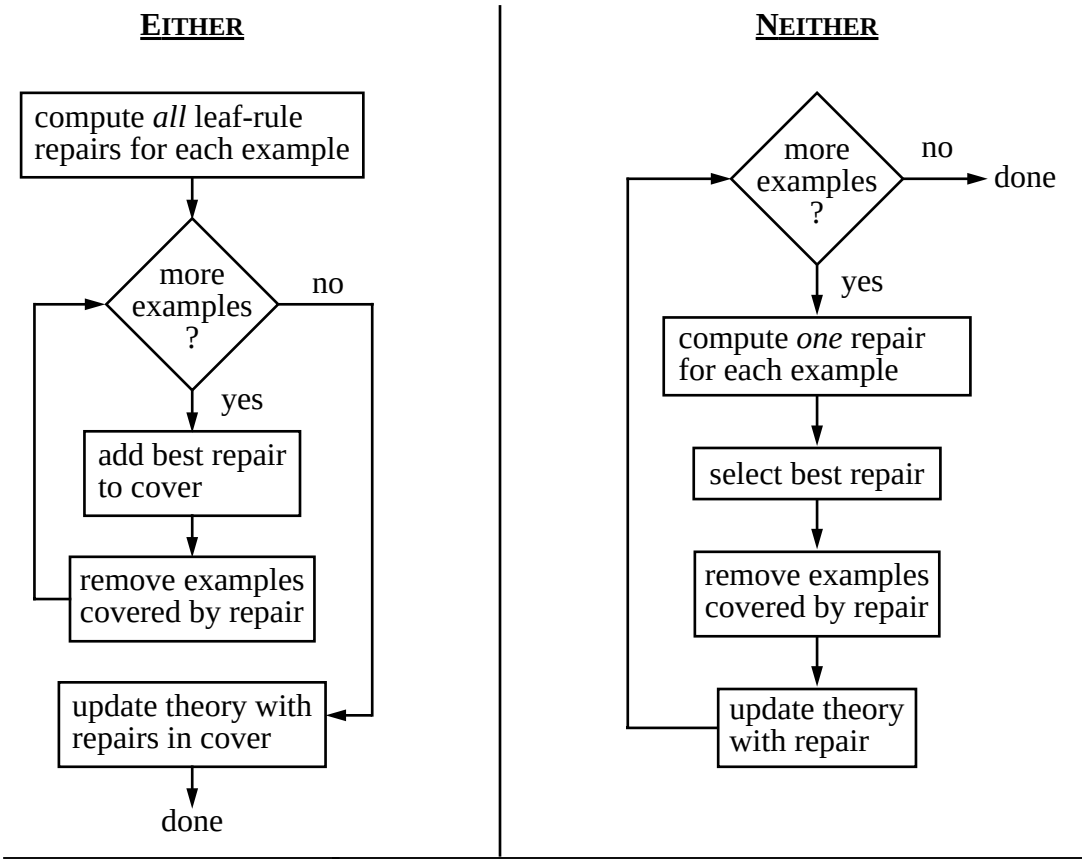


FIGURE 21 Comparison of EITHER and NEITHER main loops.

lem faced by EITHER is to find the minimum antecedent and rule covers. As this problem is NP-hard [Ourston, 1991], EITHER instead uses a greedy approach to hill climb towards the ideal antecedent and rule covers.

Antecedent and rule covers are computed separately, but each calculation uses the same three-step process depicted in the left half of Figure 21. First all the possible *leaf-rule* deletions which will fix a failing example are computed. This is done for each failing example. A leaf-rule deletion is one which occurs at a leaf rule of the theory, which is a rule whose antecedents are either observable features or intermediate concepts which are not the consequent of any existing rule. Next, EITHER enters a loop to select repairs to be added to the cover. On each iteration, the best

repair from among all the repairs for all the examples is selected based upon a *benefit-to-cost* ratio that trades off the number of failing examples fixed by the repair against the size of the repair. After selecting the best repair, all failing examples which it fixes are removed from the set of failing examples before starting the next iteration. This greedy selection is repeated until all the failing examples are fixed by the deletions in the cover. In the final step after the loop, the repairs in the cover are applied to the theory. If the application of a repair over-compensates by creating new failing examples, EITHER passes the covered examples and the new failing examples to an induction component based upon a version of ID3 [Quinlan, 1986]. The results of the induction are added as a new rule when generalizing or as additional antecedents when specializing.

The time consuming part of this process is the first step which computes all leaf-rule repairs for each failing example. As an illustration, Figure 22 shows the leaf-rule repairs for a failing positive example. The upper part of the diagram shows an input theory both as rules (on the left) and as an AND-OR graph. The middle of the diagram shows two failing positive examples which cannot prove category *a* at the top of the tree. The lower part of the diagram shows the *partial proofs* for example *E2*. A partial proof is one in which some antecedents cannot be satisfied. These antecedents become the contents of a repair for a failing positive example. Thus the proofs in Figure 22 yield four possible repairs which will fix *E2*: *delete{P, R, V}*; *delete{P, R, Y, Z}*; *delete{S, V}*; *delete{S, Y, Z}*. The combinatorial nature of computing all possible leaf-rule repairs is what makes EITHER's run-time performance too slow for interactive modeling.

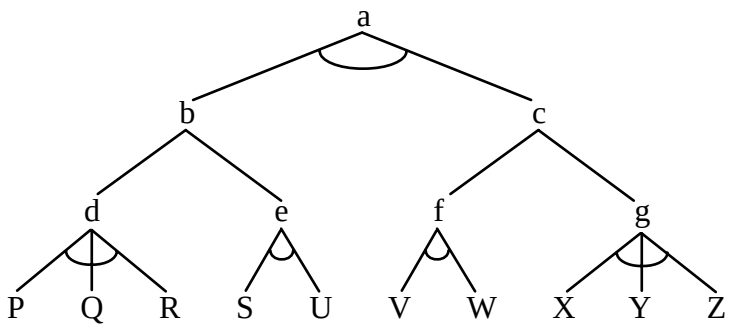
3.4 Building NEITHER from EITHER

The main loop of NEITHER, shown in the right half of Figure 21, is quite different from EITHER. Two new algorithms enable the remodeling. First, the calculation

Theory

$a \leftarrow b \wedge c$
$b \leftarrow d$
$b \leftarrow e$
$c \leftarrow f$
$c \leftarrow g$
$d \leftarrow P \wedge Q \wedge R$
$e \leftarrow S \wedge U$
$f \leftarrow V \wedge W$
$g \leftarrow X \wedge Y \wedge Z$

AND-OR Graph



Example	*Feature Vector* P	Q	R	S	U	V	W	X	Y	Z
E1	F	F	F	F	F	F	F	T	F	T
E2	F	T	F	F	T	F	T	T	F	F

Partial Proofs for Example E2

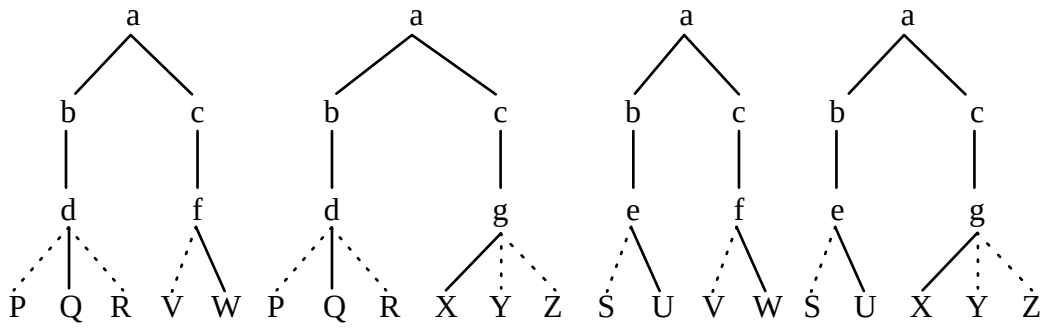


FIGURE 22 Partial proofs for an unprovable positive example. Both "E1" and "E2" are unprovable in category "a". Capital letters indicate operational features. Dotted lines indicate unprovable antecedents.

of repairs is now achieved in *linear* time. Second, all searches through the theory (for deduction, antecedent retraction and rule retraction) are optimized in NEITHER to operate also in linear time by marking the theory to avoid redundant subproofs. The combination gives NEITHER the ability to compute a repair for a failing example much more quickly than EITHER. Consequently, NEITHER can compute repairs on a

trial basis and throw away undesirable results because the computation of repairs is cheap.

NEITHER abandons the notion of searching for all leaf-rule deletions, opting instead for a method which rapidly selects a *single* repair for each example. The three steps of the EITHER algorithm can then be moved *inside* the loop which performs the greedy search. On each iteration, one repair is computed for each failing example. The best of the repairs is selected using the benefit-to-cost ratio and applied to the theory. As with EITHER, if new failing examples result from applying the repair, induction is used instead of the repair to add new antecedents or new rules to the theory. Therefore, the precise benefit-to-cost ratio used in NEITHER is to divide the number of failing examples fixed by the repair or induction by the size of the repair or the induction.

Note that the iteration must eventually halt since each circuit is guaranteed to reduce the number of failing examples by at least one. As a side benefit, NEITHER can take advantage of *interactions* among repairs for different failing examples since the theory is modified each time through the loop. Thus subsequent repairs are computed in light of any previous changes made to the theory including any inductions which may expand the rule base. This is a new feature not present in EITHER.

3.4.1 Finding Repairs in Linear Time

To illustrate how repairs are computed in linear time, refer again to Figure 22. Rather than computing all partial proofs, NEITHER follows a recursive bottom-up procedure to construct a single set of retractions. Starting at the leaves of the AND-OR graph, NEITHER collects the set of unprovable antecedents and passes them up to the parent node. At each parent, the best option from among its children is computed and passed up again. When multiple options exist at a parent node, NEITHER alter-

nates between returning the smallest option and returning the union of the options, depending whether the choice involves an AND or OR node. For generalization, deletions are unioned at AND nodes because all unprovable antecedents must be removed to make the rule provable. At OR nodes, only the smallest set of deletions is kept since only one rule need be satisfied. For specialization, these choices are reversed. Results are unioned at OR nodes to disable all rules which fire for a faulty concept. At AND nodes, the smallest set of rule retractions is selected since any single unsatisfied antecedent will disable a rule.

As an example, in Figure 22 the antecedent retraction calculations made by NEITHER would begin at the root of the graph, recursively calling nodes *b* and *c*. Retraction for node *b* would then recurse on nodes *d* and *e*. Since *P, R* and *S* are false, node *d* returns *{P, R}* and node *e* returns *{S}*. When the recursion returns back to node *b* a choice must be made between the results from nodes *d* and *e* because the theory is being generalized and node *b* is an OR node. Since node *e* requires fewer retractions, its retractions are chosen as the return value for node *b*. Recursion for node *c* follows a similar pattern: node *f* returns *{V}*, node *g* returns *{Y, Z}* and node *c* chooses the smaller results from node *f* as its return value. Finally, nodes *b* and *c* return their values to node *a*. Since node *a* is an AND node and the theory is being generalized, the results from *b* and *c* are combined. The final repair returned from node *a* is *delete{S, V}*.

This algorithm is linear in the size of the theory. No node is visited more than once, and the computation for choosing among potential retractions must traverse the length of each rule at most once. The final repair is also minimum with respect to the various choices made along the way; it is not possible to find a smaller leaf-rule repair that will satisfy the example. This new algorithm thus trades the multiple repair options available in EITHER for speed in computation and the additional capability of computing repairs in light of previous changes made to the theory.

Original Theory

$a \leftarrow P \wedge Q \wedge R \wedge S$

$a \leftarrow U \wedge V \wedge W \wedge X \wedge Y$

(a)

Example	P	Q	R	S	U	V	W	X	Y	*Desired Category*
E1	F	F	F	T	F	F	T	T	T	a
E2	F	F	F	F	F	F	F	T	T	a
E3	F	F	T	T	T	F	F	F	T	a
E4	T	F	T	T	F	F	F	F	F	¬ a

(b)

	Iteration 1			*Iteration 2*		
	repair	*benefit*	*cost*	*repair*	*benefit*	*cost*
E1	delete{U,V}	1	2	n/a	n/a	n/a
E2	delete{U,V,W}	2	3	n/a	n/a	n/a
E3	delete{P,Q}	1	3	delete{X}	1	1
	Theory after Iteration 1 $a \leftarrow P \wedge Q \wedge R \wedge S$ $a \leftarrow X \wedge Y$			*Final Theory* $a \leftarrow P \wedge Q \wedge R \wedge S$ $a \leftarrow Y$		

(c)

FIGURE 23 Example of NEITHER refinement algorithm. Part (a) shows original theory, part (b) input examples, part (c) the refinements selected by NEITHER.

3.4.2 A NEITHER Example

Figure 23 illustrates a more complete example of how NEITHER revises a theory to be consistent with a set of examples. The figure is separated into three parts. Part (a) shows an original theory consisting of two rules which define the concept *a*. Part (b) shows four examples given as input to NEITHER. The first three of these are failing positives; i.e., the original theory is unable to prove *a* true for any of these three examples. The fourth example is a negative example that the original theory correctly classifies. Though such correct examples initially pose no problem, they must be considered when selecting the best repair to apply to the theory.

Part (c) shows the effects of two iterations through the main loop of NEITHER. On the first iteration, NEITHER computes a repair for each of the three failing positive examples as outlined in the previous section. Recall that a theory must be generalized to fix failing positives, and that a concept with multiple rules is defined as an OR node in the AND-OR graph representation of a theory. Thus during calculation of repairs, NEITHER will select the smaller of the fixes proposed for the two rules. For example *E1*, the repair *delete{U, V}* is smaller than *delete{P, Q, R}*, for *E2* *delete{U, V, W}* is smaller than *delete{P, Q, R, S}*, and for *E3* *delete{P, Q}* is smaller than *delete{V, W, X}*. Since *E4* is correctly classified, it requires no repair.

Having found the three repairs, NEITHER must then choose the best among them to apply to the theory before moving to the second iteration. To rank each repair, NEITHER computes a benefit-to-cost ratio by temporarily applying each repair to the theory in turn. The benefit is measured in terms of how many failing examples will be fixed by applying the repair. The cost is the sum of the size of the repair plus the number of any *new* failing examples created. Applying *delete{U, V}* to the original theory will only fix example *E1*, giving it a benefit value of 1. Since no new failures are created by this fix, the cost for it is just the size of the repair or 2. *E2*'s repair will fix both *E1* and *E2*, giving it a benefit of 2 at a cost of 3 since it also avoids any new failures. *E3*'s repair only covers *E3*, but does so at the expense of causing *E4* to be incorrectly classified. Thus an additional penalty of one (for one new failing example) is added to the cost of *E3*'s repair. Since *E2*'s repair has the best benefit-to-cost ratio, it is selected and applied to the original theory.

NEITHER then enters the second iteration of its main loop. Note however that the original theory has been modified by the application of the repair from the first iteration. The new theory is shown at the bottom of the first iteration of part (c). At this point, only one example, *E3*, remains a problem. The repair from the last iteration fixed *E1* and *E2* without disrupting the classification of *E4*. The repair computation

for *E3* is now different from the last iteration due to the change made in the theory. Since *U*, *V* and *W* have been removed from the second rule, NEITHER selects *delete{X}* as the repair for *E3* since it is smaller than *delete{P, Q}*. Note also that this new repair for *E3* will not cause E4 to fail. Since there are no other repairs, NEITHER applies this fix to the theory resulting in the final theory shown after the second iteration. If the four examples had represented categorizations generated by a student, and if the original theory represented the correct domain knowledge, then the difference between the original theory and the final theory would represent the student model. Specifically, the student would be modeled as missing the first four conditions of the second rule of the theory.

3.4.3 Refining Rules at Higher Levels

One of the disadvantages of early versions of EITHER, which was fixed in the final EITHER algorithm, was its bias towards changing the lower level rules in a theory. This was a result of focusing changes at leaf-rule deletions when computing repairs. As described thus far, NEITHER shares the same problem; even though repairs are computed more quickly, they are still biased towards leaf-rule deletions. In terms of student modeling this is problematic since there is no a priori reason to assume that student errors are more likely to occur at leaf rules.

However, leaf-rule bias is used for a good reason: without it, changes would tend to occur at the other extreme of the theory. Keep in mind that EITHER and NEITHER also attempt to find the *smallest* change to fix a failing example. Without the leaf-rule bias, most repairs would consist of a few deletions at the top-level rules of the theory as opposed to multiple changes at lower-level rules. In effect, this would amount to throwing out the theory except for the top-level rules, which is little better than inducing a set of rules from scratch. The leaf-rule bias forces both refinement algorithms to use more of the theory to compute repairs.

```
function Repair-PCC (E:example, T:theory): deletions;
begin
   if LeafRule(T) then return LeafDeletions(T);
   else begin
      candidates = ∅;
      for c ∈ Children(T) add Repair-PCC(E, c) to candidates;
      best = choose repair(s) from candidates based on whether T is at
             an AND or OR node, and whether E is a failing positive
             or failing negative;
      if BenefitCost(best) <= BenefitCost(ParentChange(best)) then
         return best;
      else return ParentChange(best);
   end;
end
```

FIGURE 24 Pseudocode for repair comparison at different levels of the theory.

The general solution to this dilemma is to find a mechanism for comparing leaf-rule deletions with their corresponding higher-level changes. Thus, for example *E2* of Figure 22, the change *delete{P, R}* from rule $d \leftarrow P \wedge Q \wedge R$ is comparable to the change *delete{d}* from the rule $b \leftarrow d$, which is comparable to the change *delete{b}* from the rule $a \leftarrow b \wedge c$. To find the minimum repair by comparing all possible level changes corresponding to all leaf-rule deletions again results in a combinatorial search which is exponential in the size of the theory. Of course, this is precisely the kind of time sink that NEITHER is trying to avoid.

NEITHER adopts a compromise solution to this problem which is similar to that used by EITHER; specifically, NEITHER uses a *parent-child comparison* (PCC) technique. The PCC algorithm maintains the linear time, bottom-up strategy for building repairs, and adds an extra check which compares corresponding changes between parent and child nodes and keeps only the better of the two. The pseudocode for this algorithm is shown in Figure 24. The *Repair-PCC* routine recurses down towards the leaf rules of the theory. When at a leaf rule, the appropriate deletions for that rule are determined and returned to the caller as before. At a parent rule, *Repair-PCC* is invoked on each of the children and the results collected into the *candidates* vari-

able. As described in Section 3.4.1, either the smallest member of *candidates* or the union of all the members in *candidates* is selected and set to *best*. However now, instead of just returning *best* as before, NEITHER uses the benefit-to-cost measure to compare making the changes designated in *best* versus any corresponding changes to the parent rule at node *T*. If changing the parent is the same or worse, then the changes in *best* are returned. If not, the parent change is returned in place of the corresponding child changes in *best*. This maintains a bias towards leaf-rule deletions unless the corresponding parent change is better.

As an example, refer again to Figure 22, but this time focus on example E1. Without level comparisons, the repair for this example would be *delete{S, U, Y}* since the *e* rule requires fewer deletions than the *d* rule, and the *g* rule requires fewer deletions than the *f* rule. With level comparisons the repair is much different. After recursing on the *d* and *e* nodes as children of node *b*, the *candidates* variable would consist of *delete{P, Q, R}* for rule *d* and *delete{S, U}* for rule *e*. Since *E1* is a failing positive and *b* is an OR node, the smaller repair *delete{S, U}* is selected as the value for *best*. Now, however, instead of simply returning *best*, the corresponding change *delete{e}*, from the rule $b \leftarrow e$, is compared to *delete{S, U}*. Both have the same benefit since neither one completely fixes example *E1* or *E2*, but *delete{e}* is the smaller change. Thus *delete{e}* is returned as the value for the recursion of node *b*.

On the other side of the theory, *delete{Y}* is chosen over *delete{V, W}* and compared against the corresponding change *delete{g}*, from the rule $c \leftarrow g$. Again, there are equal benefits because neither change completely fixes example *E1* or *E2*. But since both deletions are the same size, they also have the same cost, so *delete{Y}* is returned based on the bias towards leaf rules. Finally, *delete{e}* and *delete{Y}* are returned to node *a* and combined in *best* since *a* is an AND node. The last comparison is thus a check of *delete{e, Y}* against the corresponding change of *delete{b, c}*

Theory

a ← b ∧ c ∧ W
b ← P ∧ Q ∧ R ∧ S
c ← U ∧ V

(a)

Example	P	Q	R	S	U	V	W	*Desired Category*
E1	F	F	T	T	T	T	T	a
E2	F	F	F	T	T	T	T	a
E3	T	T	T	T	F	T	F	a
E4	T	T	T	F	T	T	T	¬ a
E5	F	F	F	F	T	T	T	¬ a
E6	T	T	T	T	F	F	T	¬ a

(b)

	Child Rule			*Parent Rule*			*Result*
	repair	*benefit*	*cost*	*repair*	*benefit*	*cost*	
E1	delete{P,Q}	1	2	delete{b}	2	3	Parent better
E2	delete{P,Q,R}	2	3	delete{b}	2	3	Tie, use child
E3	delete{U,W}	1	2	delete{c,W}	1	3	Parent worse

(c)

FIGURE 25 Examples of level comparison algorithm. Part (a) shows theory, part (b) input examples, part (c) the refinements at different levels of the theory.

from the rule $a \leftarrow b \wedge c$. Both changes fix one example, E1, and are of the same size. Based on the bias towards lower-level changes, *delete{e, Y}* is returned as the final repair.

Figure 25 shows additional examples of the PCC algorithm. In general, a change at the parent rule can be better, worse or the same as the corresponding change at a child rule. Typically, higher-level changes require fewer deletions, resulting in a lower cost. However, most often they also create new failing examples, as is the case for all the parent rule changes in Figure 25, resulting in a higher cost. As long as these forces balance out, preference is given to the lower level changes. Only when the parent change can fix more examples at lower cost without creating too many new failing examples is it preferred. As a final point, note that level changes are not just an academic curiosity. Figure 16 on page 30 illustrates a model

for a student interacting with the C++ Tutor which required changes to different levels of the theory.

A complete example of the results of NEITHER operating in the C++ Tutor domain was shown in Figure 15 on page 29 of the overview. Those refinements fix all the mistakes in the first 10 questions made by the student whose complete interaction is shown in Appendix E.

3.5 Comparison of NEITHER and EITHER

To illustrate how EITHER and NEITHER compare, two tests were run. One compared the running times and accuracies of the two systems to test whether NEITHER could maintain the predictive accuracy of EITHER while reducing execution time. The other test compared NEITHER against NEITHER with PCC enabled (NEITHER-PCC) to determine whether the level-refinement algorithm was indeed better at detecting changes in the theory. For a more complete comparison of EITHER and NEITHER see [Baffes and Mooney, 1993].

3.5.1 Run Time and Accuracy

The DNA promoter sequences data set [Towell and Shavlik, 1990] was used to compare the running times and accuracies of EITHER and NEITHER. This data set involves 57 features, 106 examples, and 2 categories. The theory provided with the data set has an initial classification accuracy of 50%. The experiments proceeded as follows. The 106 examples were randomly divided into training and test sets. Training sets were further divided into subsets, so that the algorithms could be evaluated with varying amounts of training data. After training, each system's accuracy was recorded on the test set. To reduce statistical fluctuations, the results of this process of dividing the examples, training, and testing were averaged over 25 runs. Training time and test set accuracy were recorded for each run. Statistical significance was

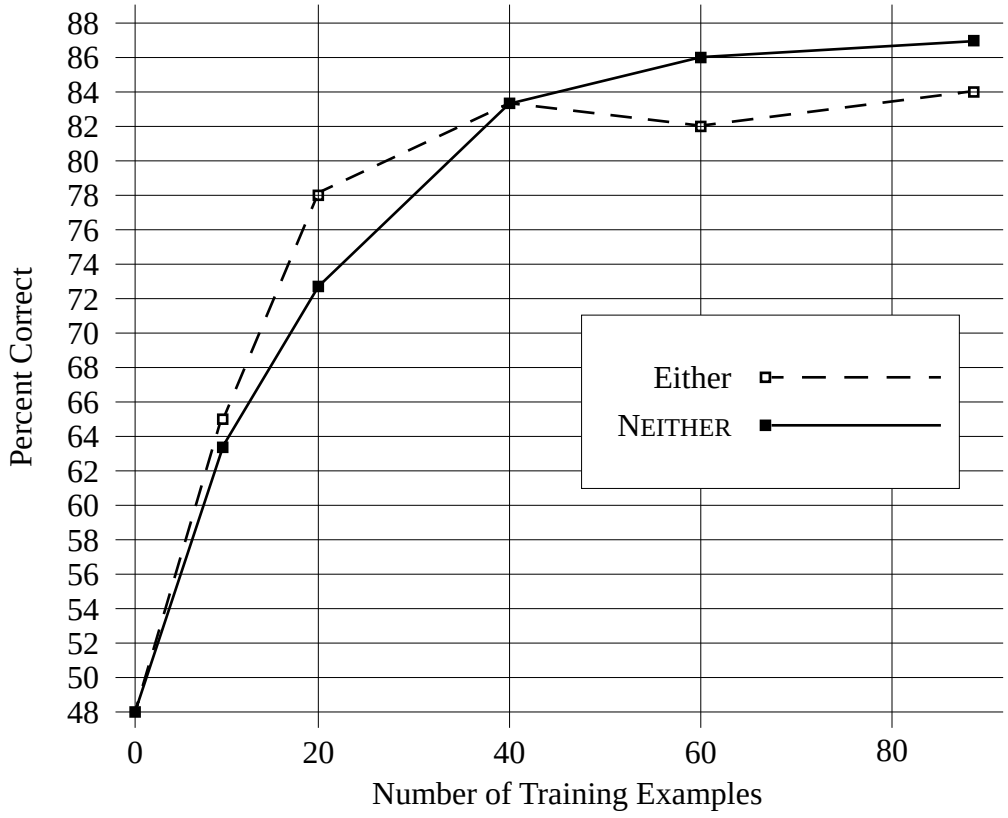


FIGURE 26 Accuracy of EITHER and NEITHER on DNA promoter test.

measured using a Student t-test for paired difference of means at the 0.05 level of confidence (i.e., 95% confidence that the differences are not due to random chance).

The results are shown in Figure 26 and Figure 27. Figure 26 compares the accuracy of NEITHER and EITHER on the test set as a function of the number of training examples. NEITHER's accuracy was lower than EITHER's for small training sets and higher for large training sets. Figure 27 compares the running times for NEITHER and EITHER for the same array of training sets. NEITHER consistently ran more than an order of magnitude faster than EITHER. These timing results were not surprising since NEITHER uses a linear approach to repair calculation as opposed to EITHER's

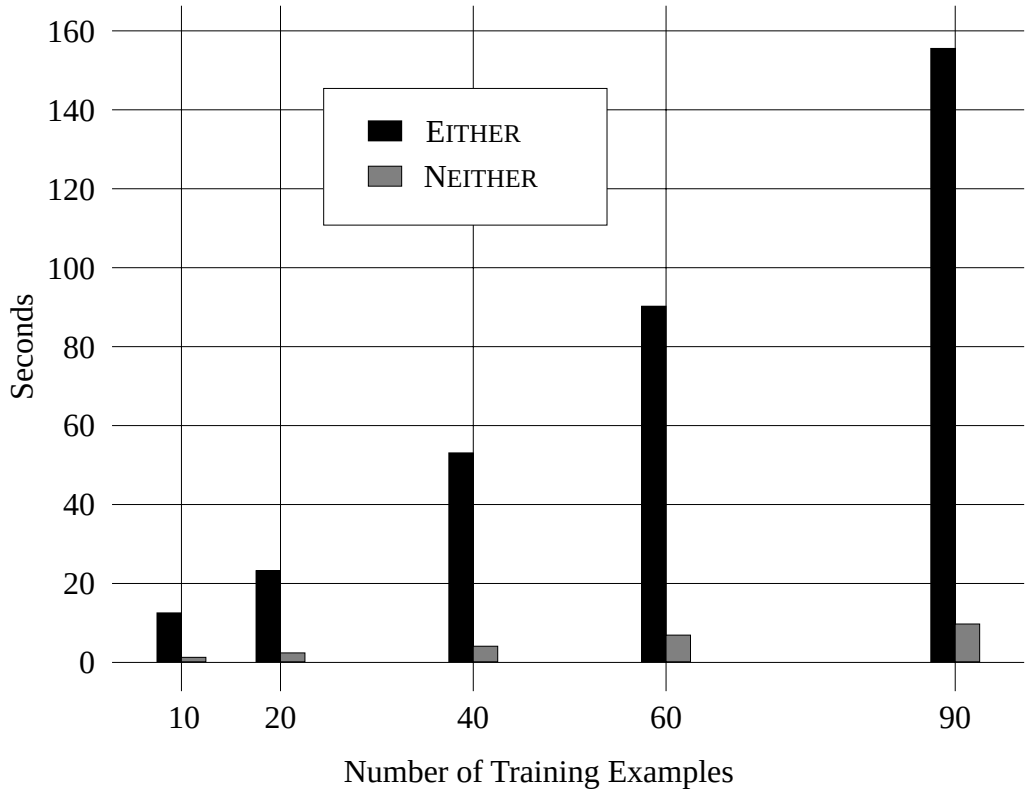


FIGURE 27 Training time of EITHER and NEITHER on DNA promoter test.

potentially exponential method. Since repairs must be computed for every failing example, faster repair calculation translates into a dramatic overall time savings.

The most surprising result of the experiment was the difference in accuracy between EITHER and NEITHER. EITHER was more accurate with fewer training examples, but its accuracy dropped off relative to NEITHER as the number of examples increased. One possible explanation for this behavior lies in the difference between how the two systems compare potential refinements. Recall that EITHER computes multiple repairs for each example, but does so only once. NEITHER, by contrast, computes one repair per example each time through its main loop. As a result, with fewer training examples, EITHER has more potential refinements to examine, apparently giving it an edge over NEITHER. Even though NEITHER computes new repairs

each time it iterates, there may not be enough iterations in some cases to generate as rich a set of repairs as is done in one step by EITHER. On the other hand, as the number of training examples grows, NEITHER undergoes many more iterations, each computing new repairs in light of any previous refinements. Since EITHER computes its repairs for each example independently, it can miss some interactions which might occur when the refinements are applied to the theory in a particular order. Capturing these interactions may be one reason NEITHER out-performs EITHER with large numbers of examples. In any event, NEITHER produces results very close to EITHER's in a fraction of the time, making it much more suitable for use in an interactive setting such as tutoring.

3.5.2 Accuracy of Repair Test

To illustrate the difference between NEITHER and NEITHER-PCC, a series of tests was run in which a modification was made to a correct rule base and the two systems were trained with a set of examples that were correctly classified by the unmodified theory. Two data points were collected for each system: whether the theory was restored to its correct original form and whether the repair was at least attempted at the correct level of the theory. Statistical significance was again measured using a Student t-test for paired difference of means at the 0.05 level of confidence.

The theory used in this experiment was the rule base for the C^{++} Tutor (see Appendix B). That rule base consists of 27 rules which classify examples into one of 3 categories. A corpus of 100 training examples correctly classified by the theory was generated using the methods outlined in Section 5.2. To ensure relatively complete coverage of the theory, examples were distributed equally among the three categories.

A total of 108 modifications to the rules were generated by creating one of each type of change shown in the taxonomy of propositional Horn-clause errors in Figure 19 on page 37. Thus four changes were made, one at a time, to each rule of the theory. Antecedent addition and deletion was performed by randomly adding or deleting one antecedent. New rules were built by combining one randomly selecting antecedent with the consequent from the original rule. Deleting a rule was straightforward. Each of these four changes was applied to each rule of the theory in turn, and the resulting modified theory given to NEITHER and NEITHER-PCC with the 100 training examples.

The results of the test are shown in Table 1, listed by the four types of changes made to the rules. The top half of the table shows the number of exact repairs found by each system, and the bottom table shows the number of repairs which were at least attempted at the correct level of the theory. In both cases, NEITHER-PCC significantly outperforms NEITHER. While these results are by no means comprehensive, they are certainly a positive indication of the value of the parent-child comparison algorithm.

3.6 Summary

The NEITHER algorithm has been described as an extension of the EITHER propositional Horn-clause theory-refinement system. The major revision made to implement NEITHER is a new method for computing repairs that vastly reduces execution time without sacrificing accuracy. Second, a new algorithm for finding repairs at different levels of the theory was described and shown to be more effective than the leaf-rule bias. The result is a modeling algorithm fast enough to be used in an interactive setting and capable of detecting errors at any rule in the theory.

Correct Repairs		
Change Type (27 of each)	*NEITHER-PCC*	*NEITHER*
Added Antecedent	23	14
Deleted Antecedent	19	13
Added Rule	23	16
Deleted Rule	10	6
Overall Accuracy	69%	45%

(a)

Repairs at Correct Level		
Change Type (27 of each)	*NEITHER-PCC*	*NEITHER*
Added Antecedent	27	18
Deleted Antecedent	19	13
Added Rule	23	17
Deleted Rule	18	14
Overall Accuracy	80%	57%

(b)

TABLE 1 Performance of NEITHER with and without the parent-child comparison algorithm. Values indicate (a) number of repairs exactly correct or (b) at least at the correct level.

CHAPTER 4

Bug Library Generation and Use

Perhaps the most unique feature of the ASSERT algorithm is its facility for automatically constructing a library of misconceptions, typically called a *bug library*, without any previous information about the errors students are likely to make. In Chapter 2, a brief overview of this process was presented along with a introduction as to how the library could be used to enhance ASSERT's student modeling capabilities. In that overview, several basic principles were presented. First, it was noted that a collection of rule refinements from different student models can be used as the foundation of a bug library. On top of this, one can add subsets of the refinements which, though not exact for any single student, might nonetheless represent parts of bugs which are common across a range of students. It was also suggested that the bugs can be rated against a hypothetical average or *stereotypical* student, providing a means for ranking the relative utility of the bugs in the library. And finally, a mechanism was introduced whereby the appropriate elements of the library can be merged with the correct rule base presented to NEITHER, thereby giving theory refinement a jump start on the modeling process.

The ability to automatically collect a library of bugs yields several distinct advantages. It can save the author a great deal of time over building a bug library by hand, or at least may help in pointing out likely areas in the domain where misconceptions may occur. It can also help the author to refine the rule base used to represent the domain by pointing out problem areas common to many students. In addition, it enhances the overall modeling capability of the system by providing a uniform mechanism whereby both common misconceptions and those unique to a

particular student can be captured by the modeler. And finally, since the library can be built incrementally, the ability of the system to accurately diagnose errors can continue to improve over time.

The purpose of this chapter to describe precisely how ASSERT constructs and uses its bug libraries. To that end, the first sections will describe how the refinements from NEITHER are collected and how *generalizations* are formed in the search for useful subset of refinements. This will include a definition of the notion of the *stereotypicality* of a bug and how that measure is used to rank the entries of the library. The second half of the chapter will then cover how the bugs which are applicable to a particular student are selected to be merged with the correct rules before those rules are passed off to theory refinement for final modeling.

4.1 Building the Bug Library

As indicated in the overview of Chapter 2, the raw materials for the bug library are the sets of rule refinements, output from NEITHER, which constitute the various student models. Having selected this format, the question becomes how to construct the bug library. The most obvious approach would be to simply list all the rule changes found across the student population, eliminating any duplicates. There are two reasons why this is unsatisfactory. First, since one function of the library is to supply the author with information as to which parts of the rule base are problematic it is important to provide some notion of which bugs are the most prevalent. Thus there must be some means for comparing bugs in the library. This can be important during modeling as well when NEITHER is faced with a choice among bugs in the library: the more common bugs are more likely to be correct and should therefore be preferred. Second, collecting just the rule changes fails to capture any similarities that might exist among bugs which alter the same portion of the rule base but are not identical. For that one needs some method of generalizing between rule changes.

Such generalizations, though not present in any particular student model, could nonetheless capture parts of rule changes which frequently occur. Section 4.2 describes how such bug generalizations are formed.

In any event, there must be a method for ranking the bugs and bug generalizations entered into the library. A simple approach would be to use the frequency of a bug's occurrence in the student population, but this will not work for bug generalizations since they never occur in a student model. What is needed is a method for ranking any kind of rule change, regardless of how it is formed. The solution used by ASSERT is to measure the extent to which a bug is *stereotypical*.

4.1.1 Stereotypicality

Recall that a student model is a set of rule changes which, when added to the correct rule base, will simulate the behavior of the student. Furthermore, each rule change amounts to a set of literal additions or deletions to a rule. So given a hypothetical space of all possible literal changes which can be made to the correct rule base, one can plot the various student models based on their number of literal additions and deletions as shown in the left half of Figure 28. In this figure, the multi-dimensional space of all possible literal changes is shown compressed into two dimensions, where the y-axis represents the space of all possible literal deletions and the x-axis represents the space of all possible literal additions. Given this notion of plotting rule bases in this multi-dimensional space, one can measure the *distance* between any two models by counting the literal additions and deletions required to *transform* one model into another [Wogulis and Pazzani, 1993; Mooney, 1994]. This is similar to the notion of Hamming distance used to measure the difference between binary vectors, and accounts for both the number and location of the differences. Thus in Figure 28, the correct rule base, which has no literal changes, is shown at the origin denoted *CR*. Four different student models are shown plotted at various

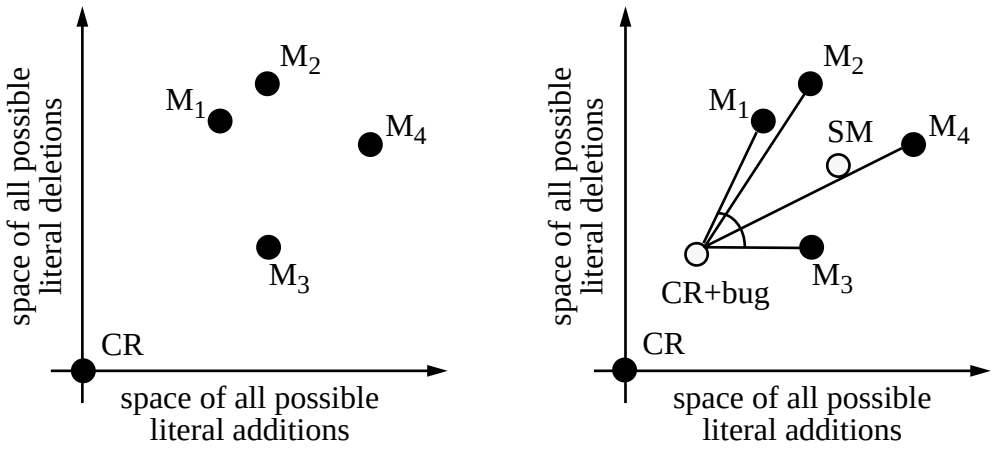


FIGURE 28 Model distance plot. "M" stands for student model, "CR" is the correct rule base and "SM" is the stereotypical student model.

points. Models M_1 and M_2 are fairly close together, indicating that they share many literal additions and deletions in common, whereas the more solitary M_3 and M_4 are each quite different from the other models. And finally, since M_3 is the closest to *CR* this indicates that it is the model with the fewest changes.

Even without actually plotting the various models, the notion of distance between models is useful. With it, the *stereotypicality* of a bug can be defined as the *change in the sum of the distances* between a hypothetical model containing just that bug and all the student models under consideration. More precisely, the stereotypicality of a bug can be defined as

$$\sum_{m \in M} distance\,(CR, m) \quad - \quad \sum_{m \in M} distance\,(CR{+}B, m)$$

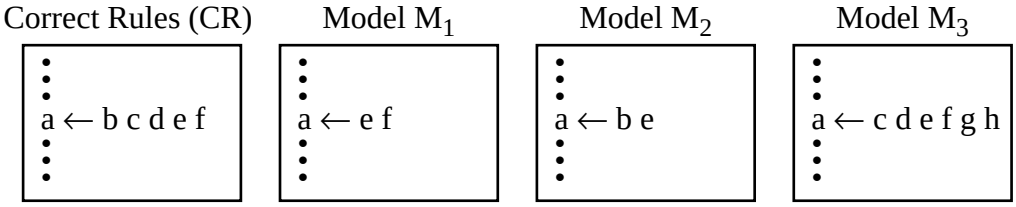
where *CR+B* is a model consisting of just the bug in question and *M* is the set of student models. What stereotypicality measures is the extent to which a given bug moves the correct rule base towards a hypothetical *stereotypical* student model. Let the *stereotypical student model* be defined as that point in the space of all literal

additions and deletions which is the minimum total distance away from all the student models. The right half of Figure 28 shows where the stereotypical student model would occur for the models M_1, M_2, M_3 and M_4.

The more frequently a bug occurs in the models the greater its stereotypicality. This is because the distance between *CR+B* and another model containing the bug *B* is guaranteed to be smaller than the distance between *CR* and the same model. In fact, it is smaller by the number of literals in the bug *B*. Thus the right hand summation decreases in proportion to how frequently the bug *overlaps* with the models. Of course, the opposite is true for less common bugs. Any model in which the bug *B* does not occur will increase the distance between *CR+B* and that model. Stereotypicality, then, is a measure of how much a bug moves the correct rule base towards the stereotypical model when it is applied to the correct rule base. The right half of Figure 28 depicts a bug with positive stereotypicality. The lines between points indicate distance and the arcs show which distances are summed together. Note that the distance between *CR+bug* and *SM* is less than the distance between *CR* and *SM*.

4.1.2 Computing Stereotypicality

An example of the computation for computing bug stereotypicality is shown in Figure 29. Three models are shown at the top of the diagram, each of which changes only one rule of the correct rule base. All the models alter the same rule, but in different ways. So, for example, the first model changes the rule $a \leftarrow b\ c\ d\ e\ f$ by deleting the set of antecedents *{b,c,d}*. The refinement for model M_1, labeled B_1, is thus *delete{b,c,d}*. Below the models are the calculations for determining the stereotypicality of each of the three bugs from the three models. The distance between *CR* and the student models is shown followed by the distances to the models when each of the three bugs is added to *CR*. Calculating the distance between two rule sets amounts to counting the number of literal changes required to convert one to the



B_1 = bug from M_1 = delete{b,c,d}
B_2 = bug from M_2 = delete{c,d,f}
B_3 = bug from M_3 = delete{b} add{g,h}

Distance from Correct Rules to Models:
distance(CR, M_1) = delete{b,c,d} = 3
distance(CR, M_2) = delete{c,d,f} = 3
distance(CR, M_3) = delete{b} add{g,h} = 3
} total = 9

Distance from Correct Rules with bug from M_1 to Models:
distance(CR+B_1, M_1) = 0
distance(CR+B_1, M_2) = delete{f} add{b} = 2
distance(CR+B_1, M_3) = add{c,d,g,h} = 4
} total = 6

Distance from Correct Rules with bug from M_2 to Models:
distance(CR+B_2, M_1) = delete{b} add{f} = 2
distance(CR+B_2, M_2) = 0
distance(CR+B_2, M_3) = delete{b} add{c,d,f,g,h} = 6
} total = 8

Distance from Correct Rules with bug from M_3 to Models:
distance(CR+B_3, M_1) = delete{c,d,g,h} = 4
distance(CR+B_3, M_2) = delete{c,d,f,g,h} add{b} = 6
distance(CR+B_3, M_3) = 0
} total = 10

Stereotypicality(B_1) = 9 - 6 = 3
Stereotypicality(B_2) = 9 - 8 = 1
Stereotypicality(B_3) = 9 - 10 = -1

FIGURE 29 Stereotypicality computation.

other. Thus, changing *CR+B1* into *M2* required changing the rule *a ← e f* into the rule *a ← b e* which is done by deleting *f* and adding *b*. At the bottom of the figure are the stereotypicality values for each of the bugs. Note that a bug may have a negative stereotypicality, indicating that adding it to the correct rules moves away from the stereotypical student model. This is not as bad as it may appear. Unless a bug is

present in more than half of the student models, it is likely to have a negative stereotypicality since there will be no overlap between the bug and the majority of the models. Thus even a bug that occurs with, say, 30% frequency in the student population may have a negative stereotypicality. What is important is the relative difference between stereotypicality values.

As a computational note it should be pointed out that, strictly speaking, it is not necessary to compute the actual distance between rule bases to calculate stereotypicality. Instead, the difference between distances can be calculated and the sum of the *differences* taken. This is an easier calculation since adding a single refinement to a rule base changes only one rule. So, for example, let D be the distance between the correct rule base and a model. Adding a bug to the correct rule base changes D by altering a single rule. The *change* in D, ΔD, can be calculated by finding the overlap between the refinements of the bug and any refinements made to the *same* rule in the model. Whatever is in the bug that overlaps with the model decreases the distance to the model because it means those literal changes already exist in the model. Anything in the bug which does not overlap increases the distance because those literal changes are not in the model. So, for example, the ΔD between $CR+B_1$ and M_1 is -3 because all of B_1 overlaps with M_1. The ΔD between $CR+B_1$ and M_2 is -1 because *c* and *d* overlap (-2) but *b* does not (+1). The fact that *f* is deleted in M_2 is irrelevant since it has to be deleted from both CR and $CR+B_1$ to yield M_2. Finally, the ΔD between $CR+B_1$ and M_3 is 1 because *b* overlaps (-1) but *c* and *d* do not (+2). The total ΔD is thus -3 + -1 + 1 = -3. Multiplying by -1 yields the stereotypicality of B_1. The computational complexity of this algorithm is linear in the number of literals in the refinements of the models.

Using stereotypicality as a metric has two important advantages. It can be used as an indication of how commonly a rule change occurs in the student population,

and it is a measure of how much work it would take NEITHER to convert between models. This makes it ideal for ranking the bugs in the library for use by the author or by NEITHER. Bugs which occur frequently among the students will result in higher stereotypicality since there is no cost to add a bug to a model which already contains it. By contrast, rare bugs will incur higher costs since the literals of the bug have to be accounted for in the target model. High stereotypicality bugs are also preferable in modeling because they indicate changes which are more likely to occur in the average student. Additionally, because stereotypicality is measured in literals it is directly related to how much work NEITHER might save by adding the bug to the correct rule base.

4.2 Generalizing Across Bugs

As mentioned above, constructing a bug library should have a facility for finding generalizations among the rule refinements of the student models. Generalization is important for finding commonalities among refinements which may be similar, but not identical. For example, in Figure 29, both B_1 and B_2 delete a common subset of antecedents from the same rule; namely, the subset *{c,d}*. Without the ability to extract such a subset, ASSERT would be severely limited in its ability to extract trends in student behavior.

Fortunately, a straightforward technique exists for forming generalizations among refinements. Since any refinement to a propositional theory can be expressed as a logic clause, one can compute generalizations using the *least general generalization* (LGG) operator [Plotkin, 1970]. When two propositional logic clauses are not identical, one can form a generalization of the two by dropping literals from the clauses. Any number of literals may be omitted, but the most specific (i.e., least general) way to generalize the two clauses is to drop only those literals which appear in just one of the two clauses. This is the same thing as taking the intersection between

the two clauses. Since refinements in NEITHER are collections of propositional logic literals, the LGG of two refinements is simply their intersection. Note that the result of forming the LGG of two refinements is also a refinement, which can be used in subsequent LGG operations.

Figure 30 shows the LGG's formed among the bugs from Figure 29. Listed with each is the stereotypicality of the resulting subset, if any is formed. Thus the intersection of bugs B_1 and B_2 is the refinement *delete{c,d}* which has a stereotypicality of 2. As might be expected, the LGG will often form a generalization that has better stereotypicality than a refinement from which it was taken. For instance, LGG(B_1, B_2) beats the value for B_2 which is 1. Likewise, LGG(B_1, B_3) is better than B_3 alone. This will be the result whenever the LGG operation captures more of what is common among the models, and avoids more of what is uncommon, than the refinement used as its input. Often times, the intersection operator accomplishes just this trick. However, note that the LGG is not beneficial in all cases; the same two LGG's mentioned above are both worse than the B_1 refinement alone, even though both use B_1 as an input. The process can even be continued, forming LGG's from LGG's, which can also result in better or worse refinements. Consequently, searching for the best set of generalizations to add to the bug library can quickly explode into a combinatorial problem.

4.3 The Bug Library Algorithm

ASSERT collects the refinements from multiple student models, combines them together using the LGG operator, and avoids combinatorial search by concentrating on finding a the best generalization it can for each refinement. The algorithm used by ASSERT borrows ideas presented in the GOLEM system for learning logic programs from examples [Muggelton and Feng, 1990]. The fundamental idea is to perform a hill climbing search using successive LGG operations. Starting with each refinement

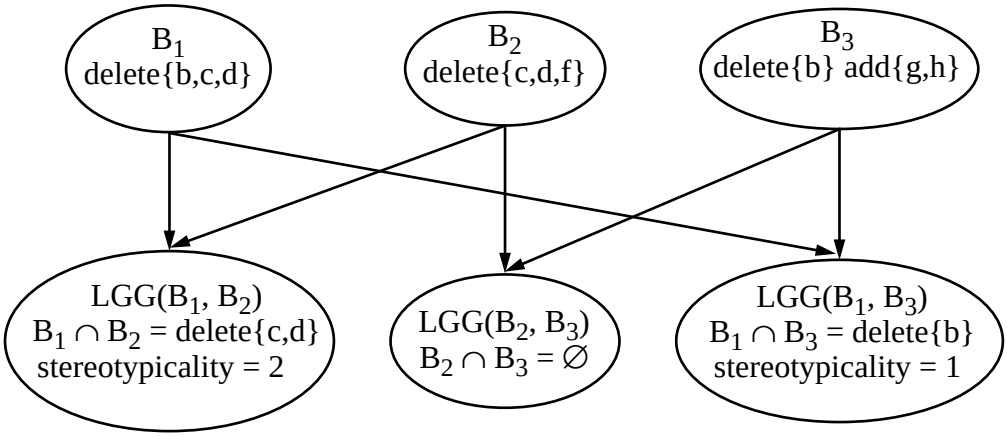


FIGURE 30 Bug generalization using the LGG operator.

as a seed, multiple calls are made to the LGG operator to combine the seed with all the other refinements from the models. As long as this continues to result in a better generalization, successive passes are made over all the refinements. After each pass of the iteration, the best generalization is kept and used as the seed for the next loop. The process halts when no generalization can be found which will improve upon the seed, which must eventually happen since continued intersections between refinements will eventually produce no change or the null set. The best generalization found starting with each refinement as the initial seed is kept and inserted into the bug library. Any duplications in the library are eliminated and the results sorted by stereotypicality. The pseudocode for this algorithm is shown in Figure 31.

The *BuildBugLibrary* routine essentially consists of two nested loops. The outer loop executes once for each unique refinement from a student model, and the inner loop continues as long as an LGG can be constructed which improves upon the best LGG found to that point. Figure 32 illustrates how a complete bug library is constructed. The top of the diagram shows the three bugs from the three models of Figure 29 plus a fourth bug from another hypothetical model which also makes a

```
function BuildBugLibrary (M:list of student models): bug library;
begin
   R = ∅;
   for m ∈ M do add refinements of m to R avoiding duplicates;
   for r ∈ R do begin
      best = r;
      S = Stereotypicality(best);
      repeat while S continues to increase begin
         B = r;
         for r ∈ R do add LGG(B, r) to B;
         BestLGG = b ∈ B with greatest stereotypicality;
         if Stereotypicality(BestLGG) > S then begin
            best = BestLGG;
            S = Stereotypicality(best);
         end;
      end;
      add best to bug library;
   end;
   return bug library sorted by greatest stereotypicality;
end
```

FIGURE 31 Pseudocode for bug library construction.

change to the same rule of the theory. This fourth bug was added to illustrate some of the subtleties of *BuildBugLibrary.*

Below the four bugs are a series of boxes, each representing one iteration of the outer loop of *BuildBugLibrary.* Thus the first box is the iteration which computes the bug to be added to the library starting with B_1 as a seed, the second starts with B_2 as the seed, et cetera. After saving the stereotypicality of B_1 in the temporary variable *S*, the inner loop is entered and an LGG is formed between B_1 and the other three bugs. Note that there is no need to compute LGG(B_1, B_1) since the result is simply B_1 which obviously cannot be an improvement. Once the LGG's are computed, *BestLGG* is found, in this case LGG(B_1, B_4), which has a stereotypicality of 4. This is compared with the current value of *S*, and since there is no improvement the inner loop halts and adds B_1 to the bug library. The second box, for bug B_2, also yields no

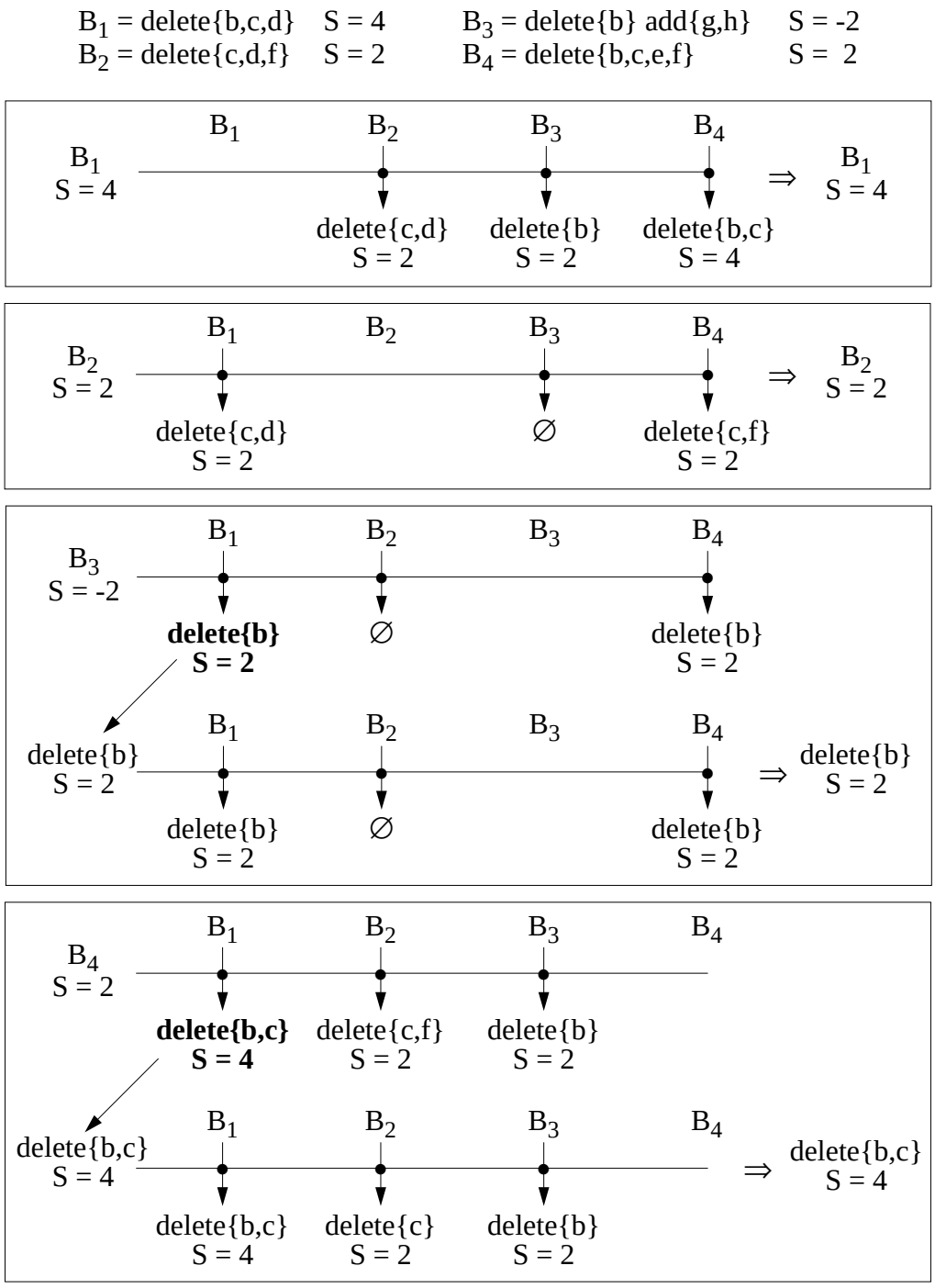


FIGURE 32 Bug library construction example. “S” stands for stereotypicality.

improvement from generalization, resulting in B_2 being added unchanged to the bug library.

The next two boxes representing the iterations of the outer loop for B_3 and B_4 are more interesting. For B_3, *BestLGG* results from combining B_3 and B_1. The generalization is *delete{b}* whose stereotypicality value of 2 is an improvement of 4 points over the value for B_3 alone. Consequently, the inner loop repeats, after setting *best* to *delete{b}* and updating *S* to 2. A second round of LGG's produces no further improvement, resulting in the addition of the generalization *delete{b}* to the bug library. For bug B_4 the process is similar. A first round of LGG's produces an improvement which cannot be surpassed by a second iteration of the inner loop.

However, two interesting LGG's are formed during the computation for bug B_4. One is LGG(B_4, B_1), which was computed and rejected in the top box where B_1 was the seed, but turns out to be a useful improvement over B_4 alone. The other is an LGG formed in the second iteration of the inner loop that cannot be constructed directly from the initial set of bugs. After *best* is set to LGG(B_4, B_1), the second round of LGG's produces the generalization *delete{c}* which is the same thing as LGG(LGG(B_4, B_1), B_2). Note that there is no binary combination of these three bugs that will yield this generalization. Thus the compounding effects of successive LGG operations are essential for finding common refinements across the student models. The final bug library consists of the following four bugs sorted in the following order: B_1, *delete{b,c}*, B_2, and *delete{c}*.

Of course, the hill climbing heuristic used in *BuildBugLibrary* is not the only way to form generalizations, nor is stereotypicality the only useful statistic that can be collected in a bug library. For example, the number of different models which exhibit a bug can also be important, especially to an author trying to refine the tutor to address the most urgent misconceptions. On the other hand, it is not immediately

obvious how one would calculate the frequency of occurrence for a bug generalization. The main advantage of stereotypicality is that it combines the prevalence of a bug with a measure of how much work might be saved by using the bug during modeling. Any bug with a positive stereotypicality will, if added to the correct rule base, reduce the average cost of converting the rules to a student model by theory refinement.

4.4 Using the Bug Library

Recall that because ASSERT is designed to use theory refinement as a black box, the only means for influencing modeling is to modify the rule base given to NEITHER as input. Using the bug library, then, amounts to determining which bugs from the library should be added to the rules. Since all the bugs are in the form of rule refinements, adding them to the rules is easy. However, the best method for selecting the right combination of bugs for a particular student is not immediately apparent.

A simple approach would be add all the bugs from the library which have a positive stereotypicality to the correct rule base, perhaps by using a hill climbing algorithm similar to that used in *BuildBugLibrary*. As discussed above, this has the advantage of decreasing the average distance between the rules and a student model. The result would be an updated rule base which roughly approximates the performance of the average student; in the ideal case, the updated rules would in fact be the stereotypical student model. An algorithm similar to this was used in previous versions of ASSERT [Baffes and Mooney, 1992; Baffes, 1994].

There are several problems with this strategy. First, only a single starting point is constructed for all students, meaning that for some students the modified rule base will actually be a worse input for NEITHER than the correct rules would have been. An obvious example of this is the student who makes no errors; starting with a

buggy rule base forces NEITHER into making unnecessary repairs. Second, it may be the case that no bugs in the library have a positive stereotypicality, rendering the information in the library totally useless. The only way to avoid these problems is to pick bugs from the library based on the data collected for a given student. Only then is there any assurance that the bugs selected are related to the needs of the individual.

A better criterion for evaluating the bugs in the library as they relate to a particular student can be drawn from an analysis of the goals of modeling. After all, since the bug library is used to enhance the modeling process, the metric used by NEITHER to evaluate a refinement can be applied when selecting a bug from the library. Since the overall goal of modeling is to produce an accurate reflection of the student's behavior, *predictive accuracy* is the most important measure of any change to the rule base. Thus, if adding a bug from the library increases the accuracy of the rule base in predicting the student's answers, then that bug is useful for modeling the student. If two bugs from the library increase accuracy by the same amount, then stereotypicality can be used as a secondary method for preferring one bug over another.

Specifically, for each bug in the library, ASSERT computes an accuracy value by adding the bug to the correct theory and counting how many of the labeled examples are correctly predicted by the modified rule base. The resulting accuracy is then compared to the accuracy of the correct rule base alone to see if adding the bug was an improvement. Since the overall goal of modeling is to produce a set of rules with perfect accuracy, all the bugs which result in improved accuracy are candidates for insertion into the correct rule base. Bugs yielding the same improvement in accuracy are distinguished by using their stereotypicality values.

There is still a question, however, as to what *order* the bugs should be added to the rules. For example, once a bug is added to the correct rule base, the accuracy values for the other bugs are no longer valid since they measure the accuracy of the bug

when added to the correct rules *alone*, not with other bugs added in. In fact, a bug which appears promising can result in a decreased accuracy if added to the correct rule base after another bug. So again, there is a combinatorial problem of selecting the best set of bugs to be added to the rules to achieve the best accuracy. ASSERT's solution to this problem is to again turn to a hill climbing algorithm, adding one bug at a time to the rules. On each iteration, a new accuracy value is computed for all the bugs in the library, and the bug with the most improvement is added to the rules. The iteration halts when no bugs result in an improvement.

There is one additional wrinkle to the algorithm which arises from the notion of statistical significance. Given that the data collected for the student is, by definition, incomplete, the accuracy values for the bugs are only approximate. In fact, any difference between the accuracy values of any two bugs may not be significant, making the bugs in question statistically equivalent. Since the accuracy values for all the bugs are computed using the same set of labeled examples from the student, one can use a paired Student t-test to estimate if the difference in accuracy between any two bugs is statistically significant (using the standard 0.05 level of confidence to indicate significance). Given this estimate, one can select a set of bugs from the library starting with the bug which improves the accuracy of the correct rules the most, and adding any other bugs which also improve the accuracy of the correct rules and are not statistically significantly more accurate than the best bug. From this set, one can then choose the bug with the highest stereotypicality for addition to the correct rule base. This allows the stereotypicality measure to have an influence on which bugs are selected, which is preferable to breaking ties by random selection.

Figure 33 puts all these ideas together into the pseudocode used by ASSERT to add bugs to the correct rule base. The goal of this function is to build the modified rule base used as input to the NEITHER algorithm (see Figure 11 on page 23). *ModifyRules* starts with the correct rule base, and loops as long as a bug can be found

```
function ModifyRules (CR:correct rule base,
                      E:labeled student examples,
                      L: bug library): modified rule base;
begin
  R = CR;
  repeat as long as R is updated do begin
    A = ∅;
    for b ∈ L do begin
      if Accuracy(R+b, E) > Accuracy(R, E) then add b to A;
    end;
    if A ≠ ∅ then begin
      best = x ∈ A with best accuracy value;
      A′ = best;
      for x ∈ A do begin
     if Paired-t-Test best, x) not significant then add x to A′;
      end;
    end;
    if A ≠ ∅ then update R with x ∈ A′ with highest stereotypicality;
  end;
  return R;
end
```

FIGURE 33 Pseudocode for bug library use.

which will increase the accuracy on the labeled examples. The first step of each loop is to find the accuracy of each bug when added to the current set of rules. All those bugs which improve accuracy are saved. Next, the bug which increases accuracy the most is found, and an inner loop is entered to pare down the list to only those bugs whose improvement in accuracy is statistically equivalent to the best bug. Finally, if there are still multiple bugs left, then the one with the greatest stereotypicality value is picked to be added to the current rule base (random selection is used as a final tie breaker). When no bugs can be found that increase the accuracy of the rule base the routine quits returning the most current version of the rules. Once the modified rule base is constructed, it is passed to NEITHER for additional refinement, since the bug library may not contain everything necessary for modeling the student.

As an example of bug library selection, refer to a trace of the execution of the "pre-model-student" routine shown in Figure 34. This function is the implementa-

```
> (pre-model-student *student-examples* *correct-theory*)

-----------------------------iteration 1--------------------------------
Trying to beat accuracy = 80.00
bug 10, Accuracy: 85.00, Stereotypicality:  -38
bug 11, Accuracy: 85.00, Stereotypicality:  -38
bug 12, Accuracy: 85.00, Stereotypicality:  -38
bug 20, Accuracy: 85.00, Stereotypicality:  -38
bug 29, Accuracy: 85.00, Stereotypicality:  -72
bug 34, Accuracy: 85.00, Stereotypicality:  -128

Picked bug 20
  type:  add-ante
  rule:  compile-error <- constant-assigned
  antes: (integer-set no)

-----------------------------iteration 2--------------------------------
Trying to beat accuracy = 85.00
bug 5, Accuracy: 90.00, Stereotypicality:  -32
bug 11, Accuracy: 90.00, Stereotypicality:  -38
bug 12, Accuracy: 90.00, Stereotypicality:  -38
bug 29, Accuracy: 90.00, Stereotypicality:  -72

Picked bug 5
  type:  del-ante
  rule:  constant-assigned <- (pointer constant) pointer-init pointer-set
  antes: (pointer constant)

-----------------------------iteration 3--------------------------------
Trying to beat accuracy = 90.00
bug 29, Accuracy: 95.00, Stereotypicality:  -72

Picked bug 29
  type:  del-rule
  rule:  operator-b-sets <- (operator-b auto-incr)
  antes: nil

-----------------------------iteration 4--------------------------------
Trying to beat accuracy = 95.00
done
```

FIGURE 34 Trace of bug selection from the bug library.

tion of the *ModifyRules* routine used by the C^{++} Tutor. The trace shown is for the same student used as an example in the overview of Chapter 2 (see Section 2.6 on

page 24). The complete bug library, not shown in the trace, consists of 34 bugs taken from the models of 45 students who used the C^{++} Tutor. Each iteration corresponding to the outer loop of *ModifyRules* is separated in the trace by a dotted line. For the first iteration, the original accuracy of the correct rule base is shown as 80%. Each bug in the library is added to the correct rules, and the six bugs which result in an increased accuracy are printed out with their stereotypicality values. Since all the bugs increase the accuracy of the correct rules by the same amount, all are candidates for addition to the rule base (i.e., the paired t-test yields no statistical difference in accuracy among the bugs). Using stereotypicality to break the tie eliminates the last two bugs, but still leaves the first four which have equal stereotypicality values. As a last resort, random selection is used to pick bug 20 as the first to be added to the rule base. Referring back to Figure 15 on page 29, note that bug 20 is the second bug shown in that figure.

Having selected bug 20, its refinement is added to the rule base which now has an accuracy of 85% as shown at the top of the second iteration. Next, all the bugs of the library are applied to this updated theory to check for any further improvement in accuracy. This time only four bugs are found, all of which again result in the same increase in accuracy. Bug 5 is a clear winner based on stereotypicality so it is selected as the bug for this iteration (bug 5 corresponds to the first bug of Figure 15 on page 29). Note that bugs 10 and 34, both of which resulted in improvements during the first iteration, are no longer useful for increasing accuracy. Also notice that bug 5 did not increase accuracy during the first iteration, meaning the addition of bug 20 enabled bug 5 to have its effect. This is a beautiful example of the ordering effects inherent in selecting bugs from the library.

At the beginning of the third iteration, the updated rule base, which now contains the refinements from both bug 20 and bug 5, has an accuracy of 90%. Only bug

29 can still improve upon this accuracy so it is selected as the third bug to be added to the rules. Note that while bug 29 continually resulted in improvements in accuracy, its relatively low stereotypicality prevented its addition to the rule base before this point. Finally, the fourth iteration results in no further improvement.

ModifyRules solves the two important problems discussed above which were present in earlier versions of ASSERT. First, by tying bug selection to the labeled examples taken from the student, *ModifyRules* uses only those bugs which are relevant to modeling the student. Second, using improved accuracy as an additional metric, *ModifyRules* can incorporate even rare bugs with low stereotypicality values. Additionally, using this two tiered method for evaluating bugs means that ASSERT can be readily altered to incorporate other preferences which the author may wish to introduce to the modeling process. By setting stereotypicality values, the author can directly influence which bugs are given preference during modeling.

4.5 Summary

ASSERT's technique for automatically constructing and using bug libraries combines some of the best ideas used in student modeling systems to date. The knowledge inherent in a library of common misconceptions allows the system to track student errors more quickly and more accurately, especially when the library is constructed from a large pool of data and when the amount of input from any single student is relatively low. The presence of a bug library essentially leverages the trends across a large number of students, allowing the system to make an educated guess about a particular student's error. Furthermore, the ability to construct a bug library automatically is a huge advantage over methods which require the library be constructed by hand. Like any other knowledge acquisition technique, the ability to extract useful information automatically can save the author of the system a great deal of time. And yet, there is nothing intrinsic to the format of ASSERT bug libraries

which would prohibit an author from modifying the bugs or adding additional bugs if he or she so desired. Finally, by integrating the use of the bug library with theory refinement, ASSERT can use common bugs when appropriate and yet still model any problems unique to an individual. And since the bug library can be updated as more students interact with the system, the performance of the modeler can continue to improve. No other system combines the ability to construct a bug library automatically with a method for updating the library incrementally and a method for modeling unique errors not already present in the library.

CHAPTER 5 Remediation

The task of remediation, as outlined in Chapter 2, has two objectives: (1) to provide an *explanation* of the correct knowledge related to the errors made by the student along with (2) one or more *examples* as an illustration. Taken together, the explanation and examples are units of remediation which the author of the tutoring system is free to combine as he or she sees fit. The question for ASSERT, and the subject of this chapter, becomes how to select the explanation to be communicated and how to generate appropriate examples. Since ASSERT uses the NEITHER theory-refinement algorithm to perform student modeling, ASSERT must generate an explanation and one or more examples for each rule modification made by NEITHER.

A review of the kinds of changes made by NEITHER reveals two important points. First, each change is made to a single rule of the rule base. Even if NEITHER must alter several rules to account for one mislabeled student example, all the modifications occur at the rule level. Consequently, generating an explanation of the correct knowledge related to a student error amounts to explaining the correct form of the rules which were modified to account for that error. Second, there are four different types of changes that NEITHER can make: it can delete antecedents from a rule, it can add antecedents to a rule, it can delete rules, and it can add new rules. Since each type of change alters a rule in a different way, each will require a different example to illustrate the change. For instance, an example showing why a condition should not be removed from a rule is necessarily different from an example showing why a condition should not be added to a rule.

Consequently, ASSERT's remediation consists of two pieces of information generated for each rule changed in the student model: an explanation of the correct form of the rule, and an example illustrating its use. These processes are described in turn below.

5.1 Explaining a Rule

When a rule base is used to express knowledge about a particular domain, the choice of that representation embodies certain assumptions about how the knowledge is structured (indeed, this is true of any knowledge representation). Given that the object of remediation is to *communicate* this knowledge implies that these assumptions must either be made plain or else are already understood by the recipient to whom the communication is directed. For a rule based representation, this means something about the *deductive* nature of the knowledge may need elucidating. That is, the remediation program must either assume or explain the notion of rule satisfaction and show how a rule's satisfaction leads to a particular category being concluded.

5.1.1 Components of a Rule Explanation

For ASSERT, this means the task of explanation boils down to stating what makes the given rule true and how the rule can be used to prove a particular category. ASSERT accomplishes this in two ways. First, it explains how the conditions of the rule lead to the conclusion of the rule. For a propositional Horn-clause this is simple, and amounts to a straightforward English transcription of all the conditions in the rule. Second, the chain of deductions which leads from the rule to the category is explained by showing how the category is *supported* by each rule of a linear chain of rules taken from the proof tree connecting the rule to the category. Each rule in this chain is explained *backwards*, indicating how the conclusion relies upon each of the

rule 1: **compile-error** ← constant-not-init
- "One way to detect a compilation error is to look for an identifier which is constant but not initialized."
  - "There must be a constant which is not initialized."

rule 2: **compile-error** ← constant-assigned
- "One way to detect a compilation error is to look for an identified which is declared constant and initialized, then later assigned a new value."
  - "There must be a constant which is initialized and later assigned."

rule 3: constant-not-init ← (pointer constant) ∧ (pointer-init false)
- "A constant identifier is uninitialized if it is declared as a constant pointer to an integer but not initialized to the address of any integer."
  - "There must be a pointer to a constant or non-constant integer, and the pointer must ITSELF be constant."
  - "A pointer declared constant must not be initialized."

FIGURE 35 Text for rule explanation. Boldface entries are category names.

antecedents. Starting with the category, ASSERT proceeds back along the chain, from the category to the rule in question, illustrating how the final category depends on the conditions of the rule being explained.

An example may help to make this process clear. Figure 35 shows the first three entries of the C^{++} Tutor rule base that embodies part of the definition of how constants can be used in the language. Each entry includes a rule, an explanation of how the conclusion of the rule relies upon the antecedents, and an explanation for each antecedent (for a complete listing of the C^{++} Tutor rule base, see Appendix B). Hence the third entry contains one more piece of text because its rule has two antecedents, whereas the first two rules have only a single antecedent. Boldface entries indicate rules which define a category in the domain; non-bold rule consequents are intermediate concepts. Taken as a group, these first three entries indicate that one may conclude that an example program contains a compile error if it has a pointer which is constant and not initialized. An example of such a program is shown in Figure 36.

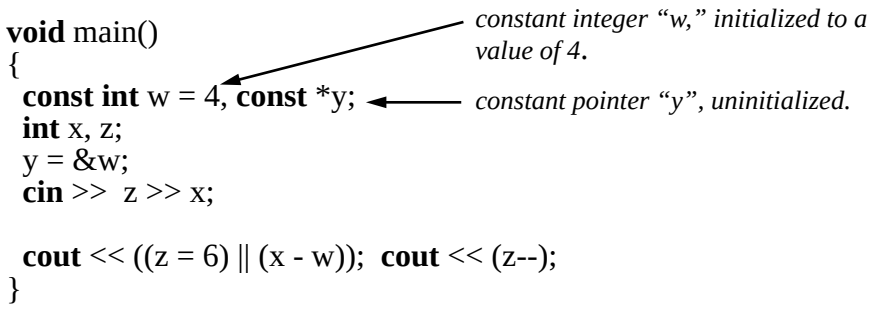


FIGURE 36 Compile error example. Constant pointer is not initialized.

Assume that a student fails to label the program of Figure 36 as a compile error, and that the modeling process determines that a modification to rule 3 of Figure 35 can account for the student's mistake. ASSERT explains the correct use of rule 3 as follows. First, the series of deductions leading from rule 3 to the correct category is be computed yielding the chain of rules *rule 3* → *rule 1* → **compile-error**. This calculation can be performed using a simple recursive descent algorithm, keeping track of the chain of rules until the rule in question is reached. In most cases, all chains leading from each rule to each category can easily be precomputed and saved before the tutorial begins. Next, the chain is traversed in reverse order, printing out the explanation associated with each rule. Lastly, the explanations for each of the antecedents of rule 3 are output. Along with some other canned text for padding, the entire explanation is generated as shown in Figure 37.

5.1.2 Selecting among multiple rule chains

As the experienced reader may have noted, it is possible for a rule to yield multiple chains leading to the same category, or to have more than one category which it supports. ASSERT's solution to this problem is to select randomly from among the possible alternatives. Of course this is not the only solution; one could just as easily justify the notion that all possible alternatives should be explained to the student so

Before proceeding with the rest of the test, let's stop to review some correctly answered examples.

In what follows, you will be shown a set of examples, one at a time. Unlike the test questions, each will be shown with its correct answer. After these examples are presented, the test will be resumed.

EXPLANATION

One way to detect a compilation error is to look for an identifier which is constant but not initialized.

A constant identifier is uninitialized if it is declared as a constant pointer to an integer but not initialized to the address of any integer.

Specifically, note the following which contribute to this type of error:
* There must be a pointer to a constant or non-constant integer, and the pointer must ITSELF be constant.
* A pointer declared constant must not be initialized.

FIGURE 37 Example of a rule explanation.

that he or she will fully understand the complete range of uses for the rule in question. It is important, then, to spell out the reasons for using random selection.

As discussed at the beginning of this chapter and in the overview of Chapter 2, ASSERT's feedback is based on the notion of a unit of remediation. ASSERT provides the most elementary information required, an explanation and an example, which the designer uses to implement his or her own remediation strategy. Hence, the main emphasis of ASSERT is not on how to generate feedback, but rather on how to model the student's actions. Yet, one still needs to generate some form of remediation as a default to test whether the modeling has an impact upon student performance. In fact, in order to test the modeler effectively, one needs a standardized, *quantifiable* unit of remediation. Then, two tests can be run, one with the modeler and one without, both of which generate the same *amount* of remediation. The only difference between such tests is the *kind* of feedback selected, which is determined by the model. If a difference is detected, the cause can be attributed directly to the model.

Random selection is just the simplest strategy for which a controlled amount of remediation can be generated.

5.2 Generating an example

Having generated an explanation for the modified rule, the second objective of ASSERT's remediation algorithm can be addressed: generating an appropriate example to illustrate the rule just explained. As already discussed, this depends upon which of the four types of changes used by NEITHER was made to the rule. Before getting into each of these types, however, it is important to describe the basics of generating an example. While each of the four types of rule changes made by NEITHER requires a slightly different form of example, there are several common parameters that can be used to define an interface for example generation. Once the interface for example generation is defined, and the general algorithm for building an example is described, the specifics of generating an example for each type of NEITHER refinement can be disclosed.

5.2.1 An Interface for Example Generation

To begin with, note that each example is accompanied by an explanation of the rule which is refined. As discussed above in Section 5.1.2, a rule may be used in the proof of more than one category, and may even participate in multiple different chains of deductions supporting a given category. Since the explanation for the rule selects only a single category supported by a particular chain of rules, it is important for the generated example to match these selections as closely as possible. It makes no sense, for instance, to follow an explanation of how a rule supports one category with an example of that rule proving another category. It would be equally confusing to generate an example that proved the same category but used an entirely different chain of rules. The example generation algorithm must therefore accept the follow-

ing two parameters: the category which the example is to prove and the chain of rules that the example must satisfy.

Furthermore, note that each of the four refinements made by NEITHER alters the rule base by either adding or removing something. Viewed from a Horn-clause perspective, what gets added or deleted is a set of literals. Since ASSERT uses a proposition Horn-clause representation, the literals which are added or deleted are propositions. Consequently, the implication for example generation is that when an example is constructed, certain propositions may have required truth values related to whether they were added or deleted from the rule. Of course each type of refinement is different, and will be explained in detail below, but consider the following example. Suppose a rule were changed by the addition of an antecedent. A simple way to generate an example illustrating why the extra antecedent is superfluous is to construct an example which satisfies the original rule *but not the extra antecedent*. To do this, the example generation algorithm must also receive information as to which propositions are to be constrained to true or false values.

The interface for example generation thus has four parameters: the category which the example must prove, the rules which must be satisfied, the propositions which must be true and the propositions which must be false. The next section describes the general example generation algorithm followed by sections outlining how each type of refinement uses the algorithm to construct an appropriate example.

5.2.2 The Basic Example Generation Algorithm

At the simplest level, example generation is an inherently recursive process whose goal is to find a set of feature-value pairs that will guarantee the proof of a particular category. During the selection of feature values there can be many points where the algorithm may pick among several possible alternatives. For example, to

guarantee a rule will be unsatisfiable, one need only force a single antecedent of the rule to be false. There are at least two ways to deal with choosing among such alternatives. One way is to exhaustively check all possible combinations of feature-value assignments until a correct feature vector is found. The disadvantage of this approach is that it can take a very long time to conclude that there is no possible combination of feature-value pairs which will prove the category since checking all combinations can be exponential in the size of the theory.

A simpler approach is to select randomly when choosing among alternatives, and check for compliance after a feature vector is found. If this check fails, the selection process is repeated until success or until some predetermined number of iterations is exceeded. While this latter technique is not guaranteed to find a correct feature vector when one exists, in practice a solution is typically found within a small number of iterations. Furthermore, because random selection is used in place of checking all possible combinations, the time required to select a feature vector is linear in the size of the theory (instead of exponential). One can thus control how long the algorithm should take trying to find a solution before giving up. This becomes essential when response time is an issue.

ASSERT uses this iterative method for generating example feature vectors, picking randomly when faced with alternatives and defaulting to failure after a fixed number of attempts. The four subroutines which comprise the bulk of the algorithm are outlined in Figure 38 and Figure 39. The *TrueProp* and *FalseProp* routines are each passed a *proposition* that the caller wants to be true or false, respectively. The proposition can be either an antecedent from a rule in the theory or a category. The task of these two routines is to select feature-value pairs which will cause that proposition to have the desired truth value. The *TrueProp* routine also takes a second argument, *SR*, which is a list of all the rules that must be satisfied in addition to the

```
function TrueProp (p:proposition, SR:satisfied rules): feature-values;
begin
  FV = ∅;
  if p is operational then FV = p;
  else begin
    R = set of rules which define p;
    if ∃ r ∈ R and r ∈ SR then begin
      FV = TrueRule(r);
      for r´ ∈ R, r´ ≠ r do add FalseRule(r´) to FV;
    end
    else begin
      randomly select r ∈ R; FV = TrueRule(r);
    end
  end
  return FV;
end

function TrueRule (R:rule, SR:satisfied rules):feature-values;
begin
  FV = ∅;
  for p ∈ R do add TrueProp(p,SR) to FV;
  return FV;
end
```

FIGURE 38 Pseudocode for setting a proposition true.

proposition being true. When faced with a choice, *TrueProp* will preferentially use these rules to select feature values.

TrueProp works as follows. If the proposition is operational, i.e. a given feature for an example rather than one defined by rules in the theory, then the feature-value pair of the proposition is simply returned. Thus, for example, the first antecedent in rule 3 of Figure 35 uses the feature *pointer* which can take on the values *constant*, *non-constant* or *absent* (the latter meaning no pointer is used). To make the proposition *(pointer constant)* true the feature value pair *⟨pointer, constant⟩* is selected.

If, on the other hand, the proposition is non-operational, then it must be defined by one or more rules in the theory. To select among the possible rules, the list of rules *SR*, which must be satisfied in the example, is scanned. If none of the defining

```
function FalseProp (p:proposition):feature-vector;
begin
  FV = ∅;
  if p is operational then return (¬p);
  else begin
    R = set of rules which define p;
    for  r ∈ R do add FalseRule(r) to FV;
  end
  return FV;
end

function FalseRule (R:rule):feature-vector;
begin
  for p ∈ R do begin
    if (¬p) already selected return (¬p);
  end
  for p ∈ R do begin
    FV = FalseProp(p);
    if FV ≠ ∅  then return FV;
  end
  return ∅;  /* failure*/
end
```

FIGURE 39 Pseudocode for setting a proposition false.

rules is a member of *SR*, then a rule is selected at random and a call is made to *TrueRule* to recursively select feature-values to make the antecedents of that rule true. If, however, one of the rules defining the proposition is a member of *SR*, then it is used in the call to *TrueRule* and all the other rules which could be used to prove the proposition are recursively set to be unsatisfied by calling the *FalseRule* routine. Doing this guarantees that only those rules which are designated, and none of their siblings, will be satisfied for the example.

As an example, consider again the misclassified program of Figure 36. Assume that the rules to be satisfied are taken from the chain of rules used to explain rule 3: *rule 3* → *rule 1* → **compile-error**. The value of *SR* is thus the list *(rule 3, rule 1)*. Assume also that *TrueProp* is invoked to prove the proposition *compile-error* true. First, *TrueProp* would determine that *compile-error* is non-operational and defined

Feature vector:
((integer constant) **(pointer constant)** (integer-init true) **(pointer-init false)**
(pointer-set true) (integer-set no) (multiple-operands false)
(position-a left-lazy) (operator-a-lazy OR) (lazy-a-left-value non-zero)
(on-operator-a-side left) (on-operator-b-side left) (operator-a assign)
(operator-b auto-incr))

Corresponding C++ example:

```
void main()
{
 const int w = 4, const *y;
 int x, z;
 y = &w;
 cin >> z >> x;

 cout << ((z = 6) || (x - w));  cout << (z--);
}
```

constant pointer "y", uninitialized.

FIGURE 40 Example of automatic feature vector selection.

by rules 1 and 2 of Figure 35. Since rule 1 is also on the list *SR*, it is selected to be satisfied by a call to *TrueRule*. Rule 2, as a sibling of rule 1, is set to be unsatisfied via a call to *FalseRule*. To satisfy rule 1, *TrueRule* must make all its antecedents true, in this case, the single antecedent *constant-not-init*. A recursive call to *TrueProp* for *constant-not-init* reveals that *constant-not-init* is also non-operational and defined by rule 3. Since rule 3 is on *SR*, another recursive call is made to *TrueRule* for rule 3 which then selects the feature value pairs *⟨pointer, constant⟩* and *⟨pointer-init, false⟩* to satisfy rule 3. This process continues until the final feature vector shown in Figure 40 is constructed.

The *FalseProp* routine is similar in structure to *TrueProp*, in that a check is first made to see if the proposition passed in is operational. If so, then a feature-value pair is selected to make the proposition false. Thus to make the *(pointer constant)* proposition false, either the *non-constant* or *absent* value would be selected at random, yielding either *⟨height non-constant⟩* or *⟨height absent⟩* as the feature-value pair returned. When the proposition is non-operational, again the list of rules which

```
function GenerateExample (C:category,
                          SR: satisfied rules,
                          FP:false propositions,
                          TP:true propositions):feature-vector;
begin
  repeat for some predetermined number of iterations
    FV = ∅;
    for p ∈ FP do add ¬p to FV;
    for p ∈ TP do add p to FV;
    add TrueProp(C, SR) to FV;
    for C´ ∈ categories, C´ ≠ C do add FalseProp(C´) to FV;
    FleshOutExample(FV);
    if CheckProof(FV, C) then return FV;
  end /* repeat */
  return ∅;
end
```

FIGURE 41 Pseudocode for the general example generation algorithm.

define the proposition are found. However, this time all the rules which define the proposition must remain unsatisfied to make the proposition false, which is done by passing each in turn to the *FalseRule* routine. To make a rule unsatisfiable, *FalseRule* need only ensure that one of the antecedents of the rule is false. Since the feature vector is constructed incrementally, it is possible that a value may have already been selected for a feature which will render the rule unsatisfiable. Thus *FalseRule* first checks to see if any such feature exists and quits if one can be found. Otherwise, a second loop is entered which will terminate as soon as an antecedent can be found which can be made false (though shown as a normal "for" loop, ASSERT actually selects randomly from among the antecedents of the rule). If no antecedent can be made false, the routine returns the empty set indicating failure.

Figure 41 puts it all together, showing how a complete feature vector example is generated to prove a given category while satisfying a given set of rules. The *GenerateExample* routine takes the four arguments discussed in Section 5.2.1: a category which the example should prove, a list of rules which should be satisfied, a list of

propositions which should be false for the example, and a list of propositions which should be true for the example. *GenerateExample* starts by adding the propositions which must be true and false to the feature vector. Next, an attempt is made to create an example which will be true in only one category and false for all others. Consequently, one call is made to *TrueProp* for the selected category and all other categories are set false using *FalseProp*. Once completed, all the necessary feature-value pairs have been selected. However, there may be other features yet unassigned. The *FleshOutExample* routine randomly assigns values to any unassigned features. Finally, a call to *CheckProof* runs the example through the theory to verify that, indeed, the example leads to a proof of only the desired category. If successful, the feature vector is returned. If not, then the process is repeated some predetermined number of times until a successful feature vector is found or a failure is returned.

5.2.3 When Example Generation Fails

Recall that the discussion of example generation at the beginning of Section 5.2.2 specifically noted that the process was designed as iterative rather than exhaustive. While the reasons for this approach are well motivated, it still leaves open the question of what happens when an example cannot be found. The solution used by ASSERT is to call *GenerateExample* again after easing some of the constraints which may have caused the failure. The trick is to show what these constraints are and how they can be reduced enough to guaranty that a solution can be generated.

There are basically two types of constraints that can cause example generation to fail. First, it may be the case that the true propositions or false propositions passed to *GenerateExample* interfere with finding a solution for which the given rule is true. A simple illustration of this is the following. Suppose *GenerateExample* is passed these four arguments:

category: category-1
rules satisfied: {category-1 ← A ∧ B ∧ C} (i.e., just one rule)
true propositions: ∅
false propositions: {C}

Obviously, there is no way for both the rule to be true and the proposition **C** to be false, since the rule requires the truth of **C**. One can imagine other similar circumstances where the specification of true and false propositions either kept the rule from being proved or ensured the satisfaction of sibling rules, either of which would cause *GenerateExample* to fail. In both cases, the straightforward solution to the problem is to remove all constraints on true and false propositions. Then, as long as the rule can lead to a solution which proves a single category without using any siblings of the rule, *GenerateExample* will work.

However, it may also be the case that the rule in question cannot be made true without also making one or more of its siblings true, regardless of whether any propositions are preset true or false. Or it may be the case that the given rule, when true, will always support the truth of more than one category. For example, in the following rules:

category-1 ← A
category-2 ← A
A ← B ∧ C
A ← C ∧ B

both problems exist; namely, the third rule cannot be true while its sibling rule is false, and it cannot be used to prove only a single category. In such an event, there is no possible setting of truth values for propositions which will have the desired effect because the theory itself violates the constraints *GenerateExample* is trying to satisfy. ASSERT avoids this problem by making the assumption that the theory does not violate these constraints. While this may seem like an unsatisfactory solution, keep in mind that the theory used by ASSERT is provided by the author designing the tutor-

ing system. Thus any extra time required to ensure that the theory is free of these violations does not effect the tutorial experience of the student. Furthermore, it is relatively simple to write a set of utilities which the author can use to automatically check if either of these two constraints has been violated.

To address a failure by *GenerateExample*, then, ASSERT just calls the routine again, leaving out any true or false propositions. This can be done as many times as needed until a solution is found since it is assumed that a solution must exist. In practice, there has never been an instance where *GenerateExample* was called more than twice, though it is possible due the random selection used in the routine. If stronger guarantees are required, one can simply run *GenerateExample* off-line before any tutoring begins and store one or more examples for each rule in the theory. Then, in the very unlikely event that the second call also results in failure, this list of precomputed examples can be tapped to return the final solution.

5.2.4 Completing the Example Generation

It is important to point out that even a complete feature vector which assures a proof using the desired rule may not be a complete example. This is because the feature vector may not be in a form conducive to communicating with the student. Depending upon the domain, more work may be required to translate the feature vector into a form appropriate for the user. This was the case for the C^{++} Tutor used to test ASSERT. In that domain, a feature vector returned by *GenerateExample* was underspecified, meaning it did not fully outline all the elements for the C^{++} example. Another look at Figure 40 on page 87 shows the difference. The first five feature-value pairs of the feature vector indicate that the example has a constant, initialized integer; a constant, uninitialized pointer; and that the pointer is eventually set to the address of an integer. However, note that the number of integers and pointers is not specified, nor are the variable names selected. Thus among other things, the transla-

tion process from feature vector to C++ example must pick the number and names of the variables, determine which are initialized and set, ensure that at least one constant integer is declared, and that a constant pointer is set to the address of an integer after its declaration (which is done in the statement “y = &w;”).

While the details of how the C++ Tutor maps its feature vectors to full C++ examples will not necessarily apply to other domains, it may be useful to point out a few general principles. The first and most important is that the same iterative approach used by *GenerateExample* can be used in translation. Starting with the feature vector as input, one can randomly select among the various options left unspecified. If contradictions are detected, the translation can be thrown away and the process repeated. This was the approach used by the C++ Tutor. Secondly, it is often the case that one can purposely limit the available options in order to generate a more effective translation. In the C++ Tutor, for example, arithmetic expressions were initially used as one of the options available for setting integer values. Later, however, it was decided that students might spend too much time performing mental calculations to determine integer values which, in fact, were irrelevant to solving the problems. Consequently, the arithmetic expressions were dropped and all integer values were set using either a constant or a simple input statement.

5.2.5 Generating Examples for Deleted Antecedents

Having explained example construction in general, we can now focus on how an example is built for each specific rule change made by NEITHER. Refer again to the partial list of rules for the C++ Tutor shown in Figure 35 on page 79. Assume that the model produced by NEITHER determined that the first antecedent of rule 3 needed to be deleted to account for an error made by the student. After printing an explanation for the rule, the question becomes how to construct an appropriate example. The obvious solution might be to simply generate an example which uses the correct

form of the rule, to go along with the explanation just given to the student. This is readily enough done by making a call to *GenerateExample* passing it *compile-error* as the category and *(rule3, rule 1)* as the list of rules to be satisfied. The drawback of this approach is that it fails to emphasize the *nature* of the change to the rule. In fact, there is no reason to expect that such an example would make any differentiation at all between the antecedents of the rule. What's needed instead is something that illustrates why the first antecedent is important.

To see how this can be done using *GenerateExample*, recall what it means when NEITHER deletes an antecedent from a rule. By making such a change, NEITHER *generalizes* the rule, making it more likely that the rule will be satisfied because fewer conditions are required. If such a change is made to model a student's behavior, then it means the student is also generalizing. Specifically, it means the student is labeling some example in a category to which it does not belong. If one can create an example that looks like something the student might generalize, and make the only mistake be the condition missing from the rule, the result will effectively illustrate the necessity of the antecedent which was deleted by NEITHER. What is needed, then, is an example which cannot be proved as a compile error *only* because the antecedent deleted by NEITHER is false.

Again, an example will help to clarify this process. Assuming the first antecedent is recommended for deletion from rule 3, ASSERT starts by *temporarily removing* this antecedent from the rule. This will allow the construction of an example that satisfies rule 3 *without* meeting this condition. Next, a call is made to *GenerateExample* leaving the *modified* rule in the rule base and passing the deleted antecedent as a false proposition. This forces the antecedent to be false in the example. The result will meet the conditions for membership in the category in every way except for satisfying the antecedent in question. If the student model is correct, this error precisely duplicates the kind of generalization made by the student. The two possible labelings

Here is an example which might appear to be a compile error
but is actually CORRECT:

```
void main()
{
  const int m = 5, p, q, *n;
  n = &m;
  cin >> m >> q >> p;

  cout << ((p | m) && (p <= q));  cout << (--m);
}
```

This example is NOT a compile error because

* The pointer 'n' is declared as a non-constant pointer to a constant integer, so it does not have to be initialized.

FIGURE 42 Deleted antecedent remediation.

for the example can then be contrasted in a presentation to the student which illustrates why the antecedent is required. The text of such a presentation for rule 3 is shown in Figure 42.

The text that surrounds the example is stored in a fashion similar to that used for rule explanations. For each feature-value pair of the domain, ASSERT needs text for explaining when the pair is true in an example and when it is false. Any randomly generated elements such as variable names are pulled from the example. When generating text to go with the example, ASSERT retrieves the text for those antecedents being highlighted. Thus in Figure 42, a canned message contrasting the two labelings for the example is printed followed by the text of the example. Then, since the first antecedent of rule 3 is false, text describing how *(pointer constant)* is false (i.e., the pointer is non-constant) is printed out. The variable name *n* is taken from the generated example, as is the fact that the integer to which *n* points is a constant. In this way, the precise error detected by NEITHER is illustrated directly by the example.

Here is an example which might appear to be correct
but is actually a COMPILE ERROR:

```
void main()
{
 int q = 1, m, n, *const p;
 p = &q;
 cin >> n >> m;

 cout << (++n);  cout << (m = n);
}
```

This example is a COMPILE ERROR because:
* The identifier 'p' is declared as a constant pointer (to a non-constant integer).
* The constant pointer 'p' is not initialized.

FIGURE 43 Deleted rule remediation. Acts as the default form of remediation if a second call is made to *GenerateExample*.

Recall that if *GenerateExample* fails in its first attempt, another call is made without setting any antecedents to a particular truth value. As discussed in Section 5.2.3, this second call can be guaranteed to return a value as long as a *correct* rule base is assumed before calling *GenerateExample*. So if an example could not be generated for the modified form of rule 3 with its first antecedent set false, ASSERT restores the original form of rule 3 and calls *GenerateExample* again. The text for such a case is shown in Figure 43. Here, instead of highlighting only one antecedent the text following the example explains how both antecedents of the rule are true. While this is not as targeted an example as Figure 42, at least the correct knowledge is communicated to the student at the level of detail where the error was detected.

5.2.6 Generating Examples for Deleted Rules

When NEITHER models a student's mistake by deleting a rule, it means the student has made an *specialization* error of omission by failing to label an example in

its proper category. By deleting the rule, NEITHER accounts for all examples which could have used the rule to prove the correct category but were not so labeled by the student. Constructing an example for this kind of error is straightforward. Since no added or missing conditions are relevant, ASSERT simply calls *GenerateExample*, and retrieves text explaining why each of the antecedents of the rule is true. An example of the type of printout generated is shown in Figure 43.

The pseudocode for generating examples for deleted antecedent changes or deleted rule changes is shown in Figure 44. Note that the two routines take different arguments since *DeletedAntes* must have additional information as to which antecedents were deleted form the rule. Each routine also specifies the antecedents that are highlighted in the explanation accompanying the text, indicating whether the truth or falsity of the antecedent is explained. Thus for the explanation of Figure 42, the *DeletedAntes* routine shows how the example is printed first, followed by an explanation of the falsity of the deleted antecedent. Finally, note that the *DeletedRule* routine is called from *DeletedAntes* if its call to *GenerateExample* fails. This is because the default action after such a failure is to generate an example for the rule in its normal form, which is precisely what *DeletedRule* does.

5.2.7 Generating Examples for Added Antecedents

Adding antecedents to a rule is another technique used by NEITHER to model specialization errors made by the student. Here, the student has failed to label an example in its proper category because of an expectation that the extra conditions added to the rule must also be true. Creating an effective counter example is done by generating an example which is true in spite of the fact that the extra conditions are false. ASSERT does this by calling *GenerateExample*, passing the list of extra antecedents as the third argument which ensures the antecedents will be false. In the text after the example the extra antecedents are highlighted, showing that they can be

```
procedure DeletedRule  (C:category, SR:satisfied rules,
                        R:rule modified);
begin
  E = GenerateExample (C, SR, ∅, ∅);
  print out E;
  for x ∈ antecedents of R do print text stating why x is true;
end

procedure DeletedAntes  (C:category, SR:satisfied rules,
                         R:rule modified,
                         DA:list of deleted antecedents);
begin
  for x ∈ DA do temporarily remove x from R;
  E = GenerateExample (C, SR, DA, ∅);
  if E ≠ ∅ then begin
    print out E;
    for x ∈ DA do print text stating why x if false;
  end;
  for x ∈ DA do put x back in R;
  if E = ∅ then DeletedRule (C, SR, R);
end
```

FIGURE 44 Pseudocode for remediating deleted rules and deleted antecedents.

false without effecting the proof of the example in the given category. The correct antecedents values are printed as well, indicating that their truth is all that is required to satisfy the example. Figure 45 shows the text generated for rule 3 when the extra antecedent *(integer constant)* is added to the rule, modeling the erroneous notion that both the pointer and integer must be constant to generate a compile error. As with deleted antecedents, if *GenerateExample* fails on its first try, then a call is made to *DeletedRule* to illustrate the correct use of the rule instead. The pseudocode for remediating added antecedents is shown in Figure 46.

5.2.8 Generating Examples for Added Rules

The final type of change made by NEITHER is to add one or more rules to the theory. As one might expect, this has the opposite effect of deleting rules and is used to model over-generalization errors on the part of the student. Recall that when removing antecedents fails to account for how a student has labeled an example in a

Here is an example which might appear to be correct
but is actually a COMPILE ERROR:

```
void main()
{
  int m, p, q, *const n;
  n = &m;
  cin >> *n >> q >> p;

  cout << ((m | q) || (p %= m));  cout << (p & q);
}
```

The following points BY THEMSELVES are enough to make this example
a COMPILE ERROR:
* The identifier 'n' is declared as a constant pointer (to a non-constant
  integer).
* The constant pointer 'n' is not initialized.

Note that this example is a compile error IN SPITE OF the
following:
* The integer 'm' is declared as a non-constant.

FIGURE 45 Added antecedent remediation.

particular category, NEITHER induces new rules to account for the error and adds them to the rule base. Unlike the other three types of changes, however, it may not be immediately clear how ASSERT can remediate an added rule. After all, if a rule is new to the rule base, then the author of the tutoring system did not anticipate its existence. Thus, there is no stored text showing how the rule supports a particular category, and *GenerateExample* cannot be used to construct an example since it depends upon a chain of rules connecting the new rule to a category.

ASSERT solves this problem by taking advantage of the manner in which new rules are added to the rule base. Specifically, NEITHER only adds new rules when deleting antecedents is unsuccessful. Therefore, each added rule is essentially linked to an already existing rule; namely, the one where NEITHER initially tries to delete antecedents. In fact, when a new rule is induced, NEITHER uses the consequent of the existing rule as the consequent for the new rule. Hence, there is an overlap between

```
procedure AddedAntes  (C:category, SR:satisfied rules,
                       R:rule modified,
                       AA:list of added antecedents);
begin
  E = GenerateExample (C, SR, ∅, AA);
  if E ≠ ∅ then begin
    print out E;
    for x ∈ antecedents of R do print text stating why x is true;
    for x ∈ AA do print text stating why x is false;
  end;
  if E = ∅ then DeletedRule (C, SR, R);
end
```

FIGURE 46 Pseudocode for remediating added antecedents.

the new rule and the original, and this can be used to generate an explanation and as the basis for setting up a call to *GenerateExample*.

ASSERT starts by finding three sets of antecedents: those common to both rules, **C**; those which belong only to the original rule, **O**; and those which belong only to the new rule, **N**. Next, the original rule is modified by temporarily removing any antecedents not in **C**. Then *GenerateExample* is called with the false propositions set to **O** and the true propositions set to **N**. This forces *GenerateExample* to construct a feature vector that satisfies the common antecedents in **C** plus those in **N**, which is equivalent to the new rule, but not the antecedents of **O**, which forces the original rule to be false. So, given the following two rules:

original rule: $A \leftarrow B \wedge D \wedge E$
added rule: $A \leftarrow D \wedge F \wedge B \wedge G$

the sets **C**, **O** and **N** would be:

C = {B, D}, **O** = {E}, **N** = {F, G}

which will generate an example where *B, D, F* and *G* are true and *E* is false.

Printing the text to go with the example emphasizes the antecedents in both set **O** and set **N**. First, text is retrieved for each antecedent in **O**, explaining why it is

Here is an example which might appear to be a compile error
but is actually CORRECT:

```
void main()
{
  const int a = 1, *d;
  int b, c;
  d = &a;
  cin >> c >> b;

  cout << (c = a);  cout << (b - a);
}
```

This example is NOT a compile error because:
*The pointer 'd' is declared as a NON-CONSTANT pointer to a CONSTANT integer. Thus it can be reassigned to point to different integers, but it cannot be dereferenced and set.

Note that this example is NOT a compile error IN SPITE OF the following:
*The pointer 'd' is assigned.

FIGURE 47 Added rule remediation.

false in the example thus keeping the correct rule from firing. Then, each antecedent of **N** is highlighted, indicating that its truth is irrelevant to the proof of the example. Figure 47 shows such a printout for the added rule

constant-not-init ← (pointer-init false) ∧ (pointer-set true)

which, compared to rule 3 of the preceding examples, has one antecedent in common with rule 3, *(pointer-init false)*, and one new antecedent, *(pointer-set true)*. The pseudocode for remediating added antecedents appears in Figure 48. Just as with *DeletedAntes*, in the event of a failure to generate an example, the *AddedRule* routine must restore the original rule in the theory before making its default call to *DeletedRule*.

```
procedure AddedRule  (C:category, SR:satisfied rules,
                      R-orig:original rule, R-new:new rule);
begin
  O = set of antecedents only in R-orig;
  N = set of antecedents only in R-new;
  for x ∈ O do temporarily remove x from R-orig;
  E = GenerateExample (C, SR, O, N);
  if E ≠ ∅ then begin
    print out E;
    for x ∈ O do print text stating why x is false;
    for x ∈ N do print text stating why x is true;
  end;
  for x ∈ O do replace x in R-orig;
  if E = ∅ then DeletedRule(C, SR, R-orig);
end
```

FIGURE 48 Pseudocode for remediating added rules.

5.3 Putting it all together

The *Remediate* procedure, which combines explanation and example generation, is shown in Figure 49. It works in two stages. Initially, all the changes made to the rule base are divided into groups such that all changes made to the same rule are bundled together. Then two nested loops are used to print out the explanation for each rule followed by examples which highlight the errors detected by NEITHER. The outer loop cycles through each rule, printing out the explanation of that rule using the sequence of rules leading from the rule to the category. The inner loop then generates examples for each type of change made to the rule and prints each in turn, using the four routines of Figure 44, Figure 46 and Figure 48. Hence the two goals of remediation outlined at the beginning of this chapter are satisfied: an explanation of the correct knowledge is provided by explaining the path leading from the category back to the rule, and examples illustrating the use of that knowledge are provided.

```
procedure Remediate  (RC:rule changes);
begin
 G = changes of RC grouped by rule changed and sorted;
 for x ∈ G do begin
   C = select a category which can be proved for rule in x;
   SR = linear chain of rules from proof tree connecting rule in x to C;
   Explain rules of SR in reverse order;
   for y ∈ x by importance do
    case (type of change in y)
      deleted antecedents : DeletedAntes(C, SR, rule in y, antecedents deleted);
      added antecedents : AddedAntes (C, SR, rule in y, antecedents added);
      deleted rule : DeletedRule (C, SR, rule in y);
      added rule : AddedRule (C, SR, original rule in y, added rule in y);
   end;
 end;
end
```

FIGURE 49 Pseudocode for the complete remediation algorithm.

5.3.1 Coherence versus Importance

Before wrapping up the description of refinement-based remediation, it is important to examine one part of the algorithm of Figure 49. The method of grouping changes by the rule changed deserves further scrutiny because it touches upon some interesting questions regarding how to generate text for human consumption. This is by no means a new issue in intelligent tutoring systems research [Clancey, 1979; Carbonnel, 1970b; Brown et al., 1975] and has been widely studied by other researchers as well. Even within the context of a relatively straightforward knowledge representation such as a rule base, a variety of questions can arise as to how best to organize the output generated for the student. For example, since ASSERT simulates student misconceptions using four types of changes, one could imagine grouping the output along these four lines. Or, because two of the changes are generalizations and two are specializations, one might organize the output as two groups. Another valid technique is to explain all changes to the same rule together since they relate to the same portion of the knowledge base. And finally, one could use addi-

tional information from NEITHER indicating which changes are the most important in an attempt to explain more serious errors before addressing peripheral ones. Regardless of the approach taken, one must use some form of *presentation knowledge* [Acker et al., 1991] to guide the selection and organization of the output text. Since this crosses into the domain of instructional design, ASSERT does not attempt to dictate which method to use.

However, ASSERT does provide a default method of presentation based on the principles of *coherence* and *importance*. A coherent presentation can be defined as one which groups together explanations that are relevant to the same portion of the knowledge base. This means that changes made to the same rule should be explained together. Thus ASSERT uses knowledge about the fact that the unit of remediation is at the rule level to organize its presentation. One benefit of this approach is that the explanation of a rule, i.e. the text indicating how the category is supported by the rule, need only be explained once. Thus a rule altered by both adding and deleting antecedents would be explained one time and then followed by two examples, one to illustrate the importance of the deleted antecedents and another to show why the extra antecedents are irrelevant. If these two examples were separated, with a change to another rule explained in between, the explanation of how the rule supports the category would be repeated. The result would be a meandering presentation full of redundancies.

The second bit of knowledge used by ASSERT to organize its default presentation is the fact that NEITHER uses a ranking system in determining the changes it makes to the rule base. While the details of this ranking are not important here, it is important to know that the *order* in which the changes are listed is significant. Recall that NEITHER works iteratively, finding the most effective change it can at each step to account for as much of the student behavior as possible. When complete, the list of changes made to the rules is ordered from most to least effective. The most effec-

tive changes account for more of the student's behavior and are therefore the most important changes to explain. This is especially salient when the resources for remediation are restricted. For instance, given a limited attention span on the part of the student, the tutoring system must be sure to explain the most important changes regardless of the rules which are changed.

As one might expect, importance and coherence can conflict. The question is how to effectively combine these two principles into a method for generating text. The approach taken by ASSERT is implicit in the *Remediate* algorithm of Figure 49. First, the changes are grouped by the rule which is changed. Each rule will have one or more changes, and each change can be assigned an integer based on importance as determined by NEITHER. Consequently, each cluster of changes to a single rule can be given an average importance by summing the importance integers and dividing by the number of changes. Sorting the rule-change groups based on average importance provides the basis for combining coherence and importance: all changes to a rule are still explained together, but the rules with the most important average change are explained first. Furthermore, within a given cluster of changes to a rule, examples for the most important changes are generated first. The first step of *Remediate* groups the changes by rule and sorts the groups by average importance of the changes made to the rule. The outer loop enforces coherence by ensuring that all changes to a rule are explained at the same time. And finally, the inner loop guarantees that the examples are generated in order of most important change first.

5.4 Summary

A number of details have been covered in this chapter to describe how ASSERT generates remediation. Yet, it is important to emphasize that remediation is not the main focus of this research. In fact, ASSERT has been purposely designed to avoid the difficult questions of remediation relating to pedagogy, leaving them instead to

the author of the tutorial. The mountain of material just explained is just the support mechanism for a very simple idea: that ASSERT can generate an explanation and an example for every refinement found in the student model. It is up to the author of the tutorial to determine the most effective method for using this information.

CHAPTER 6 Experimental Results

It can be argued that the ultimate test of any tutoring system design is whether or not it results in enhanced student performance. This is especially true for student modeling; if the use of a model cannot significantly impact the educational experience then there is little reason to construct one. As a community we can argue about various designs, but the discussion is largely vacuous without empirical evidence. Furthermore, this evidence must come from experiments involving real students, preferably large numbers of students, so that we can assess the significance of the data.

In this chapter, evidence is presented in support of the claim that the ASSERT design can be used to construct tutorials which *significantly* impact student performance. The bulk of this evidence comes from a test using 100 students who interacted with the C^{++} Tutor briefly outlined in Chapter 2 (see Section 2.6 on page 24). In addition to this evidence, experiments are presented from an artificial domain in which student responses were simulated. The advantage of this simulation domain is that it can be used to substantiate hypotheses about the results of the C^{++} Tutor test.

The rest of the chapter is organized as follows. First the simulation domain is described and an initial test is run to illustrate ASSERT's capabilities and set the context for the results expected from C^{++} Tutor test. Then the data from the C^{++} Tutor test is presented. Lastly, additional experiments are discussed using the simulation domain to illuminate the finer points of using an ASSERT-style tutorial.

6.1 Student Simulation Test

The first simulated student test explores one issue: whether or not the various aspects of an ASSERT system, namely the bug library and the NEITHER theory-refinement component, contribute to accurate student modeling. This is an important "proof-of-concept" issue, since the output of the modeling process is the basis for remediation. An accurate model directly influences the impact of an ASSERT-style tutor on the student; if ASSERT is unable to produce more accurate models than simple techniques such as guessing what the student will do or assuming the student will always be correct, then there is no chance that remediation based on an ASSERT model will have any impact. The expectation is that ASSERT will produce significantly more accurate models.

To test this hypothesis, an ablation test format was used, successively leaving out parts of the ASSERT algorithm to determine their effect on the accuracy of the model produced. There are three different configurations in which ASSERT can be used for modeling. The first, which we label simply "ASSERT," is to use everything available to construct the model. This means referencing a bug library to create a modified theory which is then fed to NEITHER for further refinement. One would expect this method to produce the most accurate models. The second technique, labeled "ASSERT-BugOnly," is to use only the bugs in the library to build the model, and the third method, labeled "ASSERT-NoBugs," is to skip the bug library and use only NEITHER. One would expect ASSERT-BugOnly to outperform ASSERT-NoBugs in those cases where a fairly complete bug library is available and relatively few examples are extracted from the student.

To run an ablation test comparing these three configurations requires four types of data: a rule base defining the correct domain knowledge, a corpus of test problems large enough to serve as both training and test examples for modeling, a set of simu-

lated students to provide “answers” to the test problems, and a bug library. The need for a bug library implies that the test must take place in two phases, where the first phase consists of simulating students solely for the purpose of constructing the bug library. Given the library and a new batch of simulated students, the second phase of the test can perform the actual comparison.

The rule base used for this test comes from an artificial domain which classifies examples into one of twelve types of animal. The full animal theory, shown in Appendix A, is an extension of a set of rules given in [Winston and Horn, 1989]. We chose to work in this domain for historical reasons; earlier versions of ASSERT were tested on these rules using simulated data [Baffes and Mooney, 1992]. Since that time, ASSERT has undergone fundamental changes in its design, making it important to show that the previous results can still be achieved with the new system. Furthermore, the animal classification rule base is rich enough to provide a good test of the three configurations of ASSERT on a variety of potential student misconceptions.

The tests were run from a pool of 180 examples randomly generated using the correct animal classification rules (15 examples for each of the 12 categories). Artificial students were generated by making modifications to the correct theory. As each student theory was formed, it was used to relabel the 180 examples to simulate the behavior of that student. These relabeled examples act as the “answers” the student would furnish for the 180 “multiple choice questions.”

Two types of modifications made to the correct theory to create students. One set of modifications was predefined, with a given probability of occurrence. These simulated common errors that occurred in the student population. Six common deviations were used, each with a 0.75 probability of occurrence. Two of these deleted a rule from the theory, two added antecedents to rules in the theory, one added a rule to the theory and one deleted an antecedent from the theory. Note that this covers the

full range of possible modifications that can be made to a propositional Horn-clause theory (see Figure 19 on page 37). To simulate individual student differences, each student theory was further subjected to random antecedent deletions with a probability of 0.10 for each antecedent in every rule.

The first phase of the test, for constructing a bug library, then proceeded as follows. First, 20 artificial students were created using the methods described above. For each student, all 180 examples were relabeled using the student's buggy theory. From these 20 students, 20 student models were generated using NEITHER on all 180 relabeled examples. Finally, bug library was built from the 20 student models using the algorithm from Chapter 4. All of the six common predefined bugs ended up in the bug library. The total size of the bug library was 29 bugs.

For the second phase, 20 new artificial students were generated using the same techniques used in phase one. For each new student, the 180 examples were relabeled using the student's modified theory. Next, 50 examples were randomly chosen from the 180 relabeled by the student to serve as training examples for ASSERT, ASSERT-BugOnly and ASSERT-NoBugs. Each new student was modeled using the same 50 examples as input to each of the three systems. The other 130 examples were reserved for testing the accuracy of each student model. Recall that the output of NEITHER is a revised theory representing the model. This output theory was used to label each of the 130 test examples. These labels were compared to those generated using the student's modified theory to compute a percentage accuracy.

Table 2 shows the results of the simulated student test. For comparison purposes, we also measured the accuracy of both an inductive learner and the correct domain rules. The inductive learner was run by starting NEITHER with no initial theory, in which case NEITHER builds rules by induction over the input examples using a propositional version of the FOIL algorithm [Quinlan, 1990]. For the correct the-

System	*Average Accuracy*
ASSERT	93.7
ASSERT-BugOnly	90.9
ASSERT-NoBugs	86.2
Correct Theory	62.2
Induction	40.8

TABLE 2 Results of simulated student test. Accuracies represent the performance of each modeling system averaged over all 20 "students."

ory no learning was performed, i.e., the correct domain rules were used without modification to predict the student's labelings. Statistical significance was measured using a two-tailed Student t-test for paired difference of means at the 0.05 level of confidence. All the differences shown are statistically significant.

All the predictions are substantiated by the results of Table 2. The ASSERT system performed the best, coming close to perfect accuracy in predicting the student's answers on the 130 test examples. It's not surprising that ASSERT beats ASSERT-NoBugs either, given the fact that the starting theory for ASSERT is the output of ASSERT-BugOnly which is nearly 30 percentage points more accurate than the correct theory serving as ASSERT-NoBugs's starting point. Also, ASSERT-BugOnly is more accurate than ASSERT-NoBugs in this case, indicating that the information in the bug library was complete enough to model the student more accurately than was possible using the 50 examples for the student. That is, on average the input examples for each student were not enough information for NEITHER to overcome the head start in the bug library. Note that this does not mean the 50 input examples were useless; without them ASSERT would not have been more accurate than ASSERT-BugOnly. And finally, notice that induction performs quite poorly in comparison to the other systems and even worse than using the correct theory without learning. This echos the results of other research; namely, when the target concept to be

learned is fairly close to the input rules, theory-refinement algorithms have a definite advantage over induction alone.

The results from Table 2 also set the context for what might be expected from a test involving real students using an ASSERT-style tutor. Ideally, similar results would occur in terms of increases in student performance; that is, the students who received remediation based on a model built by ASSERT should benefit more than students who received feedback based on a model built by ASSERT-NoBugs. Evaluating this hypothesis was the main motivation behind the C++ Tutor test.

6.2 C++ Tutor Tests

As described at the end of Chapter 2 the C++ Tutor was developed in conjunction with an introductory C++ course at the University of Texas at Austin. The tutorial covered two concepts historically difficult for beginning C++ students: ambiguity involving statements with lazy operators and the proper declaration and use of constants. These two concepts plus examples of correct programs formed three categories into which example programs could be classified (for an example of a C++ Tutor problem, see Figure 13 on page 27). A set of 27 domain rules was developed to classify problems, using a set of 14 domain features, as being either *ambiguous*, a *compile error* (for incorrectly declared or used constants) or *correct*. The latter category was the default category assumed for any example which could not be proved as ambiguous or a compile error. For the complete listing of the C++ Tutor rule base see Appendix B.

Students who used the tutorial did so on a voluntary basis and received extra credit for their participation. As an added incentive, the material in the tutorial covered subjects which would be present in the course final exam. This established a high level of motivation among the students who participated in the test. Due to the

large number of students involved, the tutorial was made available over a period of four days and students were encouraged to reserve time slots to use the program. In total, 100 students participated in the study.

Three major questions were the focus of the C++ Tutor test. First, although simulations like the one described in the previous section indicated that NEITHER is an effective modeler, there was still no proof that this would be the case for real students. In particular, all the simulations manufactured *consistent* answers for the simulated students which is probably not be the case for real students. Thus it was important to establish that ASSERT could make a significant difference in constructing an accurate student model for real students. The expectation here was that ASSERT using a bug library and NEITHER would produce more accurate models than ASSERT with NEITHER alone which, in turn, would be more accurate than simply guessing that the student was always correct. Additionally, models built using the full version of ASSERT should have a significant portion of the model contributed from the bug library if the library had any utility.

Second, even with a perfect model one may not see any increase in student performance. Though a model may be accurate in predicting *when* a student will reach a faulty conclusion, it may not be able to predict *how* that conclusion was reached. The only way to determine the efficacy of a model is to provide the student with feedback based on that model and measure any change in performance. Our hypothesis was that remediation generated using models built by ASSERT would result in increased student performance over a control group which received no feedback. Additionally, it was expected that students who were modeled with the benefit of a bug library would see greater performance increases over students who were modeled without a library.

Third, as a comparison against previous student modeling studies [Sleeman, 1987; Nicolson, 1992] we wanted to test how students receiving feedback based on student models would compare against students receiving a simple form of reteaching feedback. In this case the expectation was that remediation based on modeling would result in greater post-test performance than simple reteaching.

Testing these three hypotheses was accomplished with three experiments: one to measure the effects of remediation, another to test the utility of the bug library and a third to measure the accuracy of modeling. In the next three sections each of these tests is described in turn.

6.2.1 Remediation with the C++ Tutor

For the remediation test, students who used the C++ Tutor were divided into four groups. One group received the full benefits of ASSERT, the second used models formed without the benefit of a bug library, the third received reteaching and the fourth was a control group which had no feedback. The expectation was that these four groups would exhibit decreasing performance on a post-test as the remediation ranged from full ASSERT to no bug library to reteaching to nothing.

To test whether ASSERT can impact student performance, one needs to collect information for each student that has certain characteristics. To begin with, data must be collected both before and after any feedback given to the student to detect any change in performance. Thus the C++ Tutor was constructed as a series of two tests with a remediation session in between. Secondly, the data from the two tests must be equally representative of the student's capability and must be collected in similar ways. The only way to detect a transfer of information from the tutoring program to the student is to have both tests address similar topics from the domain at similar degrees of difficulty. One also needs to take into account the physical limits of the

student to construct tests which the student can be reasonably expected to complete during a session with the tutorial.

To that end, a program was written to generate 10 example questions using a prescribed format. Since each question from the C^{++} Tutor can be classified into one of three categories, the 10 questions were divided equally among the categories: three questions were correctly labeled as compilation errors, four were examples of ambiguous programs, and three were questions with no errors. For the ambiguous and compile error categories, example questions were constructed using the *GenerateExample* routine described in Section 5.2 on page 82 with a specific rule from the theory. The four ambiguous questions were generated using rules 13, 14, 16 and 17. The compile error questions were generated using rules 3, 4 and 6. For the three correct examples, a different technique was used since, as the default category, there are no rules to conclude an example question is correct. Instead, *GenerateExample* was called without reference to any particular rule, but with feature values selected so as to keep the example "close" to the other categories. Thus two correct examples were generated which failed to be ambiguous because of one missing condition from rules 16 and 17, and the third had missing one condition from rules 3 and 4. Note that this still leaves quite a few features unassigned for each of the 10 examples. These feature values were filled in randomly.

This process was used to generate two sets of 10 questions representing the pre-test and post-test to be given to each student. The *same* pre-test and post-test was given to every student, but the order in which the 10 questions was presented was randomized. This was done to discourage any sharing of information among students as they used the tutorial. This meant every student answered the same questions, and the only difference was the feedback given between the pre-test and post-test.

Students were randomly assigned to four groups of 25, each of which received a different kind of feedback from the C^{++} Tutor. One group of 25 received no feedback until after the post-test, acting as the control group. This group was labeled the "No Feedback" group. The other three groups were remediated using the methods outlined in Chapter 5. To ensure that the only difference between feedback groups was the *type* of feedback received, each group was given the same *amount* of feedback; specifically, four examples and four explanations for each student.

One feedback group received a form of reteaching. Specifying precisely what is meant by "reteaching" is extremely important, as it can have a profound impact on the results of the experiment. Furthermore, there are many valid approaches to reteaching, making it important to clarify the exact approach used. For this experiment, one overriding concern drove the design. The essential point of the experiment was to illustrate whether feedback based on modeling made any difference over feedback based on no modeling at all. To that end, we chose to isolate all information about the student from the form of reteaching used. Thus, no information about the student was given to the reteaching method, not even which answers the student got right or wrong. In such an informational vacuum, the option left for reteaching is to select information at random from the rule base for remediation. Thus, for the "Reteaching" group, four rules were selected at random from the rule base, and an explanation and example was generated for each.The explanation and example were generated using the *DeletedRule* routine on each rule selected because this routine explains the rule in full and generates an example of its use (see Section 5.2.6 on page 95).

The other two groups received feedback based on the models constructed for the student from his or her answers to the pre-test questions. For one group (the "ASSERT" group) the full ASSERT algorithm was used to build the model and for the other group (the "ASSERT-NoBugs" group) only NEITHER was used as in the

ASSERT-NoBugs technique described in Section 6.1 above; i.e., no bug library was used to modify the correct rule base before passing it to NEITHER. For both modeling groups, bugs were selected for remediation based on the priority assigned them by NEITHER. For the ASSERT group, bugs from the bug library were selected before those found by NEITHER in order of their stereotypicality value. In both the ASSERT and ASSERT-NoBugs groups, if fewer than four bugs were found, the remainder of the feedback was selected at random as with the reteaching group.

Students were assigned to the four groups randomly. Since the ASSERT group required a bug library, the first 45 students to take the tutorial were randomly assigned to the ASSERT-NoBugs, Reteaching and No Feedback groups. All of these students were modeled using NEITHER with the parent-child level-comparison algorithm enabled and the results used to construct a bug library. The remaining 55 students were randomly assigned to all four groups but at three times the rate to the ASSERT group until the number of students assigned to all groups was even.

Since the four groups of students each had a different average accuracy on the pre-test and post-test, they were compared using the average *improvement* in accuracy between pre-test and post-test. Also because each group consisted of different students with no pairing between groups, significance was measured using an ANOVA test. As the only variable between groups was the feedback received, the significance test used was a 1-way unpaired ANOVA test at the 0.05 level of confidence using Tukey's multiple comparison method [Tukey, 1953]. The average improvement in performance for the four groups is shown in Table 3.

The results of the experiment confirmed most of our expectations. As predicted, the average performance decreased as the feedback varied from full ASSERT to no bug library to reteaching to nothing. Moreover, both the ASSERT and the ASSERT-NoBugs students improved significantly more than students in the Reteaching group.

Group	*Average Pre-test Score*	*Average Post-test Score*	*Average Increase*
ASSERT	44.4	67.6	23.2
ASSERT-NoBugs	47.6	67.2	19.6
Reteaching	50.8	58.0	7.2
No Feedback	54.8	56.8	2.0

TABLE 3 C++ Tutor remediation test. Scores indicate percentage of problems answered correctly. ANOVA analysis on average increase results in significance between all groups except between ASSERT and ASSERT-NoBugs and between Reteaching and No Feedback.

For ASSERT-NoBugs, the improvement over Reteaching is more that 10 percentage points, which typically corresponds to an entire letter-grade improvement. For the ASSERT group, the average improvement is even greater. More importantly, in both cases nothing beyond the correct domain knowledge was required from the author of the tutorial. No prior information as to likely misconceptions was available to the system. Both modeling and bug library construction were completely automatic.

It is important to be clear about the results in Table 3. Note that there is a great deal of variance among the mean pre-test scores in the four groups, though these differences are not significant. However, this is precisely why the ANOVA test was run to compare the significance. What can be concluded from Table 3 is that ASSERT-style feedback based on a model of the student can significantly increase performance. There are no claims, however, as to how much increase one will get, whether the increase will always arise for every domain, nor what the performance will be for other forms of modeling or reteaching. What has been illustrated is that the automatic modeling and feedback performed by ASSERT can lead to significant performance improvements over feedback using no modeling at all.

This is the most important empirical result from this research. It illustrates that ASSERT can be used to build a tutorial that significantly impacts student performance

where both models and bug libraries are automatically constructed using only correct knowledge of the domain. Furthermore, it is another argument in favor of the use of student models since it shows (1) that they can have significant impact over not modeling at all and (2) that they can be constructed generically without resorting to hand crafting a library of bugs.

6.2.2 Bug Library Utility Test

However, note that the difference in Table 3 between ASSERT and ASSERT-NoBugs is not significant. This means the use of the bug library did not significantly impact the performance of the student as expected, casting doubts as its utility. Certainly having a bug library did no harm to post-test performance, and perhaps with more data the difference between the two groups would indeed have been significant. Thus it would be useful to know whether the bug library had any impact at all on the modeling process.

In an attempt to understand why ASSERT did not significantly outperform ASSERT-NoBugs, a utility test was run to determine if the contents of the bug library made any difference in the modeling process. Our hypothesis was that if the bug library were useful, the average size of the overall student models from the two groups would not change but the models which were formed using the bug library would require a significantly smaller contribution from NEITHER. If this were not the case, then clearly the bug library could be said to be irrelevant to improved student modeling.

To get at this issue, one simple test is to measure the average *contribution* which the bug library made to the student models in the ASSERT group. This is easily done by subtracting the bugs selected from the library from the overall model built for the student. What remains are the refinements which NEITHER contributed to the model.

Group	*Average Total Literal Changes by NEITHER*	*Average Total Model Size*
ASSERT	3.72	9.08
ASSERT-NOBUGS	8.48	8.48

TABLE 4 Results of bug-library utility test. Values indicate average size of model measured in literal changes to the theory. Differences in changes made by NEITHER are significant; total model size differences are not.

The size of these NEITHER refinements, measured as the total number of literals, can be compared to the models formed for the students in the ASSERT-NoBugs group. This results of such a comparison are shown in Table 4.

Apparently, the bug library is making a significant contribution to the overall model constructed for the students in the ASSERT group. In fact, considering the average total model size for the ASSERT group was 9.08 literals, nearly 60 percent of the average model comes from bugs in the library. This result, taken together with the results of the remediation test, implies that although the bugs in the library are effective at modeling the student, they are not any *more* effective than what NEITHER alone can glean from the 10 pre-test questions.

6.2.3 Modeling Performance using the C++ Tutor

To verify that this was the case requires testing the modeling performance of ASSERT in the C++ domain, checking how the various features of the algorithm impact the predictive accuracy of the resulting models. This is identical to the ablation test used in Section 6.1 which analyzed ASSERT in the simulated student domain. Recall that the hypothesis for that test was that ASSERT models would be more accurate than ASSERT-BugOnly models which are formed using only the refinements stored in the bug library. Furthermore, ASSERT-BugOnly models should be more accurate than ASSERT-NoBugs models only when the bug library can pro-

vide information which NEITHER cannot get from the pre-test. And finally, ASSERT, ASSERT-BugOnly and ASSERT-NoBugs should all outperform the correct theory, which predicts the student is always right, and induction, which must model the student from scratch.

Running this ablation test in the C^{++} domain necessitated using only the data from the No Feedback group. Because no remediation occurred between the pre-test and post-test for the students in this group, their 20 questions could be treated as a single unit from which training set and test set examples can be drawn. However, these training-test splits were generated differently that what was done in the simulated student test. Recall that the examples for the pre-test and post-test in the C^{++} domain were constructed so as to be representative across a given set of domain rules. This representative quality is important to maintain so that any effects from modeling with the training set are manifested in the test set. Therefore, the 20 examples from the pre-test and post-test were grouped into 10 pairs, where each pair consisted of the two examples (one from the pre-test and one from the post-test) which were generated for the same C^{++} Tutor domain rule. Then, training and test set splits were generated by randomly dividing each pair.

The result was 2^{10} possible training-test set splits. For each of the 25 No Feedback students, 25 training-test splits were generated, yielding 625 samples for comparing ASSERT, ASSERT-BugOnly, ASSERT-NoBugs, induction and the correct theory. Each system was trained with the training set (except for the correct theory) and accuracy was measured on the test set by comparing what the system predicted with what the student from the No Feedback group actually answered. Significance was measured using a 2-tailed paired t-test at the 0.05 level of confidence. The results are shown in Table 5.

System	*Average Accuracy*
ASSERT	62.4
ASSERT-NoBugs	62.0
ASSERT-BugOnly	56.9
Correct Theory	55.8
Induction	49.4

TABLE 5 Results for C++ Tutor modeling test. The differences between ASSERT and ASSERT-NoBugs and between ASSERT-BugOnly and the Correct Theory are not significant (all others are significant).

These results highlight why the students in the ASSERT feedback group did not fare significantly better than those from the ASSERT-NoBugs feedback group. The modeling accuracies for the two groups are largely the same. Note also how the order of the systems in decreasing accuracy is different from that of the simulated student test. Here, ASSERT-BugOnly does significantly worse than ASSERT-NoBugs whereas before the bug-library models were significantly better. This is at least part of the explanation why the post-test improvement for the ASSERT group of the remediation test was not significantly better than that of the ASSERT-NoBugs group. On the positive side, it does illustrate that the groups with significantly better models, ASSERT and ASSERT-NoBugs, are precisely the groups which performed best after remediation. This is further evidence in support of the fact that more accurate student modeling can translate directly to improved student performance via more directed remediation.

6.3 Additional Simulation Tests

Of course, this leaves open the question of why the bug library did not improve modeling over NEITHER alone. There are at least three possible reasons. First, it may be the case that there simply are no bugs common across the student population. If

this happens, then a bug library is of little use since there would be no reason to expect that anything stored in the library was likely to apply to a new student. The utility test of Section 6.2.2 seems to have discounted this possibility since it showed that the bugs in the library accounted for nearly 60% of the content of models built using the bug library. Second, it may be the case that not enough student models were used to construct the library, resulting in poor estimate of the true stereotypicality values. In such a case, the bugs applied from the library would be tried in the wrong order, possibly resulting in the mistaken application of a bug. This is unlikely for the C^{++} Tutor tests since 45 student models were used to build the bug library whereas only 20 models proved sufficient for the simulated student test. Unless the common bugs for the domain of the C^{++} Tutor are extremely rare, in which case their value is questionable anyway, it is likely that the bugs found among the 45 students were ranked correctly.

A third possibility is that the range of examples used to model students may be insufficient to produce a wide enough *variety* of bugs for the library. Without enough examples, the common bugs may not even be detected, let alone ranked correctly. Compare, for instance, the 180 examples generated from across all 21 of the domain rules of the animal classification theory to the 10 examples generated for only 7 of the 27 rules of the C^{++} Tutor domain theory. Having more examples per student not only leads to more accurate models, it also means that a misconception for any given rule is more likely to be detected and added to the bug library. It also means that the bug library, over time, can accumulate data on a wider variety of examples than one could reasonably expect to extract from a single student.

The advantage of simulating student responses is that this technique can be used to explore some of these issues related to bug-library construction. In particular, it is easy to vary the number of examples which a simulated student must answer. By

contrast, it is unreasonable to expect a real student to patiently answer 180 questions to test what effect that might have on a bug library.

To that end, three more simulated student tests were run to investigate the hypothesis that the bug library formed for the C^{++} domain had a limited utility because it contained a limited variety of bugs. The first of these tests explores the effect of small student models on bug-library construction. The second examines the impact of a bug library when the data for modeling a student is scarce. The third tests whether a useful bug library can be constructed from a large number of small student models.

6.3.1 Simulating a Bug Library Constructed with Small Models

If the bug library from the C^{++} Tutor tests did indeed suffer from a low variety of bugs, one way this could happen is if the models used to form the bug library were small. A small student model is defined as one which is formed with a scarce amount of data from the student. Small amounts of input mean that only a small potential number of misconceptions are detectable by NEITHER. If a bug library is built from small student models, it in turn is likely to contain fewer of the potential number of common misconceptions which exist in the student population. The hypothesis is that this condition can be simulated by building a bug library using simulated student models built with small amounts of input data. When the resulting bug library is compared with the bug library from the first simulated student test, one should see a drop in the number of common misconceptions which are present in the library.

Recall that the original simulated student test used 20 simulated students answering 180 questions. Each student was simulated using a method which modified the correct rule base with six common misconceptions that occurred with a high probability and additional random changes to make each student unique. To approx-

Library	*Total Students*	*Examples per Student*	*Common Bugs Found*	*Total Bugs in Library*
20-180 library	20	180	all 6	29
20-12 library	20	12	2	15

TABLE 6 Results of the small-model simulation test.

imate the conditions of the C++ Tutor tests, the simulated student test was rerun with the same 20 students but using only 12 of the 180 examples to build small student models that were used to construct a bug library. This number of examples was used to ensure that at least one example per category of the animal domain was present. As before, the first 20 students were modeled with the 12 examples, and a bug library built from their models. For comparison purposes, the total size of the bug library, as well as the number of common bugs which ended up in the library, were compared against the bug library built in the first simulated student experiment. The results are shown in Table 6.

The bug library built with the smaller student models contains less information which is useful to modeling than the large-model library. Notice how both the total number of bugs as well as the number of common misconceptions has dropped. Given that the common misconceptions were actually quite likely to occur in any given student (recall that they had a 0.75 probability of occurrence) it is rather interesting that the majority of the common bugs were not detected. These results clearly show that even though a common misconception may be present in the population, receiving only a small amount of data on each student can mean the misconception may not be detected.

6.3.2 Bug Libraries and Scarce Data

Having established the effects of narrowing the variety of examples on the bug library, the next question is whether or not a large bug library can still make a difference even when input from the student is scarce. There are many situations where one can imagine a tutoring system having to make a decision about remediation with very little input from the student. The hope is that a good bug library can offset such an eventuality by providing the tutoring system with a mechanism for making an educated guess. The expectation is that a "good" bug library, i.e., one built from large student models, should allow ASSERT to construct more accurate models than a small bug library. Thus if the two libraries from the previous test were tested using an ablation study, the library built with larger student models should perform better.

Another ablation test was run using the simulated students, and the same modeling systems as before; namely, ASSERT, ASSERT-BugOnly, ASSERT-NoBugs, the correct theory and induction. To approximate scarce input from the students, only 10 of the 180 examples were used for training each system with the remaining 170 used for testing. Each of the five systems were run twice, once with the large bug library and again with the smaller bug library. Of course, the only systems which were impacted by the differing bug libraries were ASSERT and ASSERT-BugOnly since none of the others uses a bug library. The average modeling accuracy of each system is shown inTable 7.

Notice how the predictive accuracy rises dramatically for the ASSERT and ASSERT-BugOnly systems when the larger library is used. With the smaller library neither of these two systems is significantly more effective at modeling than ASSERT-NoBugs. This result is similar to the effects found in the C^{++} Tutor test; the accuracy of ASSERT was no better than that of ASSERT-NoBugs in that test either (see Table 5 on page 121). Also, note how induction performs dismally with small

System	*Small-Model Library Average Accuracy*	*Large-Model Library Average Accuracy*
ASSERT	68.7	84.8
ASSERT-BugOnly	68.6	84.6
ASSERT-NoBugs	67.6	68.2
Correct Theory	63.1	62.6
Induction	25.4	23.9

TABLE 7 Ablation test results for differing bug libraries and scarce student input. For the small-model library, differences between ASSERT, ASSERT-BugOnly and ASSERT-NoBugs are not significant. For the large-model library, ASSERT-NoBugs is significantly smaller.

amounts of student data, nearly 40 percentage points below just guessing that the student will answer everything correctly (i.e., the correct theory accuracy). These results show that even with scarce input, a large bug library can significantly boost the accuracy of the model.

6.3.3 Incremental Bug-Library Construction

The final question to address is whether an accurate bug library can be constructed over time with low amounts of input from each student. The last section illustrated how a good bug library can have an important impact on the quality of modeling when student data is scarce. However, if useful bug libraries cannot be constructed from small student models, then the result is meaningless since one would still be tied to collecting large amounts of data on some students to construct the library. While more student data will always result in more accurate *individual* models, it is important to show that a good *collective* bug library can still be built over time using less accurate models as input. The theory behind this idea is that with a large enough student population, random selections of questions covering different parts of the correct rule base would eventually result in the detection of whatever common misconceptions exist in the population. The assumption is that a large

number of students answering a small number of questions will exhibit approximately the same degree of common misconceptions as a pool of students answering a large number of questions.

To test this hypothesis, a final student simulation test was run to construct a new bug library for an ablation test. This time, however, the bug library was constructed from 100 simulated students modeled with 12 of the 180 possible examples. For each student, the 12 examples were *randomly* selected for building that student's model. This point is important, since the only way to ensure that all possible misconceptions could be exhibited in the limit is to allow for complete coverage of the rules in the correct knowledge base. Thus, over the entire 100 students the 180 examples were equally represented. Predictive accuracy was again measured using an ablation test with the same students from the original simulation and training sets of size 10. The size of the bug library and the number of common bugs it contained were also determined. The results, combined with those of Table 6 and Table 7 for comparison purposes, are shown in Table 8.

The bug library built using 100 students with 12 examples per student (the 100-12 library) compares favorably against the other two libraries. Though all of the common bugs did not end up in the 100-12 library, 4 of the 6 did, which is much better than the 20-12 library. And while constructing a library from less accurate models cannot replace a library of more accurate models, note how the performances of ASSERT and ASSERT-BugOnly using the 100-12 library approach the marks using the 20-180 library. This illustrates that a bug library can be incrementally improved as more students interact with the system, resulting in more accurate modeling. And as the data from the C++ Tutor shows, more accurate models lead to better remediation and improved student performance. Therefore, as more students interact with the C++ Tutor by taking tests that cover different subsets of the correct domain rules,

Library	*Total Students*	*Examples per Student*	*Common Bugs Found*	*Total Bugs in Library*
20-180 Library	20	180	all 6	29
20-12 Library	20	12	2	15
100-12 Library	100	12	4	48

(a)

	Accuracy using different starting Bug Library		
System	*20-12 Library*	*100-12 Library*	*20-180 Library*
ASSERT	68.7	79.4	84.8
ASSERT-BugOnly	68.6	79.9	84.6
ASSERT-NoBugs	67.6	69.8	68.2
Correct Theory	63.1	63.5	62.6
Induction	25.4	26.0	23.9

(b)

TABLE 8 Comparison of bug libraries. Part (a) compares libraries on size and total number of bugs, part (b) compares accuracy of modeling with libraries.

it seems likely that a better bug library could be constructed which would lead to more accurate modeling and, in turn, better post-test performance.

6.4 Subjective Evaluation

6.4.1 Student Response to the Tutorial

Perhaps the most difficult topic to measure objectively is an evaluation of how much students enjoyed using the tutorial and whether they felt the experience was beneficial. For the vast majority of students who used the C^{++} Tutor the response was positive. Many students made an unsolicited effort to express an appreciation for the opportunity to use the tutorial. Several students outside of the experimental group heard about the experiment and asked to use the tutor to refresh their C^{++} skills.

There were even a few students who expressed disappointment that the tutorial did not cover more material

On the negative side, students complained about their inability to back up during the pre-test and post-test to change answers, which was not allowed by the interface. This was especially true during the early part of the pre-test, when students were still familiarizing themselves with the interface. Also students expressed a boredom with redundancies which showed up in the explanations during remediation. Many students had multiple errors detected in similar parts of the theory. As a result, the chains of rules used by remediation to generate an explanation overlapped, resulting in duplications in explanation. Both of these problems could be easily fixed to make the interface more robust.

But perhaps the most important factor responsible for the positive response was the fact that the feedback given to the student avoided negative language as much as possible. Of course, the student was told which questions he or she got wrong, but the explanation for the wrong answers did not focus on the student's mistake. Instead, the explanation described the correct reasoning, followed by an erroneous counter example which looked like something the student might misclassify, and explained why the counter example was wrong. Thus rather than saying something like "here's what you did wrong" the system instead said "here's the right way to do something and, by the way, here's an answer which is wrong for the following reasons." This impersonal style of feedback may have made it easier for the student to accept the tutor's evaluation. And finally, by giving the students a second chance to perform via the post-test, students were able to apply what they'd learned and, in the average case, achieve a much better score. Such a concrete sense of improvement probably also contributed a great deal to the positive student response.

6.4.2 "Correctness" of the Bug Library

In the simulated student tests, the correctness of the contents of a bug library could be measured directly by counting the number of common misconceptions which ended up in the library. With the C++ Tutor domain this is not possible, since there is no a priori information about what the common misconceptions might be. However, one can perform a subjective evaluation with a domain expert to determine whether or not the bugs are "reasonable" explanations of why students made their mistakes. This was done for the C++ bug library (see Appendix C) by consulting the instructor for the course with the result that the bugs did, in fact, appear to make sense. For example, several bugs in the library represented missing conditions, critical to detecting erroneous constant declarations, which the instructor felt students typically forget. The library also contained bugs capturing the notion that students failed to understand when the logical operators "AND" and "OR" were fully evaluated, as the instructor suspected. While this is an admittedly weak evaluation, it does at least illustrate that the bugs which ended up in the library could communicate information about trends in student behavior which made sense to the instructor.

6.5 Summary

To recap, the main result presented in this chapter was that ASSERT was shown to significantly improve student performance in a test involving 100 college level students using a C++ Tutor developed with ASSERT. Furthermore, it was shown that those students for which ASSERT was able to construct significantly better models were the students whose performance improved the most. And while the use of a bug library did not significantly enhance student performance, additional evidence was presented demonstrating the likelihood that the contents of the library would improve over time so as to significantly impact on the modeling process. This empirical evidence supports the two principal claims of this research: (1) that theory

refinement can be used to significantly increase student performance by modeling students using only correct domain knowledge and (2) that a bug library can be constructed automatically over time that can enhance the modeling accuracy of theory refinement.

CHAPTER 7 Related Work

Tutoring systems research has been approached from a wide range of academic pursuits including epistemology, psychology, cognitive science, education, linguistics, anthropology, human-computer interaction and artificial intelligence. In addition, because ASSERT is based on the notion of theory refinement, other theory-refinement algorithms are also relevant to the ideas presented here. A complete review of all these fields is, of course, beyond the scope of this work. Instead the most closely related research from artificial intelligence is compared with ASSERT, and the reader is referred to other sources for further investigation [Wenger, 1987]. Even so, within artificial intelligence tutoring systems approaches still vary widely. As a result, this chapter is divided into topics related to the design of ASSERT, and each is discussed in turn.

7.1 Knowledge Representations for Modeling

Not surprisingly, the underlying knowledge representation language used for constructing a student model has a profound effect upon the expressiveness and capabilities of the system. As a result, a number of approaches have been used, and each has advantages relating to the goals of the system it supports. There is no single best representation; to the contrary, there is evidence to suggest that a variety of different viewpoints on the same knowledge base may be desirable within a single tutor [Stevens and Collins, 1980]. Most systems can be roughly categorized into six styles of representation; each is described in turn below and compared with the theory-refinement representation used by ASSERT.

Overlay Models. This type of model, already discussed in some detail in Section 1.1 on page 4, essentially focuses on modeling a student as a subset of the correct domain knowledge. Early systems typical of this approach are SCHOLAR [Carbonell, 1970b], WEST [Burton and Brown, 1976] and WUSOR [Carr and Goldstein, 1977b]. Other systems have added extensions to the overlay idea, using truth-maintenance to provide a more general mechanism for keeping track of the beliefs about the student's knowledge [Finnin, 1989; Murray, 1991]. The overlay concept is not dependent on any particular knowledge representation per se; nearly every kind of knowledge base can be extended to include a system of marks to indicate evidence that the student knows a given concept. The advantage of the overlay is its simplicity; the elements of the model can be mapped directly on to the knowledge used to engineer the system. The disadvantage, as stated previously, is the restriction placed on the model. Only missing elements of the correct knowledge can be modeled; alternative notions which a student might have cannot be captured. By contrast, an ASSERT-style model can model novel behaviors, resorting to induction if it must to learn new rules to cover previously unseen behavior.

Bug Libraries. This second form of modeling has again been discussed previously, both briefly in Section 1.1 and in more detail in Chapter 3 and Chapter 4. It can be used to classify a large number of systems, encompassing any which attempt to store some form of expected or inferred misconceptions. The classic bug-library work was done by Brown, Burton and VanLehn [Brown and Burton, 1978; Burton, 1982; Brown and VanLehn, 1980], and Sleeman and Smith [Sleeman and Smith, 1981], but a host of other systems can be said to incorporate some form of stored misconceptions [Anderson et al., 1985; Reiser et al., 1985; Rich, 1989; Goldstein and Miller, 1976; Ikeda and Misoguchi, 1993; Lianging and Taotao, 1991; Miller and Goldstein, 1977a; Langley et al., 1984; Quilici, 1989; Soloway and Johnson, 1984]. The idea is a very powerful one, especially if specific responses can be tied to

whatever buggy structures are encoded. Like overlay models, the disadvantage is an absence of flexibility in addressing novel student errors, which is one of the principal contributions of ASSERT.

Model Tracing. As less of a knowledge representation and more of an implementation for the ACT* theory of cognition [Anderson, 1983], model tracing is concerned with the process of keeping track of the cognitive state of the student. John Anderson and his colleagues have implemented several model-tracing tutorials, most notably the GEOMETRY tutor [Anderson et al., 1985] and the LISP tutor [Reiser et al., 1985]. While these systems do contain information which can be used to model different student conceptions that do not conform to the correct knowledge the tutor is trying to convey, such information is provided chiefly to support the tracking process. In fact, Anderson himself has been recently quoted as saying that modeling itself is not the emphasis of the system [Sandberg and Barnard, 1994]. Instead, the goal is to keep the student from straying too far off the correct path by not allowing faulty concepts to develop in the first place. ASSERT obviously takes an alternate view. Though models may well prove to be less useful in situations where the presentation of material can be carefully monitored and controlled, there are other situations where tutorial help can be useful where such rigid assumptions do not apply. One example would be an intelligent help system which must interact with users possessing a variety of conceptual backgrounds. In fact, such systems can greatly benefit from the automatic bug-library extensions provided by ASSERT since the help environment is one in which trends in user misconceptions are precisely the information needed to improve the system over time.

Logic-based Methods. Several systems have turned to a logic-based representation for the student model [Costa et al., 1988; Ikeda and Misoguchi, 1993; Hoppe, 1994]. Here the idea is to use an analytical approach such as deduction or resolution

to search through a rule-base to determine where a misconception lies. Essentially, whenever the rule-base fails to produce a "proof" which mimics the student's actions, the points where the proof fails become candidates for querying the user about his or her beliefs. The idea is very closely related to theory refinement, but with an important distinction. Using theory refinement, ASSERT can operate autonomously, inducing rules by itself when faced with an impasse in following a student's actions. Other logic-based approaches are currently hampered by the fact that they must turn to an oracle to resolve ambiguities. This is not to say that the logic-based approach is inherently flawed; rather, that theory refinement provides an additional capability which allows it to resolve ambiguities when querying the user is not an option. Probably the best approach would be to combine the two methods, querying the user whenever possible and operating without such input whenever necessary. And finally, as techniques for first-order theory refinement such as the work by Richards [Richards, 1992] continue to develop, there is no reason why ASSERT could not be updated to these more expressive representations.

Modelers for Planning Domains. Several lines of research have addressed the problem of modeling in planning domains; specifically, the writing of computer programs [Miller and Goldstein, 1977a; Miller and Goldstein, 1977b; Soloway et al., 1981; Soloway and Johnson, 1984; Johnson, 1986; Murray, 1986]. Here the problem is more complex than either the classification or procedural domains, since one must try to capture information about the student's planning knowledge in addition to any misconceptions the student may have about the steps required to carry out a plan. None of these systems provides the general purpose capabilities available in ASSERT; on the other hand, ASSERT does not provide the rich knowledge representations which are likely to be necessary for modeling students in planning domains. It remains an open question whether ASSERT could be used to satisfy at least some of the modeling requirements for planning problems, or whether theory-refinement

techniques for these more complex domains will become available to extend the range of ASSERT's applicability.

Stereotype Modeling. Finally, the last category of modeling techniques is a knowledge representation generally referred to as *stereotypes*, though the alternate terms *script* and *frame* are equally descriptive [Rich, 1989; Cohen and Jones, 1989; Huang et al., 1989]. The basic idea here is to engineer clusters of related information together, so that as any one datum of the cluster is detected the rest of the stereotype can be incorporated as needed to enhance the response generated for the student. The advantage of this approach is the depth of knowledge available to the tutor. Unfortunately, the construction of stereotypes remains largely a hand-built and time-consuming process. However, it may be possible to use techniques like ASSERT to make inroads into the construction of stereotype information. Since ASSERT already includes a process for detecting trends across students involving individual bugs, there is no reason why it could not be extended to look for bugs that occur in tandem. This way, misconceptions which typically occur together could be detected, and when one bug was found for particular student the system could be alerted to look for the related bugs.

The six-way classification discussed above represents only one of many possible perspectives on student modeling research and is not intended to represent a complete taxonomy. Indeed, several of the systems mentioned cross the boundaries between categories, thus the distinctions are not clear-cut. This particular grouping was selected primarily to highlight the various features of ASSERT in relation to other work in the field. As a final summary, Table 9 shows five features of ASSERT, relating them to the various categories presented above.

	models missing concepts	models mis-conceptions	can infer novel bugs	multi-student input	self-improving
Overlay	x				
Bug Library	x	x			
Model Tracing	x	x			
Logic-based	x	x	x		
Planning	x	x	x		
Stereotypes	x	x			
ASSERT	x	x	x	x	x

TABLE 9 Comparison of ASSERT and other modeling paradigms. “Self-improving” indicates the technique analyzes its own output to improve performance.

7.2 Novel Bug Detection

Recall from Section 1.1 of the introduction that the main disadvantage of modeling with a bug library is that novel student misconceptions are still undetectable by the system. To avoid this shortcoming one needs some kind of learning algorithm or bug generation technique so that the space of possible new bugs can be explored to account for any previously unseen actions. There are three previous approaches which have dealt with this issue of the genesis of bugs, and they are the work most closely related to ASSERT.

The first of these approaches is VanLehn's SIERRA system [VanLehn, 1983] which builds on ideas developed in the DEBUGGY system [Burton, 1982]. In DEBUGGY, student models are formed by introducing incorrect subskills into a lattice of skills called a *procedural network*. When a subskill is missing or incorrect subskill is substituted for a correct one, the result is faulty behavior. DEBUGGY

attempts to model incorrect behavior by searching for faulty subskills via a generate-and-test method using known deviations for the subskills. VanLehn built on these ideas to develop a theory of how misconceptions are formed. He proposes that faulty behavior arises from a bad *core procedure* which the student is following. When the student's procedure fails to work in a given problem, the result is an *impasse* which the student overcomes by constructing a local *repair.* Incorrect core procedures are assumed to arise from incorrect inductions which occur when the examples given to the student do not meet a set of *felicity conditions* to ensure proper induction. VanLehn's *STEP theory* outlines the conditions for instruction which will lead to proper induction on the part of the student so that he or she may avoid the formation of incorrect core procedures. While SIERRA is perhaps the most complete theory to date on how student misconceptions arise, it is intended as a cognitive model of how bugs arise and is thus a simulation of how the student *forms* misconceptions. ASSERT, by contrast, is a simulation of how to *diagnose* the presence of misconceptions, which is the opposite perspective on the same problem. No doubt a framework like STEP theory would enhance the ability of modeling systems like ASSERT to construct more plausible bugs, but to date STEP theory has not been used in this way.

Sleeman et al. [Sleeman et al., 1990] describe two extensions to their PIXIE system, called INFER* and MALGEN, both of which can be used to extend a bug library. PIXIE is a tutoring system designed for the domain of high-school algebra whose goal is to provide appropriate feedback to improve student performance. PIXIE's underlying representation is a state-space paradigm, where the domain theory is a set of operators implemented as rules. Misconceptions which comprise the bug library are encoded as faulty rules termed *mal-rules*. Both INFER* and MALGEN attempt to generate new mal-rules when the student exhibits a problem that cannot be modeled using the mal-rules already in the bug library. The difference

between the two extensions is that INFER* attempts to patch specific faulty student solutions, whereas MALGEN generates and tests new mal-rules by altering rules in the domain theory. INFER* uses the rules it has to work forward from the problem statement and backward from the student's solution as far as it can. The remaining *gap* is filled by inferring a new mal-rule. In MALGEN, formalized perturbation operators are used to change rules in the domain theory. Unlike ASSERT, both systems require a user to discern which new mal-rules are appropriate extensions for the bug library. ASSERT further differs from these two systems in its ability to operate with no prior knowledge of misconceptions and in its ability to combine results from multiple students to learn new common bugs.

Langley and Ohlsson [Langley et al., 1984; Langley and Ohlsson, 1984; Ohlsson and Langley, 1985] describe the ACM system which uses a domain-independent induction algorithm to induce control knowledge for selecting operators to perform addition. Given a set of overly-general operators, the goal is to induce a set of control rules that will generate an operator sequence that produces the same solutions as the student. Each run of ACM starts without knowledge of when operators should be applied and induces the conditions for applying the operator from examples of student problem solving. A path connecting the problem specification to the student's solution is found, and induction is then performed by noting whether each operator lies on or off the solution path. Since ACM starts from scratch, it must spend time modeling both correct and buggy student control knowledge that could be preprogrammed. As illustrated in Chapter 6, this means that ACM will have difficulty producing accurate models unless a large number of examples are supplied by the student. Also, there is no facility within ACM for building in typical student bugs nor for generalizing across different students.

Thus, none of the three systems provides ASSERT's ability to self-improve over time by interacting with multiple students. O'Shea's quadratic equation tutor

	models missing concepts	models mis-conceptions	can infer novel bugs	uses correct domain knowledge	requires an oracle	multi-student input	self-improving
INFER*, MALGEN	x	x	x	x	x		
ACM	x	x	x				
ASSERT	x	x	x	x		x	x

TABLE 10 Comparison of ASSERT, ACM, INFER* and MALGEN.

[O'Shea, 1982] does have such a capability, but it is focused on improving the presentation of material to the student rather than on enhancing the modeling process. In fact, none of the systems discussed in this chapter is self-improving, which makes ASSERT unique among modeling algorithms. Table 9 shows a comparison of ASSERT with INFER*, MALGEN and ACM.

7.3 Empirical Studies

Part of what fuels the debate about student modeling is the existence of conflicting empirical evidence regarding the educational value of feedback using student models. In addition, the lack of a unified approach to testing feedback methods further confuses the issue. For example, in a study by White et al. [White, 1991], various levels of remediation were provided to students in much the same fashion that the experiments with ASSERT altered its feedback. The results showed that changing the level of feedback had no significant impact on student performance, which appears to contradict the results presented in Section 6.2.1. However, the White study did not attempt to provide feedback based on a specific model of the student; in particular, it only varied the *extent* to which a correct explanation was given to the

student and did not attempt to generate a corresponding *example*. Other studies, such as one performed by Tennyson [Tennyson, 1971], have shown that explanation coupled with an example beats explanation alone. This might explain why students receiving feedback from ASSERT did experience a significant increase in performance, contrary to the results found by White et al.

In a more closely related series of experiments, Sleeman [Sleeman, 1987] reported results from an experiment where human instructors provided either model-based feedback or simple reteaching in response to student errors. He found no significant difference in performance. Yet when Nicolson reran similar tests and used a machine tutor to provide the modeled feedback and reteaching, a significant difference was detected [Nicolson, 1992]. Among other observations, Nicolson points out that the added variation among human tutors might account for the difference in results between the two studies. On the other hand, it may simply be the case that what works well for human tutors may not translate to computer tutors and vice versa. Either way, the results presented here corroborate Nicolson's findings; namely, that providing modeling capabilities can significantly impact upon an automated systems ability to effect student performance.

The important conclusion to draw from these kinds of results is that one must be careful to understand a modeling experiment before passing judgement on the field as a total success or failure. It is also incumbent upon the author to make the implications of the experiment clear. This is especially true for the study of student modeling, which is currently undergoing a change in scope and definition. It may be that the very term "student modeling" is now a misnomer. In any event, the questions addressed by the ASSERT approach are as follows: (1) can modeling techniques be used to build effective systems for communicating a set of concepts to a human user, (2) can these techniques be unified under one framework; specifically, can a bug library be automatically constructed and folded into modeling and (3) will the result-

ing system self-improve over time. As the results from Chapter 6 indicate, the answers to the first two questions are both “yes” and the answer to the third is “most likely.”

7.4 Use of Synthetic Students

The use of synthetic students to test and tune a modeling algorithm predates ASSERT. One of the earliest examples of a simulated student was the construction of SYNDIE for developing coaching strategies for the WUMPUS-II project [Carr and Goldstein, 1977b]. Carr and Goldstein found SYNDIE very helpful in debugging various coaching principles, much the same way ASSERT uses simulated students to explain the strengths and limits of its bug-library construction algorithm as discussed in Section 6.3. In both cases, the motivation for the simulation was the same; by altering parameters to modify the simulated student input, one can tune the performance of the tutoring system much more quickly than by using real student input alone.

An alternative approach to student simulation is exemplified by the work done by VanLehn, Jones and Chi [VanLehn et al., 1992]. Here the goal is not to tune any particular tutoring system, but to produce a computational model that can reproduce interesting behaviors observed in real students. Thus rather than using a simulation to produce a variety of behaviors which can exercise a tutorial, the simulation is used to “explain” real student behaviors from a computational perspective. The motivation behind this method comes from at least two sources. First, it enhances the understanding of cognitive principles through the use of computational models which can be rigorously tested against real student behavior. Second, once cognitive principles are codified, it is possible to study ways in which tutoring environments can benefit from such knowledge, especially in terms of new pedagogical principles for helping students avoid misconceptions. This approach was not used in ASSERT,

though it would be interesting to study the mechanisms human tutors use to revise their own opinions of a student's knowledge as a means of enhancing the capabilities of theory refinement.

7.5 Other Tutorial Design Issues

Several other tutoring systems which influenced the design of ASSERT in various ways should be credited. The SOPHIE system [Brown and Burton, 1975; Brown et al., 1975; Brown et al., 1976] was the first ITS to make use of a simulator which could actually perform the tasks which are given to the student. This idea is central to the means by which ASSERT detects misconceptions. By being able to alter a working simulation, theory refinement essentially reprograms its correct knowledge until it matches the student's behavior. While this was not done in SOPHIE, the use of a working simulation is nonetheless of fundamental importance. The groundbreaking work of GUIDON [Clancey, 1979] illustrated that the rules of an expert system used for tutoring must be carefully crafted to include the motivations behind the operations of the rules that will be of principal concern for the student. This is as true for ASSERT as for any other rule-based tutor; the feedback presented will only be useful if the knowledge from which it is drawn is generated at the appropriate level of detail for the student.

The pedagogical ideas present in systems such as WHY [Stevens and Collins, 1977; Collins 1977], BIP [Barr et al., 1976] and WEST [Burton and Brown, 1976] influenced the remediation technique selected for ASSERT, especially in terms of problem generation. While none of these systems automatically generate examples from a modified rule base as ASSERT does, the idea of picking examples based on the needs of the user is present in all of these systems and in many others as well. What makes these three particularly relevant to the work here is that each presents examples in a different way. In WHY, Stevens and Collins go to great lengths to show

how different examples can be used to implement a Socratic style of feedback, prompting the user to think about his or her hypotheses. ASSERT's ability to generate counter examples could be used directly in such a scheme. The BIP system concentrated on selecting problems that would challenge but not overload the student, guiding him or her towards mastery of the subject matter. Again, ASSERT could be used in this paradigm by constructing examples that incrementally show the necessity of conditions in a concept. The fine grained ability to add or remove conditions from examples generated by ASSERT seems ideally suited to this method. And finally, coaching tutors such as WEST often use examples to illustrate concepts or contrast expert behavior with the actions taken by the student. ASSERT's general-purpose techniques for constructing explanations and examples make it suitable for any of these three approaches.

7.6 Theory-Refinement Algorithms

Ongoing research in theory-refinement algorithms continues to explore refinement methods for different representation schemes and problem domains. However NEITHER, like EITHER, has certain advantages over previous theory-refinement algorithms in that it can revise both overly general and overly specific theories. By contrast, some systems can only generalize a theory [Wilkins, 1988; Danyluk, 1989; Whitehall, 1990; Tecuci and Michalski, 1991] and others can only specialize a theory [Flann and Dietterich, 1989; Cohen, 1990]. Additionally, NEITHER can revise intermediate rules in the theory, giving it an advantage over systems like RTLS [Ginsberg, 1990] which cannot revise intermediate concepts. Finally, since NEITHER is a symbolic refinement process, its outputs are readily interpretable, in contrast to systems like KBANN [Towell and Shavlik, 1991] which must rely on a translation process to decode refinements.

However NEITHER has certain disadvantages which limit its applicability to other domains. First, NEITHER has no facility for handling noisy data. Second, NEITHER does not have a facility for representing rule precedence, nor for revising such ordering relationships among rules as is done in the KRUST system [Craw and Sleeman, 1991]. Third, NEITHER cannot revise probabilistic or "evidence summing" rules in which a partial matching of antecedents are enough to satisfy a rule. Initial efforts to extend NEITHER in this direction, resulting in the NEITHER-M-OF-N algorithm [Baffes and Mooney, 1993], have compared favorably with other systems [Mahoney and Mooney, 1993; Towell and Shavlik, 1990]. However, NEITHER-M-OF-N does not employ an induction algorithm which can construct evidence-summing rules, nor have the explanation and example generation elements of ASSERT been extended to cover this extended representation. And finally, NEITHER's knowledge representation contains certain limitations, such as the inability to deal with negation-as-failure and the restriction to propositional Horn-clause rules. Other systems such as FORTE [Richards, 1992] revise first-order Horn-clause theories, and Wogulis has outlined an approach for revising first-order theories which employ negation-as-failure syntax [Wogulis, 1993]. As these line of research mature, the resulting refinement system can be substituted for NEITHER in ASSERT, with the resulting increase in representational power.

CHAPTER 8 Future Work

Like other tutoring systems, ASSERT touches on a variety of research interests. Consequently, there are a number of perspectives from which one can view the work and make suggestions for improvements. These fall principally under three areas. First, the theory-refinement component of ASSERT can be extended to enhance the knowledge representation of the system and increase its applicability to other problem domains. Second, further experiments could be conducted to test the utility of ASSERT for other modeling problems, most notably, for the types of procedural problems which have dominated previous research efforts in student modeling. And finally, ASSERT could be tested more extensively to determine its ability to self-improve and its applicability to other domains. The advantage of these perspectives is that each can be driven empirically, using experimental methods similar to those outlined in Chapter 6.

8.1 Enhancements to NEITHER

NEITHER has several shortcomings which restrict its expressive power and, correspondingly, that of ASSERT as well. The two most important issues are the limits of NEITHER's rule representation and its inability to handle noisy input. Fortunately, ongoing research efforts have resulted in new approaches which can be explored directly through modification or replacement of NEITHER as ASSERT's theory-refinement component.

8.1.1 Expanding NEITHER's Knowledge Representation

NEITHER's restriction to propositional Horn-clause rules is perhaps the most significant barrier to ASSERT's long-term acceptance as a modeling system framework. This is especially true in light of the spread of expert system tools and applications, many of which include features, such as the use of variables, that go beyond a propositional rule representation. As mentioned in the previous chapter, some first-order Horn-clause refinement algorithms have been developed which might prove complete enough to serve as substitutes for NEITHER in ASSERT [Richards, 1992; Pazzani et al., 1991]. Thus an obvious approach to extending ASSERT is to substitute these algorithms for NEITHER and determine what additional changes would be necessary to generate examples and explanations for first-order rules. In all likelihood, this approach would have to be limited to non-recursive rules, as refinement of recursive rules is still an open research question. Also, current first-order Horn-clause refinement algorithms use hill climbing approaches that can lead to local maxima and a resulting theory which is not consistent with all of the input examples. It remains an empirical question to test whether or not ASSERT could produce useful remediation in such a case.

Negation-as-failure is another limitation of NEITHER; specifically, NEITHER is not designed to handle arbitrarily negated propositions. Negated propositions compound the refinement process by inverting the goal of a refinement; generalizations which are negated become specializations and vice versa. Wogulis [Wogulis, 1993] has suggested a method which addresses this problem by combining generalizations and specializations under a unified process that treats all refinements equally instead of separating rule changes into generalization and specialization phases. Implementing these ideas into NEITHER could be done using these ideas along with the uniform NAND graph representation outlined in [Koppel et al., 1994], but it would mean a major redesign of the algorithm. Since the addition of negation-as-failure provides

no additional representational power the work has not been undertaken to date. However, the notation convenience of negated propositions is such that any future change to NEITHER should incorporate this feature.

8.1.2 Allowing for Noise

Several colleagues who have criticized earlier presentations of ASSERT have pointed out the need for a student modeling algorithm to handle noisy inputs. This is essential for the simple reason that students are not always consistent in their answers, particularly when they are unsure of their knowledge. Although ASSERT has been shown to be effective even without this capability, the criticism remains a valid one. Fortunately, previous work on extending the theory refinement to handle noise provides a starting point for incorporating these changes into NEITHER and thus into ASSERT.

The LATEX system [Tangkitvanich and Shimura, 1993] is a theory-refinement algorithm that employs the minimum description length principle [Rissanen, 1978] to revise a knowledge base in the presence of noisy examples. LATEX uses an MDL metric to measure the number of bits required to represent changes to the initial theory plus the number of bits required to classify the examples once the theory is revised. LATEX resolves the problem of whether to modify the theory to account for an example or simply accept the example as noise by attempting to minimize this sum. These ideas could be incorporated into NEITHER's metric for selecting and ranking its refinements. Another technique would be to simply record the number of examples each refinement fixes and only remediate a student on those which account for multiple problems. The latter method amounts to simply adding a parameter to ASSERT, directing it to avoid mentioning the refinements it finds unless there is enough evidence to suspect they are not due to noise. Experiments could then be run similar to those of Section 6.2, comparing full remediation of all refinements found

for the student against a method which only remediated those refinements which accounted for multiple errors.

8.1.3 Refinement using an Oracle

Though NEITHER has been billed as a system which has advantages because it does not rely upon an oracle to make its revisions, the tutoring domain presents a unique environment for harnessing the power of input from a human operator for enhancing the accuracy of modeling. Specifically, one can simply ask the student to resolve choices during refinement by posing additional questions. This is the approach used in the logic-based modelers developed by Ikeda and Misoguchi [Ikeda and Misoguchi, 1993] and Hoppe [Hoppe, 1994]. Moreover, since ASSERT has the ability to generate examples dynamically, it would be a relatively straightforward change to generate questions during refinement. Some additional planning is required to ensure that the student is not overloaded, but since NEITHER can work without the additional input from the student, this original format for computing refinements can always be used as a default.

8.2 Other Modeling Domains

Since the bulk of ITS student modeling has focused on procedural problems, it would be interesting to apply ASSERT to such a domain and see how it fares against previous results. Furthermore, since extensive bug libraries have been constructed for some procedural domains, one could directly test how well ASSERT is able to duplicate such results. As with the C^{++} Tutor tests, extensive empirical results would have to be collected before any claims can be made as to whether ASSERT can be successfully extended to cover a procedural task.

Given ASSERT's current design, there are two basic approaches that one might take as a first step towards applying the architecture to procedural tasks. One method

is to assume that the interface for the task makes all the intermediate steps of the procedure clear. Said another way, this means that all the steps taken by the student from the problem specification to the solution would be made known to the system. Anderson's *model-tracing* tutors are examples of such tutorials [Anderson, 1984]. With this assumption, it may be possible to treat each step of the procedure as an individual concept, where the problem faced by the student is to correctly advance to the next step of the process. Picking the right next step would amount to the correct classification.

There are two open questions which could be problematic for such an approach. First, the domain might not lend itself to revealing the student's decisions at intermediate steps of the procedure. In such a case it is difficult to see how ASSERT could be applied, though it seems difficult to model students with such a paucity of information using any method. Second, it may prove difficult to detect errors where the student performs each step correctly but in the wrong order. It seems likely that the refinement component of ASSERT would have to be modified to check the rules associated with other steps of the process in addition to refining the rules of the correct step to catch this eventuality.

A second method for extending ASSERT to procedural domains is to focus on developing another theory refinement technique which is more appropriate to procedural domains, especially for those domains where a rigid, model-tracing interface is inappropriate and a more open, exploratory method is desirable. One potential theory refinement method which might be applicable to this situation is first-order theory refinement [Richards, 1992]. In this approach, the basic architecture of ASSERT could be used, but much of the internals would need reworking to be compatible with first-order logic rules as opposed to propositional rules.

8.3 Other Empirical Tests

Finally, there are many additional tests that could be run to test ASSERT. Other tests using the C++ Tutor could be run to collect more empirical evidence to support the results reported in Chapter 6. Specifically, larger numbers of students could be tested under more stringently controlled conditions, especially time allowed to perform the pre-test and post-test. Also, it would be interesting to test ASSERT against different styles of reteaching such as those used in previous studies [Sleeman, 1987]. Specifically, one might try a weak form of "modeling" whereby students are given feedback on the questions they got wrong. For example, one could take the questions the student answered incorrectly and explain the rules of the proof for the correct answer to the question. Or, one could simply explain the most common bugs in the bug library. Such tests could illuminate other kinds of modeling which compare favorably with ASSERT. To be fair test, however, it would be important to ensure that any such test generated its models automatically as ASSERT does.

Also, additional classification tutors could be developed to determine if the results found for the C++ Tutor map to other concept learning domains. Furthermore, other remediation techniques could be tried such as the Socratic feedback described in [Stevens and Collins, 1977; Collins, 1977] or the coaching feedback developed for WEST [Burton and Brown, 1976] to see if these are more effective at enhancing student performance. And finally, it would be interesting to see if the technique of constructing bug libraries from large numbers of small student models described in Section 6.3.3 could be realized using real student data. One way to test this would be to continue the C++ Tutor tests over multiple semesters, tracking the modeling accuracy of ASSERT and the post-test performance increases.

CHAPTER 9 Conclusions

The primary result of this research is a new student modeling technique, ASSERT, which can automatically capture novel student errors using only correct domain knowledge, and can automatically compile trends across multiple student models into bug libraries. Student models built with ASSERT have been shown to be an essential component for generating feedback that significantly improves student performance. This chapter reviews the contributions of ASSERT in three different areas: its impact on the student modeling debate, the modular features of the program, and an evaluation of the performance of the algorithm.

9.1 ASSERT as an Argument for Student Modeling

The recent debate over the utility of student modeling has criticized models as being too difficult to construct and of limited use in effecting student performance. The ideas developed for modeling are not the problem per se, it is the techniques required for bringing these ideas to bear on a real domain that is objectionable. To overcome these objections, a modeling system must exhibit two essential features. First, it should be easy to use; specifically, it must be able to successfully model a student without requiring the author of the tutoring system to perform the laborious task of encoding all possible misconceptions a student might exhibit. Second, the models produced should be useful; that is, they should provide information which plays an essential role in significantly impacting a student's performance.

ASSERT achieves both of these goals. It requires *only* the correct knowledge of the domain to construct its models. All bug-library information, as well as any novel

error modeling, is generated completely automatically without necessitating feedback from the author of the tutorial. Since the correct domain knowledge has to be encoded for a tutorial in any event, there is no added expense to construct an ASSERT-style tutorial. Furthermore, ASSERT responds quickly, operating at execution speeds which are linear in the size of the input knowledge base. This makes it possible to apply ASSERT to interactive domains. And finally, while the ultimate utility of student modeling may still be a matter of debate, the results of Chapter 6 illustrate that student modeling via ASSERT can significantly impact student performance in at least some domains.

ASSERT illustrates that the process of generating a useful student modeling component for an intelligent tutoring system need not be a time consuming task; to the contrary, modeling can serve as an extra source of input to an instructor about recurrent student misconceptions. By combining the ability to construct a bug library automatically with a method for updating the library incrementally and a method for modeling novel errors, ASSERT answers the student modeling critics with a system that can positively impact student learning without negatively impacting tutorial design.

9.2 Features of ASSERT

The structure of ASSERT incorporates several key features which contribute to its success and its potential for future enhancement. Chief among these is it modular design. Separating the various components along well defined interface boundaries allows components to be used in different ways and updated as new techniques become available.

The foundation upon which ASSERT is built is its theory-refinement component. Using theory refinement, ASSERT can model any misconception consistent within

the primitives used to define the domain. ASSERT can bring as much knowledge to bear on the modeling process as the author of the tutoring system is willing to encode. ASSERT can even work with no input knowledge, resorting to induction as previous methods have done. And most importantly, the output of theory refinement is of a form which can be easily fed back into the modeling process to improve the accuracy of the resulting model. ASSERT accomplishes this by collecting refinements from multiple students into a bug library from which bugs can be drawn to improve the accuracy of the knowledge base before refinement begins. Thus, ASSERT encompasses three important student modeling principles: (1) it can model the differences between the system's correct domain knowledge and the student's behavior (2) it can make use of a bug library to catch known misconceptions and (3) it can extend its capabilities by modeling novel student misconceptions and by automatically constructing new bug-library entries.

And finally, the remediation component of ASSERT was purposely designed to avoid the difficult questions of feedback relating to pedagogy, leaving them instead to the author of the tutorial. Instead, ASSERT provides just the support mechanism for generating an explanation and an example for every refinement found in the student model. While this simple feedback alone has proved sufficient for improving student performance in the C^{++} domain, the ability to generate salient examples is a powerful feature that can be used in a variety of ways from altering the style of interaction to selecting challenging problems.

9.3 The Performance of ASSERT

Empirical evaluations of ASSERT highlight the advantages of its approach. To begin with, Section 3.5 showed that NEITHER could operate more than an order of magnitude faster than EITHER without sacrificing accuracy. This enables ASSERT to be used in an interactive setting which is important in the design of a tutoring sys-

tem. Next, the subjective evaluation of ASSERT revealed that students who interacted with the system found it both enjoyable and informative. This is absolutely critical to the long term utility of a system since nothing can be accomplished by a tutor which no one likes to use. Students seemed to especially like the three-part presentation of a set of problems, followed by an example which highlighted issues rather than discussing mistakes, followed by a second chance at a similar set of problems. Also, a subjective evaluation of the bug library formed for the C^{++} domain indicated that it captured reasonable misconceptions typically found for students taking that class.

But the most important empirical results were presented in Section 6.2, which showed that students who received feedback based on ASSERT models performed significantly better on a post test than students who received simple reteaching feedback based on no information about the student. While not conclusive evidence of the universal applicability of ASSERT, this result is a clear indication that student models can be constructed which will result in a net positive influence on student performance. The fact that these models were built automatically with only the correct knowledge of the domain as input makes this result the most important finding of this research.

Other results from Chapter 6 indicate that theory-refinement systems such as NEITHER are clearly more effective than inductive techniques, with NEITHER posting a modeling accuracy improvement of 12 percentage points over induction for the C^{++} domain. And while the bug library did not significantly impact overall modeling accuracy, the presence of a bug library did significantly reduce the average number of refinements which NEITHER had to find to model student behavior. Furthermore, additional evidence was presented demonstrating the likelihood that the contents of the library would improve over time so as to significantly improve the modeling process and, correspondingly, the impact of the system's remediation and the performance of the students on a post test.

9.4 Summary

In summary, ASSERT is a shell for constructing student modeling tutors which operate in concept learning domains. It is able to construct student models efficiently and automatically, catching both expected and novel student behavior. It is the first modeling system which can construct bug libraries automatically using the interactions of multiple students, without requiring input from the author, and integrate the results so as to improve future modeling efforts. Finally, the empirical evidence presented supports the two principal claims of this research: (1) that theory refinement can be used to significantly increase student performance by modeling students using only correct domain knowledge and (2) that a bug library can be constructed automatically in an incremental fashion using multiple student models as input.

APPENDIX A

Animal Classification Domain

Domain Features:

feed-young:	(milk regurgitate bring none)
body-covering:	(scales hair feathers moist-skin)
birth:	(live egg)
eat-meat:	(true false)
fly:	(true false)
teeth:	(pointed flat none)
fore-appendage:	(wing leg fin)
foot-type:	(hoof clawed webbed none)
neck-length:	(short medium long extra-long)
body-length:	(small medium large huge)
color:	(white black gray tawny)
pattern:	(spots stripes patch none)
pattern-color:	(black white none)
ruminate:	(true false)

Correct Domain Theory:

mammal ← (body-covering hair)
mammal ← (feed-young milk)
mammal ← (birth live)
bird ← (body-covering feathers)
bird ← (birth egg) fly)
ungulate ← mammal (foot-type hoof)
ungulate ← mammal ruminate
carnivore ← eat-meat
carnivore ← (teeth pointed) (foot-type clawed)

giraffe ← ungulate (neck-length extra-long) (color tawny) (pattern spots) (pattern-color black)
zebra ← ungulate (color white) (pattern stripes) (pattern-color black)
cheetah ← mammal carnivore (color tawny) (pattern spots) (pattern-color black)
tiger ← mammal carnivore (color tawny) (pattern stripes) (pattern-color black)
dolphin ← mammal (fore-appendage fin) (color gray) (body-covering moist-skin) (body-length medium)
whale ← mammal (fore-appendage fin) (color gray) (body-covering moist-skin) (body-length huge)
bat ← mammal (color black) (pattern none) (pattern-color none) fly
platypus ← mammal (birth egg) (foot-type webbed)
ostrich ← bird (not fly) (neck-length medium) (color white) (pattern patch) (pattern-color black)
penguin ← bird (not fly) (color white) (pattern patch) (pattern-color black) (foot-type webbed)
duck ← bird (foot-type webbed) fly
grackle ← bird (color black) (pattern none) (pattern-color none) fly

APPENDIX B C++ Tutor Domain

Domain Features:

pointer:	(constant non-constant absent)
integer:	(constant non-constant)
pointer-init:	(true false)
integer-init:	(true false)
pointer-set:	(true false)
integer-set:	(yes no through-pointer)
multiple-operands:	(true false)
position-A:	(normal left-lazy right-lazy)
operator-A-lazy:	(AND OR)
lazy-A-left-value:	(non-zero zero)
on-operator-A-side:	(left right)
on-operator-B-side:	(left right)
operator-A:	(assign modify-assign mathematical logical comparison auto-incr)
operator-B:	(assign modify-assign mathematical logical comparison auto-incr)

Correct Domain Theory:

compile-error	←	constant-not-init
compile-error	←	constant-assigned
constant-not-init	←	(pointer constant) (pointer-init false)
constant-not-init	←	(integer constant) (integer-init false)
constant-assigned	←	(integer constant) integer-init (integer-set yes)
constant-assigned	←	(integer constant) integer-init (integer-set through-pointer)
constant-assigned	←	(pointer constant) pointer-init pointer-set
ambiguous	←	multiple-operands operands-linked

```
operands-linked         ← operand-A-uses operator-B-sets
operands-linked         ← operand-A-sets operator-B-uses
operand-A-uses          ← operand-A-evaluated operator-A-uses
operand-A-sets          ← operand-A-evaluated operator-A-sets
operand-A-evaluated     ← (position-A normal)
operand-A-evaluated     ← (position-A left-lazy)
operand-A-evaluated     ← (position-A right-lazy) lazy-A-full-eval
lazy-A-full-eval        ← (operator-A-lazy AND) (lazy-A-left-value non-zero)
lazy-A-full-eval        ← (operator-A-lazy OR)  (lazy-A-left-value zero)
operator-A-uses         ← (on-operator-A-side right)
operator-A-uses         ← (on-operator-A-side left) (not (operator-A assign))
operator-A-sets         ← (operator-A auto-incr)
operator-A-sets         ← (on-operator-A-side left) (operator-A modify-assign)
operator-A-sets         ← (on-operator-A-side left) (operator-A assign)
operator-B-uses         ← (on-operator-B-side right)
operator-B-uses         ← (on-operator-B-side left) (not (operator-B assign))
operator-B-sets         ← (operator-B auto-incr)
operator-B-sets         ← (on-operator-B-side left) (operator-B modify-assign)
operator-B-sets         ← (on-operator-B-side left) (operator-B assign)
```

APPENDIX C

C++ Tutor Bug Library

The following is a library of 34 bugs, automatically constructed using the student models of 45 students who interacted with the C++ Tutor. These students were evenly drawn from the three groups which received no feedback: ASSERT-NoBugs, Remediation, and No Feedback. The first 15 students of each group were used.

bug type: del-ante
rule: constant-assigned <- (pointer constant) pointer-init pointer-set
antecedents: (pointer-set)
stereotypicality: -26

bug type: del-ante
rule: lazy-a-full-eval <- (operator-a-lazy or) (lazy-a-left-value zero)
antecedents: ((operator-a-lazy or))
stereotypicality: -30

bug type: add-rule
rule: constant-assigned <- (pointer constant) pointer-init pointer-set
antecedents: (pointer-init)
stereotypicality: -30

bug type: del-ante
rule: operand-a-evaluated <- (position-a right-lazy) lazy-a-full-eval
antecedents: (lazy-a-full-eval)
stereotypicality: -32

bug type: del-ante
rule: constant-assigned <- (pointer constant) pointer-init pointer-set
antecedents: ((pointer constant))
stereotypicality: -32

bug type: add-ante
rule: operand-a-evaluated <- (position-a left-lazy)
antecedents: ((integer non-constant))
stereotypicality: -32

bug type: add-rule
rule: compile-error <- constant-assigned
antecedents: ((position-a left-lazy))
stereotypicality: -34

bug type: del-ante
rule: lazy-a-full-eval <- (operator-a-lazy or) (lazy-a-left-value zero)
antecedents: ((lazy-a-left-value zero))
stereotypicality: -36

bug type: add-rule
rule: ambiguous <- multiple-operands operands-linked
antecedents: ((operator-b mathematical))
stereotypicality: -38

bug type: add-rule
rule: compile-error <- constant-not-init
antecedents: ((not multiple-operands))
stereotypicality: -38

bug type: add-rule
rule: compile-error <- constant-not-init
antecedents: ((lazy-a-left-value non-zero))
stereotypicality: -38

bug type: add-rule
rule: compile-error <- constant-assigned
antecedents: ((lazy-a-left-value non-zero))
stereotypicality: -38

bug type: add-ante
rule: operator-b-uses <- (on-operator-b-side right)
antecedents: ((integer non-constant))
stereotypicality: -38

bug type: add-rule
rule: ambiguous <- multiple-operands operands-linked
antecedents: (operands-linked)
stereotypicality: -38

bug type: add-rule
rule: compile-error <- constant-assigned
antecedents: ((integer non-constant))
stereotypicality: -38

bug type: add-rule
rule: compile-error <- constant-assigned
antecedents: (operator-b-uses)
stereotypicality: -38

bug type: add-ante
rule: operator-a-uses <- (on-operator-a-side right)
antecedents: ((pointer constant))
stereotypicality: -38

bug type: del-ante
rule: operator-a-sets <- (operator-a auto-incr)
antecedents: ((operator-a auto-incr))
stereotypicality: -38

bug type: add-ante
rule: ambiguous <- multiple-operands operands-linked
antecedents: ((pointer constant))
stereotypicality: -38

bug type: add-ante
rule: compile-error <- constant-assigned
antecedents: ((integer-set no))
stereotypicality: -38

bug type: add-rule
rule: ambiguous <- multiple-operands operands-linked
antecedents: ((integer-set no))
stereotypicality: -38

bug type: del-rule
rule: operand-a-evaluated <- (position-a normal)
antecedents: nil
stereotypicality: -44

bug type: del-rule
rule: operand-a-evaluated <- (position-a left-lazy)
antecedents: nil
stereotypicality: -52

bug type: add-rule
rule: constant-assigned <- (integer constant) integer-init
(integer-set through-pointer)
antecedents: (integer-init (integer constant))
stereotypicality: -52

bug type: del-rule
rule: operator-a-uses <- (on-operator-a-side right)
antecedents: nil
stereotypicality: -60

bug type: del-rule
rule: ambiguous <- multiple-operands operands-linked
antecedents: nil
stereotypicality: -66

bug type: del-rule
rule: compile-error <- constant-not-init
antecedents: nil
stereotypicality: -68

bug type: add-rule
rule: constant-not-init <- (integer constant) (not integer-init)
antecedents: ((not integer-init) operator-b-sets)
stereotypicality: -72

bug type: del-rule
rule: operator-b-sets <- (operator-b auto-incr)
antecedents: nil
stereotypicality: -72

bug type: del-rule
rule: constant-not-init <- (pointer constant) (not pointer-init)
antecedents: nil
stereotypicality: -78

bug type: del-rule
rule: operand-a-uses <- operand-a-evaluated operator-a-uses
antecedents: nil
stereotypicality: -102

bug type: del-rule
rule: constant-not-init <- (integer constant) (not integer-init)
antecedents: nil
stereotypicality: -102

bug type: del-rule
rule: lazy-a-full-eval <- (operator-a-lazy or) (lazy-a-left-value zero)
antecedents: nil
stereotypicality: -114

bug type:	del-rule
rule:	constant-assigned <- (integer constant) integer-init (integer-set through-pointer)
antecedents:	nil
stereotypicality:	-128

APPENDIX D

C++ Tutor Raw Student Data

The following lists of numbers represent the raw data collected for the 100 students who participated in the C++ Tutor test. Each group is listed separately, and the data for each student is listed on a separate line.

Each student entry consists of 5 pieces of information listed left to right as follows: pre-test accuracy, post-test accuracy, pre-test answers, post-test answers, size of the bugs used from the bug library (see Appendix C) measured in literals, and the size of the model for the student, also measured in literals.

The correct answers for both the pre-test and post-test are "(C C B B C A A A B B)" which corresponds to the pre-randomization order of both the pre-test and post-test.

"ASSERT" Group: Feedback based on the full ASSERT algorithm.

60	80	(A C B B A C A A C B)	(C C A C C A A A B B)	4	1
50	60	(C B A A C A A A C A)	(C C C A A A A A C B)	4	6
50	90	(C C C A A A A A C C)	(C C B B A A A A B B)	6	4
40	60	(A B C A C A A A C C)	(B C C B C A A A A C)	7	4
40	80	(C A A B A C A A C C)	(C C C C C A A A B B)	5	4
50	90	(C C A C C A C A C C)	(C C B B C A A A B A)	4	3
40	80	(C C C C A A A C C C)	(C B B B A A A A B B)	6	4
20	60	(A A A A A A C A C A)	(B C B B A A C A C B)	1	0
50	80	(A C C C C A A A C A)	(C B B B C C A A B B)	5	0
50	80	(C A C C C A C B C C)	(B C B B C A A C C A)	6	4
60	80	(C B B B C B A C B C)	(B B B B C A A A B B)	3	5
40	40	(A C C C C A C A C A)	(B B C A A A A A B C)	5	3
40	40	(C C C C C C A C A C)	(B B C C A A A A C B)	5	4
50	50	(A C C B A A A C B A)	(B B B B A A A C A B)	2	8

50	70	(C B A B A C A A B A)	(B C B B A A A A C B)	5	5
40	40	(B C C C B A A B B C)	(B B B B C C B C B C)	6	5
50	50	(B C A B A A A A C C)	(B C A A A A A A B C)	7	0
30	60	(A A C C A A A A C C)	(C B B B A B A B B B)	8	2
40	80	(A C A A C C A A C C)	(B C C B C A A A B B)	7	0
20	60	(A A A A A A A C A A)	(A B C B A A A A B B)	5	7
60	90	(C B C A A A A A B B)	(C C B B A A A A B B)	6	4
50	70	(C C C C C A C A C C)	(B B B B A A A A B B)	4	3
30	80	(B A A B A C A A C A)	(C C B C C C A A B B)	6	4
40	70	(A A C B C B A C B C)	(C C C C A A A A B B)	4	7
60	50	(C C A A C A A C B A)	(B B C A C C A A B B)	4	6

"ASSERT-NoBugs" Group: Feedback based on models built with NEITHER only.

30	60	(B A C A C A C A A A)	(B B B B C C C A B B)	0	15
10	40	(A A A A A B A C C C)	(C C C C C C C A C C)	0	10
70	10	(C C C C C A A A C B	(C C B B C A A A B B)	0	4
40	70	(B B C C B A A A C B)	(B C B A C A A A B C)	0	9
50	50	(C C C C A C A A B C)	(B B C B C C A C B B)	0	8
40	50	(A C C A A A A A C A)	(C B C B C C A B C B)	0	8
20	40	(A C B A A C B C C A)	(C A C A C C A A C C)	0	11
60	90	(C C C B C C C A A B)	(C C B B C C A A B B)	0	15
50	70	(A C C B A C A A B A)	(C C C B C C C A B B)	0	8
50	60	(C C C A A C A A B C)	(B B C B C A C A B B)	0	8
40	80	(C C C C A A A B A A)	(B B B B C A A A B B)	0	10
80	90	(B C C B C A A A B B)	(C C B B C A A A B C)	0	3
70	70	(B B A B C A A A B B)	(C B B B A A A A B C)	0	4
40	70	(C C A C A A A C C C)	(B B B B C A A C B B)	0	10
60	70	(C C C C C A C A B C)	(B B B B C A A C B B)	0	6
70	90	(C C B C A A A A B C)	(C C B B C A C A B B)	0	6
30	60	(A C C C C C C A C C)	(B C C B C A C C B B)	0	14
60	70	(A C A C C A A A B C)	(C C B B A A C C B B)	0	7
40	50	(A C B B B A B C A A)	(C C C A A A B C B B)	0	11
40	60	(A C A A A A A A C A)	(B B B A A A A A B B)	0	5
50	70	(C C C C C A A C C C)	(B B B B C C A A B B)	0	6

60	80	(A B B B A A A A B C)	(C C B C C A A A B C)	0	6
50	90	(C C A C A A A A C C)	(B C B B C A A A B B)	0	7
60	60	(C C A B B A B A B C)	(B C B A C C A A A B)	0	12
20	40	(A B A B A A C C A A)	(B B B A A A A A C C)	0	9

"Remediate" Group: Random reteaching feedback.

90	80	(B C B B C A A A B B)	(C B B B A A A A B B)	0	1
40	50	(C C C C A C A A C A)	(B B B A C C A B B B)	0	10
50	50	(C C B B C B C C C A)	(B A B B C A C B B C)	0	13
50	60	(B C C C A A A A B A)	(B C B C A A A A C B)	0	9
60	50	(A A C B C A C A B B)	(B A C A C A A A C B)	0	8
40	80	(A C B C A A A C A C)	(C C B B C A B A B C)	0	13
60	40	(C C A A A A B A B B)	(B B C C C A C A C B)	0	6
70	30	(C B B B B A A B B B)	(B C C C C C A C C C)	0	6
20	50	(A B C A C B B C B A)	(B B B B A B A C B B)	0	20
60	70	(C B A A C A A A C B)	(B B B B A A A A B B)	0	9
40	60	(A C A B A B A C B C)	(C C C C C A A A C C)	0	12
50	50	(C C C C C C A A C C)	(B B B C C C C A B B)	0	7
50	70	(A C C B A A C A B C)	(C B B B C C A C B B)	0	9
40	70	(C B C A C C A A A C)	(B C C B B A A A B B)	0	13
60	70	(C B C B B A A A B A)	(B C B B C C B A B B)	0	6
50	40	(A C C A C C A A A B)	(B B C C A A A C B B)	0	13
50	70	(B B B A C A A C B A)	(B C B B A A A A B C)	0	11
50	70	(C C C A C A A C A C)	(C C B C C A A C C B)	0	8
30	50	(A C C A A A C C C B)	(B B B A C A C C B B)	0	9
60	40	(C C C A C A A A A C)	(B B A C A A B A B B)	0	5
60	50	(B B B B A A A A A B)	(B B B B A C A C B B)	0	4
30	70	(A C C A C A C C C C)	(C B B C C A A A C B)	0	8
60	50	(C B C A C A C A B B)	(C C C C A A A A C C)	0	9
70	80	(C C A B C A A A A C)	(C B B B C A A C B B)	0	7
30	50	(B C A A A C A A C C)	(C C C C A A A A C C)	0	8

"No Feedback" Group:

70	80	(C B C B C A A A B C)	(C B B C C A A A B B)	0	4
50	60	(B B C C C A A A C B)	(C C C C C A A A C C)	0	6
60	40	(C C C C C A A A C C)	(C C C C B A A C C C)	0	4
80	90	(C B B B C A A A B C)	(C B B B C A A A B B)	0	3
40	50	(A C A C C C A A C C)	(C C C C A A A A A C)	0	12
30	40	(A A A B A A C A C C)	(C C A C C C C C B C)	0	12
60	60	(C C A C C A A A A C)	(C C A A C A A A C C)	0	8
90	80	(C C A B C A A A B B)	(C C B A A A A A B B)	0	4
50	40	(C C C A A A A A C C)	(B B C C A A A A B C)	0	8
50	20	(C C A C C B A A A C)	(A C C A C C C C A C)	0	11
50	60	(B A A C C A A A C B)	(C B C A C A A A C B)	0	10
40	50	(C C C C A A C A C C)	(C C C C A A A A C C)	0	11
60	50	(A C A A C A A A B A)	(B A C C C A A A A B)	0	8
40	60	(B B B A C A C C B C)	(B B A A C A A A B B)	0	10
90	70	(C C B B C A A A B C)	(B B B B C A A A B A)	0	4
70	60	(C C C B C A A A C C)	(B B B C C C A A B B)	0	4
60	50	(C C C C C A A A C C)	(C C C C A A A A C C)	0	4
70	60	(C C A B C A A A A C)	(C B C C A A A A B B)	0	8
60	80	(C C C C C A A A C C)	(C B B C C A A A B B)	0	4
50	60	(A C C B B A A A A A)	(A A B A C A A A A B)	0	10
50	50	(C B C C C A A A C C)	(B C C C C A A A C C)	0	7
20	20	(A A C C A A C C B A)	(C A C C A C A C A C)	0	18
40	60	(A C C A B A A A A A)	(C C B A A A A C A B)	0	9
50	70	(A C C C C A A A C C)	(C C B C C A C A C B)	0	8
40	60	(B B C C C A C A C B)	(C B B B C C A C C B)	0	9

APPENDIX E Student Interaction Trace

The following is a screen dump of an interaction for one of the students who used the C++ Tutor. This student was from the "ASSERT" group, and thus received feedback based on the full ASSERT algorithm.

```
Welcome to the C++ Adaptive Tutorial

Please enter your initials (up to 3 letters) followed by the last 4 digits
of your student ID (example: ptb9971):ah2914

------------------------------------------------------------------------

This homework assignment is part of an ongoing research project to build
adaptive tutoring systems. The idea is that the program will watch how you
interact with the system and adjust its output to fit your needs.

To keep things simple, the program is built around a standard multiple
choice test format. The system will present you with a series of multiple
choice questions, each in the same format. Your job is simply to pick the
single best answer for each question.

At some point during the test, the program may interrupt the test to give
you feedback before proceeding with the rest of the questions. This feedback
will consist of examples in the same format as the test questions, but with
the CORRECT answers provided. YOU SHOULD STUDY THESE EXAMPLES CAREFULLY as
they will help you better understand the material (which will be on your
final exam!).

When you have completed the test, you will be given a printout of all your
answers, the feedback created by the system for you, and a set of the
correct answers to all the test questions. You will also be given full
credit for the homework.

Your help with this research is greatly appreciated. Thanks.

press RETURN to continue...

------------------------------------------------------------------------
```

```
+------------+
| Question 1 |
+------------+

void main()
{
  const int j = 3, *h;
  int i, k;
  h = &j;
  cin >> k >> i;

  cout << (k % j);  cout << (i %= j);
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: a



+------------+
| Question 2 |
+------------+

void main()
{
  const int d = 1, *const b = &d;
  int a, c;
  cin >> c >> a;

  cout << (c % a) << (a += c);
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: b



+------------+
| Question 3 |
+------------+

void main()
{
  int i, j, k, *h;
  h = &i;
  cin >> *h >> k >> j;
```

```
  cout << ((i >= j) && (k & j)) << (i++);
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: c



+------------+
| Question 4 |
+------------+

void main()
{
  const int p = 1, *n;
  int m, q;
  n = &p;
  cin >> *n >> q >> m;

  cout << (q *= p);  cout << ((m -= q) && (p * m));
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: c



+------------+
| Question 5 |
+------------+

void main()
{
  int x = 2, y, w, *const z;
  z = &x;
  cin >> *z >> w >> y;

  cout << (w = x) << ((y * x) && (y + 10));
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: a
```

```
+------------+
| Question 6 |
+------------+

void main()
{
  const int d = 2, *a = &d;
  int b, c;
  cin >> c >> b;

  cout << (--b) << ((b += c) || (*a != d));
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: b



+------------+
| Question 7 |
+------------+

void main()
{
  int w, x, y, *z = &w;
  cin >> w >> y >> x;

  cout << (w == y) << ((w - w) && (y += w));
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: c



+------------+
| Question 8 |
+------------+

void main()
{
  const int d;
  int a, b, c;
  cin >> d >> c >> b >> a;

  cout << (c = 9);  cout << (c = 10);
}
```

```
Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: a



+------------+
| Question 9 |
+------------+

void main()
{
  int b = 2, c, d, *a = &b;
  a = &d;
  cin >> d >> c;

  cout << (6 & d) << ((c == c) || (d = b));
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: a



+-------------+
| Question 10 |
+-------------+

void main()
{
  int j = 5, h, k, *i = &j;
  i = &h;
  cin >> j >> k >> h;

  cout << (j *= 7) << ((k - k) || (h + j));
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: b

-------------------------------------------------------------------------

Before proceeding with the rest of the test, let's stop a minute to review
some important points from lecture.
```

In what follows, you will be shown a set of examples, one at a time. Before each example, a quick review of your lecture material will be presented in textual form. Crucial elements of this explanation will be highlighted. One or more examples will then be presented to illustrate the explanation.

STUDY THESE EXAMPLES CAREFULLY. They are aimed at providing you detailed feedback that should help you sort out some of the more subtle points from lecture which will be on the final exam.

System working; please wait...
;;; Loading source file "/u/baffes/thesis/results/cpp/all-bug-lib.lisp"
done

press RETURN to continue...

EXPLANATION

One way to detect a compilation error is to look for an identifier which is declared constant and initialized, then later assigned a new value.

Specifically, note the following which contribute to this type
of error:
* There must be a constant which is initialized and later assigned.

Here is an example to illustrate this point.

press RETURN to continue...

Example

Here is an example which might appear to be correct
but is actually a COMPILE ERROR:

```
void main()
{
  const int x = 2, *const w = &x;
  int y, z;
  w = &z;
  cin >> x >> z >> y;

  cout << ((x == z) && (y <= *w));  cout << (x % 8);
}
```

The following points BY THEMSELVES are enough to make this example
a COMPILE ERROR:
* The identifier 'w' is declared constant and initialized to the address of 'x',
  and then later set in the expression 'w = &z;'.

Note that this example is a compile error IN SPITE OF the

following:
* The integer 'x' is assigned in the expression 'cin >> x >> z >> y;'.

press RETURN to continue...

EXPLANATION

One way to detect a compilation error is to look for an identifier which is declared constant and initialized, then later assigned a new value.

A constant identifier is erroneously assigned when it is declared as a constant pointer to an integer, initialized to the address of some integer, and later set to the address of another integer. It does not matter if the identifier is a pointer declared to point to an constant integer or a non-constant integer; once a constant pointer is initialized it cannot be reset to the address of another integer.

Specifically, note the following which contribute to this type of error:
* There must be a pointer declared to be constant (but not necessarily pointing to a constant object).
* A pointer declared to be constant must be initialized.
* A pointer declared a constant and initialized must be set after its initialization.

Here is an example to illustrate these points.

press RETURN to continue...

Example

Here is an example which might appear to be a compile error but is actually CORRECT:

```
void main()
{
  int x = 5, y, w, *z = &x;
  z = &w;
  cin >> w >> y;

  cout << ((y *= x) || (y > w));  cout << (w -= x);
}
```

This example is NOT a compile error because:
* The pointer 'z' is declared as a non-constant pointer to a non-constant integer, so it does not have to be initialized.

press RETURN to continue...

```
--------------------------------------------------------------------------
EXPLANATION

Ambiguous statements arise in C++ when an operator has multiple operand
expressions that are LINKED. Two operands are linked if they refer to the
same variable (call it the LINKING VARIABLE), and one operand uses the variable
and the other is an expression that alters it. Remember that in C++ operands
are evaluated before an operator is executed. Since operands can be evaluated
in ANY ORDER, this means the operand which uses the linking variable can have
a different value depending upon the order in which the operands are evaluated.

A linking variable is SET in an operation if the operator is auto increment
or auto decrement (++ or --).

Specifically, note the following which contribute to this type
of error:
* The operator used is '++' or '--'.

Here is an example to illustrate this point.

press RETURN to continue...


Example
-------
Here is an example which might appear to be correct
but is actually AMBIGUOUS:

void main()
{
  int w, x, y, *z = &w;
  cin >> w >> y >> x;
  z = &x;

  cout << ((x * y) || (*z %= w)) << (--y);
}

This example is AMBIGUOUS because:
* The operator used in the expression '(--y)' is an auto-increment/
  auto-decrement operator which USES AND SETS the variable.
* Two of the operands are linked in the output statement through variable 'y'
  which is used and set in the expression '(--y)' and used in the
  left half of a lazy '||' operator in the subexpression '(x * y)'.

press RETURN to continue...


--------------------------------------------------------------------------
EXPLANATION

One way to detect a compilation error is to look for an identifier which is
declared constant and initialized, then later assigned a new value.
```

A constant identifier is erroneously assigned when it is declared as a constant pointer to an integer, initialized to the address of some integer, and later set to the address of another integer. It does not matter if the identifier is a pointer declared to point to an constant integer or a non-constant integer; once a constant pointer is initialized it cannot be reset to the address of another integer.

Specifically, note the following which contribute to this type of error:
* There must be a pointer declared to be constant (but not necessarily pointing to a constant object).
* A pointer declared to be constant must be initialized.
* A pointer declared a constant and initialized must be set after its initialization.

Here is an example to illustrate these points.

press RETURN to continue...

Example

Here is an example which might appear to be correct
but is actually a COMPILE ERROR:

```
void main()
{
  int w, x, y, *const z = &w;
  cin >> *z >> y >> x;
  z = &y;

  cout << ((w == w) && (6 & x));  cout << (x %= 8);
}
```

This example is a COMPILE ERROR because:
* The identifier 'z' is declared as a constant pointer (to a non-constant integer).
* The constant pointer 'z' is initialized to the address of 'w'.
* The pointer 'z' is set in the expression 'z = &y;'.

press RETURN to continue...

The system will now resume the test. Try using whatever you may have gleaned from the examples to answer the rest of the questions.

press RETURN to continue...

```
+-------------+
| Question 11 |
+-------------+

void main()
{
  const int p = 3, *q;
  int m, n;
  q = &p;
  cin >> n >> m;

  cout << (p & n);  cout << (p > 9);
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: c



+-------------+
| Question 12 |
+-------------+

void main()
{
  const int p = 3, *const m = &p;
  int n, q;
  cin >> q >> n;

  cout << (q - p) << (q++);
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: a



+-------------+
| Question 13 |
+-------------+

void main()
{
  const int w;
  int x, y, z;
  cin >> w >> z >> y >> x;
```

```
  cout << ((x = w) && (y | z)) << (10 % w);
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: a



+-------------+
| Question 14 |
+-------------+

void main()
{
  const int x = 1, *y;
  int z, w;
  y = &x;
  cin >> *y >> w >> z;

  cout << (x < z);  cout << (w *= x);
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: a



+-------------+
| Question 15 |
+-------------+

void main()
{
  int c, a, d, *const b;
  b = &c;
  cin >> *b >> d >> a;

  cout << (d <= c) << (10 | d);
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: a
```

```
+-------------+
| Question 16 |
+-------------+

void main()
{
  int k = 1, h, i, j;
  cin >> j >> i >> h;

  cout << (i | h) << ((1 | j) || (h = j));
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: c



+-------------+
| Question 17 |
+-------------+

void main()
{
  const int h = 2, *const i = &h;
  int j, k;
  cin >> k >> j;

  cout << (j - h) << ((j += 7) || (*i != h));
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: c



+-------------+
| Question 18 |
+-------------+

void main()
{
  int k = 5, h, j, *i = &k;
  i = &h;
  cin >> j >> h;

  cout << ((k % h) && (*i >= j)) << (h -= k);
}
```

```
Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: b

+-------------+
| Question 19 |
+-------------+

void main()
{
  int p = 4, n, q, *m = &p;
  cin >> q >> n;

  cout << (7 <= n) << ((p > p) && (n -= q));
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: c



+-------------+
| Question 20 |
+-------------+

void main()
{
  int h, i, j, k;
  cin >> h >> k >> j >> i;

  cout << (6 >= j) << ((i == i) && (--j));
}

Is the above (A) a compile error,
             (B) ambiguous (i.e., different outputs from different compilers) or
             (C) neither A nor B ?

answer: b



Congratulations! The test is over, and you have completed the homework. However,
the system has collected a good deal of data from your answers and can give you
some feedback which you may find helpful.

Would you like the additional feedback? (Yes or No) No
```

```
--------------------------------------------------------------------------

The tutorial is over. Contact Paul and he will generate
a final printout for you to take with you.

Thanks again for your help, and good luck on the final.
NIL
> (answers)
;;; Loading source file "/u/baffes/thesis/results/cpp/Cah2914"
;;; Loading source file "/u/baffes/thesis/assert/cpp/exam-data.lisp"


Question    Your    Correct
 Number    Answer   Answer
*  1         A       C
   2         B       B
*  3         C       B
*  4         C       A
   5         A       A
   6         B       B
   7         C       C
   8         A       A
*  9         A       C
  10         B       B
  11         C       C
* 12         A       B
  13         A       A
  14         A       A
  15         A       A
  16         C       C
* 17         C       B
  18         B       B
  19         C       C
  20         B       B
NIL
>
```

Bibliography

[Acker et al.] Acker, L., Lester, J., Souther, A., and Porter, B. (1991). Generating coherent explanations to answer students' questions. In Burns, H., Partlett, C. and Redfield, C., editors, *Intelligent Tutoring Systems: Evolutions in Design*, chapter 7. Lawrence-Erlbaum Associates.

[Anderson, 1983] Anderson, J. R. (1983). *The Architecture of Cognition*. Harvard University Press, Cambridge, MA.

[Anderson, 1984] Anderson, J. R. (1984). Cognitive psychology and intelligent tutoring. In *Proceedings of the Cognitive Science Society Conference*, pages 37-43. Boulder, CO.

[Anderson et al., 1985] Anderson, J. R., Boyle, C. F., and Reiser, B. J. (1985). The geometry tutor. In *Proceedings of the Ninth International Joint conference on Artificial intelligence*, pages 1-7. Los Angeles, CA.

[Baffes, 1994] Baffes, P. (1994). Learning to model students: Using theory refinement to detect misconceptions. Technical Report AI94-215, Austin, TX: Artificial Intelligence Laboratory, University of Texas.

[Baffes and Mooney, 1993] Baffes, P. and Mooney, R. (1993). Symbolic revision of theories with M-of-N rules. In *Proceedings of the Thirteenth International Joint Conference on Artificial intelligence*, pages 1135-1140. Chambery, France.

[Baffes and Mooney, 1992] Baffes, P. and Mooney, R. J. (1992). Using theory revision to model students and acquire stereotypical errors. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, pages 617-622. Bloomington, IN.

[Barr et al., 1976] Barr, A., Beard, M., and Atkinson, R. C. (1976). The computer as a tutorial laboratory: the Stanford BIP project. *International Journal of Man-Machine Studies*, 8:567-596.

[Bloom, 1984] Bloom, B. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher*, 13(6):4-15.

[Boyle and Anderson, 1984] Boyle, C. and Anderson, J. R. (1984). Acquisition and automated instruction of geometry skills. Paper presented at the annual Meeting of the American Educational Research Association. New Orleans, LA.

[Brown and Burton, 1975] Brown, J. S. and Burton, R. R. (1975). Multiple representations of knowledge for tutorial reasoning. In Bobrow, D. and Collins, A., editors, *Representation and Understanding*. New York: Academic Press.

[Brown and Burton, 1978] Brown, J. S. and Burton, R. R. (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2:155-192.

[Brown et al., 1975] Brown, J. S., Burton, R. R., and Bell, A. G. (1975). SOPHIE: a step towards a reactive learning environment. *International Journal of Man-Machine Studies*, 7:675-696.

[Brown et al., 1976] Brown, J. S., Rubinstein, R., and Burton, R. R. (1976). Reactive learning environment for computer-assisted electronics instruction. Technical Report BBN Report No. 3314, Cambridge, MA: Bolt Beranek and Newman, Inc.

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

[Burton, 1982] Burton, R. R. (1982). Diagnosing bugs in a simple procedural skill. In Sleeman, D. H. and Brown, J. S., editors, *Intelligent Tutoring Systems*, chapter 8. London: Academic Press.

[Burton and Brown, 1976] Burton, R. R. and Brown, J. S. (1976). A tutoring and student modeling paradigm for gaming environments. *Computer Science and Education. ACM SIGCSE Bulletin*, 8(1):236-246.

[Carbonell, 1970a] Carbonell, J. R. (1970a). AI in CAI: an artificial intelligence approach to computer-assisted instruction. *IEEE Transactions on Man-Machine Systems*, 11(4):190-202.

[Carbonell, 1970b] Carbonell, J. R. (1970b). *Mixed-initiative man-computer instructional dialogues*. Ph.D. thesis, Cambridge, MA: Massachusetts Institute of Technology.

[Carr and Goldstein, 1977a] Carr, B. and Goldstein, I. (1977a). Overlays: a theory of modeling for computer-aided instruction. Technical Report A. I. Memo 406, Cambridge, MA: MIT.

[Carr and Goldstein, 1977b] Carr, B. and Goldstein, I. (1977b). WUSOR-II: a computer-aided instruction program with student modeling capabilities. Technical Report A. I. Memo 417, Cambridge, MA: MIT.

[Clancey, 1979] Clancey, W. J. (1979). *Transfer of Rule-Based Expertise through a Tutorial Dialogue*. Ph.D. thesis, Palo Alto, CA: Stanford University.

[Cohen, 1990] Cohen, W. (1990). Learning from textbook knowledge: A case study. In *Proceedings of the National Conference on Artificial Intelligence*, pages 743-748, Boston, MA.

[Cohen and Jones, 1989] Cohen, R. and Jones, M. (1989). Incorporating user models into expert systems for educational purposes. In Kobsa, A. and Wahlster, W., editors, *User models in Dialog Systems*, chapter 11, pages 313-333. New York, NY: Springer-Verlag.

[Collins, 1977] Collins, A. (1977). Processes in acquiring knowledge. In Anderson, R. C., Spiro, R. J., and Montague, W. E., editors, *Schooling and the acquisition of knowledge*. Hinsdale, NJ: Lawrence Earlbaum Associates.

[Costa et al., 1988] Costa, E., Duchenoy, S., and Kodratoff, Y. (1988). A resolution-based method for discovering student misconceptions. In Self, J., editor,

Artificial Intelligence and Human Learning. New York, NY: Chapman and Hall.

[Craw and Sleeman, 1990] Craw, S. and Sleeman, D. (1990). The flexibility of speculative refinement. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 28-32. Evanston, IL.

[Danyluk, 1989] Danyluk, A. P. (1989) Finding new rules for incomplete theories: Explicit biases for induction with contextual information. In *Proceedings of the Sixth International Conference on Machine Learning*, pages 34-36, Ithaca, NY.

[Dick and Carey, 1990] Dick, W. and Carey, L. (1990). *The systematic design of instruction*. Glenview, IL: Scott, Foresman/Little, Brown Higher Education. Third edition.

[Elsom-Cook, 1988] Elsom-Cook, M. (1988). Guided discovery tutoring and bounded user modeling. In Self, J., editor, *Artificial Intelligence and Human Learning*, chapter 10, pages 259-277. New York, NY: Chapman and Hall.

[Farrell et al., 1984] Farrell, R., Anderson, J. R., Boyle, C., and Yost, G. (1984). The geometry tutor. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, pages 106-109. Austin, TX.

[Finin, 1989] Finin, T. (1989). GUMS: A general user modeling system. In Kobsa, A. and Wahlster, W., editors, *User models in Dialog Systems*, chapter 15, pages 411-430. New York, NY: Springer-Verlag.

[Flann and Dietterich, 1989] Flann, N. S., and Dietterich, T. G. (1989) A study of explanation-based methods for inductive learning. *Machine Learning*, 4(2):187-226.

[Gilmore and Self, 1988] Gilmore, D. and Self, J. (1988). The application of machine learning to intelligent tutoring systems. In Self, J., editor, *Artificial Intelligence and Human Learning*. New York, NY: Chapman and Hall.

[Ginsberg, 1990] Ginsberg, A. (1990). Theory reduction, theory revision, and retranslation. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 777-782. Detroit, MI.

[Goldstein and Miller, 1976] Goldstein, I. and Miller, M. (1976). Structured planning and debugging: a linguistic theory of design. Technical Report A. I. Memo 387, Cambridge, MA: MIT.

[Hoppe, 1994] Hoppe, H. U. (1994). Deductive error diagnosis and inductive error generalization for intelligent tutoring systems. *Journal of AI and Education* (in press).

[Huang et al., 1991] Huang, X., McCalla, G. I., Greer, J. E., and Neufeld, E. (1991). Revising deductive knowledge and stereotypical knowledge in a student model. *User Modeling and User-Adapted Interaction*, 1:87-115.

[IDS, 1990] March issue of *Instructional Delivery Systems*. (1990) Warrington, VA.

[Ikeda and Misoguchi, 1993] Ikeda, M. and Misoguchi, R. (1993). FITS: A framework for ITS. Technical Report AI-TR-93-5, Osaka, Japan: ISIR Osaka University.

[Johnson, 1986] Johnson, W. L. (1986). *Intention-Based Diagnosis of Novice Programming Errors*. London: Pitman Publishing.

[Jones and VanLehn, 1991] Jones, R. and VanLehn, K. (1991). Computation model of acquisition for children's learning strategies. *Machine Learning*, 6:65-69.

[Koppel et al., 1991] Koppel, M., Feldman, R., and Segre, A. (1994). Bias-driven revision of logical domain theories. *Journal of Artificial Intelligence Research*, 1:1-50.

[Langley and Ohlsson, 1984] Langley, P., and Ohlsson, S. (1984). Automated cognitive modeling. In *Proceedings of the National Conference on Artificial Intelligence*, pages 193-197, Austin, TX.

[Langley et al., 1984] Langley, P., Ohlsson, S., and Sage, S. (1984). A machine learning approach to student modeling. Technical Report CMU-RI-TR-84-7, Pittsburgh, PA.: Carnegie-Mellon University.

[Laubsch, 1975] Laubsch, J. H. (1975). Some thoughts about representing knowledge in instructional systems. In *Proceedings of the Fourth International Joint conference on Artificial Intelligence*, pages 122-125.

[Lianging and Taotao, 1991] Lianging, H. and Taotao, H. (1991). A uniform student model for intelligent tutoring systems: Declarative and procedural aspects. *Educational Technology*, 31(11):44-48.

[Mahoney and Mooney, 1993] Mahoney, J., and Mooney, R. (1993). Combining connectionist and symbolic learning to revise certainty-factor rule bases. Connection Science, 5:339-364.

[Michalski, 1983] Michalski, R. S. (1983). A theory and methodology of inductive learning. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine Learning: An Artificial Intelligence Approach*, pages 83-134. Tioga.

[Miller and Goldstein, 1977a] Miller, M. and Goldstein, I. (1977a). An automated consultant for MACSYMA. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, page 789. Cambridge, MA.

[Miller and Goldstein, 1977b] Miller, M. and Goldstein, I. (1977b). Problem solving grammars as formal tools for intelligent CAI. In *Proceedings of the National ACM Conference*, pages 220-226. Seattle, WA.

[Mitchell, 1982] Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence*, 18(2):203-226.

[Mooney, 1994] Mooney, R. J. (1994). A preliminary PAC analysis of theory revision. In Petsche, T., Judd, S. and Hanson, S., editors, *Computational Learning Theory and Natural Learning Systems, (Vol. 3)*. Cambridge: MIT Press, (in press).

[Mooney and Ourston, 1989] Mooney, R., and Ourston, D. (1989). Induction over the unexplained: Integrated learning of concepts with both explainable and conventional aspects. In *Proceedings of the Sixth International Conference on Machine Learning*, pages 5-7, Ithaca, NY.

[Moore and Sleeman, 1988] Moore, J. and Sleeman, D. (1988). Enhancing PIXIE's tutoring capabilities. *International Journal of Man-Machine Studies*, 28:605-623.

[Muggleton and Feng, 1990] Muggleton, S. and Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*. Tokyo, Japan: Ohmsha.

[Murphy and Davidson, 1991] Murphy, M. A. and Davidson, G. V. (1991). Computer-based adaptive instruction: Effects of learner control on concept learning. *Journal of Computer-Based Instruction*, 18(2):51-56.

[Murray, 1991] Murray, W. (1991). An endorsement-based approach to student modeling for planner-controlled tutors. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 1100-1106. Sydney, Australia.

[Murray, 1986] Murray, W. R. (1986). TALUS: Automatic program debugging for intelligent tutoring systems. Technical Report AI TR86-32, Austin, TX: University of Texas.

[Nicolson, 1992] Nicolson, R. I. (1992). Diagnosis can help in intelligent tutoring. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, pages 635-640. Bloomington, IN.

[Nwana, 1991] Nwana, H. (1991). User modelling and user adapted interaction in an intelligent tutoring system. *User Modeling and User-Adapted Interaction*, 1:1-32.

[Pazzani et al., 1991] Pazzani, M., Brunk, C., and Silverstein, G. (1991). A knowledge-intensive approach to learning relational concepts. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 432-436, Evanston, IL.

[Ohlsson and Langley, 1985] Ohlsson, S. and Langley, P. (1985). Identifying solution paths in cognitive diagnosis. Technical Report CMU-RI-TR-85-2, Pittsburgh, PA.: Carnegie-Mellon University.

[O'Shea, 1982] O'Shea, T. (1982). A self-improving quadratic tutor. In Sleeman, D. H. and Brown, J. S., editors, *Intelligent Tutoring Systems*, chapter 13. London: Academic Press.

[Ourston, 1991] Ourston, D. (1991). *Using Explanation-Based and Empirical Methods in Theory Revision*. Ph.D. thesis, Austin, TX: University of Texas. Also appears as Artificial Intelligence Laboratory Technical Report AI 91-164.

[Ourston and Mooney, 1990] Ourston, D. and Mooney, R. (1990). Changing the rules: A comprehensive approach to theory refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 815-820. Detroit, MI.

[Ourston and Mooney, 1994] Ourston, D. and Mooney, R. J. (in press). Theory refinement combining analytical and empirical methods. *Artificial Intelligence*. 66:311-394.

[Papert, 1980] Papert, S. (1980). *Mindstorms: Children, Computers and Powerful Ideas*. New York: Basic Books.

[Plotkin, 1970] Plotkin, G. D. (1970). A note on inductive generalization. In Meltzer, B. and Michie, D., editors, *Machine Intelligence (Vol. 5)*. New York: Elsevier North-Holland.

[Quilici, 1989] Quilici, A. (1989). Detecting and responding to plan-oriented misconceptions. In Kobsa, A. and Wahlster, W., editors, *User models in Dialog Systems*, chapter 5, pages 108-132. New York, NY: Springer-Verlag.

[Quinlan, 1986] Quinlan, J. (1986). Induction of decision trees. *Machine Learning*, 1(1):81-106.

[Quinlan, 1990] Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning*, 5(3):239-266.

[Reiser et al., 1985] Reiser, B. J., Anderson, J. R., and Farrell, R. G. (1985). Dynamic student modeling in an intelligent tutor for LISP programming. In *Proceedings of the Ninth International Joint conference on Artificial Intelligence*, pages 8-14. Los Angeles, CA.

[Rich, 1989] Rich, E. (1989). Stereotypes and user modeling. In Kobsa, A. and Wahlster, W., editors, *User models in Dialog Systems*, chapter 2, pages 35-51. New York, NY: Springer-Verlag.

[Richards, 1992] Richards, B. L. (1992). *An Operator-Based Approach To First-Order Theory Revision*. Ph.D. thesis, Austin, TX: University of Texas. Also appears as Artificial Intelligence Laboratory Technical Report AI 92-181.

[Rissanen, 1978] Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, 14:465-471.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, J. R. (1986). Learning internal representations by error propagation. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing, Vol. I*, pages 318-362. Cambridge, MA: MIT Press.

[Sandberg and Barnard, 1993] Sandberg, J. and Barnard, Y. (1993). Education and technology: What do we know? And where is AI? *Artificial Intelligence Communications*, 6(1):47-58.

[Scanlon and O'Shea, 1977] Scanlon, E. and O'Shea, T. (1977). Cognitive economy in physics reasoning: Implications for designing instructional materials. In Mandl, H. and Lesgold, A., editors, *Learning Issues for Intelligent Tutoring Systems*, pages 259-277. New York, NY: Springer-Verlag.

[Self, 1974] Self, J. (1974). Student models in computer-aided instruction. *International Journal of Man-Machine Studies*, 6:261-276.

[Self, 1990] Self, J. A. (1990). Bypassing the intractable problem of student modeling. In Frasson, C. and Gauthier, G., editors, *Intelligent tutoring systems: at the crossroads of artificial intelligence and education*, chapter 5. Ablex Publishing Corp.

[Sleeman, 1983] Sleeman, D. (1983). Inferring student models for intelligent computer-aided instruction. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine Learning: An Artificial Intelligence Approach*, pages 483-508. Tioga.

[Sleeman, 1987] Sleeman, D. (1987). Some challenges for intelligent tutoring systems. In *Proceedings of the Tenth International Joint conference on Artificial Intelligence*, pages 1166-1168. Milan.

[Sleeman et al., 1990] Sleeman, D., Hirsh, H., Ellery, I., and Kim, I. (1990). Extending domain theories: two case studies in student modeling. *Machine Learning*, 5:11-37.

[Sleeman and Smith, 1981] Sleeman, D. H. and Smith, M. J. (1981). Modelling students' problem solving. *Artificial Intelligence*, 16:171-187.

[Soloway and Johnson, 1984] Soloway, E. and Johnson, W. (1984). Remembrance of blunders past: a retrospective on the development of PROUST. In *Proceedings of the Sixth Annual Conference of the Cognitive Science Society*, page 57. Boulder, CO.

[Soloway et al., 1983] Soloway, E., Rubin, E., Wolf, B., Bonar, J., and Johnson, W. L. (1983). MENO-II: an AI-based programming tutor. *Journal of Computer-Based Instruction*, 10(1):20-34.

[Soloway et al., 1981] Soloway, E., Wolf, B., Rubin, E., and Barth, P. (1981). MENO-II: an intelligent tutoring system for novice programmers. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 975-977. Vancouver, Canada.

[Stevens and Collins, 1977] Stevens, A. L. and Collins, A. (1977). The goal structure of a Socratic tutor. In *Proceedings of the National ACM Conference*, pages 253-263. Seattle, WA.

[Stevens and Collins, 1980] Stevens, A. L. and Collins, A. (1980). Multiple models of a comples system. In Snow, R., Frederico, P. and Montague, W., editors, *Aptitude, Learning, and Instruction*, volume 2. Lawrence-Erlbaum Associates, Hinsdale, NJ.

[Tangkitvanich and Shimura, 1993]. Tangkitvanich, S., and Shimura, M. (1993). Learning from an approximate theory and noisy examples. In Proceedings

of the Eleventh National Conference on Artificial Intelligence, pages 466-471. Washington, D.C.

[Tecuci and Michalski, 1991] Tecuci, G., and Michalski, R. (1991). A method for multistrategy task-adaptive learning based on plausible justifications. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 549-553, Evanston, IL.

[Tennyson, 1971] Tennyson, R. (1971). Instructional variables which predict specific learner concept acquisition and errors. Paper presented at the annual Meeting of the American Psychological Association. Washington, D.C.

[Tennyson and Park, 1980] Tennyson, R. D. and Park, O. (1980). The teaching of concepts: A review of instructional design research literature. *Review of Educational Research*, 50(1):55-70.

[Towell and Shavlik, 1991] Towell, G. and Shavlik, J. (1991). Refining symbolic knowledge using neural networks. In *Proceedings of the International Workshop on Multistrategy Learning*, pages 257-272. Harper's Ferry, W.Va.

[Towell et al., 1990] Towell, G. G., Shavlik, J. W., and Noordewier, M. O. (1990). Refinement of approximate domain theories by knowledge-based artificial neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 861-866. Boston, MA.

[Tukey, 1953] Tukey, J. W. (1953). The Problem of Multiple Comparisons. Mimeograph, Princeton, NJ. In Spence, J., Cotton, J., Underwood, B. and Duncan, C., editors, *Elementary Statistics*, fourth edition, pg 215, note 4. Prentice-Hall.

[VanLehn, 1983] VanLehn, K. (1983). *Felicity conditions for human skill acquisition: validating an AI-based theory.* Ph.D. thesis, Cambridge, MA: Massachusetts Institute of Technology.

[VanLehn, 1987] VanLehn, K. (1987). Learning one subprocedure per lesson. *Artificial Intelligence*. 31:1-40.

[VanLehn et al., 1992] VanLehn, K., Jones, R., and Chi, T. (1992). A model of the self-explanation effect. *The Journal of the Learning Sciences*, 2(1):1-59.

[Wenger, 1987] Wenger, E. (1987). *Artificial Intelligence and Tutoring Systems*. Los Altos, CA: Morgan Kaufmann.

[White et al., 1991] White, J., Troutman, A., and Stone, D. (1991). Effect of three levels of corrective feedback and two cognitive levels of tasks on performance in computer-directed mathematics instruction. *Journal of Computer-Based Instruction*, 18(4):130-134.

[Whitehall, 1990] Whitehall, B. (1990). *Knowledge-Based Learning: An Integration of Deductive and Inductive Learning for Knowledge Base Completion*. Ph.D. thesis, University of Illinois, Urbana, IL.

[Wilkins, 1988] Wilkins, D. (1988). Knowledge base refinement using apprenticeship learning techniques. In *Proceedings of the National Conference on Artificial Intelligence*, pages 646-651, St. Paul, MN.

[Winston and Horn, 1989] Winston, P. H. and Horn, B. K. P. (1989). *Lisp*. Reading, MA: Addison-Wesley.

[Wogulis, 1993] Wogulis, J. (1993). Handling negation in first-order theory revision. In *Proceedings of the IJCAI-93 Workshop on Inductive Logic Programming*. Chambery, France.

[Wogulis and Pazzani, 1993] Wogulis, J. and Pazzani, M. (1993). A methodology for evaluating theory revision systems: results with Audrey II. In *Proceedings of the Thirteenth International Joint Conference on Artificial intelligence*, pages 1128-1134. Chambery, France.

[Young and O'Shea, 1981] Young, R. M. and O'Shea, T. (1981). Errors in children's subtraction. *Cognitive Science*, 5:153-177.

Vita

Paul Thomas Baffes was born in Chicago, Illinois, on April 20, 1962, the son of Mary Lou Baffes and Thomas Gus Baffes. In 1980, he graduated from Maine Township High School East and entered Harvard University. After receiving the degree of Bachelor of Arts from Harvard in 1984 he moved to Houston, Texas. During the following years he worked both as a federal employee and as a contractor in the NASA Johnson Space Center area where he received two patents and two Space Act awards for his work in applied artificial intelligence. In September 1990 he entered the Graduate School of the University of Texas at Austin. He earned the degree of Master of Science in May of 1993 from the University of Texas.

Permanent address: 523 Airline Road, #1704, Corpus Christi, Texas 78412.

This dissertation was typed by the author.