Journal of Artificial Intelligence in Education, 7,1 (1996) pp. 75-116

Refinement-Based Student Modeling and Automated Bug Library Construction

Paul Baffes BAFFES@SCICOMP.COM
SciComp, Inc., 5806 Mesa Drive,
Suite 250, Austin, TX 78731, U.S.A.

Raymond Mooney MOONEY@CS.UTEXAS.EDU
Department of Computer Sciences, Taylor Hall 2.124,
The University of Texas at Austin, Austin, TX 78712, U.S.A.

Abstract

A critical component of model-based intelligent tutoring systems is a mechanism for capturing the conceptual state of the student, which enables the system to tailor its feedback to suit individual strengths and weaknesses. To be useful such a modeling technique must be *practical*, in the sense that models are easy to construct, and *effective*, in the sense that using the model actually impacts student learning. This research presents 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. 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.

1 Introduction

Student modeling has a long and interesting history, dating back well into the earliest efforts to produce *intelligent tutoring systems* (ITS). The best method for constructing and using a student model is still the subject of much debate. Most student modeling techniques, however, have a similar goal, which might be defined as follows:

Given: A set of *expectations* regarding student behavior in some domain and,
A set of *observations* of a specific student's behavior on one or more tasks in that domain,
Find: A *representation* accounting for any discrepancies between the expectations and observations that can be used as a basis for tutoring 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 pre-programmed into the tutor*. Using the student model, an ITS would then modify its feedback to suit specific strengths and weaknesses, enabling it to be truly adaptive to the individual.

Unfortunately, the difficulty of constructing and testing student models has discouraged many researchers from pursuing further investigations into the field. Despite the fact that more than two decades of research has resulted in a wide variety of student modeling techniques, the practical task of incorporating these techniques into a functioning tutoring system has proved to be a major roadblock. Furthermore, neither the utility nor even 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 on-line student modelling." (Sandberg & 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."

Thus the current challenge for student modeling is to show that modeling a student can be made both *practical* and *effective*. This is precisely the contribution of this work, as embodied in the ASSERT algorithm (Acquiring Stereotypical Student Errors by Refining Theories). ASSERT was designed to show that student modeling is a viable tool for building an effective tutoring system. By taking advantage of some of the latest techniques in machine learning, ASSERT is able to construct student models efficiently and automatically, catching both expected and novel student misconceptions. Also, 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. Finally, ASSERT can be used to significantly improve student performance, as will be shown in the sections which follow.

The remaining sections are organized as follows. Section 2 reviews previous work in student modeling as a motivation underlying the design of ASSERT. Section 3 then presents a description ASSERT focusing on the portion of the algorithm which captures individual student errors. Next, Section 4 describes how trends across a population of students are automatically collected into a *bug library*, and how such a library is then incorporated into the modeling process. Finally, Section 5 presents experimental results followed by discussion and conclusions in Sections 6 and 7.

2 Previous Work

2.1 Overlay Modeling

The earliest AI-based student models, embodied in systems such as SCHOLAR (Carbonell, 1970), WEST (Burton & Brown, 1976) and WUSOR (Carr & Goldstein, 1977), 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 using some form of truth-maintenance scheme (Finnin, 1989; Murray, 1991). However the marking is achieved, typically the unmarked elements of the model are used to focus tutoring on new problem areas for the student, or to ensure full coverage of the domain.

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, is the representational 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. Specifically, this means that *misconceptions* cannot be modeled. Thus, overlay models can capture the notion of a student's lack of knowledge, but they cannot be used to model the student who knows of a topic but misunderstands it.

2.2 Bug Libraries

To address the limitation of overlay models, other researchers focused on constructing databases of student misconceptions typically termed *bug libraries*. The classic bug-library work was done by Brown, Burton and VanLehn (Brown & Burton, 1978; Burton, 1982; Brown & VanLehn, 1980), Sleeman and Smith (Sleeman & Smith, 1981), and Young and O'Shea (Young & O'Shea, 1981), but a number of other systems can be said to incorporate some form of stored misconceptions (Rich, 1989; Miller & Goldstein, 1977; Quilici, 1989; Soloway & Johnson, 1984). 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.

The idea is a very powerful one, especially if specific responses can be tied to whatever buggy structures are encoded. However, two important problems remain with this simple bug-library approach. First, the construction of such catalogs by hand is a difficult and time-consuming task which must be repeated for every new domain. Second, even if great care is taken, the resulting library may still fail to cover a wide enough range of behaviors to function successfully. That is, a student may exhibit a misconception which was not anticipated by the author of the library. As with overlay models, the static nature of bug libraries renders them incapable of modeling unanticipated student behaviors.

2.3 Dynamic Modeling

To capture novel student misconceptions, one must turn to some kind of search over the space of possible bugs. Two methods have been tried to date: one attempts to extend a bug library and the other attempts to infer a model of the student from scratch using inductive machine–learning techniques. In both cases, novel errors are modeled by constructing new buggy information dynamically, using data from a student's behavior to bound the search.

2.3.1 Extending a bug library

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. In PIXIE's bug libraries, misconceptions 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* builds new mal-rules by filling *gaps* in partial representations of a student's solution, whereas MALGEN uses a generate-and-test method to create new candidate mal-rules.

The disadvantage of both systems is their reliance upon a user to decide which new mal-rules are appropriate extensions for the bug library. To their credit, the authors do raise this issue and discuss potential filters that might be used to cut down on the number of new mal-rules presented to the user. Unfortunately, to this point no general-purpose filtering mechanism which might be usable across domains has been found. In the end, the user is still faced with a number of proposed mal-rules and must decide which ones are the "keepers."

2.3.2 Modeling by Induction

In an effort to avoid the cost associated with hand-constructed bug libraries, Langley et al. turned to machine learning. Their ACM system (Langley & Ohlsson, 1984; Langley et al., 1990) was the first effort to harness machine learning techniques for diagnosis of misconceptions through the use of *induction*. ACM uses a domain-independent induction algorithm to induce control knowledge representing how students apply operators in a given domain. The output of induction is a set of conditions which predict when an *operator* will produce a *state* which lies on the *solution path* connecting the input for a given problem to the student's solution. The conditions found by induction are then used to specialize the operators, and the result is a procedure that models the student's unique problem solving behavior.

By using induction, ACM can operate automatically, constructing models that capture both correct and buggy knowledge. However, because the operators must initially be general enough to model many kinds of behavior, both correct and incorrect, the potential search space is huge. Langley et al. note this, and suggest various "psychologically plausible" conditions which can be applied to the operators to limit the search. However, the system is still fundamentally limited by the complexity of having to construct a model completely from scratch. This can only be remedied by collecting large amounts of data on each student or by imposing further constraints on the search space, which would require finding such constraints by using the very human-intensive methods this technique is designed to avoid.

2.4 Tracing Techniques

One final style of student modeler bears mentioning because it represents a blend of the various techniques described to this point. In what might be loosely termed *tracing* techniques, the underlying philosophy is to follow along with the student during his or her problem solving, stopping whenever the student deviates from the correct procedure. As such, tracing techniques must have knowledge of both correct and incorrect actions like buggy modelers, and must also have a mechanism for reproducing the steps followed by the student like ACM's solution paths. Using the correct path as a bias, tracing systems can operate very efficiently.

The pioneering efforts in this area are Anderson's *model tracing* tutors (Anderson, 1983; Anderson et al., 1985; Reiser et al., 1985), which follow student behavior by requiring all interaction with the tutor to occur through menu selection. Other tracing systems utilize a logic-based representation (Costa et al., 1988; Ikeda & Misoguchi, 1993; Hoppe, 1994) where 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. 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.

Unlike the previous methods, tracing techniques do not dynamically construct student models. Instead, they rely upon either the assumption that the student can be directed to follow along the correct path or querying an oracle whenever a deviation is detected. As such they lack the ability to handle novel student misconceptions independently.

3 Refinement–Based Modeling

This previous work on student modeling has resulted in three important ideas central to the research presented here. First, modeling systems can increase their coverage of student behavior by incorporating knowledge from a library of expected misconceptions. Second, to be truly adaptive and to avoid the costs of bug library construction, one must turn to some sort of dynamic modeling or learning algorithm. And third, tracing student actions in comparison to expected correct behavior can be an effective tool for detecting faulty behavior without requiring a great deal of search. ASSERT combines these ideas by using a machine-learning technique called *theory refinement* (Ginsberg, 1990; Ourston and Mooney, 1994; Craw and Sleeman, 1991; Towell et al., 1990). 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 correct 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-refinement process merely modifies the knowledge base until it is consistent with the examples. Thus, one can also use theory refinement by inputting a *correct* knowledge base and examples of *erroneous* behavior, and theory refinement will introduce whatever modifications are necessary to cause the knowledge base to simulate the incorrect examples.

Theory refinement, then, provides a basis for the development of a *refinement-based modeler*. Starting with a representation of the correct knowledge of the domain, and given examples of erroneous student behavior, theory refinement will revise the knowledge base to make it consistent with the student, i.e., introduce "faulty" knowledge to account for the student's mistakes.[1] The refinements made to the knowledge base then represent a model of the student, and can be used directly to guide tutorial feedback by comparing the refinements with whatever elements of the correct knowledge base they replaced.

Using theory refinement, ASSERT combines the methods used in previous modeling systems. A theory-refinement learner combines the power of both analytic (as in MALGEN and INFER) and empirical (as in ACM) learning techniques in an integrated, domain-independent way. ASSERT can model any misconception consistent within the primitives used to define the domain. And finally, ASSERT provides an extension to theory refinement that can combine the results of *multiple* student models from *different* students. This mechanism allows ASSERT to construct a bug library automatically, without the necessity of intervention on the part of the author. Section 4 describes this algorithm in detail. First, however, we turn our attention to the mechanism of theory refinement and its role in the design of ASSERT.

3.1 Outline of Theory Refinement

Having outlined the philosophy behind ASSERT, we can now turn our attention to the theory refinement algorithm around which ASSERT is constructed. It is important to point out from the start that basic design of ASSERT is not tied to a particular theory refinement algorithm. Other theory refinement systems which differ from the one presented here could be used to provide ASSERT with different or enhanced capabilities.

ASSERT uses the NEITHER algorithm (Baffes, 1994; Baffes & Mooney, 1993) which is based on the EITHER theory-refinement system (Ourston & Mooney, 1990). EITHER was chosen as a starting point because it was the most complete symbolic theory-refinement system available. NEITHER is designed to work with a propositional Horn-clause knowledge representation. It takes two inputs, a propositional Horn-clause rule base called the *theory*, which

1. Keep in mind that the language used here is highly subjective in nature. One need not take the view that student actions are "mistakes." The central point is that theory refinement can be used to detect automatically any actions which are inconsistent with its given knowledge base.

R1: compile-error ← constant-not-init
R2: compile-error ← constant-assigned
R3: constant-not-init ← (pointer constant) ∧ (pointer-init false)
R4: constant-not-init ← (integer constant) ∧ (integer-init false)
R5: constant-assigned ← (integer constant) ∧ integer-init ∧ (integer-set yes)
R6: constant-assigned ← (integer constant) ∧ integer-init ∧ (integer-set through-pointer)
R7: constant-assigned ← (pointer constant) ∧ pointer-init ∧ pointer-set

	Example 1	Example 2	Example 3	Example 4
compile-error	*true*	*false*	*true*	*true*
pointer	constant	non-constant	non-constant	non-constant
pointer-init	true	false	true	false
pointer-set	true	true	true	true
integer	constant	non-constant	non-constant	non-constant
integer-init	true	true	true	true
integer-set	through-pointer	yes	no	no

Figure 1: A Theory and Examples. The desired classification for each example is shown in italics (thus, Examples 3 and 4 are misclassified by the theory).

is repaired 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. NEITHER can generalize or specialize a theory, without user intervention, and is guaranteed to produce a set of refinements which are consistent with the input examples.

Figure 1 shows an example theory and four input examples. The top of the figure shows part of a rule-base taken from an ASSERT tutor built for teaching a subset of C^{++} concepts (for a complete listing of the C^{++} Tutor rules, see Appendix A).[2] The rules, numbered R1-R7, consist of a consequent which is considered true for an example only when the propositions to its right are provable from the feature values of that example. Propositions without values represent either intermediate concepts or are shorthand for binary features requiring “true” as a value. This simplified theory has only one category, “compile-error,” into which it can classify examples. The input examples, shown in the table below the rules, are considered classified as compile-errors only if they can satisfy rules R1 or R2 or both. Otherwise, a closed-world assumption is used to classify the example as non compile-error. So, for instance, example 1 is correctly classified as a compile-error because it can satisfy R2 by satisfying either R6 or R7. Likewise, example 2 fails to satisfy any of the rules and is thus correctly classified as non compile-error.

However, examples 3 and 4 are misclassified by the theory in its current state; that is, the

2. The source code for both ASSERT and the C^{++} Tutor is available from the authors by anonymous FTP.

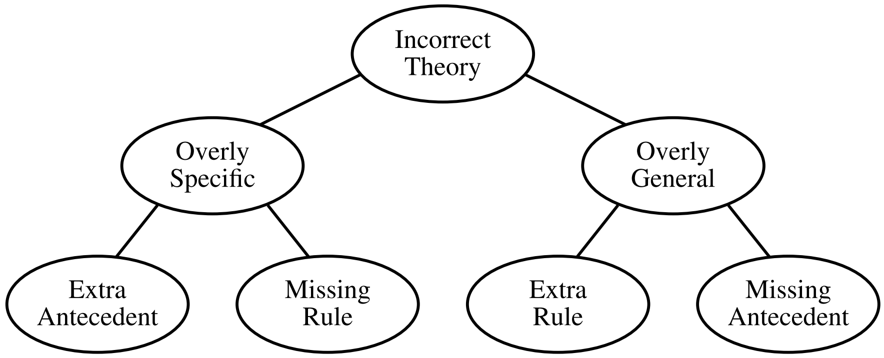


Figure 2: Theory error taxonomy for propositional Horn-clause theories.

rule base is "incorrect" since it does not produce the desired classification shown for these examples. Propositional Horn-clause theories can have four types of semantic errors, as depicted in Figure 2. An overly-general theory is one that causes an example to be classified in categories other than its own, i.e., a false positive. NEITHER 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. NEITHER retracts existing antecedents and learns new rules to fix these problems. By making these four kinds of syntactic rule changes, NEITHER can correct the semantics of the theory by altering the conditions under which rules are satisfied.

NEITHER use a combination of three computations to determine how to modify a theory. The first step is to find a *repair* for a single failing example by analyzing the rule base to determine what rules need to be changed to fix the theory for the example. For a failing positive example, a set of rule antecedents is found which, if deleted, will fix the theory for that example. Alternatively, for a failing negative example, a set of rules is computed which, if deleted, will repair the theory. The second step is to *test* a repair for a single example against all the other input examples to see if the repair will cause new misclassifications to occur. If not, then the repair can be applied to the theory directly. Otherwise, a third step is taken using *induction* to learn a set of additional conditions which will separate the examples fixed by the repair from the examples for which the repair causes new misclassifications. These additional conditions are then used to modify the repair.

As an example, in Figure 1, notice that both example 3 and example 4 are false negative examples since neither is classified as compile-error. This indicates that the theory is overly-specific and must be generalized. One way to repair the theory for example 3 would be to delete the "(pointer constant)" condition from rule R7. This allows rule R7 to be satisfied by the example, without hindering the classification of example 1, and without causing rule R7 to become so general that it would be satisfied for example 2. Testing this repair against examples 1, 2 and 4 yields no new misclassifications, and it can be applied directly to the theory.

Finding a repair for example 4, however, yields a different result. The simplest repair is to delete the "(pointer constant)" condition from rule R3. However, when this repair is tested against examples 1, 2 and 3, example 2 is erroneously classified as compile-error. The only way to fix the repair is to remove the "(pointer constant)" condition from the rule, to fix example 4, plus add a new condition to the rule which keeps example 2 from satisfying the

R1: compile-error ← constant-not-init
R2: compile-error ← constant-assigned
R3: constant-not-init ← (pointer constant) ∧ (pointer-init false)
R4: constant-not-init ← (integer constant) ∧ (integer-init false)
R5: constant-assigned ← (integer constant) ∧ integer-init ∧ (integer-set yes)
R6: constant-assigned ← (integer constant) ∧ integer-init ∧ (integer-set through-pointer)
R7: constant-assigned ← (pointer constant) ∧ pointer-init ∧ pointer-set

\- -

R1: compile-error ← constant-not-init
R2: compile-error ← constant-assigned
R3: constant-not-init ← ~~(pointer constant)~~ ∧ (pointer-init false) ∧ (integer-set no)
R4: constant-not-init ← (integer constant) ∧ (integer-init false)
R5: constant-assigned ← (integer constant) ∧ integer-init ∧ (integer-set yes)
R6: constant-assigned ← (integer constant) ∧ integer-init ∧ (integer-set through-pointer)
R7: constant-assigned ← ~~(pointer constant)~~ ∧ pointer-init ∧ pointer-set

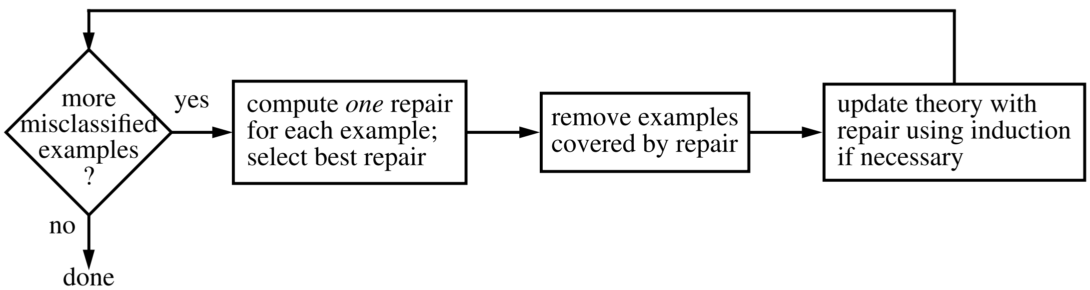
Figure 3: Example of NEITHER refinement. Above the dashed line is the rule base before refinement; below are the rules after refinement. Modified rules are shown in boldface.

rule. Passing examples 2 and 4 to an induction algorithm would return "(integer-set no)" as the condition which can discriminate between the examples. The final revised rule base, which correctly classifies all four examples, is shown in Figure 3.

Notice that the repairs chosen for examples 3 and 4 in Figure 3 are not the only possible repairs for these examples. For instance, example 3 could have been classified as compile-error by removing the conditions "(integer constant)" and "(integer-set through-pointer)" from rule R6, or by removing the conditions "(integer constant)" and "(integer-set yes)" from rule R5. For that matter, removing all of the antecedents from any one of the rules R1 through R4 would also have repaired the theory for example 3, by making either the "compile-error" or "constant-not-init" concepts provable by default. In fact, computing all possible repairs for an example in the general case is exponential in the size of the theory. Consequently, the way in which repairs are calculated, as well as when a repair is applied to the theory in relation to computing repairs for other examples, can have a profound impact on the accuracy and performance of the theory refinement algorithm.

A summary of the NEITHER algorithm is provided in Figure 4. NEITHER opts for speed in computing repairs, focusing on quickly finding one good repair for each example. The goal is to find the smallest repair in the deepest possible part of the theory. After converting the theory into a graph, NEITHER uses a recursive routine which starts at the leaf rules of the theory. Failing conditions are collected at each rule and passed up to parent rules. When a choice is possible, NEITHER always chooses the smaller repair, picking randomly to break ties. Each rule is visited only once, making the repair computation linear in the size of the theory.

Once a repair has been calculated for each example, NEITHER selects one repair from among the set to apply to the theory. A selection is made by temporarily modifying the theory with each repair, calculating how many examples it fixes and how many new misclassifications it causes. These results are combined with the size of the repair, and the smallest repair which fixes the most examples with the fewest new misclassifications is selected. The repair is then tested against the rest of the examples, and induction is performed if necessary



Criterion for computing a repair for ONE example:
Find the deepest, shortest, repair which causes the fewest new misclassifications

Criterion for selecting best repair from AMONG examples:
Select the shortest repair fixing the most examples with the fewest new misclassifications

Figure 4: NEITHER main loop.

to modify the repair to avoid new misclassifications. The whole process is embedded in a loop which continues until all misclassified examples have been accounted for. The result is that NEITHER runs very quickly (see Baffes & Mooney, 1993), giving response times that are on the order of a few seconds. This is critical to an interactive tutoring environment where feedback must be generated for the student in a timely fashion.

3.2 Overview of ASSERT

Having reviewed the basics of theory refinement, we can now turn to the details of ASSERT. 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. At its most abstract level, such a tutorial can be seen as a dialog between the student and the system as shown in Figure 5. ASSERT focuses on the question of how to construct a useful interpretation of the student's actions. This focus is depicted as a new component inserted into the diagram as shown in the right half of Figure 5. 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

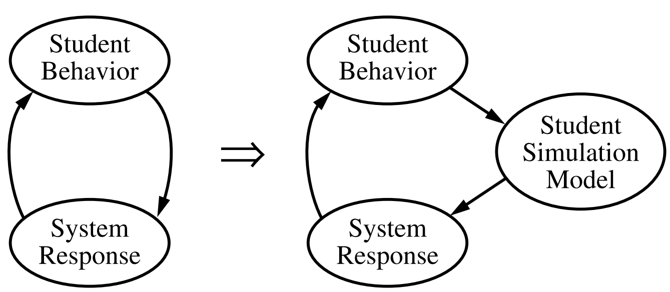


Figure 5: Abstract view of student-tutor interaction.

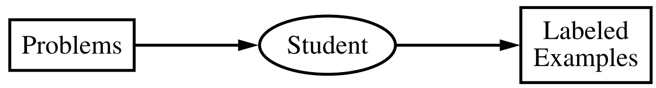


Figure 6: Student behavior diagram.

that this knowledge base can be modified to replicate the solutions furnished by the student.

3.2.1 The Student as a Classifier

Figure 6 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, called *problems*, the student will produce a set of *labeled examples* which classify each of the problems into one *category*. 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 associates 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 automated training systems. It also means that student actions will translate directly into a form usable by theory refinement, which requires labeled examples as one of its inputs.

3.2.2 Modeling by Theory Refinement

Once collected, the labeled examples generated by the student are passed to the theory-refinement component of ASSERT, depicted in Figure 7. As discussed previously, theory refinement will take an incoming knowledge base, plus an incoming set of examples, and refine the knowledge base until it is consistent with the examples. The NEITHER theory-refinement system is used to add or remove rules or parts of rules until the rule base reproduces the same answers as the student, i.e., will classify each feature vector using the same category label as the student. The resulting refined rule base is thus able to simulate the student's behavior.

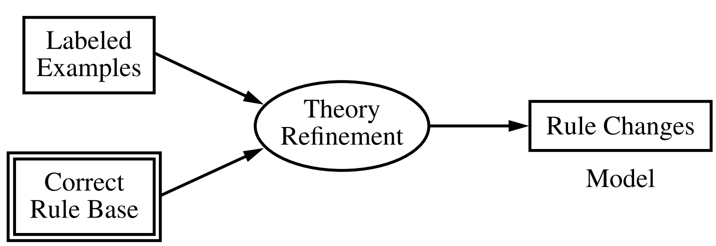


Figure 7: The student simulation model.

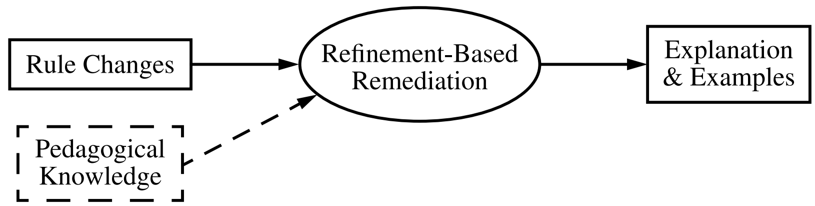


Figure 8: System response diagram.

How the correct rule base is constructed is crucial 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.

3.2.3 Refinement-Based Remediation

The last component of ASSERT, the system response, is outlined in Figure 8. Using the refinements produced by NEITHER, ASSERT generates explanations and examples to reinforce the correct form of the rule or rules modified. The underlying approach, called *refinement-based remediation*, is based on fundamental units of explanation called *units of remediation*. Rather than implementing any particular pedagogy, ASSERT supplies the most elementary information required: an *explanation* with one or more *examples*. For each refinement detected by NEITHER, ASSERT provides two functions: 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 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.

The specifics of generating explanations and examples for each refinement are explained in detail in (Baffes, 1994), but the underlying idea is straightforward. Explanations focus on describing how the correct form of the rule (not the revised version) fits into the originally correct rule base. Each rule has an associated piece of stored text, describing its role in the rule base. A full explanation is generated by chaining together the stored text for the rules lying on the proof path for the correct label (not the student's label) for the example, i.e., the label which is produced by the correct rule base for the given feature vector.

Generating examples is a bit more complicated since they are constructed *dynamically* rather than being drawn from storage. Recall that each refinement made by NEITHER results in the addition or deletion of literals from a rule in the theory. Whole rules can be added or deleted as well, but this is the same as adding or removing blocks of literals. Using normal deductive methods, the added and removed literals can be traced down to the feature vector. The result is a set of conditions in the feature vector which the student is ignoring or a set of extra conditions not present in the feature vector which the student thinks are necessary.

```
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.
```

Figure 9: Example remediation given to a student.

ASSERT can thus generate an example which is correct in every way *except* for the added or missing conditions in the refinement. The result is then presented as a counter example to the student, and the various added or missing conditions highlighted. Note that this corresponds very closely to tutorial methods outlined for conceptual domains (Tennyson, 1971).

Figure 9 shows an explanation and example pair, corresponding to one of the refinements depicted previously in Figure 3. Recall that the last rule of Figure 3 was altered by removing one of its antecedents. Figure 9 is the feedback generated for this missing antecedent, illustrating how the condition represented by the antecedent is essential to drawing a correct conclusion. The top half of Figure 9 shows the text which explains how the rule fits in to the overall rule base. As part of the explanation, the three conditions of the rule, represented by its three correct antecedents, are itemized at the end of the explanation. Then, an example is generated which highlights how the “(pointer constant)” condition bears on the correct answer to the example, showing how the truth or falsity of the condition leads to a particular

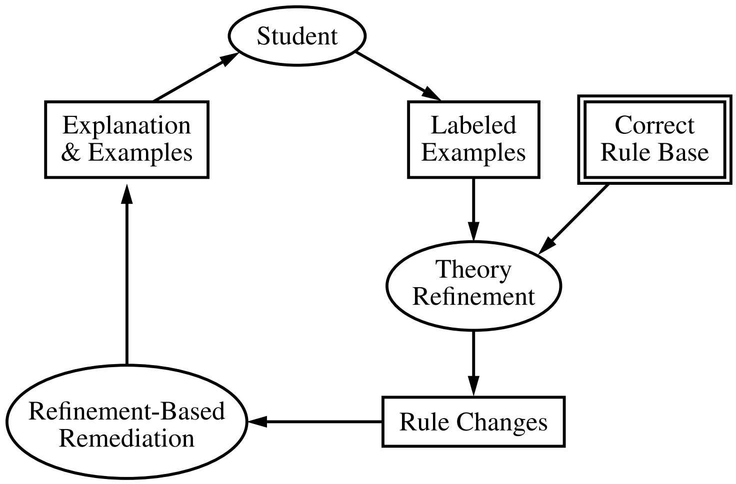


Figure 10: Basic design of the ASSERT algorithm.

conclusion.

Finally, Figure 10 combines the components from Figures 6, 7and 8, 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.

4 Extending ASSERT's Modeler

The previous three 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. As such this explains how ASSERT constructs student models by tracing the student's behavior against a known standard rule base. Nothing, though, has been said about how multiple student models are mined to construct a bug library nor how such a library is incorporated back into the modeling process.

4.1 Building a Bug Library

ASSERT uses the rule changes resulting from theory refinement for each student as the basis for constructing its bug library. This gives ASSERT two advantages. First, the rule changes are 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 directly into the modeling process.

ASSERT constructs a bug library in three stages. First, it collects copies of all the rule changes from all the student models together, eliminating any duplicates. Second, it ranks each rule change by a measure of how frequently it occurs among the various student mod-

Correct Rules (CR)	Model M_1	Model M_2	Model M_3
⋮	⋮	⋮	⋮
$a \leftarrow b\ c\ d\ e\ f$	$a \leftarrow e\ f$	$a \leftarrow b\ e$	$a \leftarrow c\ d\ e\ f\ g\ h$
⋮	⋮	⋮	⋮

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:

$$\left.\begin{array}{l} \text{distance}(CR, M_1) = \text{delete}\{b,c,d\} = 3 \\ \text{distance}(CR, M_2) = \text{delete}\{c,d,f\} = 3 \\ \text{distance}(CR, M_3) = \text{delete}\{b\}\ \text{add}\{g,h\} = 3 \end{array}\right\} \text{total} = 9$$

Distance from Correct Rules with bug from M_1 to Models:

$$\left.\begin{array}{l} \text{distance}(CR+B_1, M_1) = 0 \\ \text{distance}(CR+B_1, M_2) = \text{delete}\{f\}\ \text{add}\{b\} = 2 \\ \text{distance}(CR+B_1, M_3) = \text{add}\{c,d,g,h\} = 4 \end{array}\right\} \text{total} = 6$$

Distance from Correct Rules with bug from M_2 to Models:

$$\left.\begin{array}{l} \text{distance}(CR+B_2, M_1) = \text{delete}\{b\}\ \text{add}\{f\} = 2 \\ \text{distance}(CR+B_2, M_2) = 0 \\ \text{distance}(CR+B_2, M_3) = \text{delete}\{b\}\ \text{add}\{c,d,f,g,h\} = 6 \end{array}\right\} \text{total} = 8$$

Distance from Correct Rules with bug from M_3 to Models:

$$\left.\begin{array}{l} \text{distance}(CR+B_3, M_1) = \text{delete}\{c,d,g,h\} = 4 \\ \text{distance}(CR+B_3, M_2) = \text{delete}\{c,d,f,g,h\}\ \text{add}\{b\} = 6 \\ \text{distance}(CR+B_3, M_3) = 0 \end{array}\right\} \text{total} = 10$$

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

Figure 11: Stereotypicality computation.

els, called the *stereotypicality* of the rule change. Third, in the process of ranking each rule change, ASSERT tests generalizations of the change to see if they result in better stereotypicality. If a generalization is found which has superior stereotypicality, then the rule change is replaced by the generalization. The final bug library contains the best generalization for each rule change with any duplicates removed.

Figures 11 through 14 illustrate how a bug library is constructed for a contrived example generated for illustration purposes. Figure 11 shows how stereotypicality is computed (for details on how stereotypicality can be computed in linear time see Baffes, 1994). 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 by computing how much each

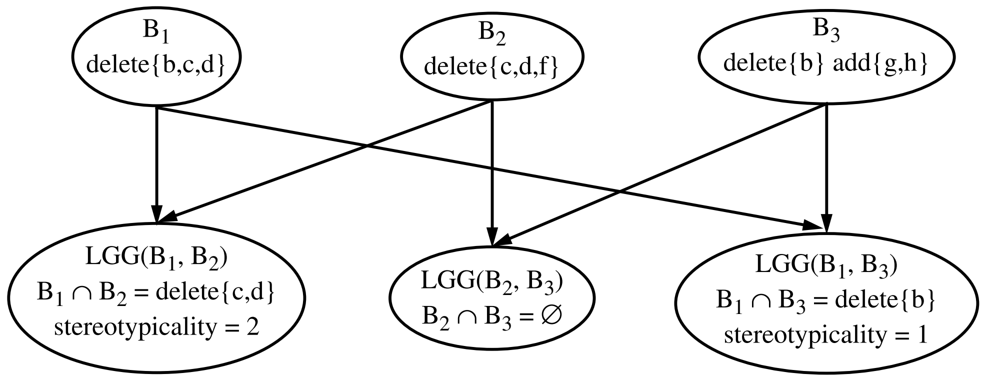


Figure 12: Bug generalization using the LGG operator.

bug decreases the *distances* between the models and the correct rule base. 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 other. Thus, changing $CR{+}B_1$ into M_2 requires changing the rule $a \leftarrow e\ f$ into the rule $a \leftarrow b\ e$ which is done by deleting *f* and adding *b*. The bottom of the figure has the stereotypicality values for each of the bugs.[3]

Figure 12 illustrates how ASSERT forms generalizations among the bugs from Figure 11. 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). For propositional logic literals, the LGG of two refinements is simply their intersection. 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. 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. Since the result of forming the LGG of two refinements is also a refinement, the process can be continued, forming LGG's from LGG's which can also result in better or worse refinements.

Figure 13 shows the pseudocode for constructing a bug library. The fundamental idea is to perform a hill-climbing search using successive LGG operations. Starting with each refinement 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. 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.

3. Note that a bug may have a negative stereotypicality. Unless a bug is present in more than half of the student models, it will have a negative stereotypicality since there will be no overlap between the bug and the majority of the models. What is important is the relative difference between stereotypicality values.

```
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
      Temp = ∅;
      for r ∈ R do add Intersection(Best,r) to Temp;
      G = member of Temp with highest stereotypicality;
      if Stereotypicality(G) > S then begin
        Best = G;
        S = Stereotypicality(G);
      end;
    end;
    add Best to bug library;
  end;
  return bug library sorted by greatest stereotypicality;
```

Figure 13: Pseudocode for bug library construction

Figure 14 shows a complete example for constructing a bug library, using the bugs from Figure 11 plus one extra bug to highlight the hill-climbing nature of the algorithm. Below the four bugs are a series of boxes, each representing one iteration of the outer loop. 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 the inner loop is entered and an LGG is formed between B_1 and the other three bugs. Once the LGG's are computed, the best is found, in this case LGG(B_1, B_4), which has a stereotypicality of 4. This is compared with the current best stereotypicality, 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 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, the best LGG results from combining B_3 and B_1. The resulting 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. 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. Note that LGG(B_4, B_1), which was computed and rejected in the top box where B_1 was the seed, turns out to be a useful improvement over B_4 alone. 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}*.

4.2 Using the Bug Library for Modeling

Once the library is built, the question becomes how to use its information to enhance the modeling process. Perhaps the most obvious way to incorporate the bugs in the library is to alter the theory-refinement algorithm to use the bugs as a means for selecting repairs to fix

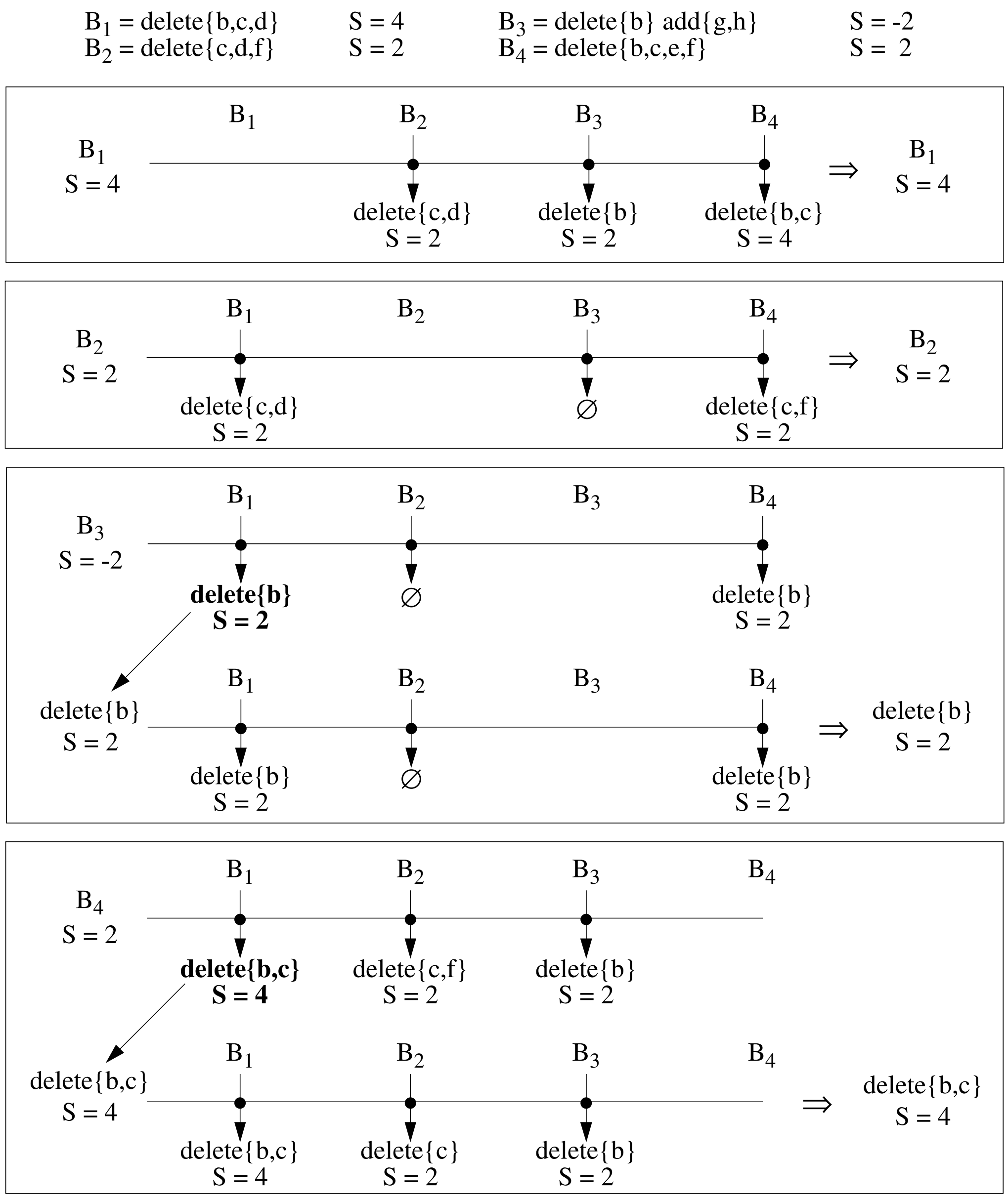


Figure 14: Bug library construction example. "S" stands for stereotypicality.

mistakes made by a student. The disadvantage of such an approach is that it would destroy the modularity of ASSERT's design. Theory refinement would no longer be an interchangeable component which could be swapped out for different refinement algorithms. A simpler approach, and the one used in ASSERT, is to modify one of the inputs given to NEITHER, leaving the refinement algorithm intact. Specifically, the correct rule base is modified before the refinement process begins by incorporating elements of the bug library which are relevant to the current student.

Figure 15 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

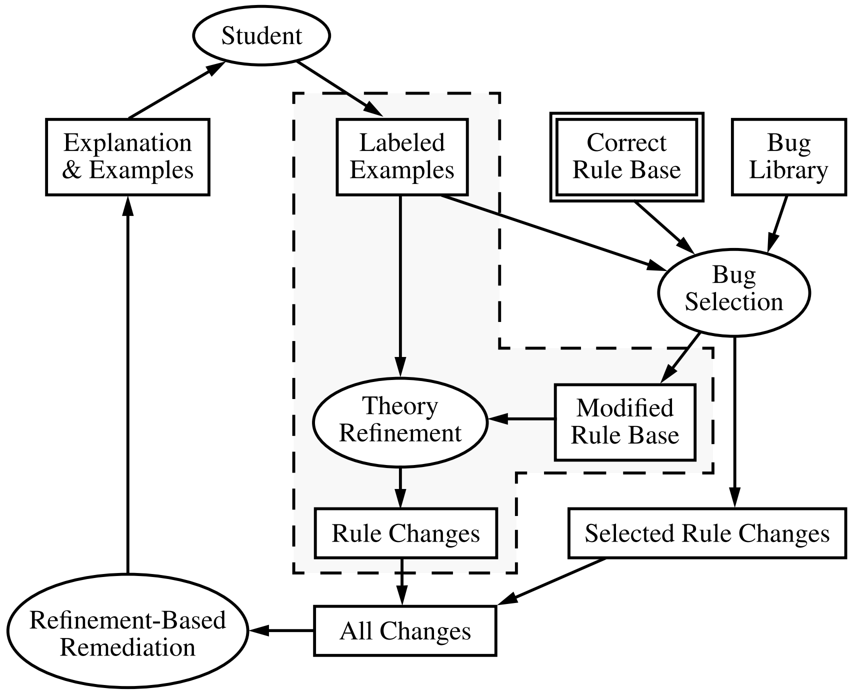


Figure 15: Overview of extended ASSERT algorithm. The area within the dashed line is the theory refinement component.

the library to be added to the rule base using a hill-climbing search. Bugs which improve upon the predictive accuracy of the rule base are added incrementally. The result is a modified rule base which resembles the student's behavior more closely than the correct rules but which may still be incomplete. The bugs which were selected are returned along with the modified rule base since they must be included with the final model of the student. 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.

The pseudocode for constructing the modified rule base is shown in Figure 16. *ModifyRules* starts with the correct rule base, and loops as long as a bug can be found which will increase the accuracy of the rule base on the set of labeled examples. The first step of each loop is to find the accuracy of the rule base when the bug in question is added to the set of rules. All those bugs which result in improved 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"[4] to the best bug, using a paired Student t-test.[5] 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

4. more precisely, whose improvement in accuracy is less, but not statistically significantly less.
5. 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).

```
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;
```

Figure 16: Pseudocode for bug library use.

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.

As an example of bug library selection, refer to a trace of the execution of the "pre-model-student" routine shown in Figure 17. This function is the implementation of the *ModifyRules* pseudocode used by the C++ Tutor mentioned in Section 3.1. The trace shown is taken from a student who interacted with the system as part of a larger test of ASSERT (described in detail below in Section 5). The bug library used in this trace consisted of 34 bugs taken from the models of 45 students who used the C++ Tutor. However, only a portion of the bug library is actually shown in the trace, corresponding to those bugs which were applicable to the mistakes made by this particular student. For a complete listing of the bug library, as well as an entire trace of this student's interaction with the system, see (Baffes, 1994).

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. But because the bug library here 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. Thus the final theory of Figure 17 has partially accounted for the student's errors and the rest gets done by NEITHER.

This completes the description of ASSERT. As has been shown, ASSERT can model both expected student errors as well as mistakes unique to an individual. Furthermore, by using a fully automated scheme for bug library construction, and by integrating the bug library with its automatic modeling algorithm, ASSERT can continue to improve its modeling accuracy over time.

```
> (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. Bug is:
  type:  add antecedent to rule
  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. Bug is:
  type:  delete antecedent from rule
  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. Bug is:
  type:  delete rule
  rule:  operator-b-sets <- (operator-b auto-incr)
  antes: nil
----------------------------iteration 4-------------------------------
Trying to beat accuracy = 95.00
done
```

Figure 17: Trace of bug selection from the bug library.

5 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. Furthermore, this evidence must come from experiments involving large numbers of students in a realistic setting so that the significance of the data can be determined.

In this section, 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 a C^{++} Tutor built for

```
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 ?
```

Figure 18: Example C^{++} problem

tutoring college-level freshmen taking an introductory C^{++} course at the University of Texas at Austin. 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 analyze the results of the C^{++} Tutor test.

5.1 C^{++} Tutor Tests

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. 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. Figure 18 shows an example question from the C^{++} Tutor (for the complete listing of the C^{++} Tutor rule base see Appendix A).

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 on 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, it was important to establish whether or not ASSERT could be an effective modeler for students in a realistic setting. This was measured by testing the model produced for a student on a set of examples taken from the student which had not been given to ASSERT. The predictive accuracy of the model on such novel examples was expected to be higher than simply using the correct rule base with no modifications. 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 utility of a model is to provide the student with feedback based on that model and measure any change in the student's 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 in 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 measure the accuracy of modeling and a third to test the utility of the bug library. In the next three sections each of these tests is described in turn.

5.1.1 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 no feedback.

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.

To that end, a program was written to generate 10 example questions using a prescribed format as follows. 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. This process was used to generate two sets of 10 questions, both of which covered the same subset of the correct rule base. This ensured that the two sets of questions covered the same concepts at the same level of difficulty, though no two questions were identical. These two sets of questions represented the pre-test and post-test to be given to each student. One set of questions was used as the pre-test for all the students, the other as the post-test, thus the same pre-test and post-test was given to every student. To discourage cheating, the order in which the 10 questions were presented was randomized. This meant every student answered the same two sets of 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, acting as the control group. This group was labeled the "No Feedback" group. The other three groups were given feedback using explanations and examples as described in Section 3.2.3. 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 automated 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, reteaching methods vary, making it important to clarify the exact approach used. For this experiment, the essential point was to illustrate whether feedback based on modeling made any difference over feedback based on no modeling at all. To that end, we chose an automated form of reteaching which used no information about the student, not even which answers the student got right or wrong. In such an informational vacuum, the option left for reteaching is to select at random from the rule base for remediation. Thus, for the "Reteaching" group, ASSERT selected four rules at random from the rule base, and an explanation and example was generated for each rule.[6]

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-Full" group) the full ASSERT algorithm was used to build the model and for the other group (the "ASSERT-NoBugs" group) only NEITHER was used, i.e., no bug library information was given to the system. For both these groups, bugs were selected for remediation based on the order they were found by NEITHER.[7] For the ASSERT-Full group, bugs from the bug library were given preference to those found by NEITHER, in order of their stereotypicality value. In both the ASSERT-Full 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-Full 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. The models from these first 45 students were then 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-Full group until the number of students assigned to all groups was even (25 students in each group).

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 1.

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-Full and the ASSERT-NoBugs students improved significantly more than students in the Reteaching group. For ASSERT-NoBugs, the improvement over Reteaching is more that 12 percentage points, and for the ASSERT-Full group, the average improvement is even greater.

It is important to be very clear about the results in Table 1. Note that there is a great deal of variance among the mean pre-test scores in the four groups. Though none of the differences among mean pre-test scores is significant, their variance is a concern when comparing the significance of the differences in average increase from pre-test to post-test. However, this is precisely why the ANOVA test was run to compare the significance. What can be con-

6. Note that we specifically avoided comparing ASSERT's reteaching against reteaching by a human tutor. Such a comparison represents an entirely different test. Here we were concerned with determining whether ASSERT's automatic reteaching was an effective feature of the algorithm.

7. NEITHER orders its refinements by preferring those which increase accuracy the most with the smallest change.

Group	*Average Pre-test Score*	*Average Post-test Score*	*Average Increase*
ASSERT-Full	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 1: 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-Full and ASSERT-NoBugs and between Reteaching and No Feedback.

cluded from Table 1 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, how much the increase depends upon the size of the pre-test data given to ASSERT's modeler, 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 automatically without resorting to the time-consuming task of hand crafting a library of bugs.

5.1.2 Modeling Performance using the C^{++} Tutor

The second important question to answer is whether or not there is a correlation between the ability of ASSERT to produce an accurate model and an improvement in student performance. This 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 can be accomplished using an ablation test format, in which various pieces of ASSERT are disabled and the resulting systems compared based on the accuracy of the models they produce. There are two different configurations in which ASSERT was used for modeling. The first, which is labeled "ASSERT-Full," uses 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. This method should produce the most accurate models. The second technique, labeled "ASSERT-NoBugs," skips the bug library and uses only NEITHER. One would expect ASSERT-Full to outperform ASSERT-NoBugs because of the additional information in the library.

In the C^{++} domain, only the data from the No Feedback group is useful for an ablation test. This is because no remediation occurred between the pre-test and post-test for the students in this group; thus, their 20 questions could be treated as a single unit from which training set and test set examples could be drawn. These training-test splits were generated

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

Table 2: Results for C++ Tutor modeling test. The differences between ASSERT-Full and ASSERT-NoBugs are not significant (all others are significant).

so as to be equivalently representative across the correct domain rules. Such a 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 covered the same 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-Full and ASSERT-NoBugs. Each system was trained with the training set 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. The results are shown in Table 2. For comparison purposes, we also measured the accuracy of both an inductive learner, using the same training and test set splits, 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 training examples using a propositional version of the FOIL algorithm (Quinlan, 1990). For the correct theory no learning was performed, i.e., the correct domain rules were used without modification to predict the student's answers. Statistical significance was measured using a two-tailed Student t-test for paired difference of means at the 0.05 level of confidence.

These results illustrate that the groups with significantly better models, ASSERT-Full and ASSERT-NoBugs, are precisely the groups which performed best after remediation in Table 1. 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. Table 2 also reinforces the finding of other studies (Ourston and Mooney, 1994) that inductive methods are simply not as effective as theory refinement in terms of accuracy.

5.1.3 Bug Library Utility Test

However, note that the differences in Table 1 and Table 2 between ASSERT-Full and ASSERT-NoBugs are not significant. This means the use of the bug library did not significantly impact the performance of the student as expected, casting doubts as to 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 more about the conditions under which a bug library, as constructed automatically by ASSERT, might be expected to impact the modeling process.

A series of tests designed to address this question, described in detail in (Baffes, 1994), can be summarized by the results shown in Table 3. This data was generated using simulated

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-Full	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 3: Comparison of bug libraries. Part (a) compares libraries on size and total number of bugs found, part (b) compares accuracy of modeling.

ablation tests like the one described in the previous section. In such a test, "students" were simulated by modifying a correct theory using six standard bugs selected probabilistically, plus additional random rule changes. The modified theory was then used to generate "answers" for 180 feature vectors representing a hypothetical "multiple-choice exam." Those answers were then passed to ASSERT to see how well it could reproduce the modified theory. Once this was done for 20 students, resulting in 20 student models, the models were combined to build a bug library.

Table 3 is a comparison of three libraries constructed using this technique, but with varying numbers of simulated students and varying amounts of example answers per student. The first library, denoted 20-180, was formed from 20 student models built with all the 180 example answers per student. The second library, 20-12, used 20 students with only 12 example answers per student chosen randomly for each student from among the 180 possible. The 100-12 library was built from 100 students answering 12 randomly selected questions. Conceptually, these three libraries were designed to compare three different conditions: a few students answering lots of questions, a few students answering a few questions, and a large number of students answering a few questions. This comparison was aimed at answering the following two-part question: (1) are ASSERT bug libraries only effective when students answer a large number of questions, or (2) can one expect effective bug libraries to emerge from a large number of students answering a more reasonable number of questions. If useful bug libraries cannot be constructed from small student models, then the utility of an ASSERT bug library is limited 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.

Part (a) of Table 3 compares the three libraries based on size and on how many of the six standard bugs were found. The 20-180 library performed the best, finding all six common bugs. That result is readily justified—by drastically reducing the amount of information on each student, individual bugs are much less likely to be found and thus much less likely to end up in the bug library. Yet, the 100-12 library outperformed the 20-12 library. Thus, even though having smaller amounts of data per student reduces the chances that a common bug will be found for any given individual, increasing the number of students improves the likelihood that the bug will be present in some student.

However, note that the total overall size of the 100-12 bug library is largest of any of the three libraries. In fact, the 100-12 library has the lowest ratio of common bugs to total bugs in the library. This dilution is a potential problem when the bug library is used in subsequent modeling efforts. Recall from Section 4.2 that the bug library is treated as a search space of refinements which is traversed in an effort to improve the accuracy of the rule base before it is passed to NEITHER. Increasing the size of the bug library widens the search, making it potentially less likely that the common bugs will be selected, which could, in turn, decrease the modeling accuracy.

This concern is addressed in Part (b) of Table 3 which shows results from more ablation tests aimed at determining if the ratio of common bugs to library size is detrimental in terms of modeling accuracy. For these tests, a new crop of simulated students was created, and each ablation test was run with this same set of students, varying the bug library. For each student only 10 of the 180 examples were used for training, leaving the remaining 170 for testing. As the numbers in the table show, the 20-12 library results in almost no improvement for ASSERT-Full and ASSERT-BugOnly (which used just the library without NEITHER) as opposed to ASSERT-NoBugs (which used no bug library). By contrast, when the "better" bug libraries are used, the accuracy of ASSERT-Full and ASSERT-BugOnly is significantly better.

This implies that a bug library can be incrementally improved as more students interact with the system, even if the student interaction is moderate, resulting in more accurate modeling. And as the data from Table 1 and Table 2 shows, more accurate models lead to better remediation and improved student performance.

5.2 Summary of Results

To recap, the main result presented in this section 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 could 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 automatically constructed refinement-based models can be used to significantly increase student performance and (2) that a bug library can be constructed automatically over time that can enhance refinement-based modeling.

6 Discussion

Several important issues have been raised by this research that must be emphasized so as to place the work into a proper context. The first issue concerns the type of knowledge being modeled. Unlike the previous modeling efforts, which focus on procedural tasks, ASSERT is designed for use in classification domains. As an example of this difference, several previous student modeling efforts have focused on the domain of writing computer programs (Miller and Goldstein, 1977; Soloway et al., 1983; Soloway and Johnson, 1984), whereas this research was tested using a classification task where students were 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 the general framework of ASSERT per se. So, for instance, as first order logic refinement methods are enhanced (Richards and Mooney, 1995), ASSERT can be updated accordingly, enabling it to address a wider range of applications. However, it is not immediately clear how easy it would be to map ASSERT to a procedural domain.

Shifting the focus of modeling to a concept-learning emphasis is not unprecedented. Other researchers, most notably Gilmore and Self (Gilmore & Self, 1988), have recognized the potential of using machine learning for tutoring conceptual knowledge. Concept learning also has a fairly well explored pedagogy (Dick & Carey, 1990) and effective techniques for responding to incorrect student classifications are already extant in the literature (Tennyson & Park, 1980). Thus a general technique for modeling in concept domains has a wide applicability and can be coupled with instructional techniques shown to be effective in the presentation of conceptual material (Tennyson, 1971).

The second issue of importance is the comparison of the C^{++} Tutor experiments to previous studies performed on the utility of student modeling. Much of the controversy over student modeling stems from a popular belief that detailed student models are enormously difficult to build and yet result in little or no practical value. In truth, however, previous studies are few in number and have reached disparate conclusions: one shows student modeling to be ineffective (Sleeman, 1987), another shows that modeling can indeed have a positive effect (Nicolson, 1992). Both of these studies used a bug library approach in which an extensive library was built by hand rather than automatically as in ASSERT. Furthermore, both previous studies used a different form of reteaching than that used in the C^{++} Tutor. Thus the C^{++} Tutor cannot be seen as a direct comparison to previous experiments. In can and should, however, be seen as evidence that effective student models can be constructed automatically which will positively impact student performance.

Which leads to the third important issue; namely, the kind of modeling done and the way in which it is used. The empirical results here show simply that automatic modeling can have a significant impact on student performance. This says nothing, however, about the sort of models one ought to build nor about the best way to use them. For example, it may be that equally significant results could be achieved by using a far simpler modeling technique, or that far better results could arise from ASSERT models if the student were allowed more control over the number or type of counter examples presented as feedback. Furthermore, one need not restrict the modeling capability of ASSERT for use only on remediation tasks. Another way to interpret a "misconception" is as an expression of a different way to solve a given problem. The most common bugs in the bug library may actually indicate that the initial "correct" rule base does not, in fact, cover all correct solutions. Thus it may be possible to use ASSERT as a knowledge acquisition tool which can learn from "creative" students. In any event, the significant features of ASSERT are that it is a general-purpose method, that it works automatically, and that it can significantly enhance student performance.

7 Conclusions

In conclusion, ASSERT is a general-purpose method 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 into future modeling efforts. Finally, the empirical evidence presented supports the two principal claims of this research: (1) that automatically constructed refinement-based models can be used to significantly increase student performance and (2) that a bug library can also be constructed automatically using multiple student models as input.

Acknowledgments

This research was supported by the NASA Graduate Student Researchers Program, grant number NGT-50732.

Appendix A. 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)

References

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

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, P. (1994). *Automatic Student Modeling and Bug Library Construction using Theory Refinement*. Ph.D. thesis, Austin, TX: University of Texas.

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, 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.

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

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

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

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

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, 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.

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

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.

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

Hoppe, H. U. (1994). Deductive error diagnosis and inductive error generalization for intelligent tutoring systems. *Journal of Artificial Intelligence in Education*. 5:27-49.

Ikeda, M. and Misoguchi, R. (1994). FITS: A framework for ITS – a computational model of tutoring. *Journal of Artificial Intelligence in Education*. 5:319-348.

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

Langley, P., Wogulis, J. and Ohlsson, S. (1990). Rules and principles in cognitive diagnosis. In Frederiksen, N., Glaser, R., Lesgold, A. and Shafto, M., editors, *Diagnostic Monitoring of Skill and Knowledge Acquisition*, chapter 10, pages 217-250. Hillsdale, NJ: Lawrence Erlbaum Associates.

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

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.

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.

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

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.

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

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, 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, B. and Mooney, R. (1995). Refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95-131.

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

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

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

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

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.

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, 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, 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, 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.

Young, R. M. and O'Shea, T. (1981). Errors in children's subtraction. *Cognitive Science*, 5:153-177.