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 mech capturing the conceptual state of the student, which enables the system to tail to suit individual strengths and weaknesses. To be useful such a modeling tecl 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 modeling technique which can automatically capture novel student errors using c domain knowledge, and can automatically compile trends across multiple studen 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 bas rectly 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 are necessary to account for the student's behavior. The efficacy of the appro demonstrated by evaluating ASSERT using 100 students tested on a classification task c ering 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 reteachi

1 Introduction

Student modeling has a long and interesting history, dating back well into to produce intelligent tutoring systems (ITS). The best method for constructing and usi student model is still the subject of much debate. Most student modeling t ever, 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 th domain,

Find: A representation accounting for any discrepancies between the expectations an 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 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 strengths and weaknesses, enabling it to be truly adaptive to the individua

Unfortunately, the difficulty of constructing and testing student models many researchers from pursuing further investigations into the field. Desp more than two decades of research has resulted in a wide variety of studen niques, the practical task of incorporating these techniques into a functi tem has proved to be a major roadblock. Furthermore, neither the utilit necessity of student modeling as a component of an ITS is a universally acc to the contrary, an interview of ten well-known ITS researchers which appea 1993 issue of *AI Communications* came to the conclusion that "most of the research longer believe in on-line student modelling." (Sandberg & Barnard, 1993). on to conclude that "instead of becoming more integrated, the field has diverged in the last few years. It appears that scientists in the field of e ogy no longer share a research paradigm."

Thus the current challenge for student modeling is to show that modeling be made both *practical* and *effective*. This is precisely the contribution of this wo embodied in the ASSERT algorithm (Acquiring Stereotypical Student Errors by Refining Theories). ASSERT was designed to show that student modeling is a viable tool f an effective tutoring system. By taking advantage of some of the lates machine learning, ASSERT is able to construct student models efficiently and auto catching both expected and novel student misconceptions. Also, it is the fi tem which can construct bug libraries automatically using the interactions dents, without requiring input from the author, and integrate the results future modeling efforts. In this sense, ASSERT is a self-improving student modeler. Final ASSERT can be used to significantly improve student performance, as will be sections which follow.

The remaining sections are organized as follows. Section 2 reviews previo dent modeling as a motivation underlying the design of ASSERT. Section 3 then presents a description of ASSERT focusing on the portion of the algorithm which captures indi dent errors. Next, Section 4 describes how trends across a population of s matically collected into a *bug library*, and how such a library is then incorporated in modeling process. Finally, Section 5 presents experimental results followe and conclusions in Sections 6 and 7.

2 Previous Work

2.1 Overlay Modeling

The application of artificial intelligence to the task of student modeling h progression of techniques for collecting information both about what a stu does not know, and about explaining the sources of misconceptions. The ea student models, embodied in systems such as SCHOLAR (Carbonell, 1970), WES & Brown, 1976) and WUSOR (Carr & Goldstein, 1977), used a form of modeling now generally referred to as *overlay modeling*. An overlay model relies on the assumptio that a student's knowledge is always a subset of the correct domain knowle dent performs actions which illustrate that he or she understands particul domain knowledge, these are marked in the overlay model. More sophisticated els can express a range of values indicating the extent to which the system understands a given topic using some form of truth-maintenance scheme (Finn ray, 1991). However the marking is achieved, typically the unmarked element are used to focus tutoring on new problem areas for the student, or to ensu the domain.

The advantage of the overlay is its simplicity; the elements of the mode directly on to the knowledge used to engineer the system. The disadvantage, tational restriction placed on the model—only missing elements of the cor can be modeled. Alternative notions which a student might have cannot be ca cally, this means that misconceptions cannot be modeled. Thus, overlay models can capt the notion of a student's lack of knowledge, but they cannot be used to m who knows of a topic but misunderstands it.

2.2 Bug Libraries

To address the limitation of overlay models, other researchers focused on c bases of student misconceptions typically termed bug libraries. The classic bug-library work was done by Brown, Burton and VanLehn (Brown & Burton, 1978; Burton, 1982; VanLehn, 1980), Sleeman and Smith (Sleeman & Smith, 1981), and Young and (Young & O'Shea, 1981), but a number of other systems can be said to inco form of stored misconceptions (Rich, 1989; Lianging & Taotao, 1991; Miller 1977; Quilici, 1989; Soloway & Johnson, 1984). With a bug library, models 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 buggy structures are encoded. However, two important problems remain with bug-library approach. First, the construction of such catalogs by hand is a consuming task which must be repeated for every new domain. Second, even if taken, the resulting library may still fail to cover a wide enough range of tion successfully. That is, a student may exhibit a misconception which wa by the author of the library. As with overlay models, the static nature of l them incapable of modeling unanticipated student behaviors.

2.3 Dynamic Modeling

To capture novel student misconceptions, one must turn to some kind of s space of possible bugs. Two methods have been tried to date: one attempts library and the other attempts to infer a model of the student from scrat machine-learning techniques. In both cases, novel errors are modeled by co buggy information dynamically, using data from a student's behavior to boun

2.3.1 Extending a bug library

Sleeman et al. (Sleeman et al., 1990) describe two extensions to their PIX INFER* and MALGEN, both of which can be used to extend a bug library. PIXIE ing system designed for the domain of high-school algebra whose goal is to priate feedback to improve student performance. PIXIE's underlying repres state-space paradigm, where the domain theory is a set of operators implem Misconceptions which comprise the bug library are encoded as faulty rule mal-rules. Both INFER* and MALGEN attempt to generate new mal-rules when the exhibits a problem that cannot be modeled using the mal-rules already in th difference between the two extensions is that INFER* attempts to patch spe dent solutions, whereas MALGEN generates and tests new mal-rules by alterin

domain theory. INFER* uses the rules it has to work forward from the problem
backward from the student's solution as far as it can. The remaining gap is filled by inferring
a new mal-rule. In MALGEN, formalized perturbation operators are used to ch
the domain theory to generate new mal-rules for the bug library.

The disadvantage of both systems is their reliance upon a user to decide
rules are appropriate extensions for the bug library. To their credit, the
issue and discuss potential filters that might be used to cut down on the nu
rules presented to the user. Unfortunately, to this point no general-purpo
nism which might be usable across domains has been found. In the end, the u
with a number of proposed mal-rules and must decide which ones are the "kee
more, since both systems were developed to address the problem of extendi
bug library, they remain strongly tied to the arduous task of preconstructi
hand.

2.3.2 Modeling by Induction

In an effort to avoid the cost associated with hand-constructed bug librar
turned to machine learning. Their inspired work on the ACM system (Langley
1984; Langley et al., 1990) was the first effort to harness machine learni
diagnosis of misconceptions through the use of induction. The underlying idea is to invent a
student model on the fly by automating the process of protocol analysis (Newell & Simon,
1972) which is used by human instructors as a means of unearthing student m

ACM uses a domain-independent induction algorithm to induce control know
resenting how students apply operators in a given domain. Starting with a s
eral operators, the goal is to induce a set of control rules that will g
sequence that produces the same solutions as the student. To model a par
ACM starts with only general knowledge of how the operators can be applied
for a path of operator applications connecting the problem specification to
tion. Given this "solution path," induction is performed by noting whether
ated by an operator application lies on or off the path. The output of in
conditions which predict when an operator will produce a state which lies
path. The conditions found by induction are then used to specialize the o
result is a procedure that models the student's unique problem solving beha

By using induction, ACM can operate automatically, constructing models
both correct and buggy knowledge. However, because the operators must initi
enough to model many kinds of behavior, both correct and incorrect, the p
space is huge. Langley et al. note this, and suggest various "psychological
ditions which can be applied to the operators to limit the search. They also
additional underlying first principles, representing a deeper level knowled
provide the basis for automated addition of such psychologically plausible
ever, the system is still fundamentally limited by the complexity of havi
model completely from scratch. This can only be remedied by collecting lar
data on each student or by imposing further constraints on the search spa
require finding such constraints by using the very human-intensive methods t
designed to avoid.

2.4 Tracing Techniques

One final style of student modeler bears mentioning because it represents a ious 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 ing, stopping whenever the student deviates from the correct procedure. A techniques must have knowledge of both correct and incorrect actions like l and must also have a mechanism for reproducing the steps followed by the ACM's solution paths. By focusing on the correct path as a bias, tracing sy 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 behavio interaction with the tutor to occur through menu selection. However, buildi student is not the primary focus of model tracing. Instead, buggy informati tem acts mostly to support the tracking process, with the goal being to kee straying too far off the correct path. Other tracing systems utilize a log tion (Costa et al., 1988; Ikeda & Misoguchi, 1993; Hoppe, 1994) where the i analytical approach, such as deduction or resolution, to search through a mine where a misconception lies. Essentially, whenever the rule-base fai "proof" which mimics the student's actions, the points where the proof fai dates for querying the user about his or her beliefs. Again the emphasis is ations from an expected path of correct student behavior.

Unlike the previous methods, tracing techniques do not dynamically cons models. Instead, they rely upon either the assumption that the student can low along the correct path or querying an oracle whenever a deviation is d 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 ide research presented here. First, modeling systems can increase their cove behavior by incorporating knowledge from a library of expected misconceptio be truly adaptive and to avoid the costs of bug library construction, one sort of dynamic modeling or learning algorithm. And third, tracing student parison to expected correct behavior can be an effective tool for detecti without requiring a great deal of search. ASSERT combines these ideas by using a relative new machine learning technique called *theory refinement* (Ginsberg, 1990; Ourston and Mooney, 1994; Craw and Sleeman, 1991; Towell and Shavlik, 1991). Theory refi method for *automatically* revising a knowledge base to be consistent with a set of Typically, the knowledge base is considered incorrect or incomplete, and the sent correct behavior which the knowledge base should be able to emulate. refinement procedure itself is blind to whether or not the input knowledge in any absolute sense; the theory-refinement process merely modifies the kn until it is consistent with the examples. Thus, one can also use theory refin a *correct* knowledge base and examples of *erroneous* behavior, and theory refinement will introduce whatever modifications are necessary to cause the knowledge base t 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 dom examples of erroneous student behavior, theory refinement will revise the kn make it consistent with the student, i.e., introduce "faulty" knowledge to dent's mistakes.[1] The refinements made to the knowledge base then represent a mod student, and can be used directly to guide tutorial feedback by comparing with whatever elements of the correct knowledge base they replaced.

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

3.1 Outline of Theory Refinement

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

ASSERT uses the NEITHER algorithm (Baffes, 1994; Baffes & Mooney, 1993) whic based on the EITHER theory-refinement system (Ourston & Mooney, 1990; Ourston, 1 EITHER was chosen as a starting point because it was the most complete sym refinement system available. NEITHER is designed to work with a propositional Horn-c knowledge representation. It takes two inputs, a propositional Horn-clause the *theory*, which is repaired using a set of input *examples*. The input examples are assumed to be lists of feature-value pairs chosen from a set of *observable domain features*. Each example has an associated category *label* which should be provable using the theory with the ture values in the example. NEITHER can generalize or specialize a theory, without use vention, and is guaranteed to produce a set of refinements which are consist examples.

Figure 1 shows an example theory and four input examples. The top of the part of a rule-base taken from a tutor ASSERT built for teaching a subset of C++ concepts (for a complete listing of the C++ Tutor rules, see Appendix A). The rules, numbered R consist of a consequent which is considered true for an example only when to its right are provable from the feature values of that example. Proposit represent either intermediate concepts or are shorthand for binary feature as a value. This simplified theory has only one category, "compile-error," classify examples. The input examples, shown in the table below the rules, classified as compile-errors only if they can satisfy rules R1 or R2 or b closed-world assumption is used to classify the example as non compile- instance, example 1 is correctly classified as a compile-error because it ca isfying either R6 or R7. Likewise, example 2 fails to satisfy any of the ru rectly classified as non compile-error.

However, examples 3 and 4 are misclassified by the theory in its current s

1. Keep in mind that the language used here is highly subjective in nature. One need not tak actions are "mistakes." The central point is that theory refinement can be used to detec 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-po
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-
pointer-init	true	false	true	fals
pointer-set	true	true	true	true
integer	constant	non-constant	non-constant	non-
integer-init	true	true	true	true
integer-set	through-pointer	yes	no	no

Figure 1: A Theory and Examples. The desired classificati
is shown in italics (thus, Examples 3 and 4 are misclass

rule base is "incorrect" since it does not produce the desired classificati examples. Propositional Horn-clause theories can have four types of erro Figure 2. An overly-general theory is one that causes an example to be cla ries other than its own, i.e., a false positive. EITHER specializes existing antecedents, add new antecedents, and deletes rules to fix such problems. An overly-specific t example not to be classified in its own category, i.e., a false negative. EITHER generalizes existing antecedents and learns new rules to fix these problems. ory-refinement systems that are subject to local maxima, EITHER is guaranteed to fix any arbitrarily incorrect propositional Horn-clause theory (Ourston, 1991).

Symbolic theory-refinement systems like EITHER use a combination of three computations to determine how to modify a theory. The first step is to *repair* a single failing example by analyzing the rule base to determine what r changed to fix the theory for the example. For a failing positive example, a cedents is found which, if deleted, will fix the theory for that example. A failing negative example, a set of rules is computed which, if deleted, wil

The second step is to *test* a repair for a single example against all the other i ples to see if the repair will cause new misclassifications to occur. If not be applied to the theory directly. Otherwise, a third step is taken using *induction* set of additional conditions which will separate the examples fixed by the examples for which the repair causes new misclassifications. These additiona then used to modify the repair.

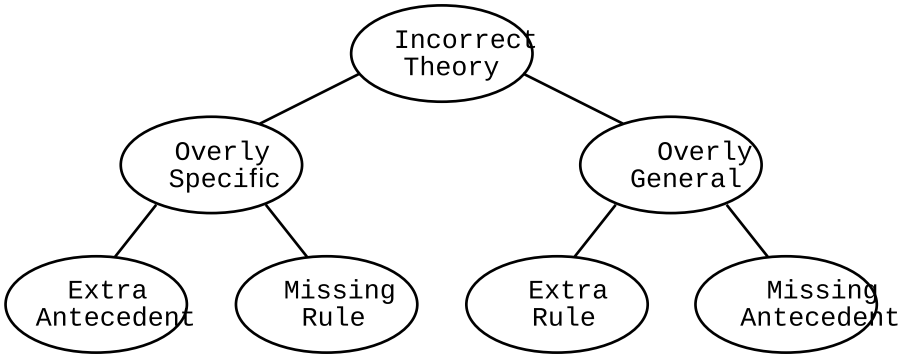


Figure 2: Theory error taxonomy for propositional Horn-

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

Finding a repair for example 4, however, yields a different result. The to delete the "(pointer constant)" condition from rule R3. However, when th against examples 1, 2 and 3, example 2 is erroneously classified as compile way to fix the repair is to remove the "(pointer constant)" condition fro example 4, plus add a new condition to the rule which keeps example 2 from rule. Passing examples 2 and 4 to an induction algorithm would return "(in the condition which can discriminate between the examples. The final revi 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 repairs for these examples. For instance, example 3 could have been classi error by removing the conditions "(integer constant)" and "(integer-set t from rule R6, or by removing the conditions "(integer constant)" and "(i from rule R5. For that matter, removing all of the antecedents from any on through R4 would also have repaired the theory for example 3, by making ei pile-error" or "constant-not-init" concepts provable by default. In fact, ble repairs for an example in the general case is exponential in the si Consequently, the way in which repairs are calculated, as well as when a re the theory in relation to computing repairs for other examples, can have a on the accuracy and performance of the theory refinement algorithm.

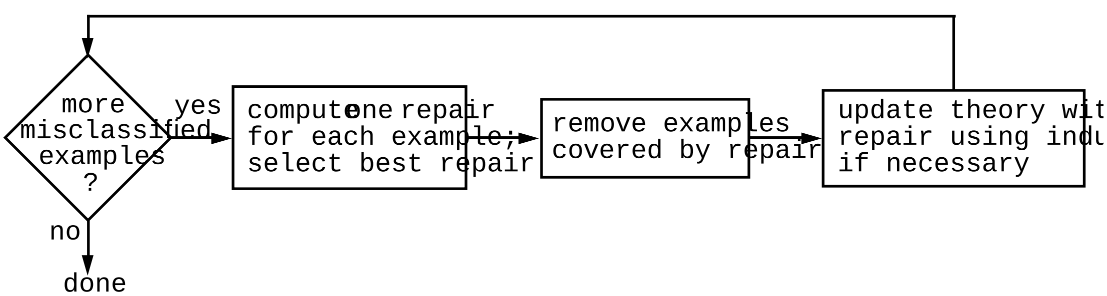
NEITHER and EITHER differ in the way the repair, testing, and induction comp theory refinement are combined. While a detailed comparison between the two scope of this article (see Baffes, 1994, chapter 3; Baffes & Mooney, 1993), NEITHER algorithm is provided in Figure 4. NEITHER opts for speed in computing repairs, focusing on quickly finding one good repair for each example. The goal is to est repair in the deepest possible part of the theory. After converting 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-po
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-po
R7: constant-assigned ← ~~(pointer constant)~~ pointer-init pointer-set
```

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

NEITHER uses a recursive routine which starts at the leaf rules of the theo
tions are collected at each rule and passed up to parent rules. When a choi
NEITHER always chooses the smaller repair, picking randomly to break ties. Eac
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 mo
ory with each repair, calculating how many examples it fixes and how many ne
cations it causes. These results are combined with the size of the repair
repair which fixes the most examples with the fewest new misclassifications i
repair is then tested against the rest of the examples, and induction is pe



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

Criterion for selecting best repair from AMONG examples
Select the shortest repair fixing the most examples with the few

Figure 4: NEITHER main loop.

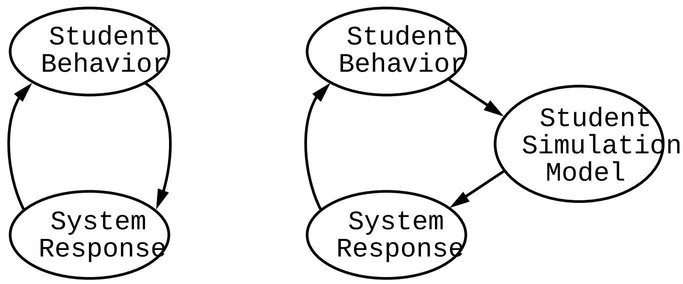


Figure 5: Abstract view of student-tutor interac

to modify the repair to avoid new misclassifications. The whole process is loop which continues until all misclassified examples have been accounted fo that NEITHER runs more than an order of magnitude faster than EITHER without losing accu- racy (Baffes & Mooney, 1993), giving response times that are on the order o This is critical to an interactive tutoring environment where feedback must the student in a timely fashion.

3.2 Overview of ASSERT

Having reviewed the basics of theory refinement, we can now turn to the deta ASSERT views tutoring as a process of communicating knowledge to a student contribution of the modeling subsystem is to pinpoint elements of the int base to be communicated. At its most abstract level, such a tutorial can b between the student and the system as shown in Figure 5. Many details are diagram and different researchers have chosen to emphasize different parts o first design decision then is one of emphasis: ASSERT focuses on the question of how to con- struct a useful interpretation of the student's actions. This decision is de ponent inserted into the diagram as shown in the right half of Figure 5. Th component, a student simulation model, implies that the system contains a knowledge that can be used to solve problems in the same context as the student mus knowledge base can be modified to replicate the solutions furnished by the s

3.2.1 The Student as a Classifier

Figure 6 depicts how ASSERT views student behavior. It is assumed that all actions a student can be broken down into 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 vec- tors describing some aspect of the problem. The task of the student is to pr each feature vector, selected from among some predetermined set of legal la student. The resulting set of labeled examples pairs each feature vecto selected by the student.

In its simplest form, a problem consists of a single feature vector prese in a multiple-choice format, where the answers available to the student are a list of possible categories. Thus, for example, the classic diagnosis pro

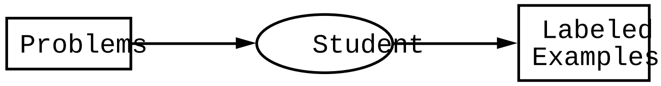


Figure 6: Student behavior diagram.

a patient's symptoms (the feature vector) and ask the student to select a d of diseases (the label). This allows ASSERT to be used in concept learning domains, whi are common applications for automated training systems. It also means that will translate directly into a form usable by theory refinement, which requi ples as one of its inputs.

3.2.2 Modeling by Theory Refinement

Once collected, the labeled examples generated by the student are passed refinement component of ASSERT, as depicted in Figure 7. As discussed previously, refinement will take an incoming knowledge base, plus an incoming set of e 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 th duces the same answers as the student, i.e., will classify each feature ve category label as the student. The resulting refined rule base is thus able dent's behavior.

The use of a propositional theory-refinement algorithm for modeling carr assumption that the author of the tutoring system will be able to provide a tion of the domain using propositional Horn-clauses. This places a good dea how the correct rule base is constructed since it becomes the language ASSERT interprets the student's actions. If the correct rules are expressed low a level of detail, the ability of the system to form accurate models wi course, this type of knowledge representation problem exists for all tutor ever, ASSERT gains an advantage by purposely isolating the correct domain kno separate component: the author can easily change the focus of the tutor by rect rule base. Moreover, if students possessing different levels of unders tutor, multiple rule bases can be written to give the system more flexibilit

3.2.3 Refinement-Based Remediation

The last component of ASSERT, the system response, is outlined in Figure 8. Us

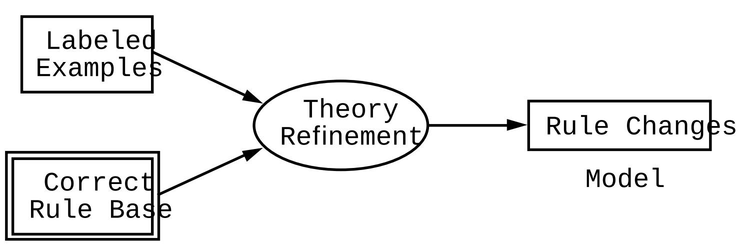


Figure 7: The student simulation model.

Figure 8: System response diagram.

refinements produced by EITHER, ASSERT generates explanations and examples to reinfo the correct form of the rule or rules modified. The underlying approach, ca refinement-based remediation, is based on fundamental units of explanation called 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 EITHER, ASSERT provides two functions: the ability to explain a corr the rule which was changed, and the ability to generate an example which us designer of a tutoring system using ASSERT has the option to generate multiple explanat or examples, to determine the circumstances when such feedback is given, whether the system or the student controls which explanations and examples By providing such explanation-example units, ASSERT supplies the raw materials for a vari- ety of remediation techniques.

The specifics of generating explanations and examples for each refinement in detail in (Baffes, 1994), but the underlying idea is straightforward. Ex describing how the correct form of the rule (not the revised version) fits correct rule base. Each rule has an associated piece of stored text, descr rule base. A full explanation is generated by chaining together the stored lying on the proof path for the correct label (not the student's label) for label which is produced by the correct rule base for the given feature vect

Generating examples is a bit more complicated since they are dynamically constructe rather than being drawn from storage. Recall that each refinement made by EITHER results in the addition or deletion of propositional literals from a rule in the th be added or deleted as well, but this is the same as adding or removing b Using normal deductive methods, the added and removed literals can be trac feature vector. The result is a set of conditions in the feature vector w ignoring or a set of extra conditions not present in the feature vector whi are necessary. 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 be counter example to the student, and the various added or missing conditio Note that this corresponds very closely to tutorial methods outlined for co (Tennyson, 1971).

Figure 9 shows an explanation and example pair, corresponding to one of t depicted previously in Figure 3. Recall that the last rule of Figure 3 was one of its antecedents. Figure 9 is the feedback generated for this missin trating how the condition represented by the antecedent is essential to dra clusion. The top half of Figure 9 shows the text which explains how the r overall rule base. As part of the explanation, the three conditions of the its three correct antecedents, are itemized at the end of the explanation.

```
EXPLANATION

One way to detect a compilation error is to look for an identi
declared constant and initialized, then later assigned a new va

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

Specifically, note the following which contribute to this type
of error:
* There must be a pointer declared to be constant (but not nece
  pointing to a constant object).
* A pointer declared to be constant must be initialized.
* A pointer declared a constant and initialized must be set aft
  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 co
  integer, so it does not have to be initialized and it can be
```

Figure 9: Example remediation given to a studen

generated which highlights how the "(pointer constant)" condition bears answer to the example, showing how the truth or falsity of the condition le conclusion.

Finally, Figure 10 combines the components from Figures 6, 7and 8, show dialog flows between the student and the system. Problems given to the studen 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 the student. The refinements are then used to generate explanations and examp ation which gets passed back to the student.

4 Extending ASSERT's Modeler

The previous three sections have described the ASSERT algorithm, showing how

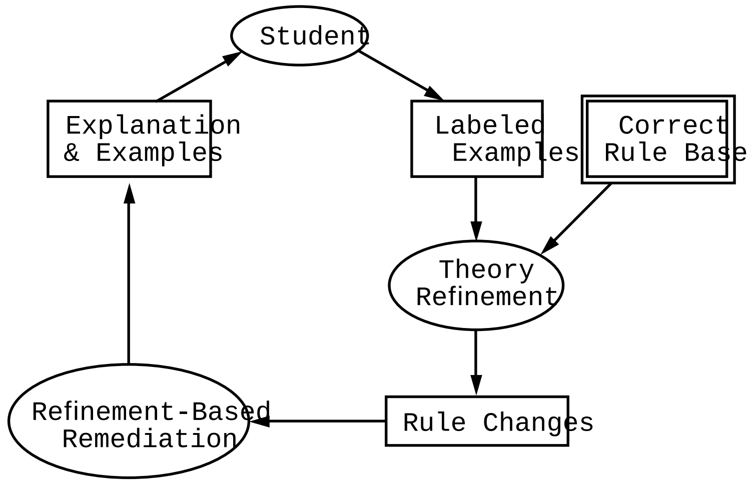


Figure 10: Basic design of the ASSERT algorithm.

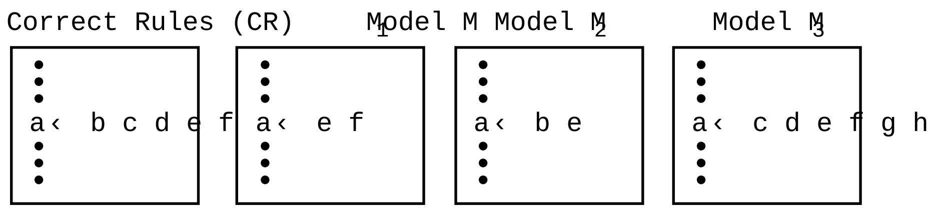
the flow of information between student and system can be implemented as refinements that highlight the differences between how the system and the s the same set of problems. As such this explains how ASSERT constructs student models by tracing the student's behavior against a known standard rule base. Nothing, said about how multiple student models are mined to construct a bug library library is incorporated back into the modeling process.

4.1 Building a Bug Library

A bug library represents a collection of data gleaned from the interactio dents with the tutoring system. ASSERT uses the rule changes output from theory refinem for each student as the basis for constructing its **bug library**. This gives ASSERT two an tages. First, the rule changes are closely related to the type of input ger of the tutoring system. Since a rule base must be supplied as input, express in terms of changes to that rule base is an effective way to communicate bu back to the author. Second, the rule-change format is precisely what NEITHER uses to simu late the behavior of the student. A bug library built of rule changes is th which can be incorporated directly into the modeling process.

ASSERT constructs a bug library in three stages. First, it collects cop changes from all the student models together, eliminating any duplicates. each rule change by a measure of how frequently it occurs among the variou els, called the *stereotypicality* of the rule change. Third, in the process of ranking change, ASSERT tests generalizations of the change to see if they result in b cality. If a generalization is found which has superior stereotypicality, th replaced by the generalization. The final bug library contains the best gene rule change with any duplicates removed.

Figures 11 through 14 illustrate how a bug library is constructed for a generated for illustration purposes. Figure 11 shows how stereotypicality i models are shown at the top of the diagram, each of which changes only one rect rule base. All the models alter the same rule, but in different ways. first model changes the rule a ← b c d by deleting the set of antecedents {b, c, d}. The



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

Distance from Correct Rules to Models:

distance(CR, M_1) = delete{b,c,d} = 3
distance(CR, M_2) = delete{c,d,f} = 3 total = 9
distance(CR, M_3) = delete{b} add{g,h} = 3

Distance from Correct Rules with bug from M_1 to Models:

distance(CR+B_1, M_1) = 0
distance(CR+B_1, M_2) = delete{f} add{b} = 2 total = 6
distance(CR+B_1, M_3) = add{c,d,g,h} = 4

Distance from Correct Rules with bug from M_2 to Models:

distance(CR+B_2, M_1) = delete{b} add{f} = 2
distance(CR+B_2, M_2) = 0 total = 8
distance(CR+B_2, M_3) = delete{b} add{c,d,f,g,h} = 6

Distance from Correct Rules with bug from M_3 to Models:

distance(CR+B_3, M_1) = delete{c,d,g,h} = 4
distance(CR+B_3, M_2) = delete{c,d,f,g,h} add{b} = 6 total = 10
distance(CR+B_3, M_3) = 0

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

Figure 11: Stereotypicality computation.

refinement for model labeled M_1, 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 co each bug decreases the distances between the models and the correct rule base. The di between CR and the student models is shown followed by the distances to the each of the three bugs is added to CR. Calculating the distance between two rule s amounts to counting the number of literal changes required to convert one t changing CR+B_1 into M_2 requires changing the rule a ← e f into the rule a ← b e which is done by deleting f and adding b. The bottom of the figure has the stereotypicality va each of the bugs.

Note that a bug may have a negative stereotypicality. Unless a bug is pre half of the student models, it is likely to have a negative stereotypicality overlap between the bug and the majority of the models. Thus even a bug that 30% frequency in the student population may have a negative stereotypicality.

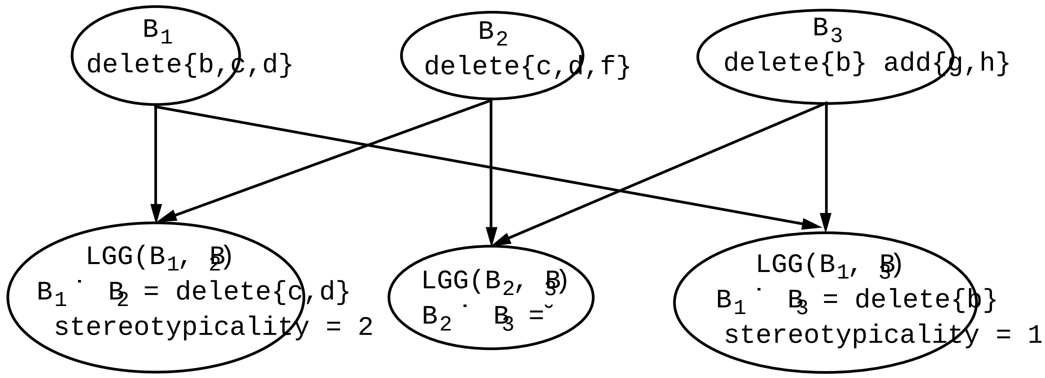


Figure 12: Bug generalization using the LGG opera

is the relative difference between stereotypicality values.[2]

Figure 12 illustrates how ASSERT forms generalizations among the bugs from Figure
Since any refinement to a propositional theory can be expressed as a logic
compute generalizations using the least general generalization (LGG) operator (Plotkin, 1970). When two propositional logic clauses are not identical, one can form of the two by dropping literals from the clauses. Any number of literals ma the most specific (i.e., least general) way to generalize the two clauses is literals which appear in just one of the two clauses. This is the same as tion between the two clauses. Since refinements in EITHER are collections of propositional logic literals, the LGG of two refinements is simply their intersection.

As might be expected, the LGG will often form a generalization that has bett than a refinement from which it was taken. For instance, the LGG(B_1,B_2) value ... B_2 which is 1. Likewise, LGG(B_1,B_3) is better than B_3 alone. This will be the result whenever the operation captures more of what is common among the models, and avoids more uncommon. However, note that the LGG is not beneficial in all cases; the same tioned 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, be continued, forming LGG's from LGG's which can also result in better or wors

Figure 13 shows the pseudocode for constructing a bug library. The fundam perform a hill-climbing search using successive LGG operations. Starting with a seed, multiple calls are made to the LGG operator to combine the seed with a ments from the models. As long as this continues to result in a better gener passes are made over all the refinements. The process halts when no generaliza which will improve upon the seed, which must eventually happen since continu

2. As a computational note it should be pointed out that it is not necessary to compute the rule bases to calculate stereotypicality. Instead, the difference between distances can be of the differences taken. So, for example, let D be the distance between the correct rule base and a model. ing a bug to the correct rule base changes a single rule in D, the change in D, can be calculated by finding the overlap between the refinements of the bug and any refinements made to the same rule in the model. Whatever is in the bug that overlaps with the model decreases the distance to the model because it means those literal changes already exist in the model. Anything in the bug which does not increases the distance because those literal changes are not in the model. The computational complex rithm is linear in the number of literals in the refinements of the models.

```
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;
end
```

Figure 13: Pseudocode for bug library constructi

between refinements will eventually produce no change or the null set. The be found starting with each refinement as the initial seed is kept and inserted in duplications in the library are eliminated and the results sorted by stereotyp

Finally, Figure 14 shows a complete example for constructing a bug libr bugs from Figure 11 plus one extra bug to highlight the hill-climbing nature Below the four bugs are a series of boxes, each representing one iteration Thus the first box is the iteration which computes the bug to be added to t with B_1 as a seed, the second starts with B_2 as the seed, et cetera. After saving the ster icality of B_1 the inner loop is entered and an LGG is formed between B_1 and the other three bugs. Note that there is no need to compute LGG(B_1,B_1) since the result is simply B_1 which obviously cannot be an improvement. Once the LGG's are computed, the best this case LGG(B_1,B_4), which has a stereotypicality of 4. This is compared with best stereotypicality, and since there is no improvement the inner loop ha B_1 to the bug library. The second box, for bug B_2, also yields no improvement from generalizati 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 poin value for B_3 alone. Consequently, the inner loop repeats. A second round of LGG 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 im which cannot be surpassed by a second iteration of the inner loop. Note th 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 sorted in the following order: B_1, delete{b,c}, B_2, and delete{c}

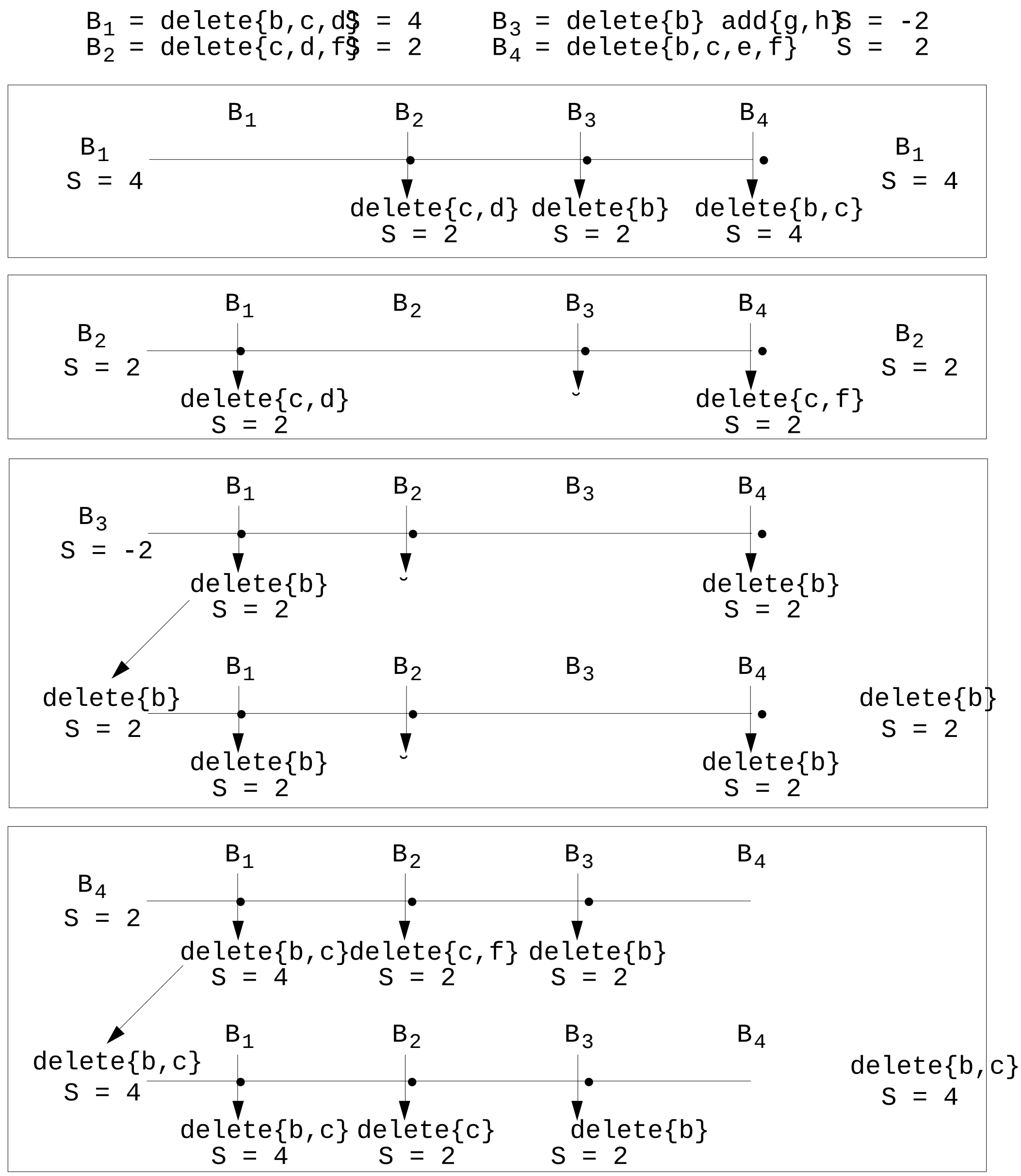


Figure 14: Bug library construction example. “S” stands

4.2 Using the Bug Library for Modeling

Once the library is built, the question becomes how to use its informati modeling process. Having the bugs in the library ranked by stereotypicali important advantages. First, stereotypicality is an indication of how commo occurs in the student population. Bugs which occur frequently among the stu in higher stereotypicality since they will overlap more often with the mode students. Additionally, because stereotypicality is measured in literal add

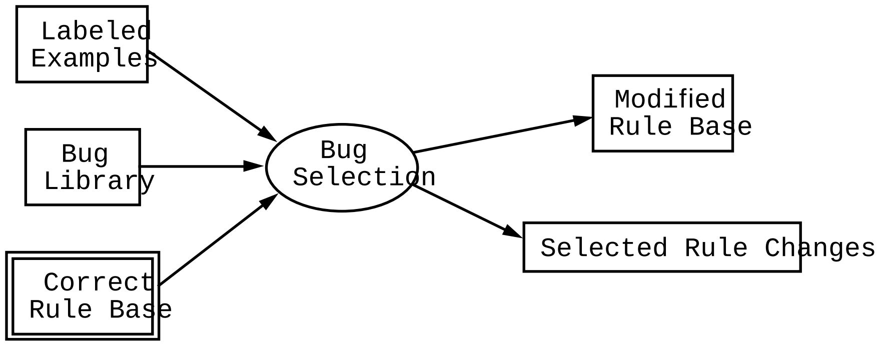


Figure 15: The bug selection module. Note that the correc
with the selected rule changes is equivalent to the mo

it is directly related to how EITHER might have by integrating the bug with th correct rule base.

Perhaps the most obvious way to incorporate the bugs in the library is to refinement algorithm to use the bugs as a means for selecting repairs to fix a student. So, for example, EITHER computed a repair for a failing example its s might include any bug in the library which applied to any rule which could l the failing example. The disadvantage of this approach is that it would des ity of ASSERT's design. Theory refinement would no longer be an interchangeable which could be swapped out for different refinement algorithms. A simpler ap one used in ASSERT, is to modify one of the inputs given to EITHER, leaving the refinement algorithm intact. Specifically, the correct rule base is modified before the begins by incorporating elements of the bug library which are relevant to t

Figure 15 shows a schematic for how this is accomplished. The bug libra rule base, and the student's labeled examples are input to a process which the library to be added to the rule base using a hill-climbing search. Bu upon the predictive accuracy of the rule base are added incrementally. The fied rule base which resembles the student's behavior more closely than the which may still be incomplete. Note that the bugs which were selected are

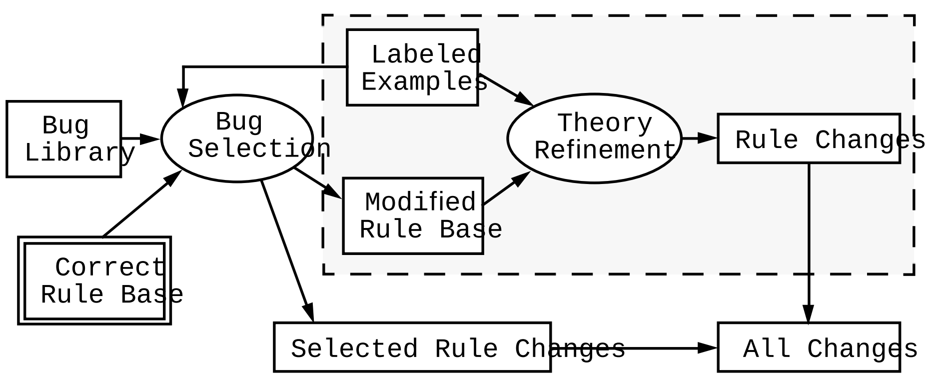


Figure 16: Extended modeling. Bug selection combined with t

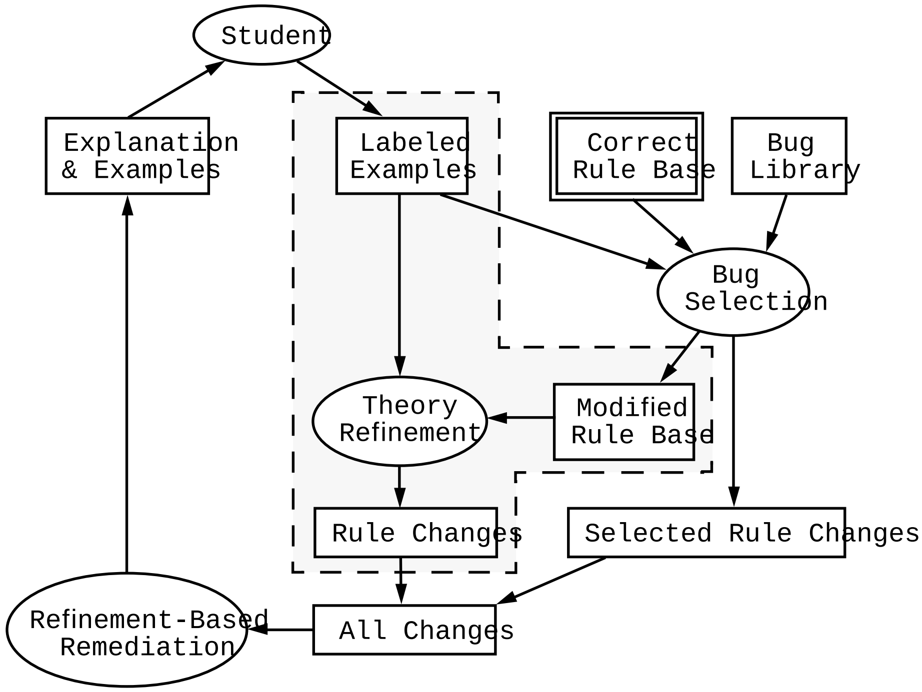


Figure 17: Overview of extended ASSERT algorithm. The shaded area rep
the theory refinement component.

with the modified rule base since they must be included with the final model o
shown in Figure 16. Once the modified rule base in constructed, it is passed
ment along with the labeled examples to determine any additional refinements
reproducing the behavior of the student. All rule changes, whether selected
or constructed by theory refinement, are returned as the final student model.
the entire ASSERT algorithm.

The pseudocode for modifying a rule base is shown in Figure 18. Modify-Rules starts with
the correct rule base, and loops as long as a bug can be found which will incr
the rule base on the set of labeled examples. The first step of each loop is t
the rule base when the bug in question is added to the set of rules. All thos
improved accuracy are saved. Next, the bug which increases accuracy the most
inner loop is entered to pare down the list to only those bugs whose improvem
“statistically equivalent”[3] to the best bug, using a paired Student t-test.[4] Finally, if there are still
multiple bugs left, then the one with the greatest stereotypicality value i
to the current rule base (random selection is used as a final tie breaker). W
found that increase the accuracy of the rule base, the routine quits return
version of the rules.

As an example of bug library selection, refer to a trace of the execution of

3. more precisely, whose improvement in accuracy is less, but not statistically significantly
4. Since the accuracy values for all the bugs are computed using the same set of labeled e
dent, one can use a paired Student t-test to estimate if the difference in accuracy betwe
tistically 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;
  return R;
end;
```

Figure 18: Pseudocode for bug library use.

dent” routine shown in Figure 19. This function is the implementation of ModifyRules
pseudocode used by the C++Tutor mentioned in Section 3.1. The trace shown is taken f
dent 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
dents who used the C++Tutor. However, only a portion of the bug library is actually
trace, corresponding to those bugs which were applicable to the mistakes made
student. For a complete listing of the bug library, as well as an entire trace
tion with the system, see (Baffes, 1994).

Each iteration corresponding to the outer ModifyRules loop is separated in Figure 19 by a
dotted line. For the first iteration, the original accuracy of the correct rul
Each bug in the library is added to the correct rules, and the six bugs which
accuracy are printed out with their stereotypicality values. Since all the bug
of the correct rules by the same amount, all are candidates for addition to t
paired t-test yields no statistical difference in accuracy among the bugs). Us
break the tie eliminates the last two bugs, but still leaves the first four whi
icality values. As a last resort, random selection is used to pick bug 20 as th
rule base.

Having selected bug 20, its refinement is added to the rule base which no
racy of 85% because of the addition of bug 20, as shown at the top of the
All the bugs of the library are now applied to this updated theory to che
improvement in accuracy. This time only four bugs are found, all of which a
same increase in accuracy. Bug 5 is a clear winner based on stereotypicalit
as the bug for this iteration. Note that bugs 10 and 34, both of which re
ments during the first iteration, are no longer useful for increasing accu
iteration. Also notice that bug 5 did not increase accuracy during the first

```
> (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-i
  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 19: Trace of bug selection from the bug lib

the addition of bug 20 enabled bug 5 to have its effect. This is a beauti ordering effects inherent in selecting bugs from the library.

At the beginning of the third iteration, the updated rule base, which refinements from both bug 20 and bug 5, has an accuracy of 90%. Only bug improve upon this accuracy so it is selected as the third bug to be added that while bug 29 continually resulted in improvements in accuracy, its rel typicality prevented its addition to the rule base before this point. Final results in no further improvement.

Perhaps the most important feature of ASSERT's bug library algorithm lies in its abil model both common and unique misconceptions. As with other bug-library bas methods, the ability to use a cache of expected errors gives the modeler a in domains where a large amount of data would otherwise be required for an nosis. But because the bug library here is used as a precursor to theory ref ASSERT is not restricted to using only those bugs present in the library. Any specif not in the bug library can still be captured by the theory refinement compo rithm. Thus the final theory of Figure 19 has partially accounted for the st 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. Furthe fully automated scheme for bug library construction, and by integrating the its automatic modeling algorithm, ASSERT can continue to improve its modeling accura over time.

5 Experimental Results

It can be argued that the ultimate test of any tutoring system design is results in enhanced student performance. This is especially true for studen use of a model cannot significantly impact the educational experience then t son to construct one. Furthermore, this evidence must come from experime large numbers of students in a realistic setting so that the significance determined.

In this section, evidence is presented in support of the claim that the ASSERT design can be used to construct tutors 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 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 a which student responses were simulated. The advantage of this simulation d 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 dif C++ students: ambiguity involving statements with lazy operators and the pr and use of constants. These two concepts plus examples of correct program categories into which example programs could be classified. A set of 27 dom developed to classify problems, using a set of 14 domain features, as bein 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 b ambiguous or a compile error. Figure 20 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 their participation. As an added incentive, the material in the tutorial co would be present on the course final exam. This established a high level of m the students who participated in the test. Due to the large number of stud tutorial was made available over a period of four days and students were reserve time slots to use the program. In total, 100 students participated

Three major questions were the focus of the C++ Tutor test. First, it was important establish whether ASSERT could be an effective modeler for students in a real

```
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
             (C) neither A nor B ?
```

Figure 20: Example C++ problem

ting. This was measured by testing the model produced for a student on a 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 th base with no modifications. Second, even with a perfect model one may not se in student performance. Though a model may be accurate in predicting what 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 that model and measure any change in the student's performance. Our hypoth remediation generated using models built by ASSERT would result in increased student performance over a control group which received no feedback. Additionally, i that students who were modeled with the benefit of a bug library would see mance increases over students who were modeled without a library. Third, as dent modeling studies (Sleeman, 1987; Nicolson, 1992) we wanted to test receiving feedback based on student models would compare against students simple form of reteaching feedback. In this case, the expectation was that on modeling would result in greater post-test performance than simple retea

Testing these three hypotheses was accomplished with three experiments: o the effects of remediation, another to measure the accuracy of modeling an the utility of the bug library. In the next three sections each of these tes

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 co had no feedback. The expectation was that these four groups would exhibit formance on a post-test as the remediation ranged from full ASSERT to bug library to reteaching to no feedback.

To test whether ASSERT can impact student performance, one needs to collect i tion for each student that has certain characteristics. To begin with, dat both before and after any feedback given to the student to detect any chang Thus the C++ Tutor was constructed as a series of two tests with a remediati between. Secondly, the data from the two tests must be equally representa dent's capability and must be collected in similar ways. The only way to de

information from the tutoring program to the student is to have both test
topics from the domain at similar degrees of difficulty.

To that end, a program was written to generate 10 example questions usin
format as follows. Since each question from the C++ Tutor can be classified into one of thre
categories, the 10 questions were divided equally among the categories: thr
correctly labeled as compilation errors, four were examples of ambiguous
three were questions with no errors. This process was used to generate two
tions, both of which covered the same subset of the correct rule base. Thi
two sets of questions covered the same concepts at the same level of diffi
two questions were identical. These two sets of questions represented the
test to be given to each student. One set of questions was used as the pre-
dents, the other as the post-test, thus the same pre-test and post-test wa
dent. To discourage cheating, the order in which the 10 questions were
randomized. This meant every student answered the same two sets of question
difference was the feedback given between the pre-test and post-test.

Students were randomly assigned to four groups of 25, each of which recei
kind of feedback from the C++ Tutor. One group of 25 received no feedback, acting
control group. This group was labeled the “No Feedback” group. The other
were given feedback using explanations and examples as described in Sect
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 f
explanations for each student.

One feedback group received a form of automated reteaching. Specifying p
is meant by “reteaching” is extremely important, as it can have a profoun
results of the experiment. Furthermore, reteaching methods vary, making it i
ify the exact approach used. For this experiment, the essential point was t
feedback based on modeling made any difference over feedback based on no mo
To that end, we chose an automated form of reteaching which used no informa
student, not even which answers the student got right or wrong. In such a
vacuum, the option left for reteaching is to select at random from the rule
tion. Thus, for the “Reteaching” group, ASSERT selected four rules at random from the ru
base, and an explanation and example was generated for each rule.

The other two groups received feedback based on the models constructed f
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
system. For both these groups, bugs were selected for remediation based on
were found by NEITHER.[5] For the ASSERT-Full group, bugs from the bug library were gi
preference to those found by NEITHER, in order of their stereotypicality value. In
ASSERT-Full and ASSERT-NoBugs groups, if fewer than four bugs were found, the re
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 rand
ASSERT-NoBugs, Reteaching and No Feedback groups. The models from these firs
dents were then used to construct a bug library. The remaining 55 students

5. NEITHER orders its refinements by preferring those which increase accuracy the most with the

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 percenta
swered correctly. ANOVA analysis on average increase resu
between all groups except between ASSERT-Full and ASSERT-NoBugs and
between Reteaching and No Feedback.

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 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 student between groups, significance was measured using an ANOVA test. As the onl between groups was the feedback received, the significance test used was a ANOVA test at the 0.05 level of confidence using Tukey's multiple compari (Tukey, 1953). The average improvement in performance for the four groups Table 1.

The results of the experiment confirmed most of our expectations. As predi age performance decreased as the feedback varied from ASSERT full 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 of variance among the mean pre-test scores in the four groups. Though non ences among mean pre-test scores is significant, their variance is a concern the significance of the differences in average increase from pre-test to p this is precisely why the ANOVA test was run to compare the significance. Wha cluded from Table 1 is that ASSERT-style feedback based on a model of the student **can** sig-nificantly 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 mu depends upon the size of the pre-test data given to ASSERT's model, or what the performance will be for other forms of modeling or reteaching. What has been illus 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 illus ASSERT can be used to build a tutorial that significantly impacts student perform models and bug libraries are automatically constructed using only correct domain. Furthermore, it is another argument in favor of the use of studen shows (1) that they can have significant impact over not modeling at all and

be constructed automatically without resorting to the time-consuming task o
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 correl
ability of ASSERT to produce an accurate model and an improvement in student per
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
This can be accomplished using an ablation test format, in which various p
ASSERT are disabled and the resulting systems compared based on the accuracy of t
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 mod
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
nique, 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
the library.

In the C++ domain, only the data from the No Feedback group is useful for
test. This is because no remediation occurred between the pre-test and pos
dents in this group; thus, their 20 questions could be treated as a sing
training set and test set examples could be drawn. These training-test spl
so as to be equivalently representative across the correct domain rules. Su
quality is important to maintain so that any effects from modeling with th
manifested in the test set. Therefore, the 20 examples from the pre-test
grouped into 10 pairs, where each pair consisted of the two examples (one
and one from the post-test) which covered the same domain rule. Then, trai
splits were generated by randomly dividing each pair.

The result was 2^{10} possible training-test set splits. For each of the 25 No F
dents, 25 training-test splits were generated, yielding 625 samples for co ASSERT-
Full and ASSERT-NoBugs. Each system was trained with the training set and acc
measured on the test set by comparing what the system predicted with what t
the No Feedback group actually answered. The results are shown in Table 2.
purposes, we also measured the accuracy of both an inductive learner, using
ing and test set splits, and the correct domain rules. The inductive learne
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 (Quin
the correct theory no learning was performed, i.e., the correct domain rule
out modification to predict the student’s answers. Statistical significance wa
a two-tailed Student t-test for paired difference of means at the 0.05 leve

These results illustrate that the groups with significantly better models ASSERT-Full and
ASSERT-NoBugs, are precisely the groups which performed best after remediati
This is further evidence in support of the fact that more accurate student
late directly to improved student performance via more directed remediatio
reinforces the finding of other studies (Ourston and Mooney, 1994) that ind
are simply not as effective as theory refinement in terms of accuracy.

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

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 sig the performance of the student as expected, casting doubts as to its utilit bug library did no harm to post-test performance, and perhaps with more dat between the two groups would indeed have been significant. Thus it would be more about the conditions under which a bug library, as constructed aut ASSERT, might be expected to impact the modeling process.

A series of tests designed to address this question, described in detail can be summarized by the results shown in Table 3. This data was generated ablation tests like the one described in the previous section. In such a t simulated by modifying a correct theory using six standard bugs selected plus additional random rule changes. The modified theory was then used "answers" for 180 feature vectors representing a hypothetical "multiple-cho answers were then passed to ASSERT to see how well it could reproduce the modified th Once this was done for 20 students, resulting in 20 student models, the mo bined to build a bug library.

Table 3 is a comparison of three libraries constructed using this technic ing numbers of simulated students and varying amounts of example answers pe first library, denoted 20-180, was formed from 20 student models built wi example answers per student. The second library, 20-12, used 20 students example answers per student chosen randomly for each student from among th ble. The 100-12 library was built from 100 students answering 12 randomly tions. Conceptually, these three libraries were designed to compare t conditions: a few students answering lots of questions, a few students answ tions, and a large number of students answering a few questions. This compa 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 effe ies to emerge from a large number of students answering a more reasonable n tions. If useful bug libraries cannot be constructed from small student utility of ASSERT bug library is limited since one would still be tied to co amounts of data on some students to construct the library. While more st 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 i

Part (a) of Table 3 compares the three libraries based on size and on how

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

(a)

System	Accuracy using different starting Bug Lil		
	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
total number of bugs found, part (b) compares accuracy

standard bugs were found. The 20-180 library performed the best, finding a
bugs. By contrast, the 20-12 library, which used the same number of student
data per student, contained only two of the six common bugs. That result
fied—by drastically reducing the amount of information on each student, indi
much less likely to be found and thus much less likely to end up in the bug
note that the 100-12 library contained four of the six common bugs, which
ment over the 20-12 library. Consequently, even though having smaller amour
student reduces the chances that a common bug will be found for any give
increasing the number of students improves the likelihood that the bug wi
some student. Again, this is not too surprising when one compares the total
ples used to build the student models which served as input to the three li
180 library there were 3,600 total examples, whereas the models used for
built with only 240 total examples. The 100-12 library, with 1,200 exam
bridges the gap in total examples and in number of common bugs found.

However, note that the total overall size of the 100-12 bug library is la
three libraries. In fact, the 100-12 library has the lowest ratio of commo
in the library. This dilution is a potential problem when the bug library i
modeling efforts. Recall from Section 4.2 that the bug library is treated a
refinements which is traversed in an effort to improve the accuracy of the r
is passed NEITHER. Increasing the size of the bug library widens the search
potentially less likely that the common bugs will be selected, which could,
the modeling accuracy.

This concern is addressed in Part (b) of Table 3 which shows results fro
tests aimed at determining if the ratio of common bugs to library size is d

of modeling accuracy. For these tests, a new crop of simulated students w each ablation test was run with this same set of students, varying the bug student only 10 of the 180 examples were used for training, leaving the re testing. As the numbers in the table show, the 20-12 library results in almo for ASSERT-Full and ASSERT-BugOnly (which used just the library without NEITHER) as opposed to ASSERT-NoBugs. By contrast, when the "better" bug libraries are used racy of ASSERT-Full and ASSERT-BugOnly is significantly better, with the 20-180 l performing the best. Even the 100-12 library, with its low ratio of common size, resulted in a significant performance improvement for ASSERT-Full and ASSERT-BugOnly.

This implies that a bug library can be incrementally improved as more st with the system, even if the student interaction is moderate, resulting in eling. And as the data from Table 1 and Table 2 shows, more accurate models remediation and improved student performance. Recall, also, that in the rem Section 5.1.1 the bug library was constructed using data from only 45 stude more students interact with the C++ Tutor by taking tests that cover different subset correct domain rules, it seems likely that a better bug library could be would lead to more accurate modeling and, in turn, better post-test perform

5.2 Subjective Evaluation

Student Response to the Tutorial

Perhaps the most difficult topic to measure objectively is an evaluation o dents enjoyed using the tutorial and whether they felt the experience was b vast majority of students who used the C++ Tutor the response was positive. Many studen made an unsolicited effort to express an appreciation for the opportunity t Several students outside of the experimental group heard about the experime use the tutor to refresh their C++ skills. There were even a few students who expressed pointment that the tutorial did not cover more material

On the negative side, students complained about their inability to back u test and post-test to change answers, which was not allowed by the interfac cially true during the early part of the pre-test, when students were stil selves with the interface. Also students expressed a boredom with redun showed up in the explanations during remediation. Many students had mul detected in similar parts of the rule base. As a result, the chains of rule to generate an explanation overlapped, resulting in duplications in explana problems could be easily fixed to make the interface more robust.

But perhaps the most important factor responsible for the positive respo that the feedback given to the student avoided negative language as much course, the student was told which questions he or she got wrong, but the e wrong answers did not focus on the student's mistake. Instead, the explanat correct reasoning, followed by an erroneous counter example which looked l the student might misclassify, and explained why the counter example was rather than saying something like "here's what you did wrong" the system "here's the right way to do something and, by the way, here's an answer whi the following reasons." This impersonal style of feedback may have made it student to accept the tutor's evaluation. And finally, by giving the student

to perform via the post-test, students were able to apply what they had l
average case, achieve a much better score. Such a concrete sense of improv
also contributed a great deal to the positive student response.

“Correctness” of the Bug Library

In the simulated student tests, the correctness of the contents of a bug li
sured directly by counting how many of the six common misconceptions ende
library. With the C++ Tutor domain this is not possible, since there is no a prior
about what the common misconceptions might be. However, one can perform a
evaluation with a domain expert to determine whether or not the bugs are
explanations of why students made their mistakes. This was done by consulti
tor for the course with the result that some of the bugs did, in fact, app
For example, several bugs in the library represented missing conditions, cr
erroneous constant declarations, which the instructor felt students typi
library also contained bugs capturing the notion that students failed to u
logical operators “AND” and “OR” were fully evaluated in C++, as the instructor suspected.
While this is an admittedly weak evaluation, it does at least illustrate
ended up in the library could communicate information about trends in st
which made sense to the instructor.

5.3 Summary of Results

To recap, the main result presented in this chapter was that ASSERT was shown to signifi-
cantly improve student performance in a test involving 100 college level
C++ Tutor developed with ASSERT. Furthermore, it was shown that those students for
ASSERT was able to construct significantly better models were the students
mance improved the most. And while the use of a bug library did not signifi
student performance, additional evidence was presented demonstrating the
the contents of the library could improve over time so as to significantly i
eling process. This empirical evidence supports the two principal claims of
that automatically constructed refinement-based models can be used to signifi
student performance and (2) that a bug library can be constructed automati
that can enhance refinement-based modeling.

6 Discussion

Several important issues have been raised by this research that must be em
place the work into a proper context. The first issue concerns the type of
modeled. Unlike the previous modeling efforts, which focus on procedural ta
ASSERT is
designed for use in classification domains. As an example of this difference
student modeling efforts have focused on the domain of writing computer pro
and Goldstein, 1977; Soloway et al., 1983; Soloway and Johnson, 1984),
research was tested using a classification task where students were asked t
rectness of program segments. This tie to classification domains is largely
the most mature theory-refinement algorithms developed thus far are designed
tion and is not a limitation of the general framework of ASSERT. For instance, as
first order logic refinement methods are enhanced, ASSERT can be updated accordingly,
enabling it to address a wider range of applications than is currently po
NEITHER’s propositional Horn-clause representation. Or, if refinement algorithm

using entirely different knowledge representations, the approach taken by ASSERT can be applied to those domains as well. However, it is not immediately clear how to map ASSERT to a procedural domain.

Shifting the focus of modeling to a concept-learning emphasis is not Other researchers, most notably Gilmore and Self (Gilmore & Self, 1988), h the potential of using machine learning for tutoring conceptual knowledge. recent trade journal survey of applications for computer-based training ind commercial products are currently available for constructing computer-based cations (IDS, 1990) in which concept learning is a primary task. Concept le fairly well explored pedagogy (Dick & Carey, 1990) and effective techniques to incorrect student classifications are already extant in the literature 1980). Thus a general technique for modeling in concept domains has a wide potential for commercial impact, and can be coupled with instructional tech 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 cont dent modeling stems from a popular belief that detailed student models a difficult to build and yet result in little or no practical value. In tru studies are few in number and have reached disparate conclusions: one shows ing to be ineffective (Sleeman, 1987), another shows that modeling can ind tive effect (Nicolson, 1992). Both of these studies used a bug library app extensive library was built by hand rather than automatically as in ASSERT. Furthermore, both previous studies compared reteaching and modeling using the questions answered incorrectly, whereas in the C++ Tutor the reteaching method was divorced from all s dent input to isolate the effects of the automatic modeling. Thus the C++ Tutor cannot be seen as a direct comparison to previous experiments. In can and should, how evidence that effective student models can be constructed automatically w tively impact student performance.

Which leads to the third important issue; namely, the kind of modeling do in which it is used. The empirical results here show simply that automatic a significant impact on student performance. This says nothing, however, ab models one ought to build nor about the best way to use them. For example, equally significant results could be achieved by using a far simpler modeli that far better results could arise from ASSERT models if the student were allowed more con trol over the number or type of counter examples presented as feedback. I significant features of ASSERT are that it is a general-purpose method, that it works ically, and that it can enhance student performance.

7 Conclusions

In conclusion, ASSERT is a general-purpose method for constructing student model which operate in concept learning domains. It is able to construct student and automatically, catching both expected and novel student behavior. It is system which can construct bug libraries automatically using the interactio dents, without requiring input from the author, and integrate the results i efforts. Finally, the empirical evidence presented supports the two princi research: (1) that automatically constructed refinement-based models can be cantly increase student performance and (2) that a bug library can also be matically using multiple student models as input.

Acknowledgments

This research was supported by the NASA Graduate Student Researchers Prog
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-poir
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. *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 Refinement*. Ph.D. thesis, Austin, TX: University of Texas.

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

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

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

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

Burton, R. R. (1982). Diagnosing bugs in a simple procedural skill. In Sle 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 parad environments. *Computer Science and Education. ACM SIGCSE Bulletin*, 8(1):236-246.

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

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

Costa, E., Duchenoy, S., and Kodratoff, Y. (1988). A resolution-based metho ing 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 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 in 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. *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 777-782. Detroit, MI.

Hoppe, H. U. (1994). Deductive error diagnosis and inductive error generali gent tutoring systems. *Journal of AI and Education* (in press).

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

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

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 co In Frederiksen, N., Glaser, R., Lesgold, A. and Shafto, M., editors, *Diagnostic Monitoring of Skill and Knowledge Acquisition*, chapter 10, pages 217-250. Hillsdale, N Lawrence Erlbaum Associates.

Lianging, H. and Taotao, H. (1991). A uniform student model for intellige tems: Declarative and procedural aspects. *Educational Technology*, 31(11):44-48.

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

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

Newell, A. and Simon, H. A. (1972). *Human problem solving*. Chapter 6. Englewood Cliffs, NJ: Prentice-Hall.

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

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

Ourston, D. and Mooney, R. (1990). Changing the rules: A comprehensive app ory refinement. In *Proceedings of the Eighth National Conference on Artificial gence*, pages 815-820. Detroit, MI.

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

Quilici, A. (1989). Detecting and responding to plan-oriented misconceptio 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 studen intelligent tutor for LISP programming. In *Proceedings of the Ninth International Joi conference on Artificial Intelligence*, pages 8-14. Los Angeles, CA.

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

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

Self, J. A. (1990). Bypassing the intractable problem of student modeling. Gauthier, G., editors, *Intelligent tutoring systems: at the crossroads of artific gence and education*, chapter 5. Ablex Publishing Corp.

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

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 student's problem solv *Artificial Intelligence*, 16:171-187.

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

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

Soloway, E. and Johnson, W. (1984). Remembrance of blunders past: a retros development of PROUST. *Proceedings of the Sixth Annual Conference of the Co tive Science Society*, page 57. Boulder, CO.

Tennyson, R. (1971). Instructional variables which predict specific learner tion and errors. Paper presented at the annual Meeting of the American Association. Washington, D.C.

Tennyson, R. D. and Park, O. (1980). The teaching of concepts: A review o design research literature. *Review of Educational Research*, 50(1):55-70.

Towell, G. and Shavlik, J. (1991). Refining symbolic knowledge using neural *Proceedings of the International Workshop on Multistrategy Learning*, pages 257-272. Harper’s Ferry, W.Va.

Tukey, J. W. (1953). The Problem of Multiple Comparisons. Mimeograph, Princ Spence, J., Cotton, J., Underwood, B. and Duncan, C., editors, *Elementary Statistics* fourth edition, pg 215, note 4. Prentice-Hall.

Wenger, E. (1987). *Artificial Intelligence and Tutoring Systems*. Los Altos, CA: Morgan Kaufmann.

Winston, P. H. and Horn, B. K. P. (1989). *Lisp*. Reading, MA: Addison-Wesley.

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