Towell, G. G., Shavlik, J. W., and Noordewier, M. O. (1990). Refinement of approximate domain theories by knowledge-based artificial neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 861-866. Boston, MA.

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

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

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

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

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

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

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

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

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

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

Sleeman, D. (1987). Some challenges for intelligent tutoring systems. In *Proceedings of the Tenth International Joint conference on Artificial Intelligence*, pages 166-168. Milan.

Sleeman, D. H. and Smith, M. J. (1981). Modelling student's problem solvin *Artificial Intelligence*, 16:171-187.

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

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

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

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

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

Costa, E., Duchenoy, S. and Kodratoff, Y. (1988). A resolution-based method for discovering student misconceptions. In Self, J., editor, *Artificial Intelligence and Human Learning*. New York, NY: Chapman and Hall.

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

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

Finin, T. (1989). GUMS: A general user modeling system. In Kobsa, A. and 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 inte

systems. In Self, J., editor, *Artificial Intelligence and Human Learning*. New York, NY: Chapman and Hall.

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

Hoppe, H. U. (1994). Deductive error diagnosis and inductive error generaliza

gent tutoring systems. *Journal of Artificial Intelligence in Education*, 5:27-49.

Ikeda, M. and Misoguchi, R. (1994). FITS: A framework for ITS – a computation

tutoring. *Journal of Artificial Intelligence in Education*, 5:319-348.

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

Langley, P., Wogulis, J. and Ohlsson, S. (1990). Rules and principles in cognitiv

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

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

Murray, W. (1991). An endorsement-based approach to student modeling for plan

trolled tutors. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 100-106. Sydney, Australia.

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

Ourston, D. and Mooney, R. (1994). Theory refinement combining analytical and empi

methods. *Artificial Intelligence*, 66:[illegible]

```
operator-A-uses ← (on-operator-A-side right)
operator-A-uses ← (on-operator-A-side left) (not-(operator-A assign))
operator-A-sets ← (operator-A auto-incr)
operator-A-sets ← (on-operator-A-side left) (operator-A modify-assign)
operator-A-sets ← (on-operator-A-side left) (operator-A assign)
operator-B-uses ← (on-operator-B-side right)
operator-B-uses ← (on-operator-B-side left) (not-(operator-B assign))
operator-B-sets ← (operator-B auto-incr)
operator-B-sets ← (on-operator-B-side left) (operator-B modify-assign)
operator-B-sets ← (on-operator-B-side left) (operator-B assign)
```

References

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

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

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

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

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

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

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

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

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

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

which operate in concept learning domains. It is able to construct student m[odels] efficiently and automatically, catching both expected and novel student behavior. It is the first modeling system which can construct bug libraries automatically using the interactions [of stu]dents, without requiring input from the author, and integrate the results into future model[ing] efforts. Finally, the empirical evidence presented supports the two principal cla[ims of this] research: (1) that automatically constructed refinement-based models can be u[sed to signifi]cantly increase student performance and (2) that a bug library can also be c[onstructed auto]matically using multiple student models as input.

Acknowledgments

This research was supported by the NASA Graduate Student Researchers Progra[m, grant] number NGT-50732.

Appendix A. C++ Tutor Domain

Domain Features:

```
pointer:             {constant, non-constant, absent}
integer:             {constant, non-constant}
pointer-init:        {true, false}
integer-init:        {true, false}
pointer-set:         {true, false}
integer-set:         {yes, no, through-pointer}
multiple-operands:   {true, false}
position-A:          {normal, left-lazy, right-lazy}
operator-A-lazy:     {AND, OR}
lazy-A-left-value:   {non-zero, zero}
on-operator-A-side:  {left, right}
on-operator-B-side:  {left, right}
operator-A:          {assign, modify-assign, mathematical, logical, comparison, aut
operator-B:          {assign, modify-assign, mathematical, logical, comparison, aut
```

Correct Domain Theory

```
compile-error      ‹  constant-not-init
compile-error      ‹  constant-assigned
constant-not-init  ‹  (pointer constant) (pointer-init false)
constant-not-init  ‹  (integer constant) (integer-init false)
constant-assigned  ‹  (integer constant) integer-init (integer-set yes)
constant-assigned  ‹  (integer constant) integer-init (integer-set through-pointer)
constant-assigned  ‹  (pointer constant) pointer-init pointer-set
ambiguous          ‹  multiple-operands operands-linked
operands-linked    ‹  operand-A-uses operator-B-sets
operands-linked    ‹  operand-A-sets operator-B-uses
operand-A-uses     ‹  operand-A-evaluated operator-A-uses
operand-A-sets     ‹  operand-A-evaluated operator-A-sets
operand-A-evaluated ‹ (position-A normal)
operand-A-evaluated ‹ (position-A left-lazy)
operand-A-evaluated ‹ (position-A right-lazy) lazy-A-full-eval
lazy-A-full-eval   ‹  (operator-A-lazy AND) (lazy-A-left-value non-zero)
lazy-A-full-eval   ‹  (operator-A-lazy OR) (lazy-A-left-value zero)
```

designed for use in classification domains. As an example, several previous student modeling efforts have focused on the domain of writing computer programs and Goldstein, 1977; Soloway et al., 1983; Soloway and Johnson, 1984), wh research was tested using a classification task where students were asked to rectness of program segments. This tie to classification domains is largely due to the fact that the most mature theory-refinement algorithms developed thus far are designed f tion and is not a limitation of the general Assert framework. So, for instance, as first order logic refinement methods are enhanced (Richards & Mooney, 1995), Assert can be updated accordingly, enabling it to address a wider range of applications. Howe not immediately clear how easy it would be to map Assert to a procedural domain.

Shifting the focus of modeling to a concept-learning emphasis is not un Other researchers, most notably Gilmore and Self (Gilmore & Self, 1988), hav the potential of using machine learning for tutoring conceptual knowledge. Co also has a fairly well explored pedagogy (Merrill & Tennyson, 1977; Klausmeier, 1990), and effective techniques for responding to incorrect student classifications are already extant in the lit & Park, 1980). Thus a general technique for modeling in concept domains has cability and can be coupled with instructional techniques shown to be benef tation 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 contro dent modeling stems from a popular belief that detailed student models are difficult to build and yet result in little or no practical value. In truth, previous studies are few in number and have reached disparate conclusions: one shows s ing to be ineffective (Sleeman, 1987), another shows that modeling can indeed hav tive effect (Nicolson, 1992). Both of these studies used a bug library approac extensive library was built by hand rather than automatically as in Assert. Furthermore, both previous studies used a different form of reteaching than that used in the C++ Tutor. Thus the C++ Tutor cannot be seen as a direct comparison to previous experiment and should, however, be seen as evidence that effective student models can be constructed automatically which will positively impact student performance.

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

7 Conclusions

In conclusion, Assert is a general-purpose method for constructing student modelin

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

However, note that the total overall size of the 100-12 bug library is the largest of any of the three libraries. In fact, the 100-12 library has the lowest ratio of common in the library. This dilution is a potential problem when the bug library is used modeling efforts. Recall from Section 4.2 that the bug library is treated as a search sp refinements which is traversed in an effort to improve the accuracy of the rule base befo is passed to 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 from more ablation tests aimed at determining if the ratio of common bugs to library size is det of modeling accuracy. For these tests, a new crop of simulated students was crea each ablation test was run with this same set of students, varying the bug. For each student only 10 of the 180 examples were used for training, leaving the rem testing. As the numbers in the table show, the 20-12 library results in almost no improvem for Assert-Full and Assert-BugOnly (which used just the library without Neither) as opposed to Assert-NoBugs (which used no bug library). By contrast, when the “bett libraries are used, the accuracy of Assert-Full and Assert-BugOnly is significantly better.

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

5.2 Summary of Results

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

6 Discussion

Several important issues have been raised by this research that must be emph place the work into a proper context. The first issue concerns the type of k modeled. Unlike the previous modeling efforts, which focus on procedural tasks, Assert is

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

(a)

System	Accuracy using different Starting Bug Library		
	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 li
total number of bugs found, part (b) compares accuracy

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

Table 3 is a comparison of three libraries constructed using this technique, ing numbers of simulated students and varying amounts of example answers per first library, denoted 20-180, was formed from 20 student models built with all example answers per student. The second library, 20-12, used 20 students with only 12 example answers per student chosen randomly for each student from among the ble. The 100-12 library was built from 100 students answering 12 randomly se tions. Conceptually, these three libraries were designed to compare three d conditions: a few students answering lots of questions, a few students answer tions, and a large number of students answering a few questions. This comparison wa at answering the following two-part question: (1) are ASSERT bug libraries only effective when students answer a large number of questions, or (2) can one expect effective bug librar-ies to emerge from a large number of students answering a more reasonable number of q tions. If useful bug libraries cannot be constructed from small student mo utility of ASSERT bug library is limited since one would still be tied to coll amounts of data on some students to construct the library. While more student data will always result in more accurate individual models, it is important to show that a good collec-tive bug library can still be built over time using less accurate models as in

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

so as to be equivalently representative across the correct domain rules. Such quality is important to maintain so that any effects from modeling with the training set are manifested in the test set. Therefore, the 20 examples from the pre-test and grouped into 10 pairs, where each pair consisted of the two examples (one fr and one from the post-test) which covered the same domain rule. Then, traini splits were generated by randomly dividing each pair

The result was 2^{10} possible training-test set splits. For each of the 25 No Fee dents, 25 training-test splits were generated, yielding 625 samples for com ASSERT-Full and ASSERT-NoBugs. Each system was trained with the training set and accur measured on the test set by comparing what the system predicted with what the the No Feedback group actually answered. The results are shown in Table 2. For comparison purposes, we also measured the accuracy of both an inductive learner, using the same training and test set splits, and the correct domain rules. The inductive learner 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 (Quinl the correct theory no learning was performed, i.e., the correct domain rules out modification to predict the students' answers. Statistical significance was measured usi a two-tailed Student t-test for paired difference of means at the 0.05 level of confidence.

These results illustrate that the groups with significantly better models, ASSERT-Full and ASSERT-NoBugs, are precisely the groups which performed best after remediation in Table 1. This is further evidence in support of the fact that more accurate student mo late directly to improved student performance via more directed remediation. It also reinforces the finding of other studies (Ourston & Mooney, 1994) that inductive methods are simply not as effective as theory refinement in terms of accuracy.

5.1.3 Bug Library Utility

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 sign the performance of the student as expected, casting doubts on its utility. Certainly having a bug library did no harm to post-test performance, and perhaps with more data the difference between the two groups would indeed have been significant. Thus it would be us more about the conditions under which a bug library, as constructed automatically by ASSERT, might be expected to impact the modeling process.

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

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.

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 much
depends upon the size of the pre-test data given to ASSERT, nor what the perfor-
mance will be for other forms of modeling or reteaching. What has been illust
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 illust ASSERT
can be used to build a tutorial that significantly impacts student performa
models and bug libraries are automatically constructed using only correct kn
domain. Furthermore, it is another argument in favor of the use of student models since
shows (1) that they can have significant impact over not modeling at all and (
be constructed automatically without resorting to the time-consuming task of
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 correlat
ability of ASSERT to produce an accurate model and an improvement in student perfo
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 pie
are disabled and the resulting systems compared based on the accuracy of th
produce. There are two different configurations in which ASSERT was used for modeling. The
first, which is labeled "ASSERT-Full," uses everything available to construct the model
means referencing a bug library to create a modified theory which is then fed NEITHER for
further refinement. This method should produce the most accurate models. The s
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 in
the library.

In the C++ domain, only the data from the No Feedback group is useful for an
test. This is because no remediation occurred between the pre-test and post-
dents in this group; thus, their 20 questions could be treated as a single
training set and test set examples could be drawn. These training-test split

is meant by “reteaching” is extremely important, as it can have a profound results of the experiment. Furthermore, reteaching methods vary, making it important to clarify the exact approach used. For this experiment, the essential point was to feedback based on modeling made any difference over feedback based on no modeling at al To that end, we chose an automated form of reteaching which used no informati student, not even which answers the student got right or wrong. In such an 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 rul base, and an explanation and example was generated for each rule.[6]

The other two groups received feedback based on the models constructed for 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.[7] For the ASSERT-Full group, bugs from the bug library were give preference to those found by NEITHER, in order of their stereotypicality value. In bo ASSERT-Full and ASSERT-NoBugs groups, if fewer than four bugs were found, the rema of the feedback was selected at random as with the Reteaching group.

Students were assigned to the four groups randomly. Since the ASSERT-Full group required a bug library, the first 45 students to take the tutorial were randomly assi ASSERT-NoBugs, Reteaching and No Feedback groups. The models from these first dents were then used to construct a bug library. The remaining 55 students were randomly assigned to all four groups but at three times the rate to the ASSERT-Full group until the number of students assigned to all groups was even (25 students in each group).

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

The results of the experiment confirmed most of our expectations. As predict age performance decreased as the feedback varied from full ASSERT to no bug library to reteaching to nothing. Moreover, both the ASSERT-Full and ASSERT-NoBugs students improved significantly more than students in the Reteaching group. For ASSERT-NoBugs, the improvement over Reteaching is more that 12 percentage points, and for the ASSERT-Full group, the average improvement is even greater.

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

---

6. Note that we specifically avoided comparing ASSERT reteaching against reteaching by a human tutor. Such comparison represents an entirely different test. Here we were concerned with determining whether ASSERT's automatic reteaching was an effective feature of the algorithm.
7. NEITHER orders its refinements by preferring those which increase accuracy the most with the sm

formance over a control group which received no feedback. Additionally, it was expected
that students who were modeled with the benefit of a bug library would see g
mance increases over students who were modeled without a library. Third, as in previous stu-
dent modeling studies (Sleeman, 1987; Nicolson, 1992) we wanted to test ho
receiving feedback based on student models would compare against students
simple form of reteaching feedback. In this case, the expectation was that re
on modeling would result in greater post-test performance than simple reteach

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

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 bene
fit of a bug library, the third received reteaching and the fourth was a control gr
had no feedback. The expectation was that these four groups would exhibit de
formance on a post-test as the remediation ranged from ASSERT with a full bug library to
reteaching to no feedback.

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

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

Students were randomly assigned to four groups of 25, each of which receive
kind of feedback from the C++ Tutor. One group of 25 received no feedback, acting as
control group. This group was labeled the “No Feedback” group. The other t
were given feedback using explanations and examples as described in Section 3.2.3
ensure that the only difference between feedback groups was the type of feedback rec
each group was given the same amount of feedback, specifically four examples and four
explanations for each student.

One feedback group received a form of automated reteaching. Specifying pre

```
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 18: Example C++ problem

tutoring college-level freshmen taking an introductory C++ course at the University of T
at Austin. In addition to this evidence, experiments are presented from an ar
which student responses were simulated. The advantage of this simulation dom
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 prop
and use of constants. These two concepts plus examples of correct programs
categories into which example programs could be classified. A set of 27 doma
developed to classify problems, using a set of 14 domain features, as being ambiguous, a compile error (for incorrectly declared or used constants) or correct. The latter category was the default category assumed for any example which could not be
ambiguous or a compile error. Figure 18 shows an example question from the C++ Tutor (for the complete listing of the C++ Tutor rule base see Appendix A).

Students who used the tutorial did so on a voluntary basis and received e
their participation. As an added incentive, the material in the tutorial cove
would be present on the course final exam. This established a high level of mo
the students who participated in the test. Due to the large number of students involved, the
tutorial was made available over a period of four days and students were
reserve time slots to use the program. In total, 100 students participated in

Three major questions were the focus of the C++ Tutor test. First, it was important to
establish whether ASSERT could be an effective modeler for students in a realistic s
ting. This was measured by testing the model produced for a student on a se
taken from the student which had not been given to ASSERT. The predictive accuracy of the
model on such novel examples was expected to be higher than simply using the
base with no modifications. Second, even with a perfect model one may not see
in student performance. Though a model may be accurate in predicting *when* a student will
reach a faulty conclusion, it may not be able to predict *how that conclusion* was reached. The
only way to determine the utility of a model is to provide the student with f
that model and measure any change in the student's performance. Our hypothesis was that
remediation generated using models built by ASSERT would result in increased student per-

```
> (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 17: Trace of bug selection from the bug libra

5 Experimental Results

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

In this section, evidence is presented in support of the claim that the Assert design can be used to construct tutorials which significantly impact student performance. The bulk of this evidence comes from a test using 100 students who interacted with a C++ Tutor built with

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

Figure 16: Pseudocode for bug library use.

breaker). When no bugs can be found that increase the accuracy of the rule ba
quits returning the most current version of the rules.

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

Perhaps the most important feature of ASSERT’s bug library algorithm lies in its ability model both common and unique misconceptions. As with other bug-library based methods, the ability to use a cache of expected errors gives the modeler an in domains where a large amount of data would otherwise be required for an accurate nosis. But because the bug library here is used as a precursor to theory refin ASSERT is not restricted to using only those bugs present in the library. Any specific student problem not in the bug library can still be captured by the theory refinement compone rithm. Thus the final theory of Figure 17 has partially accounted for the student’s errors and the rest gets done by NEITHER.

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

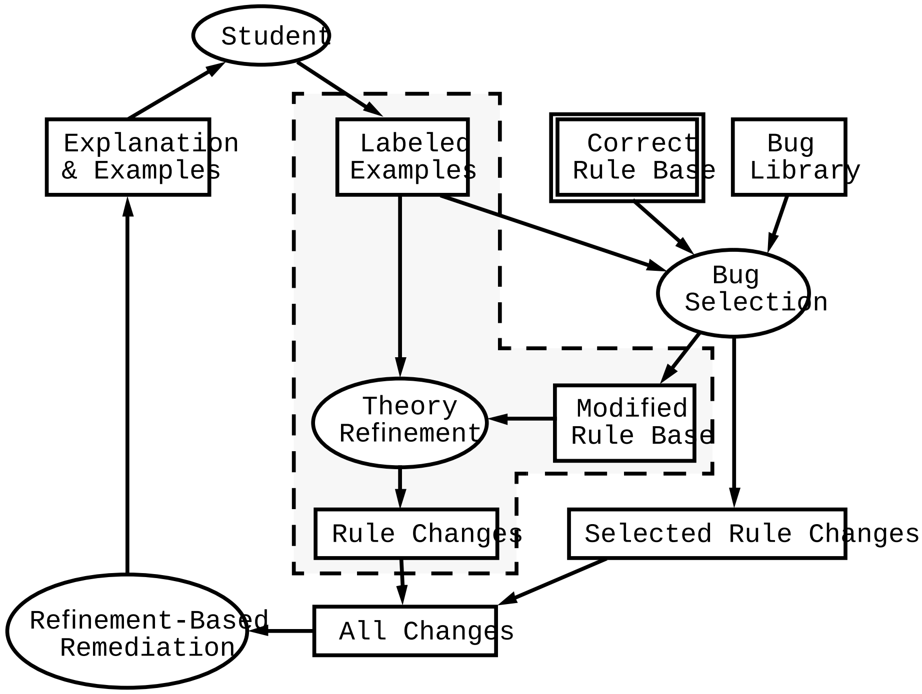

Figure 15: Overview of extended ASSERT algorithm. The area within th
line is the theory refinement component.

the library to be added to the rule base using a hill-climbing search. Bugs upon the predictive accuracy of the rule base are added incrementally. The result is a modi-fied rule base which resembles the student's behavior more closely than the correct rules b which may still be incomplete. The bugs which were selected are returned al modified rule base since they must be included with the final model of the stud modified rule base in constructed, it is passed to theory refinement along wi examples to determine any additional refinements necessary for reproducing the the student. All rule changes, whether selected from the library or constr refinement, are returned as the final student model.

The pseudocode for constructing the modified rule base is shown in Figure 16. Modi-fyRules starts with the correct rule base, and loops as long as a bug can be f increase the accuracy of the rule base on the set of labeled examples. The first to find the accuracy of the rule base when the bug in question is added to the se bugs which result in improved accuracy are saved. Next, the bug which increase most is found, and an inner loop is entered to pare down the list to only those b ment in accuracy is "statistically equivalent"[4] to the best bug, using a paired Student t-te[5] Finally, if there are still multiple bugs left, then the one with the greates value is picked to be added to the current rule base (random selection is u

4. more precisely, whose improvement in accuracy is less, but not statistically significantly less.
5. Since the accuracy values for all the bugs are computed using the same set of labeled exam dent, one can use a paired Student t-test to estimate if the difference in accuracy between any two bugs is sta-tistically significant (using the standard 0.05 level of confidence to indicate significance).

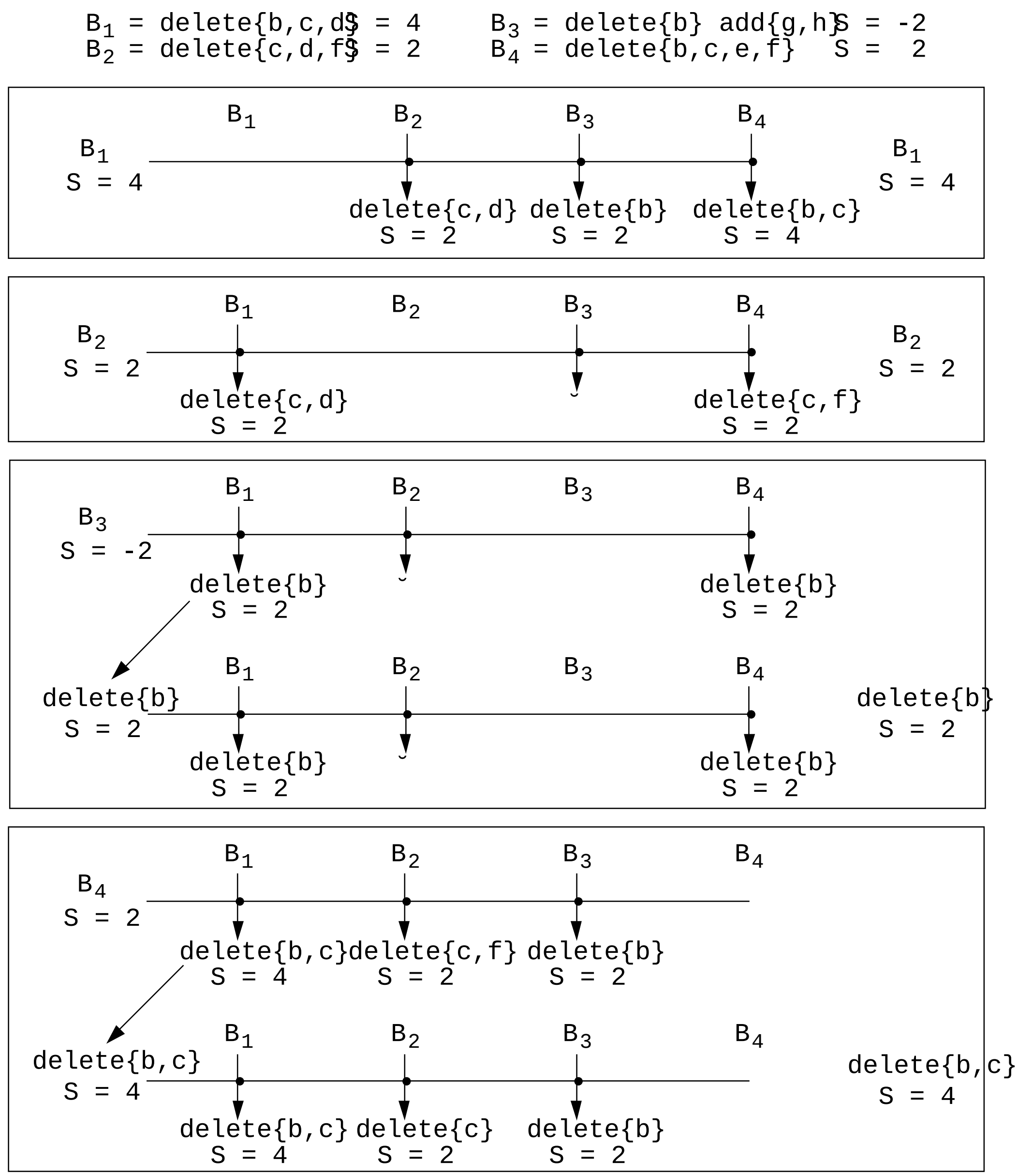

Figure 14: Bug library construction example. “S” stands fo

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

Figure 15 shows a schematic for how this is accomplished. The bug library, the correct rule base, and the student's labeled examples are input to a process which selects bug

```
function BuildBugLibrary (M:list of student models): bug
library;
begin
  R = ˘ ;
  for m ˛ M do add refinements of m to R avoiding duplicates;
  for r ˛ R do begin
    Best = r;
    S = Stereotypicality(Best);
    repeat while S continues to increase begin
      Temp = ˘ ;
      for r ˛ R do add Intersection(Best,r) to Temp;
      G = member of Temp with highest stereotypicality;
      if Stereotypicality(G) > S then begin
        Best = G;
        S = Stereotypicality(G);
      end;
    end;
    add Best to bug library;
  end;
  return bug library sorted by greatest stereotypicality;
```

Figure 13: Pseudocode for bug library construction

Figure 14 shows a complete example for constructing a bug library, using the bugs from Figure 11 plus one extra bug to highlight the hill-climbing nature of the algori four bugs are a series of boxes, each representing one iteration of the out first box is the iteration which computes the bug to be added to the library s B_1 as a seed, the second starts with B_2 as the seed, et cetera. After saving the B_1 stereotypical the inner loop is entered and an LGG is formed between B_1 and the other three bugs. Once the LGG's are computed, the best is found, in this case LGG(B_1,B_4), which has a stereotypicality of 4. This is compared with the current best stereotypicality, and since there is no improvement the inner loop halts and adds B_1 to the bug library. The second box, for bug B_2, also yields no improvement from generalization, resulting in B_2 being added unchanged to the bug library.

The next two boxes representing the iterations of the outer loop for B_3 and B_4 are more interesting. For B_3, the best LGG results from combining B_3 and B_1. The resulting generalization delete{b} whose stereotypicality value of 2 is an improvement of 4 points value for B_3 alone. Consequently, the inner loop repeats. A second round of LGG's produces no further improvement, resulting in the addition of the generalization delete{b} to the bug library. For bug B_4, the process is similar. A first round of LGG's produces an improvement which cannot be surpassed by a second iteration of the inner loop. Note that LGG(B_4,B_1), which was computed and rejected in the top box where B_1 was the seed, turns out to be a useful improvement over B_4 alone. The final bug library consists of the following fo sorted in the following order: B_1, delete{b}, B_2, and delete{c}.

4.2 Using the Bug Library for Modeling

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

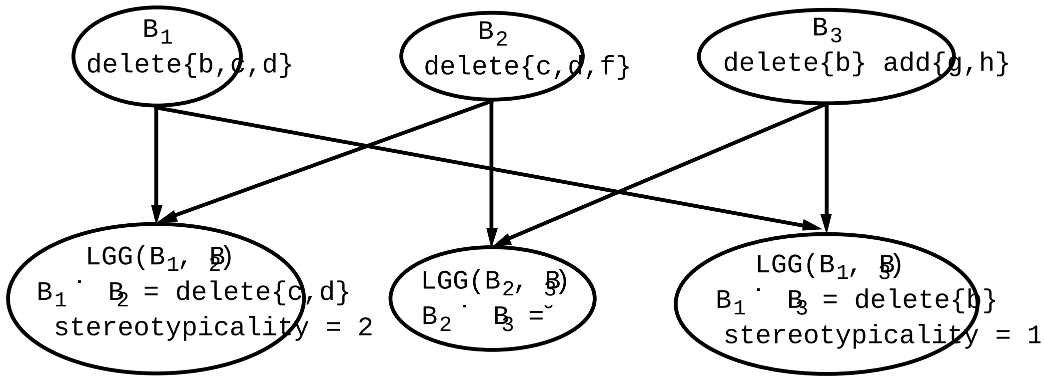

Figure 12: Bug generalization using the LGG operato

bug decreases the distances between the models and the correct rule base. The dis between CR and the student models is shown followed by the distances to the mo each of the three bugs is added to CR. Calculating the distance between two rule se amounts to counting the number of literal changes required to convert one to . Thus, changing CR+B_1 into M_2 requires changing the rule $a \leftarrow e \wedge f$ into the rule $a \leftarrow b \wedge e$ which is done by deleting f and adding b. The bottom of the figure has the stereotypicality valu each of the bugs.[3]

Figure 12 illustrates how ASSERT forms generalizations among the bugs from Figure 11. Since any refinement to a propositional theory can be expressed as a logic c compute generalizations using the least general generalization (LGG) operator (Plotkin, 1970). For propositional logic literals, the LGG of two refinements is simply tion. The LGG will often form a generalization that has better stereotypicality t from which it was taken. For instance, LGG(B_1,B_2) beats the value for B_2 which is 1. Likewise, LGG(B_1,B_3) is better than B_3 alone. This will be the result whenever the LGG operation c more of what is common among the models, and avoids more of what is uncommon. note that the LGG is not beneficial in all cases; the same two LGG's mentioned above are both worse than the B_1 refinement alone, even though B_1 is an input. Since the result of forming the LGG of two refinements is also a refinement, the process can be continued, forming LGG's from LGG's which can also result in better or worse refinements.

Figure 13 shows the pseudocode for constructing the bug library. The fundamental idea is to perform a hill-climbing search using successive LGG operations. Starting with ea a seed, multiple calls are made to the LGG operator to combine the seed with al ments from the models. As long as this continues to result in a better general passes are made over all the refinements. The process halts when no generalizati which will improve upon the seed, which must eventually happen since continued between refinements will eventually produce no change or the null set. The best found starting with each refinement as the initial seed is kept and inserted into . Any duplications in the library are eliminated and the results sorted by stereotypic

3. Note that a bug may have a negative stereotypicality. Unless a bug is present in more than half of the student models, it will have a negative stereotypicality since the between the bug and the majority of the models. What is important is the relative difference between stereotypicality values

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

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

Distance from Correct Rules to Models:

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.

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

Figures 11 through 14 illustrate how a bug library is constructed for a contr generated for illustration purposes. Figure 11 shows how stereotypicality is computed (for details on how stereotypicality can be computed in linear time see Baffes, 1994). Three mod-els are shown at the top of the diagram, each of which changes only one rule rule base. All the models alter the same rule, but in different ways. So, for example, the first model changes the rule a ← b c d e f by deleting the set of antecedents {b,c,d}. The refine-ment for model M_1, labeled B_1, is thus delete{b,c,d}. Below the models are the calculations for determining the stereotypicality of each of the three bugs by computing

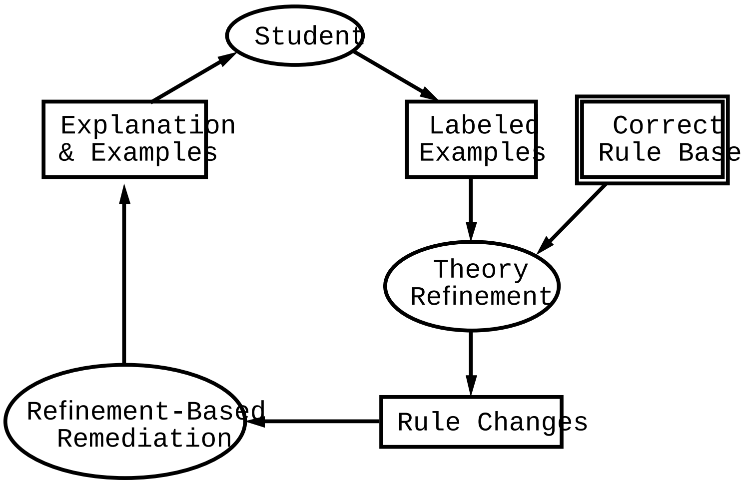

Figure 10: Basic design of the ASSERT algorithm.

conclusion.

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

4 Extending ASSERT's Modeler

The previous three sections have described the ASSERT algorithm, showing how the flow of information between student and system can be implemented as a refinements that highlight the differences between how the system and the student evalua the same set of problems. As such this explains how ASSERT constructs student models by tracing the student's behavior against a known standard rule base. Nothing, though, 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

ASSERT uses the rule changes resulting from theory refinement for each student for constructing its bug library. This gives ASSERT two advantages. First, the rule changes are closely related to the type of input generated by the author of the tutor rule base must be supplied as input, expressing the bug library in terms of rule base is an effective way to communicate buggy information back to the author. Second, the rule-change format is precisely what NEITHER uses to simulate the behavior of the stu dent. A bug library built of rule changes is thus already in a form which car directly into the modeling process.

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

```
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 student.

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

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

Figure 8: System response diagram.

How the correct rule base is constructed is crucial since it becomes the l
which ASSERT interprets the student's actions. If the correct rules are expressed at to
or too low a level of detail, the ability of the system to form accurate mod
ished. Of course, this type of knowledge representation problem exists for a
tems. However, ASSERT gains an advantage by purposely isolating the correct d
knowledge as a separate component: the author can easily change the focus of
altering the correct rule base. Moreover, if students possessing different levels of under-
standing will use the tutor, multiple rule bases can be written to give the system mo
bility.

3.2.3 Refinement-Based Remediation

The last component of ASSERT, the system response, is outlined in Figure 8. Using the
refinements produced by NEITHER, ASSERT generates explanations and examples to reinforc
the correct form of the rule or rules modified. The underlying approach, called refinement-
based remediation, is based on fundamental units of explanation called
Rather than implementing any particular pedagogy, ASSERT supplies the most elementary
information required: an explanation with one or more examples. For each refinement
detected by NEITHER, ASSERT provides two functions: the ability to explain a correc
the rule which was changed, and the ability to generate an example which uses
designer of a tutoring system using ASSERT has the option to generate multiple explanatio
or examples, to determine the circumstances when such feedback is given, a
whether the system or the student controls which explanations and examples a
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 ar
in detail in (Baffes, 1994), but the underlying idea is straightforward. Explanatio
describing how the correct form of the rule (not the revised version) fits in
correct rule base. Each rule has an associated piece of stored text, describ
rule base. A full explanation is generated by chaining together the stored
lying on the proof path for the correct label of the student's example, i.e., the
label which is produced by the correct rule base for the given feature vector

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

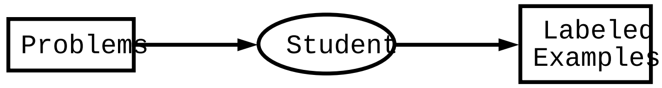

Figure 6: Student behavior diagram.

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

3.2.1 The Student as a Classifier

Figure 6 depicts how ASSERT views student behavior. It is assumed that all actions taken b a student can be broken down into a set of classification decisions. That is, given a set of inputs, called problems, the student will produce a set of labeled examples which classify each of the problems into one category. Each problem consists of one or more feature vectors describing some aspect of the problem. The task of the student is to proc each feature vector, selected from among some predetermined set of legal labels giv student. The resulting set of labeled examples associates each feature vecto selected by the student.

In its simplest form, a problem consists of a single feature vector present in a multiple-choice format, where the answers available to the student are t a list of possible categories. Thus, for example, the classic diagnosis prob a patient's symptoms (the feature vector) and ask the student to select a diagno of diseases (the label). This allows ASSERT to be used in concept learning domains, which are common applications for automated training systems. It also means that s will translate directly into a form usable by theory refinement, which require ples as one of its inputs.

3.2.2 Modeling by Theory Refinement

Once collected, the labeled examples generated by the student are passed t refinement component of ASSERT, depicted in Figure 7. As discussed previously, theory refinement will take an incoming knowledge base, plus an incoming set of exa refine the knowledge base until it is consistent with the examples. The NEITHER theory-refinement system is used to add or remove rules or parts of rules until the duces the same answers as the student, i.e., will classify each feature vect category label as the student. The resulting refined rule base is thus able to dent's behavior.

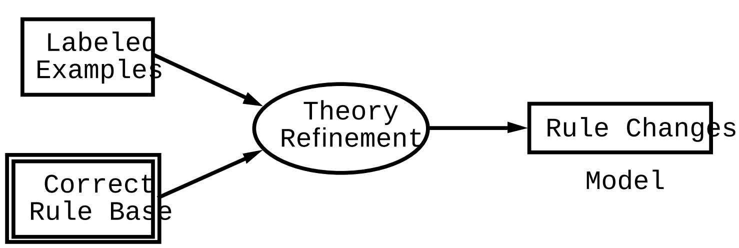

Figure 7: The student simulation model.

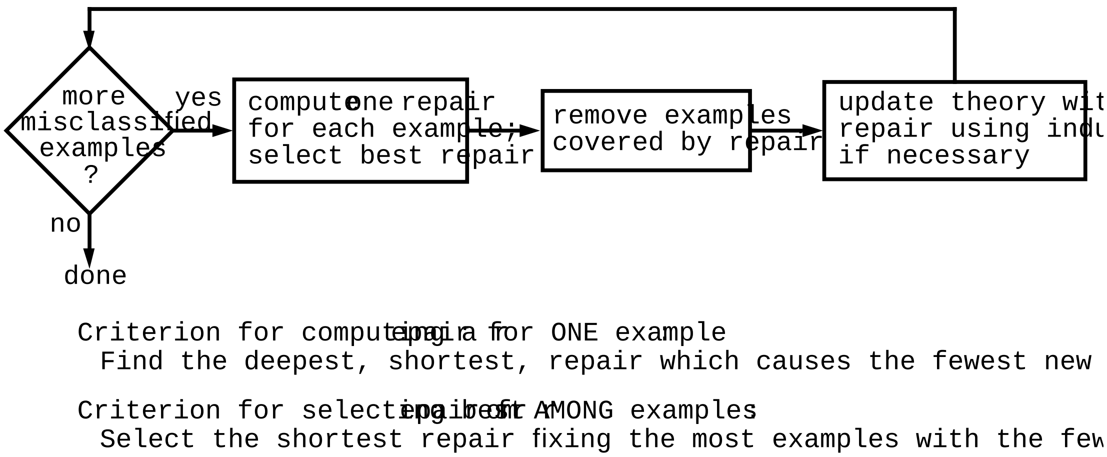

Figure 4: NEITHER main loop.

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

3.2 Overview of ASSERT

Having reviewed the basics of theory refinement, we can now turn to the detai ASSERT. ASSERT views tutoring as a process of communicating knowledge to a student, contribution of the modeling subsystem is to pinpoint elements of the inter base to be communicated. At its most abstract level, such a tutorial can be between the student and the system as shown in Figure 5. ASSERT focuses on the question of how to construct a useful interpretation of the student's actions. This focus is depicted as a new component inserted into the diagram as shown in the right half of Figure 5. The name of the new component, a student simulation model, implies that the system contains a knowl- edge base that can be used to solve problems in the same context as the stu

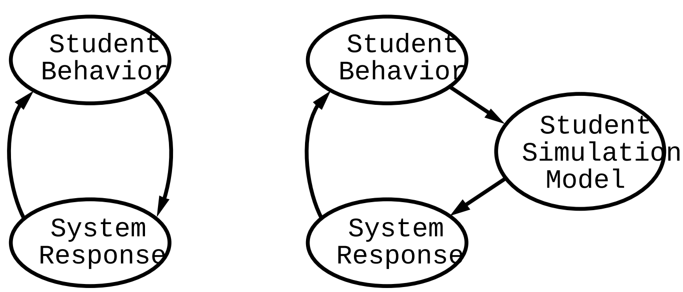

Figure 5: Abstract view of student-tutor interacti

```
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 NEITHER refinement. Above the dashed line i
base before refinement; below are the rules after refineme
shown in boldface.

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

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

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

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

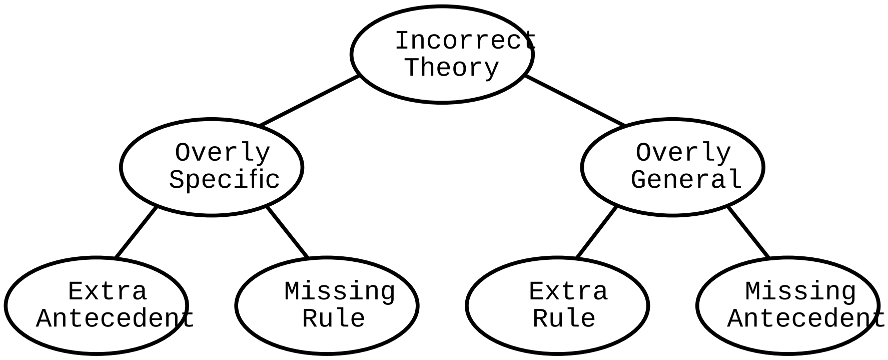

Figure 2: Theory error taxonomy for propositional Horn-cl

rule base is “incorrect” since it does not produce the desired classification examples. Propositional Horn-clause theories can have four types of semant depicted in Figure 2. An overly-general theory is one that causes an example to be in categories other than its own, i.e., a false positive. NEITHER adds new antecedents and deletes rules to fix such problems. An overly-specific theory causes an examp classified in its own category, i.e., a false negative. NEITHER retracts existing antecedents and learns new rules to fix these problems. By making these four kinds of s changes, NEITHER can correct the semantics of the theory by altering the condit which rules are satisfied.

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

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

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

```
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-constant
pointer-init	true	false	true	false
pointer-set	true	true	true	true
integer	constant	non-constant	non-constant	non-constant
integer-init	true	true	true	true
integer-set	through-pointer	yes	no	no

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

is repaired using a set of input examples. The examples are assumed to be lists of featur
value pairs chosen from a set of observable domain features. Each example has an associated
label or category which should be provable using the theory with the feature valu
example. NEITHER can generalize or specialize a theory without user intervention, and is
guaranteed to produce a set of refinements which are consistent with the input

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

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

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

behavior by incorporating knowledge from a library of expected misconceptions be truly adaptive and to avoid the costs of bug library construction, one m sort of dynamic modeling or learning algorithm. And third, tracing student a parison to expected correct behavior can be an effective tool for detecting faulty behavio without requiring a great deal of search. ASSERT combines these ideas by using a machine-learning technique called theory refinement (Ginsberg, 1990; Ourston and Mooney, 1994; Craw and Sleeman, 1991; Towell et al., 1990). Theory refinement is a method for automatically revising a knowledge base to be consistent with a set of examples. Typically, the knowledge base is considered incorrect or incomplete, and the examples repr behavior which the knowledge base should be able to emulate. However, the refinement procedure itself is blind to whether or not the input knowledge base is “correct sense; the theory-refinement process merely modifies the knowledge base until tent with the examples. Thus, one can also use theory refinement by inputting a correct knowledge base and examples of erroneous behavior, and theory refinement will introduce whatever modifications are necessary to cause the knowledge base to simulate 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 domai examples of erroneous student behavior, theory refinement will revise the knowledge base make it consistent with the student, i.e., introduce “faulty” knowledge to ac students’ mistakes.[1] The refinements made to the knowledge base then represent a model student, and can be used directly to guide tutorial feedback by comparing t with whatever elements of the correct knowledge base they replaced.

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

3.1 Outline of Theory Refinement

Having outlined the philosophy behind ASSERT, we can now turn our attention to the theor 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 algor Other theory refinement systems which differ from the one presented here could be used t provide ASSERT with different or enhanced capabilities.

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

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

with a number of proposed mal-rules and must decide which ones are the “keepe

2.3.2 Modeling by Induction

In an effort to avoid the cost associated with hand-constructed bug libraries,
turned to machine learning. Their ACM system (Langley & Ohlsson, 1984; Lang
1990) was the first to harness machine learning techniques for diagnosis of mi
tions through the use of induction. ACM uses a domain-independent induction algorithm t
induce control knowledge representing how students apply operators in a given
output of induction is a set of conditions which predict whether an operator
which lies on the solution path connecting the input for a given problem to the stud
solution. The conditions found by induction are then used to specialize the o
result is a procedure that models the student's unique problem solving behavior.

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

2.4 Tracing Techniques

One final style of student modeler bears mentioning because it represents a bl
ious techniques described to this point. In what might be termed tracing techniques,
the underlying philosophy is to follow along with the student during his or h
ing, stopping whenever the student deviates from the correct procedure. As
techniques must have knowledge of both correct and incorrect actions like bu
and must also have a mechanism for reproducing the steps followed by the
ACM’s solution paths. Using the correct path as a bias, tracing systems can o
ciently.

The pioneering efforts in this area are Anderson's model-tracing tutors (Anderson, 1983;
Anderson et al., 1985; Reiser et al., 1985), which follow student behavior
interaction with the tutor to occur through menu selection. Other tracing s
logic-based representation (Costa et al., 1988; Ikeda & Misoguchi, 1993;
where the idea is to use an analytical approach, such as deduction or resol
through a rule-base to determine where a misconception lies. Whenever the rul
produce a “proof” which mimics the student's actions, the points where the proof fail
become candidates for querying the user about his or her beliefs.

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

3 Refinement–Based Modeling

This previous work on student modeling has resulted in three important ideas
research presented here. First, modeling systems can increase their covera

directly on to the knowledge used to engineer the system. The disadvantage, i tational restriction placed on the model—only missing elements of the corre can be modeled. Alternative notions which a student might have cannot be capt ically, this means that misconceptions cannot be modeled. Thus, overlay models can captur the notion of a student's lack of knowledge, but they cannot be used to model the s who knows of a topic but misunderstands it.

2.2 Bug Libraries

To address the limitation of overlay models, other researchers focused on co bases of student misconceptions typically called bug libraries. The classic bug-library work was done by Brown, Burton and VanLehn (Brown & Burton, 1978; Burton, 1982; Brown & VanLehn, 1980), Sleeman and Smith (Sleeman & Smith, 1981), and Young and O'Shea (Young & O'Shea, 1981), but a number of other systems can be said to incorp form of stored misconceptions (Rich, 1989; Miller & Goldstein, 1977; Quilici way & Johnson, 1984). With a bug library, models are built by matching student behavio against a catalog of expected bugs which are precompiled by hand through an analysis of student errors.

The idea is a very powerful one, especially if specific responses can be ti buggy structures are encoded. However, two important problems remain with this simple bug-library approach. First, the construction of such catalogs by hand is a di 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 was not an by the author of the library. As with overlay models, the static nature of bug librarie them incapable of modeling unanticipated student behaviors.

2.3 Dynamic Modeling

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

2.3.1 Extending a bug library

Sleeman et al. (Sleeman et al., 1990) describe two extensions to their PIXIE INFER* and MALGEN, both of which can be used to extend a bug library. In PIXIE's bug libraries, misconceptions are encoded as mal-rules. Both INFER* and MALGEN attempt to generate new mal-rules when the student exhibits a problem not be modeled using the mal-rules already in the bug library. The difference between the two extensions is that INFER* builds new mal-rules by filling gaps in partial representations of a student's solution, whereas MALGEN uses a generate-and-test method to crea candidate mal-rules.

The disadvantage of both systems is their reliance upon a user to decide w rules are appropriate extensions for the bug library. To their credit, the authors do raise thi issue and discuss potential filters that might be used to cut down on the numb rules presented to the user. Unfortunately, to this point no general-purpose filtering mech nism which might be usable across domains has been found. In the end, the use

Unfortunately, the difficulty of constructing and testing student models has disco many researchers from pursuing further investigations into the field. Despit more than two decades of research has resulted in a wide variety of student niques, the practical task of incorporating these techniques into a functio tem has proved to be a major roadblock. Furthermore, neither the utility necessity of student modeling as a component of an ITS is a universally accep to the contrary, an interview of ten well-known ITS researchers which appears in 1993 issue of *AI Communications* came to the conclusion that “most of the researcher longer believe in on-line student modeling” (Sandberg & Barnard, 1993). The article went on to conclude that “instead of becoming more integrated, the field has b diverged in the last few years. It appears that scientists in the field of educ 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 wor embodied in the ASSERT algorithm (Acquiring Stereotypical Student Errors by Refining Theories). ASSERT was designed to show that student modeling is a viable tool fo an effective tutoring system. By taking advantage of some of the latest te machine learning, ASSERT is able to construct student models efficiently and automatically, catching both expected and novel student misconceptions. Also, it is the firs tem which can construct bug libraries automatically using the interactions dents, without requiring input from the author, and integrate the results so as to impro future modeling efforts. In this sense, ASSERT is a self-improving student modeler. Finally, ASSERT can be used to significantly improve student performance, as will be s sections which follow.

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

2 Previous Work

2.1 Overlay Modeling

The earliest AI-based student models, embodied in systems such as SCHOLAR (C 1970), WEST (Burton & Brown, 1976) and WUSOR (Carr & Goldstein, 1977), used of modeling which is now generally referred to as *overlay modeling*. An overlay model relies on the assumption that a student's knowledge is always a subset of the correc edge. As the student performs actions which illustrate that he or she unders elements of the domain knowledge, these are marked in the overlay model. Mo cated overlay models can express a range of values indicating the extent to w believes a student understands a given topic using some form of truth-mainte (Finnin, 1989; Murray, 1991). However the marking is achieved, typically the unmark ments of the model are used to focus tutoring on new problem areas for the ensure full coverage of the domain.

The advantage of the overlay is its simplicity; the elements of the model

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

Refinement-Based Student Modeling and Automated Bug Library Construction

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

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

Abstract

A critical component of model-based intelligent tutoring systems is a mechan capturing the conceptual state of the student, which enables the system to tailor to suit individual strengths and weaknesses. To be useful such a modeling techn 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 ne modeling technique which can automatically capture novel student errors using onl domain knowledge, and can automatically compile trends across multiple student This approach has been implemented as a computer program, ASSERT, using a machine learning technique called theory refinement, which is a method for automatically revising a knowledge base to be consistent with a set of examples. Using a knowledge base 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 ba are necessary to account for the student's behavior. The efficacy of the approac demonstrated by evaluating ASSERT using 100 students tested on a classification task cov 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 reteaching

1 Introduction

Student modeling has a long and interesting history, dating back well into the earliest ef to produce intelligent tutoring systems (ITSs). The best method for constructing and usin student model is still the subject of much debate. Most student modeling te 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 that domain,
Find: A representation accounting for any discrepancies between the expectations and observations that can be used as a basis for tutoring the student.

Ideally, a unique model is built for every student who interacts with the sys capturing misconceptions specific to each student which are not preprogrammed into the tutor. Using the student model, an ITS would then modify its feedback to s strengths and weaknesses, enabling it to be truly adaptive to the individual.