Conclusions

ASSERT demonstrates how theory refinement techniques developed in machine learning can be used to effectively build student models for intelligent tutoring systems. This application is unique since it inverts the normal goal of theory refinement from *correcting* errors in a knowledge base to *introducing* them. A comprehensive experiment involving a large number of students interacting with an automated tutor for teaching concepts in C^{++} programming was used to evaluate the approach. This experiment demonstrated the ability of theory refinement to generate more accurate student models than raw induction, as well as the ability of the resulting models to support individualized feedback that actually improves students' subsequent performance.

Acknowledgments

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

References

Baffes, P. (1994). Automatic student modeling and bug library construction using theory refinement. Ph.D. diss., Department of Computer Sciences, The University of Texas at Austin. URL: http://net.cs.utexas.edu/users/ml/

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 1: C^{++} Tutor remediation test. Scores are percentage of post-test problems answered correctly. Increase is significant between ASSERT and the other two groups

group.

Since the four groups of students each had a different average accuracy on the pre-test and post-test, they were compared using the average *improvement* in accuracy between pre-test and post-test. Also because each group consisted of different students with no pairing between groups, significance was measured using an ANOVA test. As the only variable between groups was the feedback received, the significance test used was a 1-way unpaired ANOVA test at the 0.05 level of confidence. The average improvement in performance for the four groups is shown in Table 1. As predicted, the average performance decreased as the feedback varied from ASSERT to reteaching to nothing and the ASSERT group performed significantly better than the other two groups.

Accuracy of Student Models

To test the accuracy of the learned models at predicting subsequent student behavior, the data from the No Feedback group was used. This is because no remediation occurred between the pre-test and post-test for the students in this group; thus, their 20 questions could be treated as a single unit from which training set and test set examples could be drawn. Training-test splits were generated so as to be equivalently representative across both data sets. The 20 examples from the pre-test and post-test were grouped into 10 pairs, where each pair consisted of the two examples (one from the pre-test and one from the post-test) which covered the same domain rule. Then, training and test set splits were generated by randomly dividing each pair.

The result was 2^{10} possible training-test set splits. For each of the 25 No Feedback students, 25 training-test splits were generated, yielding 625 samples. For comparison purposes, we also measured the accuracy of both an inductive learner, using the same training and test set splits, and the correct domain rules. The inductive learner was run by starting NEITHER with no initial theory, in which case NEITHER builds rules by induction over the training examples using a propositional version of the FOIL algorithm (Quinlan, 1990). Each system was trained with the training set and accuracy was measured on the test set by comparing what the system predicted with what the student from the No Feedback group actually answered. For the correct theory, no learning was performed, i.e. the correct domain rules were used without modification to predict the students' answers. The results are shown in Table 2. Statistical significance was measured using a two-tailed Student t-test for paired difference of means at the 0.05 level of confidence. As predicted, ASSERT produced more accurate models. Note that induction was even less accurate than simply assuming the student possessed totally correct knowledge, clearly indicating the problems with this approach in the typical, limited-data situation.

System	*Average Accuracy*
ASSERT	62.4
Correct Theory	55.8
Induction	49.4

Table 2: Results of accuracy test. All differences significant

Future Work

The form of reteaching used in the current experiment is very simple and does not employ any knowledge about the individual student or any knowledge of common mistakes or misconceptions. It does not even consider which questions were answered incorrectly. The experiment was designed to test if models constructed by ASSERT were better than no model at all. Experiments comparing ASSERT's approach to alternative model-based methods are needed to evaluate the specific advantages of the refinement-based approach with respect to remediation.

Unlike previous modeling efforts which focus on procedural tasks, ASSERT is designed for classification domains. As an example of this difference, several previous student modeling efforts have focused on the domain of writing computer programs (Soloway et al., 1983), whereas this research was tested using a classification task where students were asked to judge the correctness of program segments. This tie to classification domains is largely due to the fact that the most mature theory-refinement algorithms developed thus far are designed for classification and is not a limitation of the general framework of ASSERT per se. As first-order logic refinement methods are enhanced (Richards and Mooney, 1995), ASSERT can be updated accordingly, enabling it to address a wider range of applications. However, note that it is not immediately clear how easy it would be to map ASSERT to a procedural domain.

C^{++} Tutor Tests

The C^{++} Tutor was developed in conjunction with an introductory C^{++} course at the University of Texas. The tutorial covered two concepts historically difficult for beginning C^{++} students: ambiguity involving statements with lazy operators and the proper declaration and use of constants. These two concepts plus examples of correct programs formed three categories into which example programs could be classified. A set of 27 domain rules was developed to classify problems, using a set of 14 domain features, as being either *ambiguous*, a *compile error* (for incorrectly declared or used constants) or *correct*. The latter category was the default category assumed for any example which could not be proved as ambiguous or a compile error.

Students who used the tutorial did so on a voluntary basis and received extra credit for their participation. As an added incentive, the material in the tutorial reviewed subjects which would be present on the course final exam. This established a high level of motivation among the students who participated in the test. Due to the large number of students involved (75), the tutorial was made available over a period of four days and students were encouraged to reserve time slots to use the program.

Three major questions were the focus of the test. First, it was important to establish whether or not ASSERT could be an effective modeler for students in a realistic setting. This was measured by testing the model produced for a student on a set of examples taken from the student which had not been given to ASSERT. The predictive accuracy of the model on such novel examples was expected to be higher than simply using the correct rule base or one induced from scratch from student behavior. Second, even with a perfect model one may not see any increase in student performance. Our hypothesis was that remediation generated using models built by ASSERT would result in increased student performance over a control group which received no feedback. Third, as in previous student modeling studies, we wanted to test how students receiving feedback based on student models would compare against students receiving a simple form of reteaching feedback. The expectation was that remediation based on modeling would result in greater post-test performance. Testing these hypotheses was accomplished with two experiments: one to measure the effects of remediation and another to measure the accuracy of modeling.

Effect of Remediation on Student Performance

For the remediation test, students who used the C^{++} Tutor were divided into three groups. One group received the benefits of ASSERT, the second received a very simple form of reteaching, and the third was a control group given no feedback.

To test whether ASSERT can impact student performance, one needs to collect information for each student that has certain characteristics. To begin with, data must be collected both before and after any feedback given to the student to detect any change in performance. Thus the C^{++} Tutor was constructed as a series of two tests with a remediation session in between. Secondly, the data from the two tests must be equally representative of the student's capability and must be collected in similar ways.

To that end, a program was written to generate 10 example questions as follows. The questions were divided equally among the categories: three questions were correctly labeled as compilation errors, four were examples of ambiguous programs, and three were questions with no errors. This process was used to generate two sets of 10 questions, both of which covered the same subset of the correct rule base. This ensured that the two sets of questions covered the same concepts at the same level of difficulty, though no two questions were identical. These two sets of questions represented the pre-test and post-test to be given to each student. One set of questions was used as the pre-test for all the students, the other as the post-test, thus the same pre-test and post-test was given to every student. To discourage cheating, the order in which the 10 questions were presented was randomized. Thus every student answered the same questions, and the only difference was the feedback given between the pre-test and post-test.

Students were randomly assigned to three groups of 25, each of which received a different kind of feedback from the C^{++} Tutor. One group of 25 received no feedback, acting as the control group. The other two groups were given feedback using explanations and examples as described previously. To ensure that the only difference between feedback groups was the type of feedback received, both groups were given the same amount of feedback; specifically, four examples and four explanations for each student.

For the "Reteaching" group, ASSERT selected four rules at random from the rule base, and an explanation and example was generated for each rule. The ASSERT group received feedback based on the models constructed for the student from his or her answers to the pre-test questions. Bugs were selected for remediation based on the order they were found by NEITHER.[2] If fewer than four bugs were found, the remainder of the feedback was selected at random as with the Reteaching

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

feature vector presented to the student in a multiple-choice format, where the answers available to the student are taken from among a list of possible categories. This allows ASSERT to be used in concept learning domains, which are common applications for automated training systems. It also means that student actions will translate directly into a form usable by theory refinement. Once collected, the labeled examples generated by the student are passed to the NEITHER theory-refinement system which modifies the rule base until it reproduces the same answers as the student.

Using the refinements produced by NEITHER, ASSERT generates explanations and examples to reinforce the correct form of the rule or rules modified. The underlying approach, called *refinement-based remediation*, is based on fundamental units of explanation called *units of remediation*. Rather than implementing any particular pedagogy, ASSERT supplies the most elementary information required: an *explanation* with one or more *examples*. For each refinement detected by NEITHER, ASSERT provides two functions: the ability to explain a correct use of the rule which was changed, and the ability to generate an example which uses the rule. The designer of a tutoring system using ASSERT has the option to generate multiple explanations or examples, to determine the circumstances when such feedback is given, and to decide whether the system or the student controls which explanations and examples are generated.

Explanations focus on describing how the correct form of the rule (not the revised version) fits into the originally correct rule base. Each rule has an associated piece of stored text, describing its role in the rule base. A full explanation is generated by chaining together the stored text for the rules lying on the proof path for the correct label (not the student's label) for the example, i.e., the label which is produced by the correct rule base for the given feature vector.

Examples are constructed *dynamically* rather than being drawn from storage. Recall that each refinement made by NEITHER results in the addition or deletion of literals from a rule in the theory. Using normal deductive methods, the added and removed literals can be traced down to the feature vector. The result is a set of conditions in the feature vector which the student is ignoring or a set of extra conditions not present in the feature vector which the student thinks are necessary. ASSERT can thus generate an example which is correct in every way *except* for the added or missing conditions in the refinement. The result is then presented as a counter example to the student, and the various added or missing conditions highlighted. Note that this corresponds very closely to tutorial methods outlined for conceptual domains by (Tennyson and Park 1980). An explanation and example pair is shown in Figure 3. This is the explanation generated for the deleted antecedent of the last rule of Figure 1.

EXPLANATION

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

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

Specifically, note the following which contribute to this error:
* There must be a pointer declared to be constant.
* A pointer declared constant must be initialized.
* A pointer declared constant and initialized must be set after its initialization.

Here is an example to illustrate these points:

<u>Example</u>
Here is an example which might appear to be a compile error but is actually CORRECT:

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

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

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

Figure 3: Example remediation given to a student

Experimental Results

The ultimate test of any tutoring system design is whether or not it enhances student performance. This is especially true for student modeling; if the use of a model cannot significantly impact the educational experience, then there is little reason to construct one. Furthermore, this evidence must come from experiments involving large numbers of students in a realistic setting so that the significance of the data can be determined. The importance of student modeling is currently a controversial issue (Sandberg & Barnard, 1993); however there are very few controlled studies, with somewhat contradictory results (Sleeman, 1987; Nicolson, 1992). In this section, we presented evidence supporting the claim that ASSERT can be used to construct tutorials which significantly impact student performance.

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 by-pointer)
R7: constant-assigned ← (pointer constant) ∧ pointer-init ∧ pointer-set

	Example 1	Example 2	Example 3	Example 4
category	*compile-error*	*not compile-error*	*compile-error*	*compile-error*
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	by-pointer	yes	no	no

Figure 1: Sample theory and examples. The original theory, shown in plain text, misclassifies examples 3 and 4. The corrected theory is shown with two deleted antecedents underlined and an added antecedent in boldface.

require user interaction to determine which new bugs to add to an initial hand-constructed library (Sleeman et al., 1990). By contrast, the theory-refinement approach implemented in ASSERT is completely automatic, and by taking advantage of existing correct domain knowledge, it is able to learn more accurate models from limited training data compared to inducing a complete model from scratch.

Background on Theory Refinement

For its theory refinement component, ASSERT uses NEITHER (Baffes & Mooney, 1993) a successor to the EITHER system developed by Ourston & Mooney (1990, 1994). NEITHER employs a propositional Horn-clause knowledge representation. It takes two inputs, a propositional rule base called the *theory*, which is repaired using a set of input *examples*. The examples are lists of *feature-value pairs* chosen from a set of *observable* domain features. Each example has an associated label or *category* which should be provable using the theory given the feature values in the example. NEITHER can generalize or specialize a theory, without user intervention, and is guaranteed to produce a set of refinements that are consistent with the training examples.

Although space limitations prevent us from providing details on theory refinement (see Baffes, 1994), a summary of the technique is as follows. Propositional Horn-clause theories can have four types of errors. An overly-general theory is one that causes an example to be proven in an incorrect category, i.e. a false positive. NEITHER adds new antecedents and deletes rules to fix such problems. An overly-specific theory causes an example not to be proven in its own category, i.e., a false negative. NEITHER deletes existing antecedents and learns new rules to fix these problems. By making these four kinds of syntactic rule changes, NEITHER can correct the semantics of the theory by altering the conditions under which rules are satisfied.[1] A sample theory and examples are shown in Figure 1.

Description of ASSERT

ASSERT views tutoring as a process of communicating knowledge to a student, where the contribution of the modeling subsystem is to pinpoint elements of the internal knowledge base to be communicated. Figure 2 shows a schematic of the ASSERT algorithm. It is assumed that all actions taken by a student can be broken down to a set of *classification decisions*. That is, given a set of inputs, called *problems*, the student will produce a set of *labeled examples* which classify each of the problems into one category. Each problem consists of one or more feature vectors describing some aspect of the problem. The task of the student is to produce a label for each feature vector, selected from among some predetermined set of legal labels given to the student.

In its simplest form, a problem consists of a single

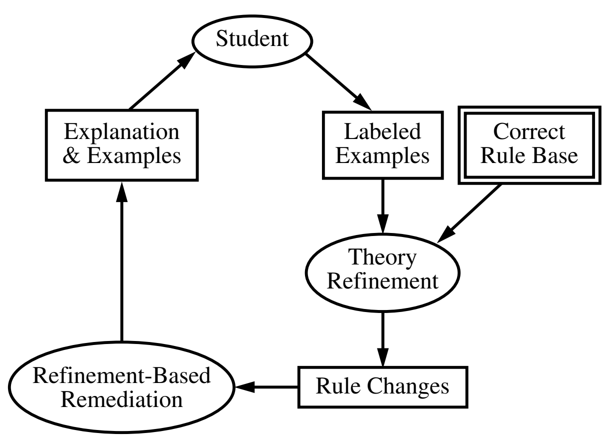

Figure 2: Basic design of the ASSERT algorithm

1. NEITHER's running time is linear in the size of the theory.

Appears in *Proceedings of the Thirteenth National Conference on Artificial Intelligence*
pp. 403-408, Portland, OR, August 1996 (AAAI-96 Best Paper Award)

A Novel Application of Theory Refinement to Student Modeling

Paul T. Baffes
SciComp Inc
5806 Mesa Drive, Suite 250
Austin Texas 78731
baffes@scicomp.com

Raymond J. Mooney
University of Texas at Austin
Department of Computer Sciences, Taylor Hall 2.124
Austin Texas 78712
mooney@cs.utexas.edu

Abstract

Theory refinement systems developed in machine learning automatically modify a knowledge base to render it consistent with a set of classified training examples. We illustrate a novel application of these techniques to the problem of constructing a *student model* for an *intelligent tutoring system* (ITS). Our approach is implemented in an ITS authoring system called ASSERT which uses theory refinement to *introduce* errors into an initially *correct* knowledge base so that it models *incorrect* student behavior. The efficacy of the approach has been demonstrated by evaluating a tutor developed with ASSERT with 75 students tested on a classification task covering concepts from an introductory course on the C^{++} programming language. The system produced reasonably accurate models and students who received feedback based on these models performed significantly better on a post test than students who received simple reteaching.

Introduction

Theory refinement methods developed in machine learning were designed to aid knowledge acquisition by using a database of classified examples to automatically make revisions that improve the accuracy of a knowledge base (Ginsberg, 1990; Ourston & Mooney, 1990 Towell & Shavlik, 1990). These learning techniques have been used to correct errors in an imperfect rule base elicited from an expert and thereby produce a more accurate knowledge base than purely inductive learning methods. In this paper, we present a particularly novel application of theory refinement to a very different problem, that of producing a *student model* for an *intelligent tutoring system (*ITS). By inverting the standard goal of theory refinement, we show how it can be used to produce a model of a student's knowledge that is useful for automated tutoring.

Typically, the knowledge base in theory refinement is considered incorrect or incomplete and the examples represent correct behavior which the knowledge base should be able to emulate. However, the refinement procedure itself is blind to whether or not the initial knowledge base is "correct" in any absolute sense; the theory-refinement process merely modifies the knowledge until it is consistent with the examples. Thus, one can also start with a *correct* knowledge base and examples of *erroneous* conclusions, and use theory refinement to *introduce* errors that cause the knowledge base to model the incorrect conclusions illustrated in the examples. In this way, theory refinement provides a basis for *refinement-based student modeling*. Starting with a representation of the correct knowledge of the domain, and examples of erroneous student behavior, theory refinement can introduce "faulty" knowledge that accounts for the student's mistakes. The resulting changes constitute a model of the student which can be used directly to guide tutorial feedback by comparing the refinements with the elements of correct knowledge they replaced.

We have implemented this approach in an ITS authoring system called ASSERT, which was then used to develop a tutor for teaching concepts in C^{++} programming. A controlled experiment with 75 students was conducted to evaluate the resulting tutor. The system produced reasonably accurate models, and students who received directed feedback based on these models performed significantly better on a post test than students who received simple reteaching.

Background on Student modeling

In order to tailor instruction to individual students, one of the primary tasks of most intelligent tutoring systems is to construct a model of the student's knowledge which is then used to guide the feedback and information presented. The simplest type of model is an *overlay model* (Carbonell, 1970; Carr & Goldstein, 1977) which assumes that a student's knowledge is a subset of the correct domain knowledge. Unfortunately, this approach is unable to model incorrect student knowledge. Other researchers have focused on constructing databases of student misconceptions typically termed *bug libraries* (Brown & Burton, 1978; Sleeman & Smith, 1981). However, hand-constructing such libraries by analyzing student protocols is a difficult, time-consuming task and the result is incapable of modeling unanticipated student behavior. More recent work has focussed on using machine learning techniques to automate the construction of student models. However, existing methods require inducing a complete model of student knowledge (both correct and incorrect) from limited training data (Langley & Ohlsson, 1984; Langley et al., 1990) or