ISSN 0918-2802
Technical Report

The Application of Machine Learning to Student Modeling: Survey and Analysis

Raymund Sison and Masamichi Shimura

TR96-0010 June

Department of Computer Science
Tokyo Institute of Technology
Ôokayama 2-12-1 Meguro Tokyo 152, Japan
http://www.cs.titech.ac.jp/

©The author(s) of this report reserves all the rights.

An abridged version of this report is to appear in the Proceedings of the European Conference on Artificial Intelligence in Education.

1 Introduction

Student modeling (SM) is the analysis of student behavior and the induction of a model that explains this behavior. The output of a *student modeling system* (SMS) is usually used by an *intelligent tutoring system* (ITS) or an *intelligent assistant* to determine what task to give its user next. Figure 1 illustrates the interaction between the student modeling and tutoring subsystems of an ITS.

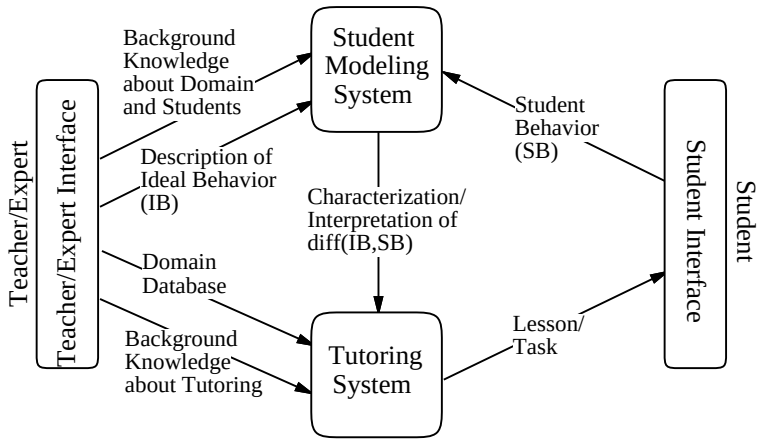


Figure 1: Architecture of an Intelligent Tutoring System

As the figure indicates, a student modeling system must know the ideal behavior in order for it to be able to recognize and interpret the behavior of a student. The ideal behavior, together with the knowledge about interpreting differences between the ideal and the actual student behaviors, form part of the background knowledge of an SMS. The background knowledge of the tutoring subsystem, on the other hand, consists of rules about teaching the subject domain. These rules are used in conjuction with a database of lessons and problems.

While we should be thankful that student modeling systems such as those of Anderson's ACT*-based tutors (Anderson *et al.*, 1990) have attained significant success, we must confess that most systems' knowledge bases remain fossilized unless extended with human help. The use of *machine learning* (ML) techniques by more recent systems has improved the situation a bit, but recognition and interpretation problems remain.

The aim of this paper is threefold. The first is to present a four-process student modeling framework within which the main approaches in the literature can be studied in a unified manner. The second is to review, using the same framework, the different ways in which ML has been used in student modeling. The third is to outline how two recent ML strategies, namely, relational theory revision and relational clustering, can be used to overcome the main limitations of the usual approaches, and how these various strategies are integrated in the multistrategic learning student modeling system, MULSMS (mŭl´sms). Sections 2, 3 and 4 address these objectives respectively.

2 A Four-Process Model of Student Modeling

We view *student modeling* as a process entailing four, possibly concurrent, activities: (1) *behavior recognition,* (2) *behavior characterization,* (3) *student model induction,* and (4) *student model maintenance.*

Definition 1 *(Student) behavior is any observable response that is used as input to the student modeling process.*

A behavior may be decomposed recursively into components and subcomponents. Behavior complexity is thus a factor of the number of subcomponents and subcomponent levels, and the degree of interrelationships among these.

Definition 2 *Behavior recognition is a matching process undertaken to identify current behavior as correct or incorrect from background knowledge.*

There are two main approaches to recognizing behavior, namely, (1) transformation and (2) verification.

1. *Transformation* uses a possibly empty sequence of *operators* to convert student behavior to an ideal one, or to an initial state, or vice versa. Successful transformation means behavior recognition. The main classes of operators for transforming one behavior, $\beta 1$, to another, $\beta 2$, are:

   (a) *variation,* where $\beta 1$ and $\beta 2$ are equivalent,

   (b) *perturbation,* where $\beta 1$ and $\beta 2$ are not equivalent,

   (c) *decomposition/aggregation* (*"decagg"*), in which $\beta 2/\beta 1$ is a set of component behaviors,

   (d) *generalization/specialization* (*"genspec"*), where $\beta 2$ is more general/specific than $\beta 1$, and

   (e) *analogy,* in which $\beta 2$ is structurally similar to $\beta 1$, though $\beta 2$ would otherwise be unobtainable using only the previous operators.

   We call the first two operators *basic* operators, since they are used by all transformation-based systems. The third operator is found only in systems which deal with *complex,* i.e., decomposable, behavior. We call the last two *learning* operators, as only systems with learning capabilities have these. In such systems, all five operations on behavior can also be performed on operators.

   Transformation-based recognition can be viewed as a form of *state space search.* This space can be constructed *offline* (e.g., PROUST (Johnson & Soloway, 1984), PIXIE (Sleeman, 1987), and the ACT* tutors),[1] *online* (e.g., ADAPT (Gegg-Harrison, 1994)), or both offline and online(e.g., SMS1 (Sison & Reyes, 1995)).

[1] PROUST has no explicit transformation operators.

2. *Verification* uses a theorem prover to demonstrate that the student behavior is equivalent to an ideal. TALUS (Murray, 1986), for example, employs a Boyer-Moore induction theorem prover, while Hoppe (1994) applies resolution theorem proving. PDS (Shapiro, 1983) uses, together with an oracle, the sequence of procedure calls that an interpreter like the resolution-based PROLOG would make when executing a program.[2]

Transformation's appeal lies in the fact that the operators used for behavior recognition are the same ones used in the other student modeling processes. Its strength depends, however, on the power of the variation operators and the completeness of the perturbation operators. In applications involving complex behavior, these cannot be guaranteed. Verification, on the other hand, will recognize any arbitrary behavior – as long as it can be cast in the form required by the theorem prover. However, the results of verification are insufficient to characterize errors.

Behavior recognition is more important when dealing with complex behavior (e.g., programs) than when the behavior is, say, a single number. Hence, DEBUGGY (Burton, 1982) and ACM (Langley and Ohlsson, 1984), which deal with numbers in subtraction, and THEMIS (Kono *et al.*, 1994) and ASSERT (Baffes & Mooney, 1996), which deal with labeled propositions, perform relatively trivial behavior recognition. The bulk of their work is in the induction of the student model from a set of noncomplex behavior.[3]

Definition 3 *Behavior characterization is the process of differentiating between the student behavior and an ideal, and interpreting this difference.*

Any difference between the ideal and the student behaviors is called a *discrepancy.* If the discrepancy results in incorrect behavior, then we call it a *bug.* Bug characterization involves two subprocesses, namely, (1) bug specification and (2) bug interpretation.

1. *Bug specification* is the enumeration of the specific differences between the student and ideal behaviors. We have earlier alluded to the fact that transformation operators can actually differentiate between the ideal and student behaviors. That is, the operators used to recognize buggy behavior are also used to specify the bugs. Verification can also recognize incorrect behavior and can, using the residue of a failed proof, identify the buggy component, but it cannot precisely specify the bug itself.

2. *Bug interpretation* involves *identifying relationships* among bugs and *making generalizations* about these. These generalized relationships are what we call *misconceptions.* All systems prior to ASSERT and MULSMS which can interpret low level bugs can do so only with the help of hand-coded, built-in bug interpretation libraries. A *bug interpretation library*, or bug library for short, is a collection of bugs together with their corresponding interpretations.

[2] TALUS and PDS are program *debuggers* rather than student modeling systems. Hence, they correct rather than interpret (v. Definition 3) behavioral discrepancies.

[3] We view the student modeling systems cited so far as representative of the modelers in the literature. Of particular interest, or course, are those that have learning capabilities, namely, PIXIE, SMS1, Hoppe's modeler, DEBUGGY, ACM, THEMIS, ASSERT and MULSMS, all of which will be discussed in the sections that follow. The ACT*-based tutors, while capable of learning the ideal model, cannot recognize unexpected student behavior.

Definition 4 *A student model is a representation of student knowledge about a particular domain. The generation of this model is called student model induction.*

This is generally an inductive process since there is no way to guarantee the correctness of arbitrary student models. Approaches to SM induction lie on a spectrum with synthetic induction on one end and transformational induction on the other.

1. *Synthetic model induction* involves selecting and combining model components and subcomponents, and then iteratively making modifications on this set until it finally explains student behavior. This is the approach of DEBUGGY, ACM, and THEMIS (Kono *et al.*, 1994), which we shall take up later.

2. *Transformational model induction,* on the other hand, involves modifying an existing, usually ideal, model so that the final model explains student behavior. This is the approach taken by systems which use transformation operators for recognition and characterization. In some cases (e.g., PROUST), the induction of the student model is implicit.

Definition 5 *Student model maintenance is the process of incorporating newly learned knowledge about a student into (a) the specific student model for this individual or (b) to the collection of ideal and buggy models for the domain. The latter is called the General Student Model.*

So far, we have used the term student model to refer to a model of a *specific* student. Actually what we have called *background knowledge* (to avoid confusion at the start) is a student model, albeit a general one.

For specific student models, maintenance approaches vary from near-zero (i.e., the SM is induced afresh without regard to the previous model) to the explicit use of truth maintenance subsytems (e.g., THEMIS). For general student models, maintenance ranges from absolutely zero (i.e., the general model is static), which is the case in all nonlearning systems, to fully automatic bug library construction and maintenance (e.g., ASSERT, MULSMS).

Recognition and Characterization Failures

Failure to recognize and characterize behavior arises when the background knowledge and the basic transformation operators or verification axioms are incomplete. One solution to this problem would be to look for more powerful variation operators and domain heuristics, and to carefully circumscribe the modeler's area of competence (e.g., ADAPT). Because this can lead to loss of generality, however, some systems have opted to use a *hybrid* approach. Thus, some transformation-based systems resort to verification in the face of recognition failure (e.g., SCENT-3 (McCalla *et al.*, 1988)) while some verification-based systems use transformation operators to offset their inability to specify bugs (e.g., (Hoppe, 1994); TALUS). However, recognition and interpretation limitations will remain as long as only the basic transformation operators are used. Machine learning has the potential to remedy this limitation.

3 Varieties of Learning in Student Modeling

We roughly classify machine learning (ML) methods into three types according to the primary approach they take to inferencing: induction, deduction, and analogy.

Inductive learning methods hypothesize premises for given consequents. These hypotheses are, however, attended by varying degrees of uncertainty. *Deductive* methods, on the other hand, derive consequents from premises using rules in their background knowledge. Assuming that the background knowledge is complete and correct, deduction therefore guarantees inference and the truth of this inference. *Analogical* methods can be viewed as a combination of the first two types. Basically, learning by analogy involves mapping the conceptual structure of a so-called "base" system onto a "target" system which the base system should match to some threshold degree.

Using our modeling framework, we now review the ways in which ML techniques have been used in student modeling. It will be noted that the systems we will be reviewing first – PIXIE, SMS1 and (Hoppe, 1994) – induce specific student models in the process of recognizing/specifying student behavior. There are at least two reasons for this integration. First, these systems use a transformation-based approach for both behavior recognition and model induction, and the transformation operators they use in both processes are essentially the same. Second, they deal with complex, i.e., decomposable, behavior. This, in turn, implies that the model underlying the behavior is likewise complex, making it difficult to obtain, let alone analyze a set of such behavior from one student, although this very same complexity makes it possible in general to obtain useful information from a single behavior.

In contrast, systems such as DEBUGGY, ACM, THEMIS, and ASSERT deal with noncomplex (i.e., nondecomposable) behavior. Since the recognition and characterization of noncomplex behavior is trivial, it is not generally possible to induce a model from a single behavior. For this, a *behavior set* is necessary.

3.1 Machine Learning and Behavior Recognition

The simplest learning method used in behavior recognition involves *operator specialization.* For example, the specialization of variation operators becomes a form of deductive learning when the new operator is compiled into the general model, as it is done in SMS1. The specialization of perturbation operators in PIXIE/MALGEN (Sleeman *et al.*, 1990)), on the other hand, is inductive in that the falsity of the more general perturbation operator is preserved. However, this can also be viewed as a form of deduction in the sense that no new knowledge is really learned. This is the view taken by SMS1.

The only other main type of learning that has so far been used in behavior recognition is "pre-inductive." When a verification-based system, for example, generalizes the residue of a failed proof and then maps the structure of a similar but correct behavior onto this residue, the system in effect induces a new rule. This approach to malrule construction, which Hoppe (1994) calls "structure mapping," is similar to the "gap filling" approach used in PIXIE/INFER* (Sleeman *et al.*, 1990) and to the "analogical generalization" approach in SMS1. In all three cases, structural similarities between the buggy and the ideal behaviors are identified and the structure of the latter imposed upon the former. We call this approach *morphing,* which we view as "pre-inductive" (or

pre-analogical) because it lacks the stricter testing for correctness that attends inductive learning algorithms.

3.1.1 Morphing and operator specialization in PIXIE

PIXIE (Sleeman, 1987) recognizes and interprets student behavior by selecting from a set of offline-generated correct and buggy models one that produces the student's behavior. In PIXIE, a model is a collection of rules and malrules that can be used to produce correct and incorrect answers to a set of problems in arithmetic and introductory algebra.

Given a correct model, PIXIE generates student models by first using variation operators to modify the ordering of rules in the model without changing its original meaning. Then it replaces specific rules with associated buggy rules, which are predefined in (Sleeman, 1987) and inferrable in (Sleeman *et al.*, 1990)), in order to account for buggy behavior.

Two algorithms for inferring buggy rules in introductory algebra, namely, INFER* and MALGEN, have been experimentally coupled with PIXIE (Sleeman *et al.*, 1990). The first algorithm, INFER*, involves a bidirectional search of the transformation space, the initial and goal states of which are the initial equation and the student's answer, respectively. Malrules are inferred pre-inductively in order to complete paths in this search space. Table 1 describes INFER*'s basic algorithm.

Table 1: Basic INFER* morphing algorithm

1. Apply variation and heuristic genspec rules to transform the student's final equation to the initial equation. If the initial equation could not be derived, call the leaf nodes of this subtree S-nodes.
2. Apply variation and heuristic genspec rules to transform the initial equation to the student's final equation. If the final equation could not be derived, call the leaf nodes of this subtree T-nodes.
3. Use heuristics to find "reasonable" numerical relationships between the coefficients of the S and T nodes. A numerical relationship constitutes a potential malrule. Forward these potential malrules to the investigator, who decides which malrule(s) to accept.

Whereas INFER* infers malrules using morphing, MALGEN specializes general perturbation operators. MALGEN views an operator as a rule made up of four parts, namely, a *pattern* against which the current state is matched, a set of *correctness conditions* specifying when the rule is appropriate to apply, a set of *actions* to be carried out, and a *result,* which is the state that ensues from the action. MALGEN perturbs all of these parts except the first in order to infer a malrule. The perturbation operators are shown in Table 2.

3.1.2 Operator specialization and morphing in SMS1

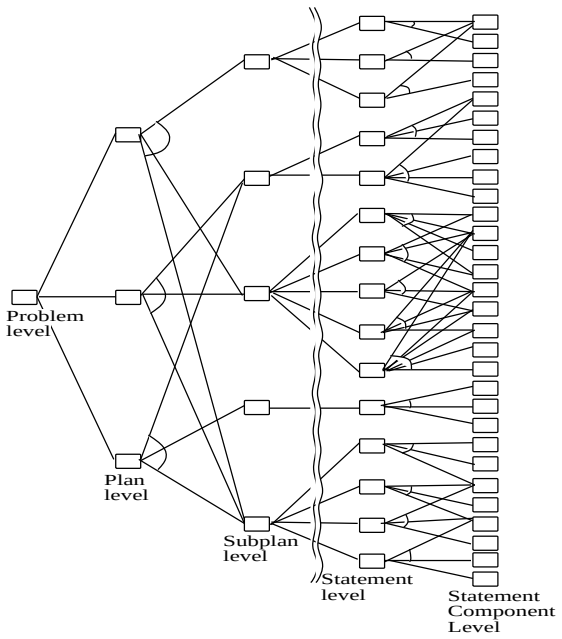
Since SMS1 (Sison & Reyes, 1995) deals with more complex behavior than that which can be handled by PIXIE, SMS1's transformation operators and transformation space

Table 2: General Perturbation Operators of MALGEN

1. To perturb the correctness condition, negate it as follows. Negate a disjunction as in De Morgan's. Negate a conjunction of n expressions by producing n new conjunctions such that each conjunction has exactly one negated expression. Each of these new conjunctions can be negated further.
2. To perturb an action, either (a) replace it with a similar action, (b) remove it, or (c) switch its arguments noncommutatively.
3. To perturb results, switch the operands in it, if any.

are, therefore, correspondingly more complex .

For example, SMS1, which represents the general student model as an AND/OR hierarchy of correct and buggy program components and subcomponents (Figure 2), has parallel general variation and perturbation operators for each AND level of this hierarchy. These general operators are summarized in Table 3. The specialization of any of these operators results, by default, in the creation of a new node in the respective AND levels. (The default option can be turned off to save space at the expense of search time.)



Note: There can be several AND-OR levels between what we have labeled as the Subplan and Statement levels.

Figure 2: General Student Model Representation in SMS1

Whereas deductive learning occurs only in the AND levels, pre-inductive learning spans all levels. Pre-inductive learning is performed only after operator specialization has been tried without success. The algorithm for morphing is shown in Table 4.

Table 3: General Variation and Perturbation Operators of SMS1

	General Variation operators	General Perturbation operators
Plan level	Perform a topological sort on the *dependency* graph of the subplans of a plan to generate equivalent serializations.	Reorder the subplans without regard for dependency constraints. Replace a control/IO construct with another from the same category. Delete a subplan. (Subplans cannot be added arbitrarily.)
Statement level	Transform an expression observing the precedence, associativity, and distributivity constraints.	Switch/Rearrange the operands of an expression without regard for the precedence etc. constraints. Replace an operator with another from the same category, or an identifier with another from the same program. Delete a subexpression or argument. (Operands cannot be added arbitrarily.)

Table 4: Morphing in SMS1

1. From the generalized components of the statement that could not be recognized as being part of a known plan, infer a new type of statement for the domain, if one does not yet exist. (That is, create a new node at the statement level.)
2. From the structure of the new statement, infer its purpose. (That is, create a node at the subplan level.)
3. Infer a plan whose structure is based on the plan most similar to the student's. (That is, create a node at the plan level whose subplans are those subplans of the student that have already been recognized.)
4. Link all the the newly created nodes to form an explanation for the newly encountered bug.

3.1.3 Morphing in (Hoppe, 1994)

Unlike PIXIE and SMS1, which use a purely transformation-based approach to behavior recognition and specification, Hoppe's (1994) approach is a hybrid one. First, using resolution, Hoppe's system attempts to prove a student's answer to a symbolic differentiation problem given the rules for this domain. A failed proof means that the student behavior is incorrect, assuming of course that the domain rules are correct and complete. The residue of this failed proof can be used to identify the student's misconception (Costa *et al.*, 1988).

Although the system uses verification to recognize behavior, it has to resort to transformation to learn a new malrule. That is, after collecting the residue of the failed proof, it generalizes (morphs) this residue to form a new malrule.[4] It is not clear how the new malrule is incorporated into the original theory.

3.2 Machine Learning and Behavior Interpretation

Apart from the approaches just outlined, the application of ML to behavior interpretation has been minimal, so that most interpretations of bugs are made with the help of hand-coded bug libraries. ASSERT's bug generalization procedure, however, which extracts commonalities between two newly learned bugs is a step in the right direction. Table 5 gives the basic algorithm for this procedure.

Table 5: Bug library construction in ASSERT

1. Collect all bugs from all student models and remove duplicates.

   For each bug, B_i, from $Model_i, M_i$, compute a "stereotypicality" value S as follows:

   $$S(B_i) = \sum_{j=1}^{n} Distance(C, M_j) - \sum_{j=1}^{n} Distance(C+B_i, M_j)$$

   where C is the Correct Rule Base, and n, the number of models.

2. Add the "best" generalization of each bug to the bug library.

   To compute for the generalization of a bug, iteratively get the intersection between it (or its generalization computed in the previous iteration) and the other bugs collected in step 1 until the stereotypicality of this intersection no longer improves on the previous one.

3. Rank the bugs in the bug library according to stereotypicality.

3.3 Machine Learning and Student Model Induction

The synthetic approach to SM induction, used in DEBUGGY, ACM, and THEMIS, is similar to *empirical induction,* arguably the most popular form of learning in ML. Empirical induction is "data-driven" in that the *features* that will make up a concept's definition are chosen based on the amount of data (i.e., the number of *training examples*) they cover correctly.

[4] The procedure is quite straightforward so we no longer present it in a table.

3.3.1 Synthetic Model Induction in DEBUGGY

DEBUGGY (Burton, 1982) learns a subtraction malrule by selecting predefined bug specifications (the features) and then iteratively removing, combining or coercing elements of the evolving set until a student's answers to a set of subtraction items (the training examples) are explained. The basic procedure is shown in Table 6.

Table 6: Basic learning algorithm of DEBUGGY

1. From a set of predefined bugs, select all those that explain at least one wrong answer in the student's behavior set, B. Call this initial hypothesis set, H.
2. From H, remove bugs that are "subsumed" in others. A bug Hi is subsumed in another bug Hj if $Hi \leq Hj$ on every answer in B and $Hi < Hj$ on at least one. $Hi < Hj$ if the level of "goodness" of Hi is less than that of Hj.
3. Pair each bug in the reduced hypothesis set, H', with all the other bugs in H' to see if the resulting compound bug covers more answers than either of its constituent bugs. If so, add this compound to H'. Repeat the procedure on the newly added compound bugs.
4. From the set of compound bugs, H'', select those that explain a given percentage of the student's answers. Call this set P. For every bug in P, first identify the student answers for which the bug predicts a different answer, then coerce (using heuristic perturbation operators) the bug so that it reproduces the student's behavior.
5. Classify and rank the bugs in P according to the number of predicted correct and incorrect answers, the number and type of mispredictions, and the number and type of coercions. Choose the bug with the highest score.

DEBUGGY first selects from a set of predefined bugs those that explain at least one wrong answer in the student's behavior set. From this initial hypothesis set, bugs that are "subsumed" in others are removed. Step 2 in Table 6 gives DEBUGGY's operational definition of subsumption. For this definition, DEBUGGY uses three levels of bug goodness, namely (from highest to lowest), bugs that agree with the student's answer, those that predict the correct answer when the student's answer is incorrect, and those that predict a wrong answer different from the answer the student gave.

The bugs in the reduced hypothesis set are then *compounded* by pairing each bug with all the others in the set. If this bug covers more answers than either of its constituent bugs, it is added to the hypothesis set. This procedure is repeated on the newly added compound bugs until the number of bugs in a compound reaches a system parameter. Additional heuristics are used to curb the explosiveness of this pairing process.

From the set of compound bugs, those that explain a given percentage (system parameter) of the student's answers are selected, and every bug in this smaller set that predicts an answer different from that of the student is perturbed. Finally, the bugs are ranked (see step 5, Table 6) and the bug with the highest score is chosen.

3.3.2 Synthetic Model Induction in ACM

The approach of ACM (Langley & Ohlsson, 1984), which also deals with learning subtraction malrules, is also empirical, although it first builds a *decision tree* in a manner

similar to ID3/TDIDT (Quinlan, 1986). This tree is then collapsed into a malrule. Table 7 shows the basic procedure of model induction in ACM.

Table 7: Basic learning procedure of ACM

1. For every answer in the student's behavior set, contruct the search space using the answer as *goal,* the problem given as *start state,* and the subtraction operators as search *operators.* Label every application of an operator that lies on the solution path as positive while those that lie one step off this path as negative. Collect all such positive and negative instances for all the answers in the student's behavior set.
2. Construct a decision tree that uses conditions (e.g., $number1 > number2$) to discriminate negative from positive instances. The conditions (or features, in empirical induction terminology) are chosen and ordered based on a measure

   $$E = max(S, 2 - S) \text{ where } S = M_+/T_- + U_-/T_-$$

   where M_+ is the number of positive instances matching the condition, M_-, the number of negative instances that fail to match the condition, and T_+ and T_-, the total number of positive and negative instances respectively.
3. Eliminate all branches of the decision tree whose leaves are negative instances. Collapse the remaining subtree into a condition-action rule.

To build a decision tree, ACM first generates a set of *positive* and *negative examples.* It does this by constructing a search tree for every answer in the student's behavior set. Every application of a subtraction operator that lies on the solution path is taken as a positive example; those that lie one step off this path are taken as negative examples.

Using these examples, ACM builds a decision tree to discriminate between positive and negative instances. The features of this tree (which are conditions for applying subtraction rules) are chosen and ordered based on a simple measure (see step 2 in Table 7). Finally, the decision tree is collapsed into a single malrule after all branches leading to negative instances have been removed.

3.3.3 Transformational Model Induction in THEMIS

Based on MIS (Shapiro, 1983), THEMIS' (Kono *et al.*, 1994) approach to SM induction is also empirical but, unlike the previous two modelers which process a set of behaviors as one batch, THEMIS is incremental, processing behaviors one by one as they come. THEMIS deals with behavior in the form of a labeled proposition, i.e., a behavior has the form $< p, v >$ where p is a proposition and v a truth value assigned to the proposition by the student.

The basic algorithm in Table 8 differs from Shapiro's MIS in three ways. First, since THEMIS is apparently meant to be used with a Socratic tutor, it makes quick assumptions about the student (step 1) and deals with student inconsistencies. It therefore requires an ATMS (de Kleer, 1986) to explicitly maintain consistency among beliefs of and about the student. Second, the algorithm does not build the student model from scratch. Instead, it starts with either the ideal model or a perturbation of it (step 1), in line with its policy of assumption-making. Thus, THEMIS is transformational by

Table 8: Basic learning procedure of THEMIS

1. Construct the initial model as follows. If the initial behavior is correct, instantiate the student model to the ideal model. Otherwise, instantiate the student model to the ideal model with one component removed.
2. For every new behavior provided by the student in response to a system query:
   (a) If the student model returns `false` on a proposition labeled by the student as `true`, then find a true proposition not covered by the model, search for a rule that covers this proposition, and add that rule to the model.
   (b) If the model returns `true` on a proposition labeled by the student as `false`, then detect a false rule in the model and remove it.
   (c) Repeat the previous steps to ensure that all known behavior are covered by the model. If this is not possible, then the student's reasoning must be inconsistent. Resolve this using heuristics (e.g., `give-priority-to-correct-answers`).

our definition, although MIS is synthetic. Third, the algorithm lacks MIS' test for program nontermination. This may be due to the fact that the domains that THEMIS can handle are limited to those that do not require recursive reasoning.

3.3.4 Transformational Model Induction in ASSERT

Unlike synthetic SM induction, transformational SM induction (PIXIE, SMS1, Hoppe's) has no direct counterpart in inductive ML, although *theory revision* comes close. For example, using the propositional theory reviser NEITHER (Baffes & Mooney, 1993), ASSERT (Baffes & Mooney, 1996) induces an SM from the ideal model and a student's answers to a set of multiple-choice questions, and it is the SM, rather than the student behavior, that is examined for misconceptions. Table 5 shows the basic procedure of model induction in ASSERT.

Like THEMIS, ASSERT deals with behavior in the form of a labeled proposition. Specifically, a behavior in ASSERT has the form $< f, v >$ where f is a vector of values, each value representing a property of a program, and v a truth value assigned to the feature vector by the student. To induce the student model, ASSERT, like the preceding three systems, relies on a set of such behavior.

3.4 Discussion

Table 10 summarizes the various student modeling and machine learning approaches used by the systems just reviewed. Deductive specialization (SMS1) and pre-inductive generalization of gap-filling perturbation operators (PIXIE, SMS1, (Hoppe, 1994)) alleviate to some extent the limitations caused by incomplete transformation operators. However, we cannot accept a new perturbation operator that adds a component into, say, the ideal behavior, without much sounder justification than the component's mere presence in the student behavior. Moreover, the operators learned via transformation are too low level to be useful as an interpretation (i.e., individual operators can hardly be

Table 9: Basic learning procedure of ASSERT

1. Construct the initial model by initializing the student model to the ideal model.
2. Refine the student model by iterating through the following three steps until all student answers in the behavior set are covered:
   (a) Find a student answer in the behavior set that is not covered by the current model. Find a revision for this failing instance. That is, for a failing positive instance, compute the set of antecedents whose deletion will fix the model. For a failing negative, compute the set of rules whose deletion will fix the model.
   (b) Test the revision against all the other answers in the behavior set. If the entire behavior set is covered, apply the revision to the model.
   (c) If the behavior set is not covered entirely, induce new rules/antecedents using a propositional variant of an inductive logic programming (ILP) learner such as FOIL (Quinlan, 1990).

considered misconceptions). Empirical induction (DEBUGGY, ACM) could solve the problem of weak justifications to some extent. However, it wouldn't solve the second limitation. In the next section, we propose a multistrategic learning student modeling system called MULSMS that takes advantage of more recent ML techniques to address both problems.

4 Toward a Multistrategic Learning Student Modeling System

MULSMS is multistrategic in both the student modeling and the machine learning senses. It is a multistrategic student modeler in that it employs several strategies for behavior recognition and interpretation, albeit all transformational. It is a multistrategic learner in the sense of (Michalski & Tecuci, 1994) in that integrates several inferencing and computational learning strategies. It is unique as a multistrategy learner in that it achieves this integration within, and through, the (four-process) student modeling framework (Figure 3).

4.1 SMS1

MULSMS extends SMS1 (Sison & Reyes, 1995), in which the General Student Model is represented as a hierarchy of correct and buggy program components. If the general model cannot recognize student behavior, then variation and perturbation operators are specialized and applied. If recognition remains incomplete, analogical generalization operators are induced via morphing. These and the specialized operators are automatically learned (i.e., incorporated into the general student model).

As discussed earlier, SMS1 is a learning system that is deductive and pre-inductive. Given an initial background knowledge of only 7 plans culled from 10 programs, SMS1

Table 10: Comparison of Learning Student Modeling Systems

System	Student Behavior	Behavior Recognition	Behavior Characterization	Student Model Induction	Student Model Maintenance
PIXIE (INFER*, MALGEN)	Linear equation. Complexity level: Intermediate.	Integrated transformation-based behavior recognition, specification and SM induction. Malrules learned via deduction and pre-induction. Limited behavior interpretation via transformation operators used. (Note: All learning is done offline. Online modeling is thus reduced to a matter of selecting the specific SM from the general SM.)			Minimal SM maintenance. Specific SM erased after session. Learned malrules are used to generate new buggy models offline.
SMS1	Pascal program. Complexity level: High.	Integrated transformation-based recognition, specification and SM induction. Malrules learned via deduction and pre-induction. Limited behavior interpretation via the transformation operators used.			Minimal SM maintenance. Specific SM incorporated automatically into general SM.
(Hoppe, 1994)	Polynomial equation. Complexity level: Intermediate.	Verification-based recognition (resolution theorem proving).	Transformation-based specification. Malrules learned via pre-induction. Behavior interpretation minimal.	Transformation-based SM induction.	Minimal SM maintenance. Specific SM erased after session. Learned malrules added to a pool of malrules.
DEBUGGY	Difference of two numbers. Complexity level: Low	Trivial recognition of single behavior.	Trivial specifica-tion of single behavior. Thus the need for a behavior set. Limited interpretation via the induced SM aka buggy rule.	Synthetic SM induction via empirical rule induction. SM constitutes one malrule.	Minimal SM maintenance. Specific SM erased after session. Specific SM can be inserted into general SM. Otherwise, general SM exists in the form of a pool of primitive correct and buggy subtraction rules.
ACM	Difference of two numbers. Complexity level: Low	Trivial recognition of single behavior.	Trivial specification of single behavior. Thus the need for a behavior set. Limited interpretation via the induced SM aka buggy rule.	Synthetic SM induction via decision tree induction. SM constitutes one malrule.	Minimal SM maintenance. Specific SM erased after session. Specific SM can be inserted into general SM. Otherwise, general SM exists in the form of correct subtraction rules.
THEMIS	Labeled proposition. Complexity level: Low.	Trivial recognition of single behavior.	Trivial specification of single behavior. Thus the need for a behavior set. Limited interpretation via the justification set of each component of the induced SM.	Data-driven transformation-based SM induction via MIS-style induction with assumptions.	ATMS used to maintain consistency of specific SM in a session.
ASSERT	Labeled feature vector. Complexity level: Low.	Trivial recognition of single behavior.	Trivial specification of single behavior. Thus the need for a behavior set. Limited interpretation via the revision operators used and the generalization mechanism for automatic bug pool construction.	Data-driven transformation-based SM induction via theory revision.	SM erased after session. General SM exists in the form of a pool of correct and buggy rules.
MULSMS	Prolog program. Complexity level: High.	Transformation-based recognition. Malrules learnable online via deduction and inductive theory revision.	Transformation-based specification. Interpretation via dynamically updated bug interpretation library. Library induced via incremental relational clustering.	Transformation-based SM induction. SM induced during behavior recognition and specification.	SM retainable after session. Specific SM incorporated automatically into general SM.

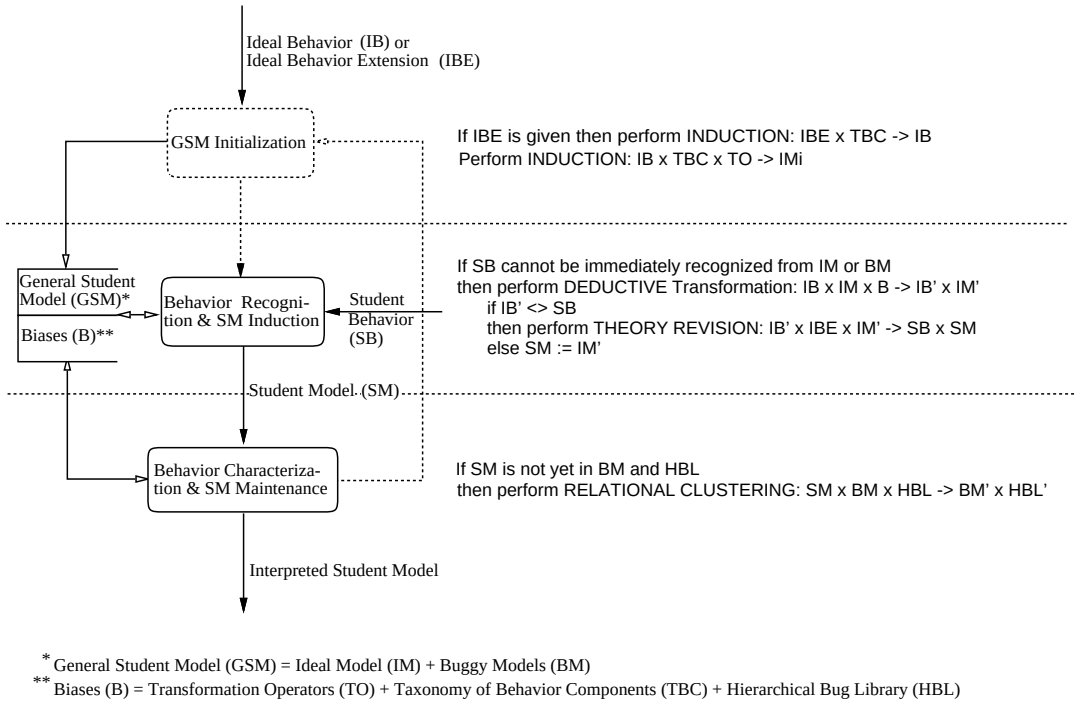


Figure 3: Architecture of MULSMS, a Multistrategic Learning Student Modeling System

was able to recognize and characterize all 30 bug classes found in the bug catalogue of Cutler *et al.* (1983), culled from 206 students.

4.2 MULSMS

MULSMS extends SMS1's capabilities in two important ways. First, it enables the addition of a new component to the buggy program whenever it can be shown that doing so leads to a new solution. The new solution constitutes the justification for the added component. Second, it allows the possibility of clustering individual instances of perturbations so as to acquire new meaning from bugs that are otherwise too low level to be interpreted by themselves. These two extensions can be achieved using techniques from *relational theory revision* and *conceptual clustering.* We now outline how this is possible.

4.2.1 Theory Revision

Theory revision (TR) is the process of inducing a correct set of rules defining some concept from an initial set of incorrect or incomplete rules, using a set of positive and negative training examples. A few relational TR systems (i.e., systems which can handle theories in the first order predicate logic, which are our concern here) have been proposed, which include Rx (Tangkitvanich & Shimura, 1992) and FORTE (Richards & Mooney, 1995).

TR is especially useful when the student behavior has more components than the ideal. As noted previously, adding a component to the correct behavior cannot be accomplished nonarbitrarily by perturbation. Although morphing can do this, it does so in a manner not unlike brute force. In contrast, a malrule learned using TR can be justified in terms of the correct behavior TR produces, which, incidentally, would constitute a totally new solution variant.

To illustrate this, consider the following buggy PROLOG program for finding a path in a directed graph. Note the unusual recursion scheme in `(1)`.

```
          path(H1,H2) :- edge(H1,H2).
(1)       path(H1,H2) :- edge(H1,B1), edge(B2,H2), path(B1,B2).
```

Given the correct program,

```
          path(H1,H2) :- edge(H2,H1).
(2)       path(H1,H2) :- edge(H1,B1), path(B1,H2).
```

perturbation cannot, in principle, transform the correct recursive clause `(2)` to the buggy recursive clause `(1)` (i.e., it should not simply add a literal), while morphing will only either delete the extra `edge(B2,H2)` literal in `(1)` or "plug" this literal into `(2)`. In contrast, a relational theory reviser like FORTE would yield the following revised code:

```
          path(H1,H2) :- edge(H1,H2).
(3)       path(H1,H2) :- edge(H1,B1), edge(B1,H2).
          path(H1,H2) :- edge(H1,B1), edge(B2,H2), path(B1,B2).
```

Note that both buggy clauses were left untouched, and a new clause `(3)` was added instead.

Not all relational theory revision systems can produce the same effect, however. Of particular interest to student modeling are those revisers such as FORTE which use a *distance* metric to guide the transformation process. This favors theories that make small incremental changes rather than major overhauls of the initial buggy theory.

4.2.2 Relational Clustering

We have already noted earlier that the application of perturbation or morphing rules yields discrepancies whose characterizations are too low level to be interpreted as misconceptions. For example, consider the following buggy PROLOG program for `reverse/2`:

```
reverse([],[]).
reverse([H1|H2],[H3|H1]) :- reverse(H2,H3).
```

Given the following correct program,

```
reverse([],[]).
reverse([H1|H2],H3) :- reverse(H2,B1), append(B1,[H1],H3).
```

SMS1 will yield the following perturbations:

```
head :          replace H3 with [H3|H1]
goal1:          replace B1 with H3
goal2:          delete
```

The problem with the above characterization is that although it is an accurate low level specification of how the correct and buggy programs differ, individual `replace` operations do not immediately reveal anything other than the specification of the error.

Now suppose SMS1 is given another buggy program:

```
reverse([],[]).
reverse([H1|H2],H3) :- reverse(H2,B1), H3 = [B1|H2],
```

SMS1 first applies a variation operator (`substitute equalities`) on the new program giving:

```
reverse([],[]).
reverse([H1|H2],[B1|H2]) :- reverse(H2,B1).
```

for which SMS1 will return the following perturbations:

```
head :          replace H3 with [B1|H2]
goal1:          (correct)
goal2:          delete
```

Now, MULSMS (but not SMS1) will recognize that `H3` has been replaced with a list expression (`=[|]`) for the second time. This constitutes a similarity and can be a ground for clustering the two discrepancies.

Noting that in both cases the `append(B1,[H1],H3)` subgoal is missing, and that both this goal and the list expression compute the value of `H3`, MULSMS now tries to equate `append` with `=[|]`. In so doing, MULSMS discovers (i.e., learns without supervision) that: some students think that the list operator `[|]` can be used like `append/3` to append a list in front of an atom/list. MULSMS will then place these two cases in one cluster representing a newly acquired higher-level perturbation rule, which now corresponds to a *misconception.* Moreover, this cluster and the one containing `append/3` can be placed under one higher-level concept representing the merging of lists, and so on.

The examples given above are part of a set of 15 actual buggy `reverse/2` programs. These and the algorithm we have developed for clustering relational descriptions are discussed in more detail in (Sison & Shimura, 1996). It will be noted that clustering algorithms in the literature (e.g., UNIMEM (Lebowitz, 1987), COBWEB (Fisher, 1987)) can only handle attribute-value representations, whereas here we are dealing with more complex nonground formulae in first order logic.

5 Conclusion and Future Work

We have accomplished three tasks in this paper. First, we have presented a four-process framework of student modeling and, using this as a unifying framework, we have summarized the main SM approaches in the field. Second, we have reviewed

the main applications of machine learning to student modeling in the last decade and a half. Third, we have outlined how relational theory revision and unsupervised relational clustering can be used to improve behavior recognition and characterization, and how these can be integrated in a multistrategic learning student modeling system called MULSMS.

Future work will involve the improvement of the relational theory revision system that we are currently using and of the relational clustering algorithm that we have developed. Evaluation of MULSMS will be done on two levels. On the "AI" level, the learning algorithms developed will need to be generalized, analyzed and compared against other ML systems. On the "education" level, the effectiveness of MULSMS in classifying bugs will be verified empirically using actual student data.

Acknowledgment

We thank Bradley Richards for letting us use the FORTE relational theory revision system and his original set of training examples for logic programming.

References

Anderson, J., Boyle, C., Corbett, A. & Lewis, M. (1990). Cognitive modeling and intelligent tutoring. *Artificial Intelligence,* 42:7-49.

Baffes P., & Mooney, R. (1993). Symbolic revision of theories with m-of-n rules. In *Proc. International Joint Conference on Artificial Intelligence '93.*

Baffes P., & Mooney, R. (1996). Refinement-based student modeling and automated bug library construction. To appear in *Jl. Artificial Intelligence in Education.*

Burton, R. (1982). Diagnosing bugs in a simple procedural skill. In Sleeman, S. & Brown, J. (eds). *Intelligent Tutoring Systems.* London: Academic Press.

Costa, E., Duchenoy, S. & Kodratoff, Y. (1988). A resolution-based method for discovering students' misconceptions. In Self J. (ed.) *Artificial Intelligence and Human Learning.* Chapman and Hall.

Cutler, B., Draper, S., Johnson, W.L., & Soloway, E. (1983). *Bug Catalogue I.* Cognition and Programming Project, Dept. of Computer Science, Yale University.

de Kleer, J. (1986). An assumption-based TMS. *Artificial Intelligence.* 28:127-162.

Fisher, D. (1987). Knowledge acquisition via incremental conceptual clustering. *Machine learning.* 2:139-172.

Gegg-Harrison T. (1994). Exploiting program schemata in an automated program debugger. *Jl. Artificial Intelligence in Education.* 5(2):255-278.

Hoppe H. (1994). Deductive error diagnosis and inductive error generalization for intelligent tutoring systems. *Jl. of Artificial Intelligence in Education.* 5(1):27-49.

Kono, Y., Ikeda, M., & Mizoguchi, R. (1994). THEMIS: A nonmonotonic inductive student modeling system. *Jl. of Artificial Intelligence in Education,* 5(3):371-413.

Johnson, W. & Soloway, E. (1984). Intention-based diagnosis of program errors. In *Proc. AAAI-84.*

Langley, P. & Ohlsson, S. (1984). Automated cognitive modeling. In *Proc. AAAI-84.*

Lebowitz, M. (1987). Experiments with incremental concept formation. *Machine Learning* 2:103-138.

McCalla, G., Greer, J. & the SCENT Research Team. (1988). Intelligent advising in problem solving domains: The SCENT-3 architecture. In . *Proc. ITS-88.*

Michalski, R., & Tecuci, G. (1994). *Machine Learning: A Multistrategy Approach,* Vol IV. Morgan Kaufmann.

Murray, W. (1986). Automatic program debugging for intellligent tutoring systems. *Computational Intelligence,* 3(1):1-16.

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

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

Richards, B. & Mooney, R. (1995). Automated refinement of first-order Horn-clause domain theories. *Machine learning* 19: 95-131.

Shapiro, E. (1983). *Algorithmic program debugging.* MIT Press.

Sison, R. & Reyes, R. (1995). A self-extending student modeling system for novice Pascal programming. In *Proc. Intl. Conf. on Computers in Education '95.*

Sison, R. & Shimura, M. (1996). Incremental clustering of relational descriptions. Technical Report TR96-0011. Dept. of Computer Science, Tokyo Institute of Technology.

Sleeman, D. (1983). Inferring student models for computer-aided instruction. In Michalski, R., Carbonell, J., & Mitchell, T. (eds). *Machine Learning, Vol. I* Palo Alto: Tioga Publ.

Sleeman, D. (1987). PIXIE: A shell for developing intelligent tutoring systems. In Lawler, R. & Yazdani, M. (eds.) *Artificial Intelligence and Education, Vol. I* New Jersey: Ablex Publ.

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

Tangkitvanich, S. & Shimura, M. (1992). Refining a relational theory with multiple faults in the concept and subconcepts. In *Proc. Ninth International Workshop on Machine Learning.*