Student Modeling and Machine Learning

Raymund Sison, Masamichi Shimura

International Journal of Artificial Intelligence in Education (1998), 9, 128-158

Student Modeling and Machine Learning

Raymund Sison, Department of Computer Science, Tokyo Institute of Technology
2-12-1 Ohokayama, Meguro, Tokyo, Japan 152-8552, sison@cs.titech.ac.jp

Masamichi Shimura, Department of Industrial Administration, Science University of Tokyo,
2641 Yamazaki, Noda, Chiba, Japan 278-8510, shimura@ia.noda.sut.ac.jp

Abstract. After identifying essential student modeling issues and machine learning approaches, this paper examines how machine learning techniques have been used to automate the construction of student models as well as the background knowledge necessary for student modeling. In the process, the paper sheds light on the difficulty, suitability and potential of using machine learning for student modeling processes, and, to a lesser extent, the potential of using student modeling techniques in machine learning.

INTRODUCTION

In general terms, student modeling involves the construction of a qualitative representation that accounts for student behavior in terms of existing background knowledge about a domain and about students learning the domain. Such a representation, called a student model, can assist an intelligent tutoring system, an intelligent learning environment, or an intelligent collaborative learner in adapting to specific aspects of student behavior (McCalla, 1992).

While we can be thankful that some student modeling systems, notably those of the ACT* tutors (Anderson, Boyle, Corbett & Lewis, 1990), have attained significant success, we have to admit that the construction of the background knowledge of many modelers, including those of the ACT* tutors, involves the manual, tedious, time consuming and error-prone analysis of student protocols, the result of which are then hand-coded into knowledge bases, which, once coupled with a student modeler, remain fossilized until extended with human help. Machine learning techniques have been used to automatically extend the background knowledge, as well as to automatically induce the student model, but limitations as well as potentials remain.

This paper examines the various ways in which machine learning or machine learning-like techniques have been used in the induction of student models and in the extension or construction of the background knowledge needed for student modeling. We will see that despite the apparent resemblance between the acquisition of a concept definition on one hand and the induction of a student model on the other, the use of supervised machine learning techniques to construct student models is not that straightforward, since student modeling, especially in problem solving domains, is much more complex than the tasks that most extant machine learning systems are capable of handling. However, we shall also see the promise of using machine learning techniques, particularly those for unsupervised learning, for constructing or extending the background knowledge needed for student modeling. Moreover, we will see that student modeling techniques might also be useful in machine learning.

The paper is organized as follows. First, it defines the essential elements of student modeling, and identifies important issues of the field. Then it outlines the major goals, approaches and paradigms of machine learning, and the main ways by which these approaches have been used in student modeling. Two sections then review how machine learning or machine learning like techniques have been used in student model induction and in background knowledge construction, respectively. Finally, the paper summarizes the difficulties and potentials of each main approach, and identifies and discusses important issues that need to be addressed further.

STUDENT MODELING

Simply put, student modeling is the construction of a qualitative representation, called a *student model*, that, broadly speaking, accounts for *student behavior* in terms of a system's *background knowledge*. These three - the student model, the student behavior, and the background knowledge - are the essential elements of student modeling, and we organize this section around them.

Student Behavior

We use the term *student behavior* to refer to a student's observable response to a particular stimulus in a given domain, that, together with the stimulus, serves as the primary input to a student modeling system. This input (i.e., the student behavior) can be an action (e.g., writing a program) or, more commonly, the result of that action (e.g., the written program). It can also include intermediate results (e.g., scratch work), and verbal protocols. In intelligent tutoring systems, stimuli from the tutor would typically come in the form of selected questions or problems about a particular topic.

Behavior Complexity.

It is sometimes expedient to distinguish between simple and (relatively) complex student behavior, where a behavior's complexity is determined by the number of its components and sub-components, and the kinds and degrees of interrelationships among these. Thus, one can regard an integer, say 565 (which can be a student's erroneous solution to the problem of subtracting 65 from 500), as a simple behavior, while a Prolog program, such as the erroneous two-clause recursive program for reversing a list in Figure 1, as (relatively) complex.

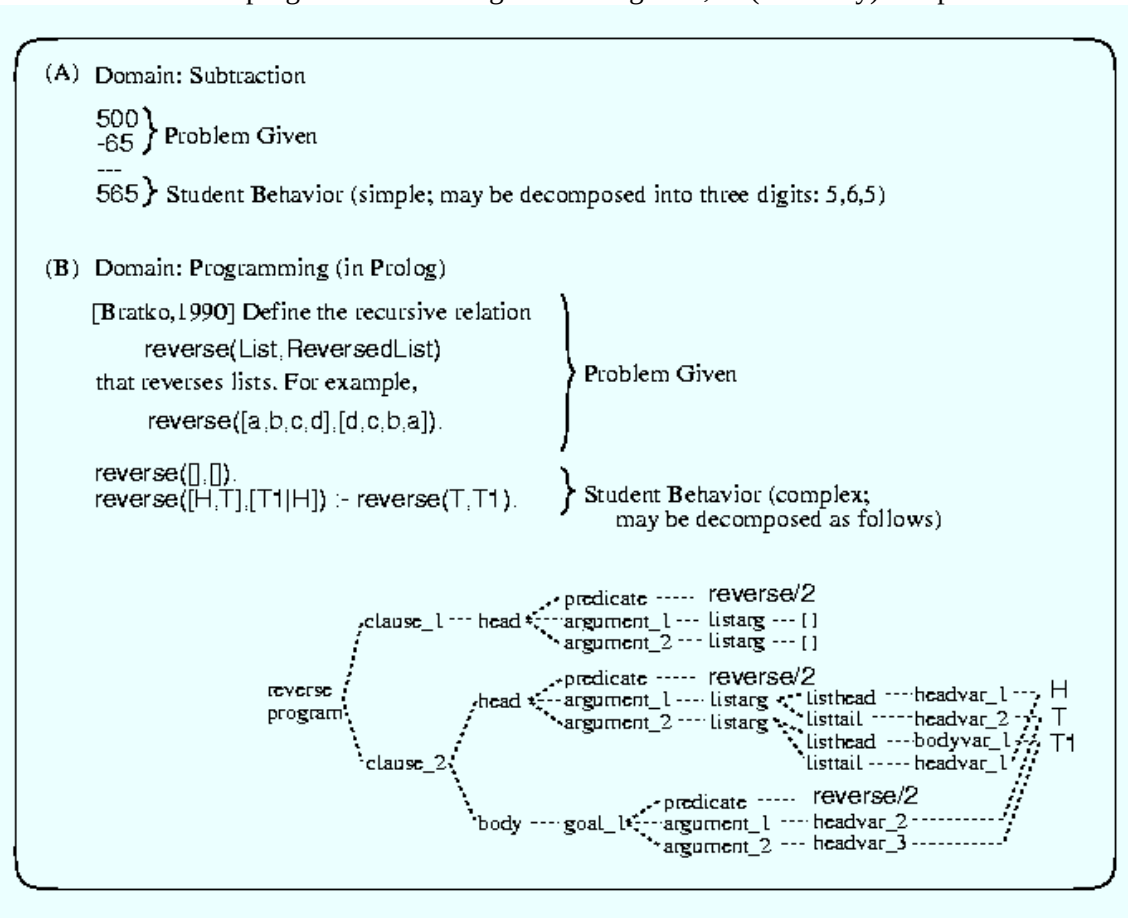

Figure 1. Examples of simple and complex student behaviors

Behavior Multiplicity.

Student modeling systems in classification (i.e., concept learning) and problem solving domains can be classified as to whether they can construct a student model from a single piece of behavior or require multiple behaviors to accomplish their task (Figure 2). Currently, systems that deal with relatively simple behaviors such as integers (e.g., DEBUGGY, Burton, 1982; ACM, Langley & Ohlsson, 1984), or propositions (e.g., ASSERT, Baffes & Mooney, 1996) require multiple behaviors to construct a student model. This is understandable, since a student model that is inferred from a single item of simple behavior such as an integer, will generally not be reliable. Each of the systems just mentioned uses a machine learning or machine learning like technique such as rule induction (DEBUGGY), decision tree induction (ACM), or theory revision (ASSERT) to synthesize a model that accounts for most, if not all, items in the input *behavior set*.

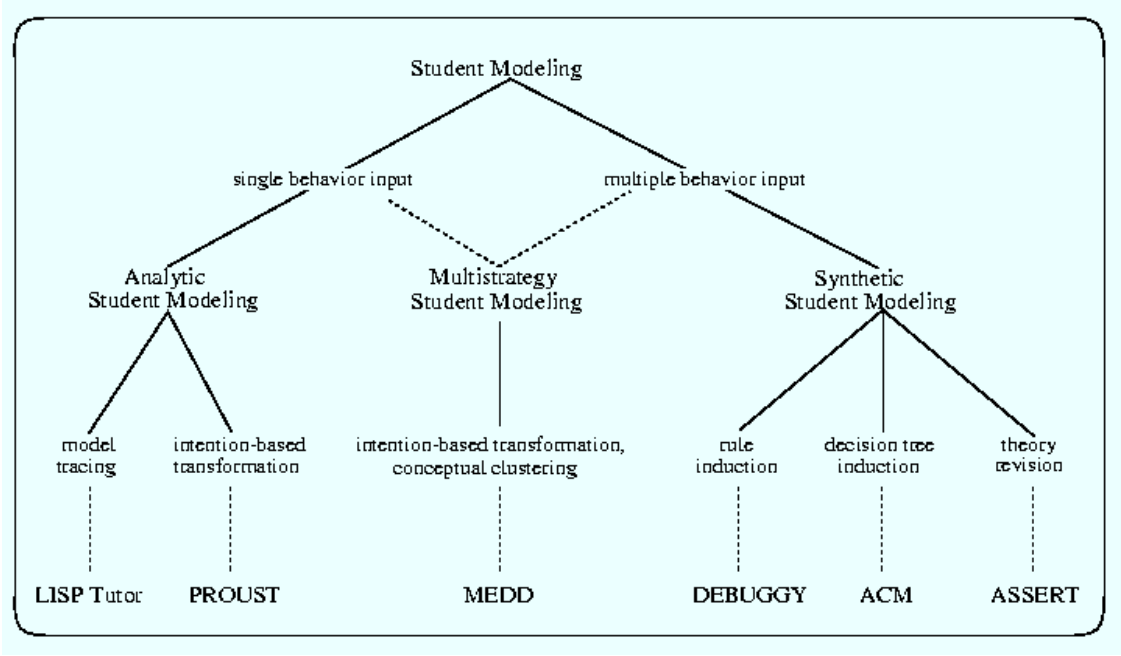

Figure 2. Some approaches to student modelling

On the other hand, systems that deal with relatively complex behaviors such as computer programs (e.g., PROUST, Johnson & Soloway, 1984; Johnson, 1990; the Lisp Tutor, Anderson & Reiser, 1985) can construct a student model from a single behavior, and use transformation operators (and, in the case of the Lisp Tutor, constrain the student to enable *model tracing*, which is a step-by-step analysis of his or her solution) for this. Their ability to construct a student model from a single behavior usually comes, however, of necessity rather than choice, since it is not practical in their domains, specifically, in programming domains, to assign multiple problems to students and only analyze program errors after the solutions have been turned in (Johnson, 1990).

A more recent system, MEDD (Sison, Numao & Shimura, 1998a), also deals with programming. It can, like PROUST, also construct a student model from a single behavior, though, in addition, MEDD also makes use of knowledge automatically acquired from multiple behaviors.

Background Knowledge for Student Modeling

We now turn to the *background knowledge* of a student modeling system. This background knowledge comprises:

- the correct facts, procedures, concepts, principles, schemata and/or strategies of a domain (collectively called the *theory* of that domain); and
- the misconceptions held and other errors made by a population of students in the same domain (collectively called the *bug library*).

The background knowledge may also contain historical knowledge about a particular student (e.g., past qualitative models and quantitative measures of performance, student preferences and idiosyncracies, etc.), and stereotypical knowledge about student populations in the domain.

Background Knowledge Construction.

While it is sometimes possible for the theory of a domain to be completely specified (as it is possible, for example, to enumerate all the correct rules for arithmetic or high school algebra), it is difficult, if not impossible, to enumerate all the misconceptions and other errors that students may possibly have, even when one only considers those errors that students generally tend to make. That is to say, it is generally impossible to have a complete bug library. And even if it were possible to have, at the start, a bug library that contained at least the most common errors of a group of students, the results of (Payne & Squibb, 1990) suggest that different groups or populations of students (e.g., students from different schools) may need different bug libraries.

In spite (or because) of the above difficulties, very few systems have the capability to automatically extend, let alone construct, their bug libraries. Specifically, only PIXIE (Sleeman, Hirsh, Ellery & Kim, 1990), (Hoppe, 1994), ASSERT, and MEDD can extend their background knowledge automatically. These systems will be examined later in this paper.

Domain Complexity.

We have sometimes alluded to differences in the complexities of domains and domain tasks. We now informally describe this notion of domain complexity:

- Following Gagne's (1985) hierarchy of intellectual skills, we can view problem solving tasks as more complex than concept learning or classification tasks in the sense that problem solving ability requires classification ability. Thus, we say that PIXIE, for example, which models students solving linear equations, deals with a more complex domain, or more specifically, a more complex domain task than, say ASSERT, which models students' classification of programs represented as feature vectors.
- Moreover, following Newell and Simon's (1972) problem space theory of human problem solving, we can regard some problem solving domains or tasks as more complex than others, depending on the complexity of state representations, the well-definedness of the operators, and the nature of the search process (algorithmic vs. heuristic), among others. Complexity in problem solving domains can therefore be viewed as a spectrum, possibly multidimensional, with mathematics (e.g., DEBUGGY, ACM, and PIXIE's domains) on one end; medical (e.g., GUIDON's domain, Clancey, 1982) and cognitive diagnosis (e.g., student modeling) on the other end; and programming (e.g., PROUST, the Lisp Tutor, and MEDD's domains) and electronic diagnosis (e.g., the SOPHIE programs' domain, Brown, Burton & de Kleer, 1982) somewhere in between (Clancey, 1986).

The Student Model

Finally we turn to the third element of student modeling, which is the output of the student modeling process itself, i.e., the *student model*. A student model is an approximate, possibly

partial, primarily qualitative representation of student knowledge[1] about a particular domain, or a particular topic or skill in that domain, that can fully or partially account for specific aspects of student behavior.

That student models are qualitative models means that they are neither numeric nor physical; rather, they describe objects and processes in terms of spatial, temporal, or causal relations (Clancey, 1986). That student models are approximate, possibly partial, and do not have to fully account for all aspects of student behavior means that we are interested in computational utility rather than in cognitive fidelity (Self, 1990). A more accurate or complete student model is not necessarily better, since the computational effort needed to improve accuracy or completeness may not be justified by the extra if slight pedagogical leverage obtained (Self, 1994).

By 'accounting for' behavior, we mean identifying specific relationships between the input behavior and the system's background knowledge. Such relationships can of course be analyzed at various levels.

Levels of Error Analysis

At the most basic level, which we call the *behavior level*, relationships, particularly *discrepancies* (i.e., syntactic mismatches), between actual and desired behaviors are determined. We call these discrepancies *behavioral* or *behavior-level errors* when they occur in incorrect behavior. While the detection of behavior-level errors is trivial when dealing with simple behaviors like integers, it quickly becomes a nontrivial task when one begins to deal with more complex behaviors like programs, in which correct stylistic variations of reference programs are not uncommon.

At a higher level, which we call the *knowledge level*, the relationships, particularly causal relationships, between behavioral discrepancies become significant. It is at this level that misconceptions and other classes of knowledge errors are recognized. *Misconceptions* are incorrect or inconsistent facts, procedures, concepts, principles, schemata or strategies that result in behavioral errors. Not every error in behavior is a result of incorrect or inconsistent knowledge, however, as behavioral errors can also be due to insufficient knowledge. We use the term *knowledge error* to include both incorrect or inconsistent knowledge (i.e., misconceptions) and missing or incomplete knowledge. Moreover, it will be noted that what could seem to have been caused by some knowledge error may actually be a slip (Corder, 1967) due, for example, to fatigue, boredom, distraction, or depressed affect. We call the errors at this level, both knowledge errors and slips, *knowledge-level errors*.

At an even higher level, relationships between misconceptions and corresponding or analogous correct pieces or chunks of knowledge become important, as they might indicate the former's origins. For example, a misconception can be traced to an overgeneralization of an existing piece of knowledge. It is at this level that architectures of cognition such as ACT*/ACT-R (Anderson, 1983; 1993) and SOAR (Laird et al., 1986; 1987; Newell, 1989), or theories of (mis)learning such as the REPAIR and STEP Theories (Brown & VanLehn, 1980; VanLehn, 1983; 1987) become especially helpful. We call this level the *learning level*.

Note that each level can be viewed as explaining or generalizing the level below it; that is, knowledge-level errors explain the occurrence of errors at the superficial behavior level, while learning-level errors explain the occurrence of misconceptions and other errors at the knowledge level. Higher-level knowledge is therefore more efficient to store, though more expensive to acquire. Student modeling systems are mainly concerned with errors at the knowledge level.

Our use of the term knowledge level is somewhat related to, but not the same as, Newell's (1981), Dietterich's (1986), and Anderson's (1989) notions. Newell used the term to distinguish between descriptions of computer systems, the other levels being the program, register-transfer, circuit and electronic levels. Dietterich used it to distinguish between machine learning systems: knowledge level learners acquire new knowledge, while *symbol level* learners yield

[1] We could have used 'belief' instead of 'knowledge'. We use the term 'knowledge' because of our use of the term 'knowledge level', to be explained shortly.

improvement in computational performance without the addition of new knowledge. Anderson (1989) inserted an *algorithm level* between Dietterich's two levels. Our use of the term knowledge level, on the other hand, indicates the explanatory capability of errors at this level for errors at the behavior level. Our use of the term also indicates that it might not (always) be possible or useful to distinguish between knowledge errors and slips (see e.g., Payne & Squibb, 1990).

Student Model Construction

There are of course a myriad of detailed ways of computing the above relationships, but three general approaches can be discerned. The most basic approach is called the *overlay* approach (Carr & Goldstein, 1977), in which the student model is assumed to be a subset of the expert model. Most early systems (e.g., SCHOLAR, Carbonell, 1970; the SOPHIE programs; WEST, Burton & Brown, 1979; WUSOR, Goldstein, 1982) used this approach.

The student-model-as-subset-of-expert-model assumption of the overlay approach, however, necessarily precludes the diagnosis of misconceptions, i.e., incorrect (as opposed to missing) knowledge. The other two approaches to computing the student model can be viewed as attempts to deal with this limitation. The first of these two uses the background knowledge to transform the student behavior to the problem given (or to the desired behavior), or vice versa, or to verify if the student behavior and the problem given (or the desired behavior) are equivalent or not. The specific operators needed to perform the transformation make up a student model. We call this approach the *analytic* or *transformational* approach. The other approach involves obtaining a set of behaviors and computing a generalization of these by, for example, synthesizing elements from the background knowledge or input data. The synthesized generalization makes up a student model. We call this approach the *synthetic* approach.

Normally, systems that need to be able to construct their student models from a single behavior (e.g., PROUST, Lisp Tutor) adopt an analytic approach, while systems that construct their student models from multiple behaviors (e.g., DEBUGGY, ACM) adopt a primarily synthetic approach (recall Figure 2). ASSERT's technique, which is called theory revision, can be viewed as transformational, though it involves the transformation of a model (rather than a behavior). Moreover, its transformation is guided by the data, i.e., by a set of behaviors, rather than the background knowledge. In the sense that ASSERT's procedure relies on multiple behaviors, it is closer to the synthetic approaches. MEDD is both analytic and synthetic. With the exception of PROUST and the Lisp Tutor, all of the systems just mentioned induce their student models using machine learning techniques, and will be examined later in this paper.

Summary of Student Modeling Elements and Issues

We have so far defined the essential elements of student modeling and identified several important student modeling issues. To summarize, student behaviors, i.e., the observable responses of students (to domain stimuli) that are used, together with the stimuli, as the primary input to student modeling, can be simple or complex, and student models can be constructed from single or multiple behaviors. The background knowledge used to infer student models from behaviors includes a domain theory and a bug library. These are usually built manually; ideally, they should be automatically extensible, if not automatically constructible from scratch.

A student model is an approximate, possibly partial, primarily qualitative representation of student knowledge about a particular domain that fully or partially accounts for specific aspects of student behavior. Accounting for behavior involves identifying relationships between the behavior and the background knowledge, and can be done at the behavior, knowledge, and/or learning levels. Constructing a student model which can deal with misconceptions as well as other knowledge-level errors can be achieved using either an analytic or a synthetic approach, or a combination of the two.

One can of course think of other issues regarding intelligent tutoring or coaching, particularly issues related to pedagogical activities, but those that have been mentioned suffice for this paper. For other intelligent tutoring issues, the reader is referred to (Wenger, 1987),

which provides as well a comprehensive description of the major tutoring systems of the 1980's. We now turn to this paper's second concern, namely, machine learning.

MACHINE LEARNING

Learning is the induction of new, or the compilation of existing, knowledge, which, in turn, may lead to improvements in the performance of a task. Most machine learning research has focused on improving accuracy (and efficiency) in classification tasks, but research has also been done on improving efficiency (and accuracy) in problem solving. In this section, we summarize the main approaches to machine learning (Figure 3), and identify the main ways by which machine learning techniques have been used in student modeling.

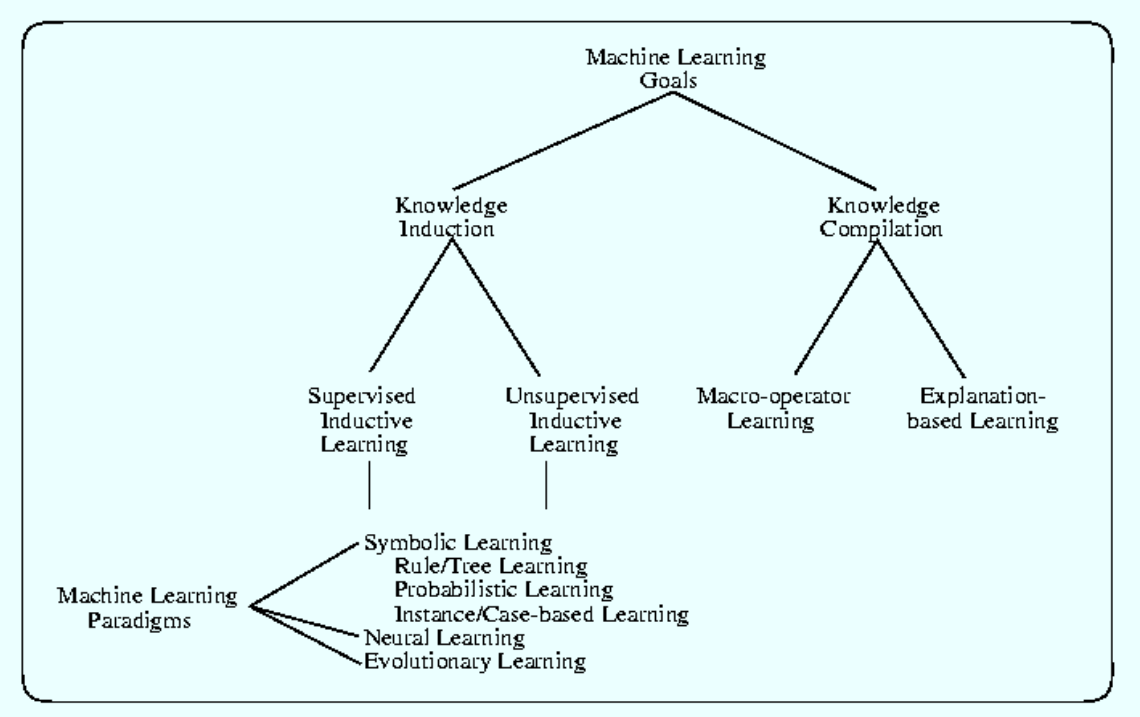

Figure 3. A classification of machine learning techniques

Supervised Knowledge Induction

The construction of new knowledge has been called *inductive* or *empirical* learning, because it relies heavily on data, i.e., specific experiences or objects, to produce hypotheses that generalize the data. The hypotheses produced in this manner are therefore implicitly attended by varying degrees of uncertainty.

The objects that inductive learners generalize from can be *labeled* or not. Since the label can be thought of as being predetermined and provided to the learning system by an entity (the 'supervisor'), learning from labeled objects has been called *supervised* learning. It has also been called *learning from examples*, since a labeled object is an example or instance of what the label represents. Since most labels represent classifications, which in turn can be viewed as implying concepts, learning from labeled instances has also been called *concept acquisition*.

A labeled instance can be viewed as a pair $(x; f(x))$; where x is the instance itself and the function $f(x)$ returns its label. The goal of supervised inductive learning is therefore to compute a function f' that approximates f, which, in turn, defines the *target* concept. The function f' can be represented in a variety of ways, e.g., using if-then rules, decision trees, neural networks, or genetic patterns. Moreover, f' can be learned *incrementally* so that the learner does not have to reprocess (and therefore remember) all the instances it has already examined whenever a new instance arrives; alternatively, f' can be learned *nonincrementally* so that it has (and in fact, must

have) access to all or most of the instances, which it can then use to compute statistics that can assist in hypothesis construction.

Symbolic Learning

Although there may be a variety of ways of learning *symbolic* descriptions[2], a couple of general approaches can be discerned. One approach is typically incremental, and involves taking an instance from a *training* set and revising the current hypothesis or hypothesis set(s) so that it is consistent with this instance. This process normally continues until all instances in the training set have been processed. We call this approach *successive revision*. Early systems that used this approach are Winston's (1975) arch learning program, and Mitchell's (1977) candidate-elimination algorithm, which involves the revision of two (rather than just one) hypothesis sets, called a *version space*, in which one set contains maximally specific hypotheses while the other contains hypotheses that are maximally general. The other approach is typically nonincremental, and involves examining the training set, selecting a sub-set of instances from the training set, and revising a hypothesis or hypothesis set so that it *covers* (i.e., is satisfied by) this subset of instances. This process normally continues until no more instances remain to be covered. We call this approach *iterative covering*, an early example of which is AQ (Michalski, 1983). Whatever the approach, *biases* (Mitchell, 1980) are needed to constrain the *form* of (e.g., conjunctive form, conjunctive normal form, disjunctive normal form etc.) and the *search* for (using, e.g., variants of Occam's razor) a hypothesis that correctly classifies most if not all the examples in the training set. Moreover, all empirical learners, which rely heavily on data, need to have mechanisms such as stopping criteria, pruning, or cross-validation for dealing with the problem of *noise* (e.g., wrong or missing attribute values or class information) in the data.

Much of the earlier work in machine learning dealt with *propositional* or *zeroth order* logic expressions of Boolean *features* (e.g., red-color, round-shape) or *attribute-value* pairs (e.g., color = red $\wedge$ shape = round). While rules made up of such features or attribute-value pairs are straightforward to work with, *decision trees*, which are fully expressive within the class of propositional languages, can sometimes afford a more compact representation. The nonleaf nodes of a decision tree denote attributes, edges denote attribute values, and leaves denote classifications. ID3 (Quinlan, 1986) and CART (Breiman, Friedman, Olshen & Stone, 1984) are the most in uential decision tree induction algorithms, the latter being particularly important in the statistics community. Both are nonincremental and can be viewed as using variants of the basic covering scheme.

While propositional representations are useful in many domains, their expressive power is limited to simple, unstructured objects. For structured objects or concepts, relations among components will have to be described using some relational representation. Systems that learn such relational concept descriptions, as well as domain theories, in *first order* logic are called *inductive logic programming* (ILP) learners. Incremental ILP systems such as MIS (Shapiro, 1983) use the successive revision approach. FOIL (Quinlan, 1990), probably the most in uential ILP learner, uses a variant of the covering scheme.

Not all symbolic learners generalize instances in terms of summary conjunctive/disjunctive class descriptions, however. For example, *probabilistic concept* learners such as *naive Bayesian classifiers* (see e.g., Langley, Iba, & Thompson, 1992) represent classes in terms of their prior probabilities and the conditional probabilities of the attributes given the class. *Instance-based* learners (e.g., Kibler & Aha, 1987) and *case-based reasoning* (CBR) systems (e.g., CHEF, Hammond, 1989) represent a class simply as a set of instances or episodes. Learning and classification in the former, i.e., in probabilistic concept learners, involves adjusting probabilities; in the latter, i.e., in instance/case-based learners, it involves learning prototypes and indexing mechanisms.

We also mention at this point *analogy* (e.g., Gentner, 1989; Carbonell, 1986). Both analogical and case-based reasoning involve, first, the retrieval of a similar or useful episode in

[2] We use the term 'symbolic' to refer to the traditional AI representation schemes, i.e. rules, logic, semantic nets, and frames, which are directly understandable by humans.

memory, followed by the mapping of elements of the retrieved episode into those in the episode being learned or solved, and finally the adjustment of the analogical inferences made in the previous step to better suit the current situation. Much of the research in analogy has focused on the mapping/inferencing step, while much work in CBR has focused on the retrieval/adjustment step.

Neural and Evolutionary Learning

An alternative to symbolic propositions and logic programs are *neural* or *connectionist* networks, which are composed of nodes called units connected by weighted, directed links. Each unit has a current activation level, a set of input links from other units, functions for computing the unit's next activation level given its inputs and their weights, and a set of output links. Inputs to a network typically come in the form of a vector of Boolean features. The output of a network is the activation of the designated output unit(s).

In most *feedforward networks* (i.e., networks with only unidirectional links and no cycles), whether single-layer or multilayer (i.e., with hidder layers between the input and output layers of units), learning is accomplished by making small adjustments in the weights of the links using some rule (e.g., the perceptron rule, Rosenblatt, 1958; the delta rule, Rumelhart, Hinton & Williams, 1986), to reduce the error/mean squared error between the observed and predicted output values, thereby making the network consistent with the examples. Of course, the procedure is easier for single-layer networks; for multilayer networks, the error would typically have to be *backpropagated* from the output layer to the first hidden layer. Nonetheless, learning in both cases can be viewed as a variant of the successive revision approach described earlier, but in which the hypothesis being revised is the network itself (or more specifically, the weights of the links in the network) and in which a revision step or 'epoch' may actually take hundreds of weight revisions. This and other connectionist learning algorithms are reviewed in (Hinton, 1989).

Rather than using a weighted network to represent acquired knowledge, *genetic* or *evolutionary* algorithms use a set of 'genetic patterns'. Each genetic pattern denotes an individual and is usually represented as a string of bits. Moreover, each pattern has an associated fitness value that summarizes its past performance. Learning using genetic algorithms involves updating the fitness values of the individuals, performing operations such as crossover (gene splicing) and mutation on the fitter individuals to produce new ones, and using these new individuals to replace those which are less fit. Genetic algorithms are a primary learning component of so called classifier systems (Booker, Goldberg & Holland, 1989).

We also mention at this point *Bayesian* or *probabilistic networks* (Pearl, 1988), which are probabilistic directed acyclic graphs used for reasoning under uncertainty. By ensuring that each node (which denotes a random variable) in the graph is conditionally independent of its predecessors given its parents, Bayesian networks allow a more convenient representation and manipulation of the entire joint probability distribution of a domain. Specifically, one need only specify the prior probabilities of the root nodes and the conditional probabilities of nonroot nodes given their immediate predecessors. This also allows us to view a link in the graph as representing direct dependency or causality. When the network structure is known and all variables are observable, learning in Bayesian networks reduces to estimating, from statistics of the data, the conditional probabilities of the networks' links. However, when not all variables can be readily observed, learning becomes strikingly similar to that which occurs in feedforward neural networks (see e.g., Neal, 1991).

Unsupervised Knowledge Induction

Most machine learning systems learn from labeled instances. It is, however, also possible, albeit more difficult, to learn from unlabeled objects. Learning from unlabeled objects is called *unsupervised learning*, since no classification information at all is provided by any supervisor. Compared to supervised learning, unsupervised learning techniques are less understood. This is not surprising since, unlike supervised learning, the goals and success criteria of unsupervised

learning are not as easy to define. In particular, there is no given function f to approximate and measure against.

The main machine learning approach to the problem of generalizing unlabeled instances is *conceptual clustering* (Michalski & Stepp, 1983b), which is basically the grouping of unlabeled objects into categories for which conceptual descriptions are formed. Conceptual clustering differs from numerical taxonomy in the sense that clusters formed using the former have simple conceptual interpretations, as expressed in their conceptual descriptions, which are more meaningful to humans (Michalski & Stepp, 1983a). Though the most influential conceptual clusterers (e.g., UNIMEM, Lebowitz, 1987; COBWEB, Fisher, 1987) are symbolic (COBWEB having probabilistic concepts), unsupervised neural learners also exist (e.g., Rumelhart & Zipser, 1985; Kohonen, 1982). Moreover, like their supervised counterparts, conceptual clusterers can also be incremental (e.g., UNIMEM, COBWEB) or nonincremental (e.g., AUTOCLASS, Cheeseman, 1988). The unsupervised learning of concepts via incremental, hierarchical conceptual clustering has been called *concept formation* (Gennari, Langley & Fisher, 1989). Other approaches to unsupervised learning (e.g., extracting algebraic laws empirically from tables of numeric and symbolic data) usually fall under the banner of *scientific* or *empirical discovery* (Langley & Zytkow, 1989).

Knowledge Compilation

The second kind of learning that can be accomplished by programs (recall Figure 3) involves the modification of existing knowledge such that what is learned remains within the knowledge base's deductive closure; thus, this kind of learning is sometimes called *deductive learning*. In contrast, knowledge induction involves the addition of knowledge that cannot be inferred from the original knowledge base. The modification of knowledge is usually performed with the view of improving performance in terms of efficiency; thus, this type of learning has also been called *speedup learning* or *compilation*. Finally, this kind of learning has also been called *knowledge-intensive* or *analytic* in the sense that it involves an examination of existing knowledge in order to improve the knowledge. In contrast, inductive/empirical approaches rely less on the background knowledge and more on the training data. Thus, whereas empirical approaches need to deal with the problem of *noisy data*, analytic approaches have to deal with the problem of what we call *noisy knowledge* (i.e., inaccurate or incomplete knowledge).

Macro-operator Learning.

Arguably the earliest form of knowledge compilation involved the construction of *macro-operators*, which can be viewed as abbreviations of sequences of simpler operators. In STRIPS (Fikes, Hart & Nilsson, 1972), for example, the sequence of operators used to solve a novel problem are analyzed and compiled into one general macro-operator, which can then reduce the length of a solution (but which can also increase the branching factor). The *composition* and *chunking* mechanisms of the ACT* (Anderson, 1983) and SOAR (Laird et al., 1987) architectures, respectively, can be viewed as variants of macro-operator learning.

Explanation-based Learning

Of course, decreasing the length of a solution is not the only way to speed up performance. Alternatively, one can narrow down the choice of good operators at each step. That is, one can decrease the branching factor, by learning *search control* heuristics in the form of operator selection/rejection rules. This can be done using a technique called *explanation-based generalization* (Mitchell, Keller & Kedar-Cabelli, 1986) or *explanation-based learning* (DeJong & Mooney, 1986), which involves recasting the deductive definition or proof of an operator (or concept) in *operational terms*, using, for example, easily computable features of the problem state (or predicates found in the example description). Explanation-based generalization is accomplished by first constructing an explanation, from the background knowledge, of why a particular operator is useful (or why an example is an instance of some concept). This

explanation is then generalized to obtain a set of sufficient conditions for the operator (or concept). The resulting rule (or concept definition) can be viewed as a generalized 'macro-explanation' that is at the same time a specialization of the high-level definition used in the explanation. This compiled rule can then be subsequently used to select the operator (or recognize the concept) more readily and efficiently. The LEX2 integral calculus problem solver (Mitchell, 1983) does exactly this. ACT*'s *proceduralization* mechanism can also be viewed as a kind of explanation-based learning.

Of course, the learning of operator selection rules can also be done inductively, as in the case of ACM which we will return to later, or as in the case of *reinforcement learning*, whose objective can be viewed as that of learning which operator or action leads to a 'reward.' One approach to reinforcement learning is to learn a function that assigns an *expected utility* value to taking a given action in a given state (Watkins & Dayan, 1992). This can be achieved, for example, by having a table of states and actions, with expected utility values for each state-action pair, and modifying the utility values using an equation that considers the utilities of an action's successors as well as its own. One can also further increase the learning rate by replacing the table with an implicit representation, which in turn can be learned inductively using, say decision-tree induction (see e.g., Chapman & Kaelbling, 1991).

Summary of Machine Learning Approaches

We have identified the main approaches and paradigms of machine learning and brie y sketched each. To summarize, machine learning algorithms can be classified as to whether they induce new knowledge or compile existing knowledge. Knowledge induction can be supervised, with the target concepts known and provided a priori, or unsupervised; and incremental, so that previously processed instances need not be reprocessed whenever a new instance arrives, or nonincremental. Moreover, knowledge induction can be achieved using propositional/relational formulas, neural networks, or genetic patterns. In contrast, almost all work on knowledge compilation, including macro-operator and explanation-based learning, deal with symbolic (propositional/relational) descriptions. One can of course have hybrid or multistrategy learning systems that, for example, combine knowledge induction and knowledge compilation, or combine two or more paradigms (e.g., symbolic and neural; neural and genetic, and so on).

The machine learning algorithms and systems we have identified above are only a small sampling of the vast number of learning algorithms out there. However, we believe that those we have mentioned are both representative of the field and sufficient for the purposes of this paper. For other machine learning systems, techniques, and issues, the reader is referred to (Shavlik & Dietterich, 1990; Langley, 1996; Mitchell, 1997).

Utility of Machine Learning in Student Modeling

Learning Student Modeling Systems

From our discussion of machine learning approaches and student modeling issues, it quickly becomes clear how student modeling can benefit from machine learning. Recall that machine learning is the induction or compilation of knowledge, normally leading to improvements in the ability of an agent to classify objects or solve problems in its domain. Thus, a *learning student modeling* system is one that could extend or compile its background knowledge, particularly its bug library, so that sub-sequent modeling would be more accurate or more efficient, or even possible where earlier it was not. PIXIE (Sleeman et al., 1990), Hoppe's (1994) system, ASSERT (Baffes & Mooney, 1996), and MEDD (Sison et al., 1998a) are examples of such learning student modeling systems.

Student Model Induction from Multiple Behaviors

There is, however, another way by which techniques developed in the machine learning community could be used in student modeling. Recall that student modelers that deal with

simple behaviors can usually afford to collect, from a student, a set of behaviors from which to induce a student model. This task of constructing the student model from multiple behaviors can be regarded as an inductive learning task, and supervised symbolic inductive learning techniques or systems can therefore be used to address this task. Thus, ASSERT uses a propositional *theory revision* system (i.e., a symbolic inductive learner that does not induce a theory from scratch, but is provided an almost correct theory at the outset), THEMIS (Kono et al., 1994) uses an MIS-based incremental ILP learner, and ACM (Langley & Ohlsson, 1984) uses an ID3-like decision tree induction algorithm.

Constructing a student model is, however, far more complex than constructing a concept description in the sense that student behaviors are likely to be *noisier* compared to the examples one can obtain for a 'normal' concept. In the context of student modeling, noise in the data (i.e., in the behavior set) would mean inconsistent or incomplete behaviors, which, in turn, can be due to any of the following:

- slips;
- the (quick) eradication of a knowledge error;
- the recurrence of old knowledge errors; or
- the sudden appearance of new knowledge errors.

Except for slips, all of the above can be considered as forms of what is known in the machine learning literature as *concept drift* (Schlimmer & Granger, 1986), in which the target concept itself may change. Note, however, that the degree of concept drift that we can expect in the student modeling domain is considerably higher than what most machine learning systems normally deal with. For example, student errors can disappear, appear, or reappear any moment before, during, or after remediation. In addition, the way that a student may compensate for his or her incomplete knowledge may vary from one problem to another: the so called *bug migration* phenomenon (VanLehn, 1987). Finally, and most importantly, the eradication of a student's knowledge error is actually one of the main functions of an intelligent tutoring system. All these indicate that concept drift is not an occasional, but rather a regular, phenomenon in the student modeling domain. That is, concept drift is not the exception in student modeling, but the norm. Thus, while it is tempting to simply use supervised inductive learning systems or techniques to induce student models from multiple behaviors, one must be careful to note that the kind and degree of noise in student modeling might be too much for existing inductive learning techniques to handle.

With this at the back of our minds, we now turn to examine how machine learning and machine learning-like techniques have been used to automate student modeling processes. Basically, there seems to be three ways:

1. *Supervised inductive* learning techniques have been used or developed to induce student models from multiple behaviors (DEBUGGY, ACM, THEMIS, ASSERT);
2. *Supervised deductive* learning techniques have been used to extend bug libraries (PIXIE; Hoppe, 1994); and
3. *Unsupervised inductive* techniques have been developed to construct and extend bug libraries while, at the same time, inducing partial student models (MEDD).

MACHINE LEARNING AND MULTIPLE-BEHAVIOR STUDENT MODEL INDUCTION

We now look at systems that use machine learning or machine learning-like techniques to construct student models from multiple behaviors. Systems that deal with simple behaviors such as integers or propositions quite necessarily have to rely on multiple behaviors to induce a fairly useful student model. In such systems, the major issue is the complete and consistent covering of the examples in the behavior set so that the student model is neither overly general

(incorrectly covering certain behaviors) nor overly specific (missing certain behaviors). Since this also happens to be a main concern of supervised machine learning research, supervised machine learning techniques such as those for the empirical induction of rules or decision trees have proved quite useful.

DEBUGGY

DEBUGGY (Burton, 1982) is a diagnostic system that extends an earlier BUGGY model (Burton & Brown, 1978), in which a skill such as place-value subtraction (BUGGY/DEBUGGY's domain) is represented as a *procedural network*. A procedural network is a recursive decomposition of a skill into subskills/subprocedures. A student modeling scheme based on such a network requires that the background knowledge contain all the necessary subskills for the general skill, as well as all the possible incorrect variants of each subskill. A student modeling system can then replace one or more subskills in the procedural network by one of their respective incorrect variants, in an attempt to reproduce a student's incorrect behavior.

1. From a set of (130) predefined buggy operators, select those that explain at least one wrong answer in the student's behavior set, B. Call this initial hypothesis set *H*.
2. Reduce *H* giving *H'* by removing buggy operators that are subsumed by others.
3. *Compound* every pair of buggy operator in *H'* to see if the resulting bug covers more answers than either of its constituents. If so, add this compound to *H'*. Repeat this step for each of the new compounds, giving *H"*.
4. From *H"*, select a set of bugs, *P*, that explain a given percentage of the student's answers. 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 one with the highest score.

Figure 4. Basic procedure of DEBUGGY

The incorrect variants of skills/subskills are called *bugs* in the BUGGY framework. Note that a bug a la BUGGY denotes a knowledge error (e.g., a misconception), rather than a behavioral one. Thus, it is not to be confused with *programming bugs*, which refer to errors in a program (Joni, Soloway, Goldman & Ehrlich, 1983; Eisenstadt, 1997), which, in the context of student modeling, are errors at the behavior, rather than at the knowledge, level. Note, too, that in the terminology of (Burton, 1982), the term *bug* is sometimes used synonymously with what we would call a *buggy model*, i.e., the procedural network that results when the correct subprocedure that corresponds to a particular bug is replaced with that bug. This synonymous treatment is due to the fact that bugs, rather than buggy models, are what DEBUGGY manipulates explicitly; a buggy modelcan always be generated at any time for any bug. We retain this synonymous treatment of the terms bug and buggy model in Figure 4 for simplicity.

A student model in DEBUGGY's framework is therefore an individualized procedural network. To construct the student model, a naive diagnostic system could simply generate a set of buggy models, one model for each primitive bug in the bug library, and select the model that reproduces all or most of a student's answers to a set of problems. To deal with the explosive search space when *compound* (i.e., multiple) bugs are responsible for the student's answers, DEBUGGY assumes that the primitive bugs that interact to form a compound bug are

individually detectable, so that there is at least one problem in which they appear in isolation. Thus, to determine a student's compound bug (and, in the process, the student's buggy model) given the student's behavior set, DEBUGGY forms, from 110 predefined primitive bugs and 20 compound bugs, an initial hypothesis set of bugs (Figure 4, step 1), whose elements it then removes, compounds, or coerces (Figure 4, steps 2, 3, and 4 respectively). The elements of the final set are then ranked, and the bug with the highest score is outputted (Figure 4, step 5).

From the procedure outlined in Figure 4, we can see that DEBUGGY actually performs a kind of supervised inductive learning of a student model, or more specifically, of a compound bug, that uses a multipass variant of the covering approach, in which the hypothesis set is examined and refined, and the entire data set is examined, several times.

ACM

Unlike DEBUGGY, ACM (Langley & Ohlsson, 1984) has no need for a library of predefined buggy subtraction rules, since it tries to induce them instead.

ACM adopts a *production system* (Newell & Simon, 1972) framework within the *problem space* paradigm (Newell, 1980). Thus, a domain is defined in terms of problem *states* and *operators* (condition-action rules) for state transition, and problem solving is viewed as the successive application of rules so that a *goal state* is reached from an *initial state*. Six primitive rules (i.e., rules with minimal conditions) for subtraction are listed in (Langley et al., 1987), where they also mention ten tests that can be used as conditions of a rule. The sequence of rules needed to transform the start state to the goal state constitutes a *solution path*. A student model in ACM's framework consists therefore of a set of production rules.

1. For every answer in the student's behavior set, *B*, contruct a partial problem space using the answer as goal, the problem as start state, and the (6) primitive subtraction operators as state 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 positive and negative instances (of operator application). Each instance denotes a particular condition (e.g., *number*1 > *number*2) when a particular operator was used.
2. Construct a decision tree for each operator, whose nonleaf nodes are conditions (there are 10 possible conditions, represented as binary tests), edges denote whether a condition holds or not, and leaves denote classifications (i.e., positive or negative). Conditions are chosen according to some measure of discriminating power, and added to the decision tree until all or almost all of the instances of operator application are covered correctly and consistently.
3. For each decision tree, eliminate all branches whose leaves are negative instances, and then collapse the remaining subtree into an if-then rule (or a disjunction of rules, in the case where many leaves remain in a pruned tree).

Figure 5. Basic procedure of ACM

The applicability of each of the rules that make up a student model is determined by inferring the conditions when the student used the primitive form of the rule. To identify these conditions, ACM first constructs a partial problem space for each of the student's answers to a set of problems, with the student's answers serving as goal states, and then classifies every instance of every operator's application as positive or negative, depending on whether the operator appears on a solution path or not (Figure 5, step 1). This procedure is called *path finding*. Since this classification reflects the student's correct and incorrect notions of the specific conditions in which each of the primitive operators are applicable, learning to make these classifications is tantamount to learning the student model. To learn these classifications, ACM builds a decision tree (Figure 5, step 2) for each of the operators following an earlier

version of Quinlan's (1986) ID3 algorithm. It then collapses the trees into rules (Figure 5, step 3).

Like DEBUGGY, ACM uses supervised inductive learning (in step 2 above) to infer a student model. However, since ACM does not have a notion of bugs, it has to construct these bugs from scratch, hence the analytic path finding in step 1. To speed up ACM's original path finding procedure, its authors later proposed several psychologically plausible heuristics, resulting in their Diagnostic Path Finder (Ohlsson & Langley, 1988). Regarding step 2, Quinlan (1986) notes that the choice of the test is crucial if the decision tree is to be simple, and, in ID3, uses an information-theoretic measure to help guarantee this. ACM's measure,

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

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,

is less powerful than ID3's entropic measure, although Wenger (1987, p. 216) mentions that later versions of ACM have replaced the equation above with statistical (χ^2) measures.

THEMIS

Whereas ACM uses a nonincremental propositional learner (that is a variant of Quinlan's ID3) to construct a student model, THEMIS (Kono et al., 1994) uses an incremental first order learning algorithm (SMIS, Ikeda, Mizoguchi & Kakusho, 1989) that is a variant of Shapiro's MIS.

1. Initialize the student model to the ideal model (whenever possible).
2. Repeat the following until no more inconsistencies are found.
   (a) Ask the student a question usually answerable by yes, no, or don't know. This is a 'fact' that the student believes. If the student's answer is correct with respect to the student model, do nothing; otherwise identify the fault in the model using two of Shapiro's (1983) diagnostic techniques (for missing and incorrect rules), and revise the model to eliminate the fault (insert the missing rule; delete the incorrect rule).
   (b) Use an ATMS (de Kleer, 1986) and a set of heuristics to maintain consistency among all rules and facts.

Figure 6. Basic procedures of THEMIS

In THEMIS, a student model is a consistent set of rules and facts in predicate logic. To build a student model, the system asks the student a series of questions on, say geography, and revises the student model each time the student responds, carrying out revisions so that the model is consistent with the student's answers (Figure 6, step 2a). In order to reduce the number of questions to ask a student, THEMIS makes reasonable assumptions about the student's knowledge or beliefs. For example, the system starts with the ideal student model (Figure 6, step 1), whenever this is reasonable to assume. It might turn out later, however, that earlier assumptions were wrong; thus, THEMIS uses de Kleer's ATMS (Assumption-based Truth Maintenance System, 1986) to maintain consistency among the rules and facts (Figure 6, step 2b).

Inconsistencies detected within the model can, however, be used by a Socratic tutor (e.g., WHY, Stevens & Collins, 1977). Thus, THEMIS distinguishes among several kinds of inconsistencies, ranging from inconsistencies in the data due to noise (e.g., slips, eradication of knowledge errors) to inconsistencies in the model due to wrong assumptions on the part of the modeler, or incorrect knowledge on the part of the student. Modeling the appearance and disappearance (the latter hopefully as a result of tutoring) of the latter kind of inconsistency, though undoubtedly ideal, greatly increases the complexity of the student modeling process.

(The abstracted procedure of THEMIS outlined above belies the complexity of the system.) Other approaches that use truth maintenance systems to model inconsistencies are found in (Huang et al., 1991), (Murray, 1991), and (Giangrandi & Tasso, 1995).

ASSERT

Unlike DEBUGGY and ACM, which synthesize student models from primitives, ASSERT (Baffes & Mooney, 1996) immediately starts with a model, albeit a correct model (Figure 7, step 1), which it then transforms to one that covers the items in a student's behavior set (Figure 7, step 2). Thus, unlike DEBUGGY and ACM's approaches which are synthetic, this approach is transformational. Yet, like DEBUGGY and ACM, this approach is inductive (since the transformation is nevertheless guided by the data). This transformational yet inductive approach to supervised learning is called *theory revision* (Ginsberg, 1989) or *theory refinement*. Incidentally, by starting with an incorrect theory, MIS (and therefore THEMIS) can also be viewed as carrying out a coarser-grained kind of theory revision in which revisions to the model involve the replacement of whole clauses, regardless of the correctness of some individual components.

1. Initialize the student model to the ideal model.
2. Revise 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 either covered by the current model when it shouldn't (a false positive), or not covered by the current model when it should (a false negative). Find a revision (delete rule, delete condition) for this falsely covered/uncovered example.
   (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/conditions using an inductive rule learner (e.g., a propositional variant of an ILP learner such as FOIL).

Figure 7. Basic procedure of ASSERT

The reader acquainted with machine learning may already have noticed that ASSERT adopts an interesting variant of the basic theory revision setup. Whereas the basic setup transforms an incorrect theory so that it covers the training examples correctly and consistently, ASSERT transforms a correct theory (i.e., the correct model) so that it covers the student's set of possibly incorrect behavior. Incidentally, THEMIS also begins with a correct model (Figure 6, step 1), though this choice was more pragmatic (less questions to ask the student) rather than paradigmatic. Step 2 of the procedure outlined above is actually carried out by the NEITHER (Baffes & Mooney, 1993) propositional theory revision system.

Each student behavior that ASSERT takes as input comes in the form of a pair, $<f; l>$, where f is a list of attribute-value pairs that describe an instance, and *l* is a *label* denoting what the student believes as the class of the instance. An attribute-value pair, $<a_i ; v_j >$, is composed of a predefined attribute, a_i, and the *value*, v_j, of that attribute for a particular instance. Unlike DEBUGGY and ACM which model students performing problem solving (subtraction) tasks, ASSERT models students performing classification tasks, specifically, the classification of C++ programs (represented as feature vectors) according to the type of program error the programs may have, e.g., constant-not-initialized, constant-assigned-a-value.

Discussion

All of the systems described above have the capability to inductively construct student models that are consistent with a majority of the items in the behavior sets of their students.

Understandably, older systems such as DEBUGGY and ACM are less efficient in accomplishing this than newer ones such as ASSERT. However, the older systems dealt directly with problem solving tasks (specifically, subtraction), whereas ASSERT and THEMIS, for example, deal with concept learning tasks, although one can, with considerable ingenuity and simplifying assumptions, possibly recast problem solving tasks such as subtraction into classification tasks.

The ability to cover the data is, however, not the only concern of empirical learners that work on real-world data (as opposed to artificial data). Dealing with real data means dealing with imperfect, i.e., noisy, data. It will be recalled that noisy data are the norm rather than the exception in student modeling. To avoid *overfitting* of noisy data (i.e., to avoid covering noisy data, which should not be covered in the first place), many machine learning systems either use stopping criteria which allow them to preterminate induction before all examples are covered, or review their output at the end of the induction process in order to delete or otherwise modify certain components according to certain criteria. There still is no general solution to the problem of noise, however, and not all machine learning prototypes deal with this issue.

DEBUGGY addresses noise, particularly noise due to slips, via its coercion operator (Figure 4, step 4). However, slips cannot account for all the noise. As for ACM, Wenger (1987, p. 216) notes that the use of statistical measures when building decision trees in later versions of ACM allow it to tolerate noise, but the extent to which ACM avoids overfitting is not clear. THEMIS uses an ATMS to deal with noise in the form of inconsistencies. ASSERT's NEITHER does not deal explicitly with noise.

Finally, most supervised inductive machine learning systems require fairly complete background knowledge in the form of features and relations. In student modeling, however, the completeness of conditions and operators (correct as well as incorrect) simply cannot be guaranteed, especially in more complex problem solving domains. In supervised inductive machine learning research, this problem is dealt with to some extent by *constructive induction* (Michalski, 1983), which involves the invention of new features or relations by generalizing given or computed examples. DEBUGGY's technique of learning compound buggy operators can be viewed as a form of such. However, in addition to the efficiency issue raised earlier, DEBUGGY does not remember these buggy operators (and therefore has to recompute them every time), and even if it did, a principled method of adding these to the background knowledge is itself a separate major issue (which we shall turn to in the next section). Moreover, it is not clear how this technique can be used in a tenable manner in problem solving domains that are more complex than subtraction.

MACHINE LEARNING AND BACKGROUND KNOWLEDGE EXTENSION

We now turn to systems that use machine learning or machine learning-like techniques for extending or constructing the background knowledge, particularly the bug library.

To acquire new buggy problem solving operators from a *single* incorrect behavior, systems such as PIXIE (Sleeman et al., 1990) and (Hoppe, 1994) have used two machine learning or machine learning-like techniques. The first involves the *specialization* of meta-rules, called *perturbation rules*, that are distinct from the rules of a domain. These meta-rules indicate how domain rules can be modified to fit the data (cf. DEBUGGY's coercion operator); the modified rules are then learned (i.e., stored). For example, in a state space paradigm of problem solving, given a solution path that cannot be completed (i.e., there is a gap between two states in the solution path that cannot be bridged using any existing domain operator), meta-rules can be used to perturb existing operators to complete the solution path.

The second technique used to acquire new buggy problem solving operators from a single incorrect behavior is what we call *morphing*. Morphing involves the generalization of specific differences (or other relationships) between two states or behaviors that should be transformable to each other, but are not. For example, given again a state space paradigm of problem solving and a solution path that cannot be completed, morphing can be used to generate a new operator,

based on the differences between the unconnectable states, to complete the solution path. The left and right branches in Figure 8 illustrate the differences between these two techniques.

Operator specialization and morphing are, however, not the only techniques used to construct or extend bug libraries. More recent systems such as MEDD (Sison et al., 1998a) acquire new buggy operators from *multiple* behaviors using techniques such as conceptual clustering.

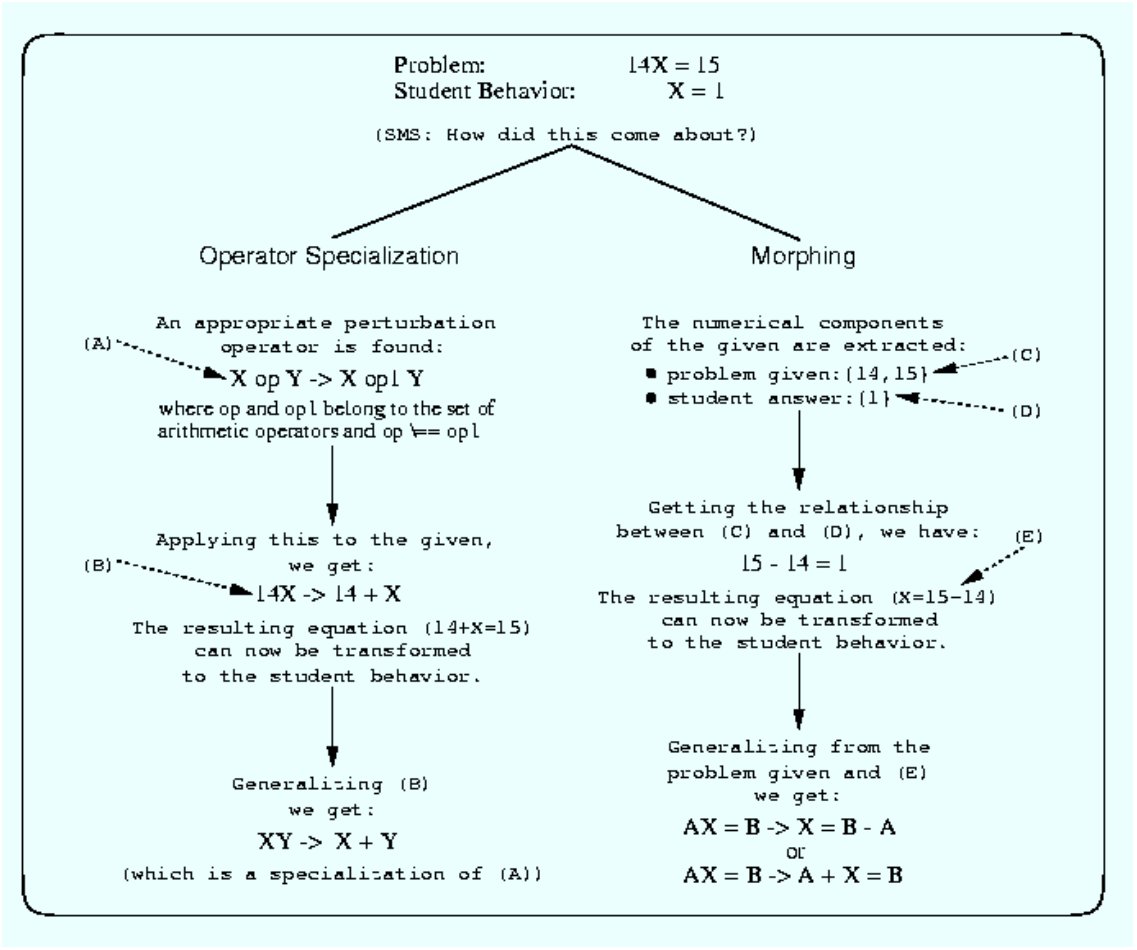

Figure 8. Examples of operator specialization and morphing techniques

PIXIE

Like ACM, LMS/PIXIE (Sleeman, 1982; 1987) assumes the production rule-problem space paradigm of problem solving. Thus, PIXIE views the solution of an algebraic equation (its domain) as the successive application of operators or rules to reach a goal state (the solution of an equation) from the start state (the equation to be solved). In PIXIE, therefore, a student model is an ordered set of rules and malrules, i.e., incorrect rules, that can be used to reproduce a student's correct or incorrect answer. Sleeman (1983) has identified 7 correct production rules that were claimed sufficient to solve basic algebraic equations, and which form part of PIXIE's theory for solving linear equations involving a single unknown.

On-line student modeling in PIXIE is a matter of selecting from among a set of precomputed (off-line) possible models, the one that best matches a student's answers to a set of problems. The off-line subsystem generates a generic problem space for each type of predefined problem using applicable correct and incorrect rules. The set of rules used in each solution path in this generic problem space corresponds to a possible model. The malrules used in model construction were predefined in (Sleeman, 1987) and inferrable in (Sleeman et al., 1990).

Two algorithms for inferring malrules, namely, INFER* and MALGEN, have been experimentally and independently coupled with PIXIE (Sleeman et al., 1990). The first algorithm, INFER*, initially applies valid rules to attempt a bidirectional traversal of the problem space (Figure 9, steps 1 and 2), whose initial and goal states are the initial problem equation and the student's answer, respectively. Malrules are then inferred via morphing (in a manner close to that shown in Figure 8) in order to complete a solution path in this problem space (Figure 9, step 3).

1. Apply valid 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 valid 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 implies a potential malrule. Forward these potential malrules to the investigator, who decides which malrule(s) to accept and generalize.

Figure 9. Behavior recognition and morphing in INFER*

Whereas INFER* learns malrules by morphing, MALGEN accomplishes the same by specializing perturbation operators. MALGEN views a problem space operator as a rule made up of four parts: (1) a *pattern* against which the current state is matched; (2) a set of *correctness conditions* that further specify when the rule is appropriate to apply; (3) a set of *actions* to be carried out; and (4) a *result*, which is the state that ensues from the action. MALGEN's perturbation rules (Figure 10) specify how the parts of a domain operator can be systematically modified to produce a malrule.

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 a result, switch the operands in it, if any.

Figure 10. Perturbation operators of MALGEN

(Hoppe, 1994)

Unlike PIXIE, which uses rather ad hoc procedures expressly written to infer malrules, Hoppe's (1994) system uses the domain rules and a resolution (Robinson, 1965) theorem prover, specifically Prolog, to explain students' answers to symbolic differentiation problems. A failed proof means that the associated student behavior is incorrect, assuming of course that the domain rules that have been supplied to the prover as axioms and theorems are correct and complete.

Costa, Duchenoy & Kodratoff (1988) have previously hinted at how residues of failed resolution proofs could be used to identify student misconceptions. Hoppe's system therefore takes advantage of such residues by morphing them to produce new malrules. The generalization procedure involves mapping equal constants to the same variable.

ASSERT and Bug Library Construction

Recall from the previous section that ACM and ASSERT can construct a student model without the need for any library of primitive buggy rules in the background knowledge; it is actually in the process of constructing a model that these systems learn new buggy rules. By storing these buggy rules, in a principled way of course, into the background knowledge for later reuse, these systems should therefore be able to construct their bug libraries from scratch. Unfortunately, ACM does not take advantage of this, though ASSERT does.

1. Collect the refinements (i.e., bug) of each student model into a list B, and remove duplicates. Compute the stereotypicality (see text) of each bug, B_i, in B.

2. For each bug, B_i , in B:
   (a) Compute the intersection between B_i and every bug in B, and determine the intersection I_i with the highest stereotypicality. If the stereotypicality of I_i exceeds that of B_i , redo this step using I_i in place of B_i .
   (b) Add I_i to the bug library.

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

Figure 11. ASSERT's procedure for building a bug library

Recall that ASSERT constructs a student model by refining the correct model (deleting a rule, deleting a condition, etc.) The set of refinements in a student model are collectively called a *bug* (cf. DEBUGGY's compound bug). ASSERT does not just store every bug into the bug library, however. Rather, it stores the most 'stereotypical' generalization of each bug. To do this, ASSERT first extracts the bug of each (buggy) student model (Figure 11, step 1) and then determines the frequency, or what its developers call 'stereotypicality,' of each bug using the following formula:

$$S(B_i) = \Sigma^{n}_{j=1} Distance(C; M_j) - \Sigma^{n}_{j=1} Distance(M_i; M_j)$$

where B_i are the refinements in model M_i, C is the correct model, and n is the number of (buggy) student models. It then tries to determine the generalization of each bug (i.e., the intersection of this bug, or its generalization in a previous iteration, with every bug other than itself) that has the highest stereotypicality value (Figure 11, step 2). This generalized bug, rather than the original bug (though the generalized bug might be the same as the original bug) is what ASSERT adds to the bug library.

MEDD

Like ASSERT, MEDD (Sison, Numao & Shimura, 1998a) also builds its bug library from multiple behaviors. However, unlike ASSERT, MEDD's domain task is problem solving, and one that is more complex compared to the domain tasks of DEBUGGY and ACM.

MEDD addresses the problem of automatically discovering and detecting novice (Prolog) programmer errors at the knowledge as well as behavior levels. MEDD tackles this problem in two steps. The first step involves intention-based error detection, in which the programmer's intention is discerned (Figure 12, step 1b), in a manner similar to that used in PROUST (Johnson, 1990), and the discrepancies between the student's actual and intended programs are computed and then further analyzed if they (the discrepancies) are indeed program errors or are correct stylistic variants (Figure 12, step 1c). This step is carried out by a system called MDD, part of which can be viewed as indirectly performing some kind of morphing: first it tries to explain the behavior; when it could not, it computes the discrepancies between the ideal and incorrect behaviors.

The second step involves *multistrategy conceptual clustering*, in which the behavior-level errors computed in the previous step are incrementally clustered (Figure 12, step 2) by an algorithm called MMD (Sison, Numao & Shimura, 1997) into an error hierarchy, whose main subtrees form intensional definitions of error classes that denote knowledge-level errors. Note that MMD considers causal relationships as well as regularities in the data.

1. Determine the intention, correctness, and errors (if any) of/in a student's program.
   (a) Canonicalize names and simplify the student program (*sp*).
   (b) Determine the reference program that best matches *sp*. In the process, compute the discrepancies δ between the student and reference programs.
   (c) Determine if δ is already in the bug library/error hierarchy (computed in step 2). If so, simply output the corresponding knowledge-level error(s) of δ. If not, test *sp* for correctness. If it is correct, either learn as a transformation rule or learn *sp* as a new reference program; other wise, either acquire (directly, or indirectly via debugging) the student's intention, or pass δ together with the intention to step 2.

2. Determine the classes of errors made by a student as well as a population of students.
   (a) Match δ with the children of a given node of the error hierarchy associated with the intention. Also determine causal relationships among the discrepancies in δ.
   (b) Position δ with respect to the matching children determined in step 1. Also increment weight counters.
   (c) Determine the directionalities of causal relationships within and among the nodes generated in step 2. Also sever a child from a parent with which it has no causal relationships, and reclassify it into the hierarchy.
   (d) Remove nodes whose weight counters fall below a threshold.

Figure 12. Basic procedure of MEDD

MEDD's outputs - behavior and knowledge-level errors in student programs - while not in narrative form, can be used in several ways (Sison, Numao & Shimura, 1998b). For example, the intensional definition of a knowledge-level error detected in a student's program can be used to: (a) provide graded hints and explanations to the student; (b) select (in a manner similar to case-based retrieval) or construct similar buggy programs for the student to analyze; or (c) select similar problems for the student to solve. These hints or cases are useful for obtaining confirmatory evidence for the partial model of the student, as well as for assisting the student as he or she assesses and refines his or her knowledge. In addition, MEDD's error hierarchy can also be used to differentiate severe and common errors from minor or infrequently occurring ones, which may be useful not only for remediation but also for curriculum design.

MEDD's incremental approach enables it to automatically construct and extend bug libraries at the same time that it infers partial student models in the form of knowledge-level errors. Moreover, such student models are each constructed by MEDD from a single behavior, with the help of knowledge learned from past multiple behaviors (recall Figure 2).

Discussion

The specialization of perturbation rules can be viewed as deductive in the sense that it involves the modification (specialization) of existing knowledge, specifically, perturbation rules, so that what is learned remains within the system's deductive closure. However, perturbation rules are: (1) not necessarily axiomatic or psychologically valid, and (2) are usually too powerful - anything can be perturbed. Their specialization in effect becomes inductive in the sense that the learned rule is implicitly attended by uncertainty.

Morphing can be viewed as similar to deductive learning in the sense that morphing entails the prior determination that two behaviors or states could not be transformed to the other, when they should be. Thus, prior to morphing, there needs to be some kind of explanation or proof

construction process (cf. explanation-based learning/EBL) that fails. However, the generation of a new operator that can bridge the gap between two states is completely inductive in morphing, unlike in operator specialization or EBL, in which a 'new' operator is specialized from an existing one. For this reason, we call morphing per se *pre-inductive*; since it generalizes empirically, albeit on the basis of a single behavior. Of course, one can attach frequency weights to operators acquired in this way, and modify the weights in a principled manner as more data are acquired. Alternatively, a morphed operator can be incrementally revised. However, this is not usually the case in the systems that use the technique.

The main difficulty with malrules that have been inferred from a single behavior (e.g., MALGEN; INFER*; Hoppe, 1994) is their credibility. The use of perturbation rules can ameliorate this to some extent, but one has to constrain the application of perturbation rules to psychologically plausible situations. This credibility or psychological plausibility issue is one reason why PIXIE requires that the malrules inferred by MALGEN or INFER* be examined by a human before these can be added to the bug library.

MEDD/MMD provides an alternative to human filtering. Specifically, it demonstrates that conceptual clustering, specifically a multi-strategic one, can be used to automate the acquisition of knowledge-level errors and their conceptual definitions. ASSERT's procedure, when transformed to one that is incremental, can probably achieve a similar effect to some extent, though in a more ad hoc way.

SIERRA

At this point we would also like to mention SIERRA, which can be used to model *hypothetical* students.

SIERRA embodies a theory of errors at the learning level, i.e., a theory of how knowledge errors are formed. To explain the genesis of specific incorrect procedures for subtraction, the developers of the BUGGY model proposed the REPAIR theory (Brown & VanLehn, 1980), which states that when incomplete knowledge (e.g., a missing subskill) leads to an *impasse*, students attempt some local problem solving tactic called a *repair* (e.g., using an analogous operation used in similar circumstances), that will allow them to continue with the task at hand. To further explain the genesis of a (hypothetical) student's incorrect procedural network (called the *core procedure*, represented as an AND/OR graph), VanLehn (1983) introduced a set of *felicity conditions*, similar to those used in linguistics, which have to be observed for students to correctly learn from examples. These conditions, which are the basic consituents of his STEP theory, include constraints that at most one disjunct (e.g., a conditional branch) be introduced per lesson; that there be a minimal set of examples per lesson; and that in introductory lessons, all 'invisible objects' be banned. The REPAIR and STEP theories are embodied in a program called SIERRA, which has essentially two parts: a learning component and a problem solving component. The learning component uses version spaces (Mitchell, 1977) to learn one disjunct at a time from a set of examples, while the problem solving component follows a version of REPAIR theory to generate bugs when the felicity conditions are not adequately met (VanLehn, 1987). Such computational models of human learners as SIERRA can be used to simulate a collaborative learner.

GENERAL DISCUSSION

We have described various student modeling systems and the various machine learning or machine learning-like approaches they use. We are now in a better position to see: (a) what still needs to be done in terms of using machine learning to address student modeling issues, and (b) whether and how what we have learned in student modeling research can be used in machine learning.

Machine Learning in Student Modeling

First of all, notice that almost every major approach to symbolic machine learning, whether for supervised induction (e.g., ID3 in ACM, MIS in THEMIS, FOIL-style induction in ASSERT), unsupervised induction (e.g., conceptual clustering in MEDD), or compilation (e.g., a form of explanation-based generalization in MALGEN), has been applied to the construction of student models and background knowledge bases (Figure 13)[3].Notice, too, that there has been significant progression from the use of ad hoc or inefficient machine learning procedures (e.g., DEBUGGY; ACM; ASSERT's bug library construction procedure) to more principled and efficient ones (e.g., ASSERT's student model construction procedure; MMD). This is a good trend. Now, two questions arise. First, which approach is best? Second, what else needs to be done?

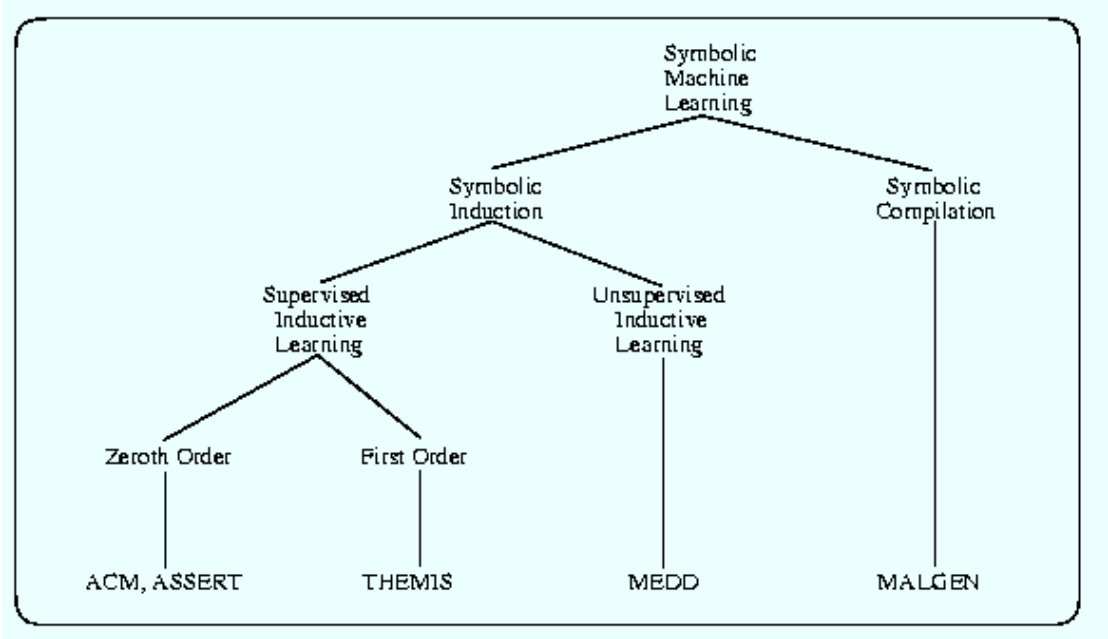

Figure 13. Machine learning in student modeling

Which approach is best?

Obviously, there is no best approach, i.e., there is no general-purpose machine learning approach that will simultaneously address all issues of student modeling. We can note, however, the following:

- Theory revision (e.g., ASSERT, THEMIS) seems a reasonable and efficient approach to the inductive construction of student models from multiple behaviors in concept learning domains where the correct theory is known (which is usually the case, at least in intelligent tutoring). Baffes & Mooney (1996) have demonstrated the utility of this approach by showing that remediation based on ASSERT-constructed student models led to performance improvements. However, the suitability of this approach to noisy concept learning domains, or to more complex domains such as problem solving domains, remains an open question.
- Conceptual clustering (e.g., MEDD), particularly one that is guided by background causal knowledge as well as data, seems a promising approach to constructing bug libraries inductively. Sison, Numao & Shimura (1998b) have demonstrated the potential of this approach by showing that error classes acquired in this manner correspond to knowledge-level errors found by human experts in incorrect behaviors. However, the

[3] The other major symbolic approaches, namely, analogical learning and reinforcement learning, seem more appropriate for learning-level analysis and tutoring, respectively.

power of this multistrategy clustering approach in domains in which it is not so easy to examine causal relationships among behavioral components, remains to be seen.

It will be noted that though both theory revision, and multistrategy conceptual clustering a la MMD, are inductive and therefore rely heavily on data, both approaches also rely more heavily on the background knowledge compared to other inductive approaches. This seems to indicate that a *multistrategy* learner that takes advantage of the background knowledge as well as the data seems suitable for dealing with the requirements of student modeling processes.

What else needs to be done?

From what we have seen above, it is clear that a lot of important questions still need to be answered. These include:

- How should noise be dealt with?
- Can theory revision be used in problem solving domains?
- Can multistrategy conceptual clustering (a la MMD) be as effective in simpler domains?
- Can't subsymbolic machine learning be used?

We now look at each of these questions. First, how should noise, which is ubiquitous in student modeling, be dealt with? There seems to be at least two ways of dealing with noise when constructing student models from multiple behaviors: one can either deal directly with noise (*noise handling*), or avoid it (*noise avoidance*). That is, one could use machine learning algorithms that explicity handle noise (or augment a learning algorithm with noise handling mechanisms such as those mentioned earlier). Alternatively, one could avoid having to deal with noise by, for example, performing only partial modeling (e.g., MEDD), constraining the student's behavior (e.g., model tracing in the ACT* tutors), or recasting domain tasks to reduce the scope of modeling (e.g., ASSERT does not ask students to write programs, but rather to classify programs represented as feature vectors).

When constructing bug libraries (as opposed to student models) from behaviors of a population of students, noise becomes less of a problem. For example, noise in the form of concept drift ceases to be an issue, while noise in the form of infrequent and isolated errors and slips can be handled, say by filtering them out (e.g., MEDD's 'forgetting' mechanism, Figure 12, step 2d).

Next, can theory revision be used in problem solving domains? While theory revision is certainly useful for constructing student models from multiple behaviors in concept learning domains, using it in problem solving domains might not be straightforward. First of all, there is the need to deal with multiple classes per instance (i.e., multiple knowledge-level errors, e.g., multiple incorrect problem solving operators, per incorrect behavior), though a simple albeit inefficient way to solve this problem would be to deal with one class at a time. However, more difficult than this problem is that of having to deal with possibly multiple, *alternative* sets of classes per instance (i.e., several possible combinations of knowledge-level errors, e.g., several possible sequences of incorrect as well as correct operators, per incorrect behavior). Dealing with this latter problem seems to require another process, in addition to theory revision. ACM's path finding procedure (recall Figure 5, step 1) is an example of such a process, though experiments of (Webb & Kuzmycz, 1996; Sison et al., 1998b) suggest that it might be possible to construct useful models in problem solving domains without such a 'reconstructive', pathfinding process. Clearly, this requires further study.

Next, can conceptual clustering be as effective in simpler domains? The success of conceptual clustering in MEDD's domain lies in its exploitation of causal relationships in the data, in addition to the regularities in the data. These causal relationships (e.g., argument dependencies) are not difficult to obtain in a domain such as logic programming. However, they might not be as easy to find in simpler problem solving domains such as subtraction, or in concept learning domains. On the other hand, perhaps causal relationships are less important at

these lower levels, and ordinary, similarity-based clustering (e.g., UNIMEM; COBWEB; SMD, Sison et al., 1998b) can be used. This would be interesting to find out.

Finally, can't subsymbolic machine learning be used? Admittedly, the use of so called subsymbolic learning approaches, e.g., connectionist and genetic learning, in student modeling, has so far been practically nil. This is partly due to the fact that, whereas student models need to be inspectable (Wenger, 1987), inspectability is not a main concern of evolutionary, and especially neural, systems. Nonetheless, this uninspectability may change soon. As we noted earlier, systems have already been developed that combine symbolic and subsymbolic learning approaches. The genetic classifier system of (Rendell, 1985), for example, uses non-string representations. Moreover, attempts have recently been made to automatically extract if-then rules from multilayer feedforward networks (Weijters, van den Bosch & van den Herik, 1998). We also mention at this point the representation of student knowledge in the form of Bayesian belief networks, since we mentioned earlier that learning in such networks, particularly in Bayesian networks with unknown variables, resembles neural learning. The use of Bayesian networks for student modeling has been studied in (Villano, 1992; Martin & VanLehn, 1993; Reye, 1996), though these have so far only dealt with networks with known structures and no unknown variables.

To summarize, theory revision and conceptual clustering seem to be suitable for the automatic construction of student models and bug libraries, respectively. However, theory revision may need to be augmented with noise handling as well as pathfinding or similar procedures for it to be useful in problem solving domains. The feasibility or necessity of using causal background knowledge for conceptual clustering in simpler domains also needs to be examined. Finally, the use of multistrategy genetic and neural learning in student modeling might prove useful very soon.

Student Modeling in Machine Learning

It is now becoming increasingly clear how machine learning techniques can be used in student modeling. But what about the reverse? Can student modeling techniques, or techniques that arise from student modeling research, be useful in machine learning?

Perhaps the student modeling approach that has made the most significant impact on machine learning would be Kurt VanLehn's SIERRA, particularly the felicity conditions of his STEP theory. The condition that there be a single disjunct per lesson constitutes a useful computational bias for constraining search, that is at the same time psychologically plausible.

Then there is also the diagnostic pathfinding method of Langley, used in ACM, that can be viewed as forming a basis of the relational pathfinding method of the FORTE system for first order theory revision (Richards & Mooney, 1995). Baffes & Mooney's (1993) m-of-n propositional theory revision system NEITHER, used in ASSERT, might (or might not) have been developed with student modeling and remediation in mind, but Sison et al.,'s (1997) MMD was certainly initially built to overcome the problem of multiple knowledge- level errors manifested in student programs (or, in machine learning parlance, multiple classes per object). MMD's use of causal background knowledge to guide the hierarchical reorganization of error hierachies was eventually found to be more effective and efficient than other techniques for mitigating so called *ordering effects* (Sison, Numao & Shimura, 1999), a problem of incremental learners that summarize information, in which different orderings of the input data produce different classifications. This use of knowledge in conceptual clustering for multiple class learning and disambiguation, and ordering effect mitigation, is potentially applicable outside of MEDD's student modeling domain, though this remains to be demonstrated empirically. Finally, the different noise-avoidance approaches of ACT*, ASSERT, and MEDD might also prove useful in other machine learning domains.

Machine Learning and Student Modeling Utility

Although this paper has so far assumed that student modeling is useful, we cannot end the paper without asking the question. Is student modeling really useful? One can answer in the negative

and cite two reasons (Ohlsson, 1991): (1) Sleeman et al.'s (1989) negative result that error-specific feedback is no more useful to the learner than generic feedback, and (2) the intractability of student modeling (Self, 1990).

Regarding the first point, however, more recent results (e.g., Nicolson, 1992; Baffes & Mooney, 1996) show that a student model can help intelligent tutoring. (Note, too, that Baffes & Mooney used machine-constructed student models.) Thus, as Sleeman et al. (1989) themselves have pointed out, there is an evermore pressing need to find out when - i.e., in what domains and tasks; for which pedagogical strategies; for what kinds or ages of students - student modeling can help.

Regarding the second point, the general problem of student modeling is, without doubt, intractable (see e.g., Self, 1990) . But there are ways to deal with this intractability (Self, 1990; McCalla, 1992). One way would be to constrain the student, or more specifically, the student's behavior, and, as McCalla (1992) has pointed out, this need not be as 'dictatorial' as the model tracing technique used by the ACT* tutors. Another way would be to constrain the tasks for which student modeling will be used. This can be viewed as the approach taken by ASSERT, which models students' incorrect programming conceptions not by examining the programs they write, but the vector representations of the programs they classify. Finally, one could constrain the system, or more specifically, the model. As discussed in (Bierman, Kamsteeg & Sandberg, 1992), tutors hardly have complete, detailed models of students. Instead, their representation of a student seems to consist of a global classification of the student plus some very localized diagnoses. This is in fact the approach taken by MEDD (as well as other nonlearning systems such as that of Bierman, et al., 1992).

In short, there is evidence that (1) student modeling is useful, and that (2) student modeling need not be detailed and complete to be useful. However, as has already been pointed out in the previous subsection, there is a need to identify and investigate more closely the factors that affect the utility of (each of the various approaches to) student modeling.

SUMMARY AND CONCLUSION

We started by defining the essential elements of student modeling, namely, the student behavior, the background knowledge, and the student model, and identified several important student modeling issues. Next, we identified the main approaches and paradigms of machine learning and briefly sketched each. After that, we examined the various ways in which machine learning techniques have been used in the induction of student models and in the extension or construction of the background knowledge needed for student modeling.

We have seen that almost every major approach to symbolic machine learning has been applied to these student modeling processes. Although there is no best approach, theory revision and conceptual clustering seem to be the most promising machine learning approaches for the automatic construction of student models and bug libraries, respectively.

While there is an unmistakable trend toward more principled and efficient learning procedures, there still is a lot that remains to be done, particularly in the investigation of the utility of theory revision for student model construction in noisy and more complex domains; the utility of multistrategy conceptual clustering a la MMD for bug library (and partial student model) construction in less complex domains; and the utility of multistrategy subsymbolic learning approaches. Finally, we have also seen how results in student modeling research have been, and might be, useful in machine learning research. This should not come as a surprise, for after all, it should be clear by now that student modeling is machine learning complete[4], that is, student modeling requires dealing with (sometimes indirectly, e.g., noise avoidance) nearly all the problems of machine learning, and more.

[4] As intelligent tutoring is artificial intelligence complete (Woolf, 1988).

Acknowledgments

We thank the anonymous reviewers, and Geoff Webb and John Self for their helpful comments.

References

Anderson, J. R. & Reiser, B. (1985). The LISP tutor. *Byte*, 10, 159-175.

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

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

Anderson, J. R. (1989). A theory of the origins of human knowledge. *Artificial Intelligence*, 40, 313-351.

Anderson, J. R. (1993). *Rules of the Mind*. Hillsdale, NJ: Erlbaum.

Baffes, P. & Mooney, R. (1993). Symbolic revision of theories with m-of-n rules. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence.*

Baffes, P. & Mooney, R. (1996). Refinement-based student modeling and automated bug library construction. *Journal of Artificial Intelligence in Education*, 7(1), 75- 116.

Bierman, D., Kamsteeg, P., & Sandberg, J. (1992). Student models, scratch pads, and simulation. In E. Costa (Ed.), *New Directions for Intelligent Tutoring Systems*. Berlin: Springer Verlag.

Booker, L., Goldberg, D., & Holland, J. (1989). Classifier systems and genetic algorithms. *Artificial Intelligence*, 40, 235-282.

Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). *Classification and Regression Trees*. Belmont, CA: Wadsworth.

Brown, J. & Burton, R. (1979). An investigation of computer coaching for informal learning activities. *International Journal of Man-Machine Studies*, 11, 5-24.

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

Brown, J., Burton, R., & de Kleer, J. (1982). Pedagogical, natural language, and knowledge engineering in SOPHIE I, II, and III. In D. Sleeman & L. Brown (Eds.), *Intelligent Tutoring Systems*. London: Academic Press.

Burton, R. & Brown, J. (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2, 155-191.

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

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

Carbonell, J. G. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Volume II. Los Altos, CA: Morgan Kaufmann.

Carr, B. & Goldstein, I. (1977). Overlays: A theory of modeling for computer-aided instruction. *AI Lab Memo 406*. Massachusetts Institute of Technology, Cambridge, Massachusetts.

Chapman, D. & Kaelbling, L. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparison. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*.

Cheeseman, P., Kelly, J., Self, M., Stutz, J., Taylor, W., & Freeman, D. (1988). AUTOCLASS: A Bayesian classification system. *Proceedings of the Fifth International Workshop on Machine Learning*.

Clancey, W. (1982). Tutoring rules for guiding a case method dialog. In D. Sleeman & L. Brown (Eds.), *Intelligent Tutoring Systems*. London: Academic Press.

Clancey, W. (1986). Qualitative student models. *Annual Review of Computer Science*, 1, 381-450.

Corder, S. (1967). The significance of learners' errors. *International Review of Applied Linguistics*, 5, 161-170.

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

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

DeJong, G. & Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, 1, 145-176.

Dietterich, T. (1986). Learning at the knowledge level. *Machine Learning*, 1, 287- 316.

Eisenstadt, M. (1997). My hairiest bug war stories. *Communications of the ACM*, 40(4), 30-37.

Fikes, R., Hart, P., & Nilsson, N. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3, 251-288.

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

Gagne, R. (1985). *The Conditions of Learning* (4th Ed.). New York: Holt, Rinehart and Winston.

Gennari, J., Langley, P., & Fisher, D. (1989). Models of incremental concept formation. *Artificial Intelligence*, 40, 11-61.

Gentner, D. (1989). The mechanisms of analogical learning. In S. Vosniadou & A. Ortony (Eds.), *Similarity and Analogical Reasoning*. London: Cambridge University Press.

Giangrandi, P. & Tasso, P. (1995). Truth maintenance techniques for modeling student's behavior. *Journal of Artificial Intelligence in Education*, 6, 153-202.

Ginsberg, A. (1989). Knowledge base refinement and theory revision. *Proceedings of the Sixth International Workshop on Machine Learning*.

Goldstein, I. (1982). The genetic graph: A representation for the evolution of procedural knowledge. In D. Sleeman & L. Brown (Eds.), *Intelligent Tutoring Systems*. London: Academic Press.

Hammond, K. (1989). CHEF. In C. Riesbeck & R. Schank (Eds.), *Inside Case-based Reasoning*. Hillsdale, NJ: Erlbaum.

Hinton, G. (1989). Connectionist learning procedures. *Artificial Intelligence*, 40, 185-234.

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

Huang, X., McCalla, G., Greer, J., & Neufeld, E. (1991). Revising deductive knowledge and stereotypical knowledge in a student model. *User Modeling and User- Adapted Interaction*, 1, 87-115.

Ikeda, M., Mizoguchi, R., & Kakusho, O. (1989). Student model description language SMDL and student model inference system SMIS. *Transactions of the IEICE*, J72-D-II, 112-120. In Japanese.

Johnson, W. L. & Soloway, E. (1984). Intention-based diagnosis of program errors. *Proceedings of the Second National Conference on Artificial Intelligence*.

Johnson, W. L. (1990). Understanding and debugging novice programs. *Artificial Intelligence*, 42, 51-97.

Joni, S., Soloway, E., Goldman, R., & Ehrlich, K. (1983). Just so stories: How the program got that bug. *Proceedings of the SIGCUE/SIGCAS Symposium on Computer Literacy*.

Kibler, D. & Aha, D. (1987). Learning representative exemplars of concepts: An initial case study. *Proceedings of the Fourth International Workshop in Machine Learning*.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 1982, 59-69.

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

Laird, J., Rosenbloom, P., & Newell, A. (1986). Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1, 11-46.

Laird, J., Rosenbloom, P., & Newell, A. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33, 1-64.

Langley, P. & Ohlsson, S. (1984). Automated cognitive modeling. *Proceedings of the Second National Conference on Artificial Intelligence*.

Langley, P. & Zytkow, J. (1989). Data-driven approaches to empirical discovery. *Machine Learning*, 40, 283-312.

Langley, P., Wogulis, J., & Ohlsson, S. (1987). Rules and principles in cognitive diagnosis. In N. Fredericksen (Ed.), *Diagnostic Monitoring of Skill and Knowledge Acquisition*. Hillsdale, NJ: Lawrence Erlbaum.

Langley, P., Iba, W., & Thompson, K. (1992). An analysis of Bayesian classifers. *Proceedings of the Tenth National Conference on Artificial Intelligence*.

Langley, P. (1996). *Elements of Machine Learning*. San Francisco, CA: Morgan Kaufmann.

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

Martin, J. & VanLehn, K. (1993). OLAE: progress toward a multi-activity, Bayesian student modeler. *Proceedings of the International Conference on Artificial Intelligence in Education '93*.

McCalla, G. (1992). The central importance of student modeling to intelligent tutoring. In E. Costa (Ed.), *New Directions for Intelligent Tutoring Systems*. Berlin: Springer Verlag.

Michalski, R. & Stepp, R. (1983). Automated construction of classifications: Conceptual clustering versus numerical taxonomy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(4), 396-410.

Michalski, R. & Stepp, R. (1983). Learning from observation: Conceptual clustering. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, CA: Tioga.

Michalski, R. (1983). A theory and methodology of inductive learning. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, CA: Tioga.

Mitchell, T., Keller, R., & Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1, 47-80.

Mitchell, T. (1977). Version spaces: A candidate elimination approach to rule learning. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*.

Mitchell, T. (1980). The need for biases in learning generalizations. Technical Report CBM-TR-117, Department of Computer Science, Rutgers University, New Brunswick, NJ.

Mitchell, T. (1983). Learning and problem solving. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*.

Mitchell, T. (1997). *Machine Learning*. New York: McGraw-Hill.

Murray, W. (1991). An endorsement-based approach to student modeling for planner-controlled tutors. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*.

Neal, R. (1991). Connectionist learning of belief networks. *Artificial Intelligence*, 56, 71-113.

Newell, A. & Simon, H. (1972). *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall.

Newell, A. (1980). Reasoning, problem solving, and decision processes: The problem space hypothesis. In R. Nickerson (Ed.), *Attention and Performance*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Newell, A. (1981). The knowledge level. *AI Magazine*, 2, 1-20.

Newell, A. (1989). *Unified Theories of Cognition*. Hillsdale, NJ: Harvard University Press.

Nicolson, R. (1992). Diagnosis can help in intelligent tutoring. *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*.

Ohlsson, S. & Langley, P. (1988). Psychological evaluation of path hypotheses in cognitive diagnosis. In H. Mandl & A. Lesgold (Eds.), *Learning Issues for Intelligent Tutoring Systems*. New York: Springer-Verlag.

Ohlsson, S. (1991). System hacking meets learning theory (Viewpoint column). *Journal of Artificial Intelligence in Education*, 2(3), 5-18.

Payne, S. & Squibb, H. (1990). Algebra malrules and cognitive accounts of errors. *Cognitive Science*, 14, 445-481.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann.

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

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

Rendell, L. (1985). Genetic plans and the probabilistic learning system: Synthesis and results. *Proceedings of the First International Conference on Genetic Algorithms and their Applications*.

Reye, J. (1996). A belief net backbone for student modeling. *Proceedings of the International Conference on Intelligent Tutoring Systems '96*.

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

Robinson, J. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12, 23-41.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65, 386-408.

Rumelhart, D. & Zipser, D. (1985). Feature discovery by competitive learning. *Cognitive Science*, 9, 75-112.

Rumelhart, D., Hinton, G., & Williams, R. (1986). Learning internal representations by error propagation. In D. Rumelhart & J. McClelland (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: MIT Press.

Schlimmer, J. & Granger, R. (1986). Incremental learning from noisy data. *Machine Learning*, 1, 317-354.

Self, J. (1990). Bypassing the intractable problem of student modeling. In C. Frasson & G. Gauthier (Eds.), *Intelligent Tutoring Systems: At the Crossroads of Artificial Intelligence and Education*. New Jersey: Ablex.

Self, J. (1994). Formal approaches to student modeling. In G. McCalla & J. Greer (Eds.), *Student Models: The Key to Individualized Educational Systems*, New York. Springer Verlag.

Shapiro, E. (1983). *Algorithmic Program Debugging*. Cambridge, MA: MIT Press.

Shavlik, J. & Dietterich, T. (1990). *Readings in Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Sison, R., Numao, M., & Shimura, M. (1997). Using data and theory in multistrategy (mis)concept(ion) discovery. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*.

Sison, R., Numao, M., & Shimura, M. (1998). Detecting errors in novice programs via unsupervised multistrategy learning. *Proceedings of the Fourth International Workshop on Multistrategy Learning*.

Sison, R., Numao, M., & Shimura, M. (1998). Discovering error classes from discrepancies in novice behaviors via multistrategy conceptual clustering. *User Modeling and User-Adapted Interaction* (Special Issue on Machine Learning for User Modeling), 8(1/2), 103-129.

Sison, R., Numao, M., & Shimura, M. (1999). Incremental multistrategy relational conceptual clustering and ordering effects. *Journal of the Japanese Society for Artificial Intelligence*, 14(1). To appear.

Sleeman, D., Kelly, A., Martinak, R., Ward, R., & Moore, J. (1989). Studies of diagnosis and remediation with high school algebra students. *Cognitive Science*, 13, 551-568.

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

Sleeman, D. (1982). Assessing aspects of competence in basic algebra. In D. Sleeman & J. Brown (Eds.), *Intelligent Tutoring Systems*. London: Academic.

Sleeman, D. (1983). Inferring student models for intelligent computer-aided instruction. In R. Michalski, J.G. Carbonell, & T. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, California: Morgan Kaufmann.

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

VanLehn, K. (1983). Human procedural skill acquisition: Theory, model and psychological validation. *Proceedings of the National Conference on Artificial Intelligence*.

VanLehn, K. (1987). Learning one procedure per lesson. *Machine Learning*, 31(1), 1-40.

Villano, M. (1992). Probabilistic student models: Bayesian belief networks and knowledge space theory. *Proceedings of the International Conference on Intelligent Tutoring Systems '92*.

Watkins, C. & Dayan, P. (1992). Q learning. *Machine Learning*, 8, 279-292.

Webb, G. & Kuzmycz, M. (1996). Feature based modelling: A methodology for producing coherent, consistent, dynamically changing models of agents' competencies. *User Modeling and User-Adapted Interaction*, 5(2), 117-150.

Weijters, T., van den Bosch, A., & van den Herik, J. (1998). Interpretable neural networks with BP-SOM. *Proceedings of the Tenth European Conference on Machine Learning*.

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

Winston, P. (1975). Learning structural descriptions from examples. In P. Winston (Ed.), *The Psychology of Computer Vision*. New York: McGraw-Hill.

Woolf, B. (1988). Intelligent tutoring systems: A survey. In H. Schrobe & AAAI (Eds.), *Exploring Artificial Intelligence*. Los Altos, CA: Morgan Kaufmann.