Learning About Systems That Contain State Variables

Thomas G. Dietterich
Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract

It is difficult to learn about systems that contain state variables when those variables are not directly observable. This paper formalizes this learning problem and presents a method called the *iterative extension method* for solving it. In the iterative extension method, the learner gradually constructs a partial theory of the state-containing system. At each stage, the learner applies this partial theory to interpret the I/O behavior of the system and obtain additional constraints on the structure and values of its state variables. These constraints can be applied to extend the partial theory by hypothesizing additional internal state variables. The improved theory can then be applied to interpret more complex I/O behavior. This process continues until a theory of the entire system is obtained. Several sufficient conditions for the success of this method are presented including (a) the observability and decomposability of the state information in the system, (b) the learnability of individual state transitions in the system, (c) the ability of the learner to perform synthesis of straight-line programs and conjunctive predicates from examples and (d) the ability of the learner to perform theory-driven data interpretation. The method is being implemented and applied to the problem of learning UNIX file system commands by observing a tutorial interaction with UNIX.

1. Introduction

Many important learning tasks involve forming theories about systems that contain state variables. Virtually all software systems, for example, contain state variables that are difficult to observe. Examples include operating systems, editors, and mail programs. These systems contain state variables such as mode switches, default settings, initialization files, and checkpoint mechanisms. Many problems in the sciences also involve learning about systems that contain state variables. In molecular biology, for example, the "state" of an organism includes the sequence of its DNA molecules. Existing techniques of molecular genetics provide only very indirect means for observing this state information.

Learning about a system that has internal state variables is difficult because the system does not always produce the same outputs when given the same inputs. Hence, in addition to solving the inherently underdetermined problem of guessing the relationship between the inputs and the outputs, the learner must also face the problem of guessing the structure and value of the state information and the relationship between the state information and the inputs and outputs. This paper presents a method, called the *iterative extension method* for learning about certain state-containing systems.

The problem of learning about systems with state can be formalized as follows. A state-containing system, M, is a function from $D \times S$ to $R \times S$, where D is the domain set of possible input values, R the range set of output values, and S the set of internal states of M. When M is given an input value and a state, it produces an output value and a new state. Let I be a sequence of input values, $\langle i_1, i_2, \ldots i_k \rangle$ and s_0 be the initial state. Then the sequence, O, of output values is generated as $M(i_1,s_0)=(o_1,s_1)$, $M(i_2,s_1)=(o_2,s_2)$, ... The learning task is to develop a theory of M given only the sequence I of inputs and the sequence O of corresponding outputs. This theory must be strong enough to predict o_j and s_j given any input i_j and previous state s_{j-1}.

In the most general case, this learning problem is unsolvable. However, if I, O, and M satisfy certain conditions, then it is possible for a learning system to form a theory of M by employing the iterative extension method. During iterative extension, the learning system develops a sequence of ever more accurate theories, T_i. Each theory explains additional behavior of M either by (a) proposing additional procedures that access and modify known state variables or (b) proposing the existence of additional state variables. At each step i, the learning system applies theory T_i to infer the values of the known state variables of M at as many points in the training sequence as possible. Given this knowledge of the state of M, the learning system then examines the remaining, unexplained data points, looking for some points that can be explained by simple extensions to T_i. First, the learner looks for points that can be explained by proposing additional procedures that compute some function of the inputs and the known state and produce the observed output values and state changes. If no such points can be found, then the learning system looks for data points that could be explained by hypothesizing the existence of an additional state variable inside M. A new theory T_{i+1} is developed by extending T_i to include either the new procedure or the new state variable, and the process is repeated. The key assumption underlying this method is that the learner will be able to identify such procedures and state variables at each point in the learning process. Notice that this is a greedy algorithm that attempts to minimize the number of state variables introduced.

This learning method is significant because it demonstrates how a learning system can exhibit something other than "one-shot" learning. Most existing learning systems start with some body of knowledge T_0, move to a larger body of knowledge T_1, and halt. In this iterative extension method, each partial theory T_i is applied to interpret the data so that the next partial theory T_{i+1} can be developed. There is no point at which the method necessarily halts.

The outline of the paper is as follows. First, previous research on learning about systems with state is reviewed and compared to the present effort. Second, a detailed example of the iterative extension technique is presented and its underlying assumptions are formalized. Third, a system, called EG, is described that applies the method to form theories of 13 UNIX file system commands from a trace of a tutorial session with UNIX. The paper concludes with a summary of the main issues.

2. Review of previous work

Most research on learning has focused on the learning of pure functions, predicates, and procedures. All of the concept learning work, for example, has dealt with the problem of determining a definition for a predicate concept in terms of some concept description language (e.g., Mitchell, 1977; Michalski, 1969, 1983; Quinlan, 1982; Winston, 1975). Research on automatic programming from examples has, for the most part, focused on the learning of functions in pure Lisp (Hardy, 1975; Shaw, Swartout, & Green, 1975), pure Prolog (Shapiro, 1981), and similar languages (Amarel, 1983, Bauer, 1975; Siklossy & Sykes, 1975; Sussman, 1975). The main problems addressed by this body of research are (a) *generalization* (determining the class of input values for which the procedure is defined), (b) *loop introduction* (determining when to introduce a loop or recursive call), (c) *subroutine introduction* (determining when to create a subroutine to share code among different parts of the system), (d) *conditional induction* (determining which boolean function of the inputs should be tested at a particular choice point in the program), and (e) *planning* (determining a sequence of actions (or a functional expression) that will compute the output as a function of the input). These are very difficult problems, but they are orthogonal to the problem of learning about state. For these authors, there are no state variables that retain their values from one invocation of the system to the next. If there are any variables at all, they serve only as temporary variables that disappear when each output is produced.

One body of research that is superficially similar to the current effort is research on automatic programming from traces (Biermann & Krishnaswamy, 1976; Neves, 1981; VanLehn, 1983). The goal of such work is identical to the work described above—namely, to synthesize a pure procedure. The similarity with the state-learning task stems from the fact that each individual step within a trace takes place in the context of some global variables. However, the values of such global variables are provided to the learner at each point, so this body of research is not relevant to the present task.

The body of research most similar to that described in this paper is the work on synthesis of Turing machines and finite-state machines from traces (Biermann, 1972; Biermann and Feldman, 1972). In the case of Turing machines, for example, the contents of the tape and the action of the machine at each step are given. The learning task is to infer the finite-state controller for the machine. This involves hypothesizing the number of states and the state transition matrix. It is appropriate to view these systems as having a single state variable whose value gives the current state of the controller. The internal state bears a particularly simple relationship to the output. The output is a simple table lookup given the current state and the input. Hence, the kind of learning taking place is rote learning of I/O pairs subject to the organization imposed by the attempts to minimize the number of states in the finite-state machine. The bulk of the state of the system is stored on the tape—and that state information is known to the learner. Hence, these methods are not relevant to the present task either.

The conclusion to be drawn from this review of the literature is that little or no progress has been made on the problem of learning about systems that contain state variables.

Now that we have reviewed the literature, we present the iterative extension method, which can be employed to learn about certain kinds of state-containing systems.

3. The iterative extension method

The easiest way to describe the iterative extension method is by example. Suppose that the system to be learned, M, is the following PASCAL-like program that computes the balance of some checking account. The account has an overdraft limit, and if a check would cause the balance to go below this limit, then it is refused and a message is printed. The special input "OK" causes an additional $100 to be added to the overdraft limit.

```
BALANCE := 0;
LIMIT := -100;
WHILE TRUE DO BEGIN
   READ(I);
   IF I=0 THEN PRINT(BALANCE)
   ELSE IF (I<0) AND (BALANCE+I<LIMIT)
        THEN PRINT("CHECK REFUSED")
   ELSE IF I="OK"
        THEN LIMIT := LIMIT - 100;
             PRINT("OK")
   ELSE BALANCE := BALANCE + I;
        PRINT("NEXT?");
   END;
```

This system contains two state variables: BALANCE and LIMIT. BALANCE is directly observable when I=0, but LIMIT can only be observed indirectly by knowing BALANCE and I when the message **CHECK REFUSED** is printed. Now suppose the learning system is given the following sequence of I/O pairs (i_j,o_j).

```
<(0, 0)                 (0, -100)
 (0, 0)                 (-5, CHECK REFUSED)
 (10, NEXT?)            (0, -100)
 (0, 10)                (OK, OK)
 (0, 10)                (-5, NEXT?)
 (-110, NEXT?)          (-95, NEXT?)
 (0, -100)              (0, -200)
 (-1, CHECK REFUSED)    (-2, CHECK REFUSED)>
```

Given this I/O sequence, the following paragraphs present one possible path of inferences that the learner might make in applying the iterative extension method. Many other paths are possible.

The iterative extension process begins with a null partial theory*, T_0. The learner looks for points in the sequence for which a simple theory can be developed. The first two I/O pairs provide such a point. The learner can propose that whenever a 0 in given to M, a 0 is printed. This is theory T_1. Of course, T_1 is immediately contradicted by the fourth and fifth I/O pairs. However, this case triggers one of the learner's *state introduction heuristics*. This heuristic—called the *constant-change-constant rule*—says: *If M exhibits one constant behavior and then shifts to another constant behavior, hypothesize that there is a state variable responsible for the behavior and that its value has changed.* Hence, the learner guesses that there is a state variable (SV1) that is printed whenever $i_j=0$. The input of $i_3=10$ changed the value of SV1. This is theory T_2.

Now, by applying this theory, it is possible to interpret several points in the I/O sequence and infer the value of SV1. In particular, the learner can determine that after the step in which $i_3=10$, SV1 = 10, and before that step, SV1 = 0. This is very nice, because it reduces the problem of learning about state-containing systems to the problem of synthesizing pure programs from I/O pairs. In this case, the inputs are x = 10 and y = 0, and the output is z = 10. Existing methods of expression induction (Langley, 1980; VanLehn, 1983) can be employed at this point to guess that z = x + y. Translating this back into the program M, the learner can obtain theory T_3 that M is performing the operation SV1 := SV1 + I.

Now, T_3 is employed to interpret the I/O sequence. T_3 seems to hold true for every point in the sequence at which **NEXT?** is printed.

*In addition to T_0, the learning system must have some prior knowledge and bias about the space of possible programs.

However, for the two cases where **CHECK REFUSED** is printed, something else is happening. It appears that some conditional behavior is occurring. Techniques of concept learning can be employed at the points where $i_8 = -1$ and $i_{10} = -5$ to determine that **CHECK REFUSED** is being printed when SV1 + I < −100. This provides theory T_4.

However, T_4 breaks down later after the OK command. At that point, it appears that **CHECK REFUSED** is being printed when SV1+I < −200. The constant-change-constant rule can be applied again here to propose that there is a second state variable (SV2) that is changed by the OK command. The final theory says that the CHECK REFUSED message is printed whenever SV1 + I < SV2.

This example shows how the learner can form theories about the "easy cases" and then apply these theories to simplify the remaining learning problem in order to expose additional easy cases. This is the key idea behind the iterative extension method.

4. Conditions on the applicability of the method

What must be true in order for the iterative extension method to work? This section attempts to formalize (a) the conditions that must hold for the system M, (b) the conditions that must be satisfied by the training sequences I and O, and (c) the capabilities required of the learning system.

Three sufficient conditions on M can be stated: (a) state observability, (b) learnability of individual state transitions, and (c) state decomposability. The condition of state observability says that every distinct point in the state space, S, of M must lead to an observable behavioral difference. That is, given any two distinct states, s_a and s_b, there must be some sequence of inputs I' such that the sequence of outputs O'_a obtained by placing M in s_a and feeding it I' differs from the sequence of outputs O'_b obtained when M is started in s_b.

To describe the remaining two conditions, several auxiliary definitions are needed. First, let us define the *decision-tree decomposition* of M as follows. Assume that the true theory of M is known. Rewrite M to gather all conditional tests into a decision tree and all actions into straight-line programs in the leaves of that tree. Define a new subprogram, M_i, for each leaf of this tree. The new subprogram contains a long conditional test for applicability, C_i, (obtained by traversing the decision tree from the root down to the leaf) plus the straight-line program, P_i taken from the leaf of the tree. This decomposition can be viewed as representing M as a set of production rules such that, in any given situation, the antecedent of exactly one rule is satisfied**.

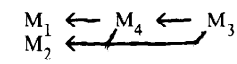
In the checking account example of the previous section, the decision-tree decomposition is

```
M1:  IF I=0 THEN PRINT(BALANCE)
M2:  IF (I<0) AND (BALANCE+I<LIMIT)
          THEN PRINT("CHECK REFUSED")
M3:  IF I="OK" THEN LIMIT := LIMIT - 100;
                          PRINT("OK")
M4:  IF I>0 OR (I<0) AND (BALANCE+I>=LIMIT)
          THEN BALANCE := BALANCE + I;
               PRINT("NEXT?").
```

Roughly speaking, at each point in the iterative extension process at least one M_i is accessible because enough is known about the state variables of M for the learner to observe the effects of M_i. Hence, the learner is able to form a partial theory of at least one M_i at each iteration. This partial theory is then applied to interpret more data so that additional M_i's will be made accessible.

It should be emphasized that the decision-tree decomposition is an analytical fiction developed from a privileged viewpoint. The learner need not represent its theories in production-rule or decision-tree form.

From the decision-tree decomposition, we can define an interaction graph as follows. The nodes of the graph are the subprograms, $\{M_i\}$. Two nodes M_j and M_k are connected by a directed edge from M_j to M_k if M_j modifies state information that is accessed by M_k. The interaction graph for the checking account example is



Since M satisfies the state observability condition, it follows that the interaction graph can be spanned by a root-directed forest. In this case, the roots of the forest are M_1 and M_2. The iterative extension process begins by developing (partial) theories of the roots of this forest and then working backwards along the edges until all nodes have been learned.

Two more definitions are needed. Define S_j to be that portion of the state information that is directly observable according to theory T_j. In the checking account example, for instance, S_2 includes only the state variable BALANCE, but not LIMIT. Also, define $M_i|S_j$ to be the partial theory of M_i involving only the state information of S_j. In the checking account example, $M_4|S_2$ is the rule **IF I<>0 THEN BALANCE := BALANCE + I; PRINT("NEXT? ")** in which all mention of LIMIT has been removed. $C_i|S_j$ and $P_i|S_j$ denote the condition and action parts of $M_i|S_j$.

Given these definitions, the second condition—learnability of state transitions—can be defined as follows. For each j such that T_j is a partial theory, there must exist some M_i such that $P_i|S_j$ is learnable from examples. The intuition behind this condition is that given only the state information in S_j, it must be possible to form a straight-line procedure for $P_i|S_j$.

The third condition—state decomposability—is the most interesting. Its role is to ensure that each C_i can be learned and additional state variables can be hypothesized. In order to learn each such conditional, it is important to be able to gather examples of situations in which it is true and false. Such examples can be gathered by establishing known prior states, $\{s_m\}$, exercising M_i, and then observing the resulting states $\{s_{m+1}\}$. But, the process of establishing known prior states and observing the resulting states requires that the learner apply its current theory T_j. Since this theory is a partial theory, there might be unknown side-effects of it that would interact with M_i and hence confuse the process. The state decomposability condition guarantees that this will not happen. It requires that for each M_i in the context of S_j it must be possible to force the overall system M into a region Q_{ij} of state space in which C_i is true and all changes wrought by P_i or any of the $\{M_k\}$ in T_j either change state information in S_j or else change state information that does not take M out of the region Q_{ij}.

This condition as stated is difficult to understand. The problems to look for are cases in which P_i (or one of the $\{M_k\}$ in T_j) changes a state variable that is tested by C_i. If this state variable is not observable according to the current partial theory, then it must be possible—by controlling the inputs and the values of other state variables—to keep C_i true.

Each of the M_i in the checking account example satisfies this decomposability condition. Notice in particular that although C_4 tests the value of BALANCE and P_4 modifies this value, BALANCE is

**This can always be accomplished, even for embedded loops and subroutines—by encoding control information in additional state variables.

already observable according to T_1. Suppose for a moment that BALANCE is not observable or is modified by M_1. M_4 would still satisfy the condition because C_4 does not test BALANCE in the region of state space for which I>0.

We can obtain a system that violates the decomposability condition by modifying M_2 to read **IF (I<0) AND (BALANCE+I<LIMIT) THEN LIMIT:=LIMIT - 20; PRINT("CHECK REFUSED").** This rule M_2 violates the condition because it modifies LIMIT in such a way that the condition C_2 is no longer true, and LIMIT is not in S_j. Every time the learner tries to observe the value of LIMIT, it changes. This makes it impossible for the iterative extension approach to succeed.

Now that we have described the requirements for the learned system, M, we turn our attention to the requirements for the training sequence. The training sequence of I/O pairs must exercise a directed spanning forest of the interaction graph. Furthermore, at the point in the training sequence where the learning system is attempting to form a theory of $C_i|S_j$, the training sequence must force M into the region Q_{ij} where valid training examples can be obtained and the sequence must include appropriate surrounding inputs so that the states before and after M_i can be inferred. In other words, the training sequence must include controlled experiments for each $C_i|S_j$. The exact requirements for the training sequence depend somewhat on the power of the learning system.

The requirements for the learning system are quite stringent. First, the learner must be able to perform *theory-driven data interpretation.* In other words, given a partial theory T_j, the learning system must be able to apply that theory to interpret the training data and thereby infer the values of the observable state variables. In the case of procedural theories, this involves reasoning both forwards and backwards through a partial program to obtain constraints on the state variables accessed by that program. This problem is very difficult because of the combinatorial explosion of alternative interpretations of the partial program when the values of the state variables are unknown and because programs are generally not invertible.

Second, the learner must be able to perform *program synthesis from I/O pairs.* The principle difficulty here is the straight-line planning task of finding a sequence of actions P_i that will produce the outputs from the inputs. Most of the problems encountered in standard AI planning tasks are met here (e.g., goal interaction, the desire to plan a single act to achieve multiple goals, combinatorial explosion of operator choices).

Third, the learner must be able to induce the conditions C_i under which the P_i occur. This is an instance of *concept learning* with the additional twist that the learner is permitted to introduce new state variables. The learner must have a set of state-introduction heuristics similar to the constant-change-constant heuristic. Two other heuristics deserve mention here. One is the *toggle rule.* It says: *If repeated inputs i_j seem to shift the system from one behavior to another and back again, then propose that i_j causes a boolean state variable to be toggled and that the M_j's test this variable.* Another heuristic is the *information flow rule: If unusual input i_j appears as an output at a later time, then suggest the existence of a new state variable that stores the value of i_j.*

5. An application of the method: forming theories of UNIX

A program, called EG, is being developed that applies the iterative extension strategy to the task of learning the file system commands of the UNIX operating system. This section gives a brief overview of the UNIX learning task and of the two principle components of EG: the program reasoner and the theory-formation engine.

The UNIX learning task is shown in Figure 5-1. This task was selected with the goal of developing an automatic knowledge acquisition system for the Stanford IA project. The IA project is an attempt to build an intelligent front-end for the diverse operating systems of the Arpanet. UNIX is notoriously difficult to learn (Norman, 1981). Nonetheless, this learning task satisfies the conditions of learnability set forth in the preceding section.

Given:	A programming language and a set of primitive operations
	The syntax for 13 UNIX file system commands (ls, mv, cp, rm, ln, mkdir, rmdir, chmod, umask, create, type, pwd, cd)
	A partial theory for 2 of these commands (ls, type)
	A tutorial session with UNIX where each of the commands is exercised in detail
Find:	Procedural theories for each of the 13 commands.

Figure 5-1: The UNIX learning task

UNIX is clearly a state-containing system. The principle state variables are (a) the file system (including the directory structure, the attributes and contents of every file, and so on), (b) the current working directory, and (c) the default file protection code. Within the file system there is some state information that is only indirectly observable. For example, information indicating which files are alias file names for one another is not printed by the default **ls** command. This information can be observed by, for example, modifying one file and then checking the contents of the other files. Also information about the configuration of the file system across several disk devices is not directly observable. UNIX commands are sufficiently complex that the training sequence must be carefully designed to guarantee that the conditions described in the previous section are met.

Notice in Figure 5-1 that EG is given some information besides the I/O training sequence (tutorial session). In particular, EG is given an initial theory of the **ls** and **type** commands. This was necessary because EG does not have a state-introduction heuristic capable of guessing the structure of the file system merely by observing the training sequence. Indeed, for most applications of the iterative extension method, it will be necessary to provide the learner with a starting theory that connects some part of the internal state information to some observable output.

EG is also given the syntax of the UNIX commands. This simplification is intended to insulate EG from user interface issues so that the basic problem of learning about state can be addressed.

The EG program contains two major subprograms: the program reasoner and the theory-formation engine. The program reasoner is a general interpreter and symbolic executor for programs expressed in the language of the programmer's apprentice "deep plans" representation (Rich and Shrobe, 1976). It operates in a manner similar to the EL system (Stallman and Sussman, 1977). EG uses the program reasoner to perform theory-driven data interpretation. Given a theory T_j and some input and output values, the program reasoner is invoked to propagate the input and output values through T_j to infer the values

of UNIX state variables. For example, given the output of a directory listing and a theory of the directory listing command, the program reasoner can infer the names and attributes of the files in the given directory.

The program reasoner operates by propagating input and output values through the partial program just as EL propagates values through a circuit. As with EL, when the program reasoner cannot propagate a value, it creates a variable and propagates expressions involving that variable around the program. One important difference between EL and the EG program reasoner is that in EG, constraints on the possible values of the variable can also be propagated. Hence, EG may not know the exact value of a list, but it may know that the list begins with (A B C). Another important departure from EL is that the program reasoner pursues several interpretations in parallel. This is essential, because it is a rare case that the I/O data admit of only one interpretation.

The theory-formation engine is a means-ends analysis planner similar to NOAH (Sacerdoti, 1977). Given starting and ending states of UNIX, it attempts to construct a plan that will get from the starting state to the ending state. The operators available to the planner are the primitive operators in the language (e.g., operators to manipulate lists, sets, and finite mappings) and any procedures that were included in one of the previous theories, T_j. EG is capable of developing conditional plans, but not loops or recursive programs.

6. Summary and Concluding Remarks

The problem of learning about state-containing systems is difficult to solve because, in addition to solving standard problems of induction from I/O pairs, the learner must also hypothesize the structure and values of the internal state variables of the system. For systems that satisfy the three conditions of state observability, state decomposability, and state-transition learnability, the iterative extension strategy can be applied to learn them.

The iterative extension method shows how a learning system can go beyond "one-shot" learning. Prior knowledge is applied to acquire further knowledge. The way in which the prior knowledge aids the learning process is by enabling the learner to interpret additional data from the training sequence. Theory T_j can be applied to interpret additional data so that T_{j+1} can be developed.

A critical condition for the success of the iterative extension method is that the training sequence be properly structured. An important question for future research is whether a learning system can be built that develops its own training sequence by performing controlled experiments. What additional constraints on the learned system must hold in order for experimentation to succeed?

A system, called EG, is being constructed that applies the iterative extension strategy to learn the semantics of UNIX file system commands.

7. Acknowledgments

I wish to thank James Bennett and Bruce Buchanan for valuable criticism of drafts of this paper. Advice from Bruce Buchanan and Mike Genesereth has been extremely valuable in guiding this research. I thank IBM for supporting this research through an IBM graduate fellowship.

8. References

Amarel, S., Program synthesis as a theory formation task--problem representations and solution methods, Rep. No. CBM-TR-135, Dept. of Computer Science, Rutgers University, 1983.

Bauer, M., A basis for the acquisition of procedures from protocols, IJCAI-4, 226-231, 1975.

Biermann, A. W., On the inference of Turing machines from sample computations, *Artificial Intelligence*, Vol. 3, 181-198, 1972.

Biermann, A. W., and Feldman, J. A., On the synthesis of finite-state machines from samples of their behavior, *IEEE Transactions on Computers*, Vol. C-21, 592-597, 1972.

Biermann, A. W., and Krishnaswamy, R., Constructing programs from example computations, *IEEE Transactions on Software Engineering*, Vol. SE-2, 141-153, 1976.

Hardy, S., Synthesis of LISP functions from examples, IJCAI 4, 240-245, 1975.

Langley, P. W., Descriptive discovery processes: Experiments in Baconian science. Rep. No. CS-80-121, Computer Science Department, Carnegie-Mellon University, 1980.

Michalski, R. S. On the quasi-minimal solution of the general covering problem, in *V International Symposium on Information Processing, FCIP 69*, Yugoslavia, Vol. A3, 2-12, 1969.

Michalski, R. S., A theory and methodology of inductive learning, *Artificial Intelligence*, Vol. 20, 111-161, 1983.

Mitchell, T. M. Version spaces: an approach to concept learning. Rep. No. STAN-CS-78-711, Computer Science Dept., Stanford University. (Doctoral dissertation.) 1977.

Neves, D. M., *Learning procedures from examples*. Unpublished doctoral dissertation, Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA, 1981.

Norman, D., The trouble with UNIX, *Datamation*, 139-150, November, 1981.

Quinlan, J. R. Learning efficient classification procedures and their application to chess end-games, in *Machine Learning*, Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., eds., Palo Alto: Tioga, 1982.

Rich, C., and Shrobe, H. E., Initial report on a Lisp programmer's apprentice, Rep. No. AI-TR-354, Artificial Intelligence Lab, MIT, 1976.

Sacerdoti, E. D., *A structure for plans and behavior*, North-Holland, 1977.

Shapiro, E. Y., Inductive inference of theories from facts, Res. Rept. 192, Department of Computer Science, Yale University, 1981.

Shaw, D. E., Swartout, W. R., Green, C. C., Inferring LISP programs from examples, IJCAI4, 351-356, 1975.

Siklossy, L, and Sykes, D., Automatic program synthesis from examples problems, IJCAI-4, 268-273, 1975.

Stallman, R. M., and Sussman, G. J., Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence*, Vol. 9, No. 2, 1977.

Sussman, G. J., *A computer model of skill acquisition*, New York: American Elsevier, 1975.

Utgoff, P. E., Adjusting bias in concept learning, *Proceedings of the International Machine Learning Workshop*, Department of Computer Science, University of Illinois, Urbana, 1983.

VanLehn, K., Felicity conditions for human skill acquisition: validating an AI-based theory, Rep. No., CIS-21, Xerox Palo Alto Research Center, 1983.

Winston, P. H., Learning structural descriptions from examples, in *The psychology of computer vision*, New York: McGraw-Hill, 157-209, 1975.