From: AAAI-87 Proceedings. Copyright ©1987, AAAI (www.aaai.org). All rights reserved.

Forward Chaining Logic Programming with the ATMS[1]

Nicholas S. Flann, Thomas G. Dietterich and Dan R. Corpron
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

Abstract

Two powerful reasoning tools have recently appeared, logic programming and assumption-based truth maintenance systems (ATMS). An ATMS offers significant advantages to a problem solver: assumptions are easily managed and the search for solutions can be carried out in the most general context first and in any order. Logic programming allows us to program a problem solver declaratively—describe *what* the problem is, rather than describe *how* to solve the problem. However, we are currently limited when using an ATMS with our problem solvers, because we are forced to describe the problem in terms of a simple language of forward implications. In this paper we present a logic programming language, called FORLOG, that raises the level of programming the ATMS to that of a powerful logic programming language. FORLOG supports the use of "logical variables" and both forward and backward reasoning. FORLOG programs are compiled into a data-flow language (similar to the RETE network) that efficiently implements deKleer's consumer architecture. FORLOG has been implemented in Interlisp-D.

I. The ATMS and Consumer Architecture

Truth maintenance systems provide several benefits to problem solving systems [deKleer, 1986a,b]. The ATMS allows the problem solver to store multiple, contradictory states during search. Each state corresponds to a *context*, a different set of assumptions made during the search. To provide maximal sharing between different contexts, a global database is employed with each fact assigned a unique *ATMS-node* that holds a *label* describing the set of contexts to which it belongs. The problem solver informs the ATMS of each inference. The ATMS caches the new derived fact and its justification in a global dependency structure.

By caching all inferences performed, the ATMS can ensure that no inference is performed more than once. By maintaining updated labels on each fact, the ATMS ensures that facts are maximally shared among different contexts. By maintaining the dependency structure, dependency-directed backtracking can be supported.

[1]This work was supported in part by the National Science Foundation under grants DMC-8514949 and IST-8519926 and by a contract from Tektronix, Inc.

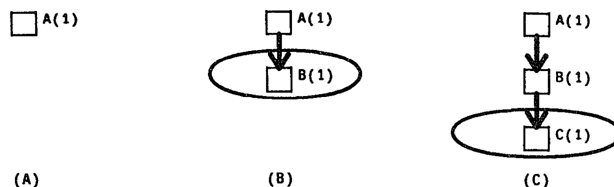

Figure 1: Consumer Architecture Example

One problem solver architecture that exploits these ATMS benefits is called the *consumer architecture* [deKleer, 1986c]. To characterize the kinds of computation the consumer architecture can perform, we can view it as directly executing the following logical language. In this language, (referred to as the CA language) a problem is described as a set of well-formed formulas each of which must have one of the following forms:

$$\forall \bar{x}\, P_1(\bar{x}) \supset P_2(\bar{x})$$
$$\forall \bar{x}\, P_1(\bar{x}) \equiv P_2(\bar{x})$$
$$L$$

where L is a ground atomic formula and each P_i is a disjunction of the form

$$E_1 \vee E_2 \vee \ldots \vee E_n,$$

Each E_j is a formula of the form

$$\forall \bar{y}\, P_1 \wedge P_2 \wedge \ldots \wedge P_m.$$

The basic formula is a universally quantified biconditional or implication. The left- and right-hand sides are both disjunctions of conjunctions (with nesting permitted to any depth). The third form—the ground atomic formula—is employed to state basic facts to the system. The main limitations of this language are that only universal quantification is permitted and values are restricted to constants (zero order terms).

To understand the consumer architecture, consider the example below:

$$\forall s\, A(s) \supset B(s) \tag{1}$$

$$\forall q\ B(q) \supset C(q) \quad (2)$$

$$A(1). \quad (3)$$

The problem solver performs forward chaining. Whenever the antecedent pattern (e.g., $A(s)$) of any implication is satisfied by a conjunction of facts in the database (e.g., $A(1)$), the implication "fires." A job, called a *consumer,* is put on the problem solver's global agenda. Each consumer is assigned an ATMS-node that is justified by the ATMS-nodes of the facts that satisfied the antecedent pattern. Hence, consumers are already linked into the dependency structure even though the resulting assertion, $B(1)$, has not been entered into the database (see Figure 1(b)). The consumer contains the "detached" consequent of the implication ($B(1)$), formed by instantiating the right hand side of (1) with the bindings generated from the antecedents. Problem solving proceeds by picking consumers off the agenda and running them. When a consumer is run, the new detachment is asserted into the database, possibly satisfying other implications and generating new consumers. When $B(1)$ is asserted, a new consumer is generated (through satisfaction of (2)) and put on the agenda, as illustated in Figure 1(c).

The consumer architecture has many advantages. Inferences can be made in any order and in any context. No inferences are missed, and no inferences are done twice. This is possible because the consumer architecture solves the *un-outing* problem. This problem arises when a line of reasoning is interrupted, so that reasoning can proceed in a different context. The un-outing problem is the problem of resuming the reasoning in the previous context. DeKleer solves this problem by imposing three conventions on the problem solver:

- The problem solver must keep the ATMS fully informed. All information that the problem solver uses in making an inference (i.e., all the antecedent facts) must be reported to the ATMS as justifications for the resulting consumer.
- A clear distinction is drawn between the problem solver and the ATMS. The problem solver must generate all consumers irrespective of the status of their labels. Even if a consumer's support is currently believed to be contradictory, the consumer must be produced and put on the agenda. This ensures that the inference will be made if ever the supporting context is un-outed.
- The problem solver's state is explicit. A consumer, although it is part of the problem solver's state, is like a "virtual" fact. It has an ATMS-node and is linked into the dependency network, but has not been asserted, and hence has no consequences. By linking it into the dependency structure, the ATMS ensures that any changes to the consumer's underlying support will be reflected in the consumer's label.

The un-outing problem is elegantly solved. To move between contexts, we choose consumers from the agenda (by inspecting their labels), rather than choosing implications to fire.

The advantages afforded by the consumer architecture and the CA language are apparent. Unfortunately, there are significant limitations that currently restrict the usefulness of the ATMS. We identify the following related problems:

Lack of expressiveness. The CA language limits variables to be universally quantified—we are unable to exploit the power of the existentially quantified variable. Values are limited to constants. The representational power of *terms* for capturing structured objects, such as trees and lists, is not available.

Lack of reversibility. Both the CA language and consumer architecture offer no support for logical variables. This is because we are restricted to ground facts. To illustrate the usefulness of logical variables, consider the time-honored *append* program in Prolog [Clocksin and Mellish, 1984], [Sterling and Shapiro, 1986]. Without logical variables, we can only ask questions of the form: *append([1],[2,3],[1,2,3])*. Logical variables stand for computed answers and allow us to "run" the program in many ways—as a generator of solutions rather than as just a test of solutions. For example, in Prolog we can solve, *append([1],[2,3],A)* and get back that *A=[1,2,3]*. In fact, if we give Prolog *append(X,Y,[1,2,3])* we get all four possible answers back as values for X and Y. Hence, by employing logical variables, we can exploit the *denotational* aspect of logic programs through reversible execution.

Only forward reasoning is possible. The consumer architecture and ATMS are limited to forward reasoning only. This is because the ATMS relies on each derived fact having a *justification*—a set of antecedent facts. Hence, the direction of inference must be the same as the direction of *logical support*. In forward chaining, we reason from antecedents to consequences, but in backward chaining we go the other way—from consequences to antecedents. Hence, we have no justifications (until the problem solving is complete).

II. Introducing FORLOG

FORLOG raises the level of programming the consumer architecture to that of a powerful programming language, while retaining all the advantages afforded by the ATMS. This is achieved by significantly extending the CA language and the underlying consumer architecture to overcome the problems identified above. The CA language is generalized to include most of first order logic, including existentially-quantified variables and arbitrary term structures. The consumer architecture is correspondingly extended to support these new features. Logical variables are implemented through their logical equivalent, Skolem constants, while unification is replaced by equality reasoning. Finally, FORLOG supports backward reasoning by reformulating it into a form of forward reasoning that yields identical behavior (but different logical semantics).

A. FORLOG Syntax

A FORLOG program is a set of well-formed formulas similar to CA language, except:

Each P_i is a disjunction of the form

$$F_1 \lor F_2 \lor \ldots \lor F_n,$$

where each F_j is an *existentially quantified* formula of the form

$$\exists \overline{y}\, P_1 \wedge P_2 \wedge \ldots \wedge P_m$$

or

$$G.$$

The basic formula allows both the left- and right-hand sides as disjunctions of existentially-quantified conjunctions (with nesting permitted to any depth). The ground literal is replaced by G—a predicate that may contain constants, Skolem constants and arbitrarily term structures.

B. FORLOG example

To clarify our description of FORLOG, we first give an example of FORLOG running the ubiquitous *append* program.

Consider the following FORLOG program, which is derived from the Prolog *append* program.

$$\forall x,y,z\ append(x,y,z) \supset (x=[\,]\wedge y=z)\vee \exists x_1,x_r,z_r\ x=[x_1|x_r]\wedge z=[x_1|z_r]\wedge append(x_r,y,z_r) \quad (4)$$

$$append([1],[2,3],sk_1) \quad (5)$$

The symbol sk_1 is a Skolem constant. From this program, FORLOG generates the following inferences.

By satisfying (4) with (5), FORLOG obtains

$$([1]=[\,]\wedge[2,3]=sk_1)\vee (\exists x_1,x_r,z_r\ [1]=[x_1|x_r]\wedge sk_1=[x_1|z_r] \wedge append(x_r,[2,3],z_r)) \quad (6)$$

The first branch of this disjunction is obviously false, because the list [1] is not nil. Hence, FORLOG can infer that the second branch is correct. Because the *cons* function is a constructor, we can infer from $[1]=[x_1|x_r]$ that $x_1=1$ and $x_r=[\,]$, and we can substitute these values for all occurrences of the existential variables x_1 and x_r. Unfortunately, we cannot infer the value of z_r, but we can Skolemize it to be sk_2. This reasoning produces

$$sk_1=[1|sk_2] \quad (7)$$

$$append([\,],[2,3],sk_2) \quad (8)$$

Formula (8) is, in a way, a recursive subgoal, and it can be applied to axiom (4) in the same way as the original assertion (5). When this is done, FORLOG obtains

$$([\,]=[\,]\wedge[2,3]=sk_2)\vee (\exists x_1,x_r,z_r\ [\,]=[x_1|x_r]\wedge sk_2=[x_1|z_r] \wedge append(x_r,[2,3],z_r)) \quad (9)$$

This time, the second branch of the disjunction can be ruled out, for it is impossible for $[x_1|x_r]$ to ever be equal to [ ]. In the first branch of the disjunction, [ ] = [ ] tells us nothing, but the remaining fact

$$[2,3]=sk_2 \quad (10)$$

completely determines the value of sk_1. Indeed, from assertions (7) and (10), FORLOG infers that

$$sk_1=[1,2,3]. \quad (11)$$

This example demonstrates that FORLOG can execute *append* in much the same way that Prolog does. However, where Prolog would use "logical variables," FORLOG uses Skolem constants, and where Prolog would use unification, FORLOG employs equality reasoning.

Thus, we can do "backward chaining" in FORLOG but in such a way that the direction of inference is the same as the direction of logical support. Hence, we have satisfied one of the constraints of the consumer architecture. We obtain the FORLOG *append* program directly from the Prolog program by first taking the *completion* [Clark, 1978]. The basic idea of the completion operation is to find all the *known* ways of proving a predicate P (i.e., all the clauses with *append* as head) and assert that these are the *only* way to prove P. Once we have done this we can, without affecting the semantic content, change the implication (from clauses to head) into an equivalence ($\equiv$). The FORLOG program is simply the other half of this equivalence (compared with Prolog). Hence, through this method, we can reverse the implication and align it with the direction in which we wish to reason. For more details, see [Corpron *et al.*, 1987] and [Corpron, 1987].

III. Implementing FORLOG in the Consumer Architecture

Before we describe how FORLOG is implemented, we review the requirements imposed on a problem solver by the consumer architecture:

1. If the antecedent pattern of any implication is satisfied by a set of facts in the global database, then the consequent must be detached.
2. Detaching a consequent must generate all the relevant consumers, each having a form suitable for satisfying other antecedent patterns and each justified by all the antecedent facts.

The following section describes how the FORLOG problem solver satisfies these requirements.

A. FORLOG and the Consumer Architecture

Running a FORLOG program entails applying a restricted theorem prover to the set of FORLOG expressions. The theorem prover basically applies *modus ponens*—detaching consequences when the antecedents are satisfied. In other words, we *implement* this theorem prover in the consumer architecture. In our description of FORLOG below, we emphasize both logical and implementation issues.

First, consider point 1 above, the satisfaction of antecedents. In both FORLOG and the CA language, an antecedent pattern consists of a disjunction of conjunctions of predicates, nested to any depth. We need only consider a DNF form, since any deeply nested expression can always be "multiplied out" to DNF form. Each disjunct can easily be implemented as a separate conjunctive pattern in a RETE network [Forgy, 1982]. When we are dealing only with ground facts, the RETE implementation provides an efficient method of determining when patterns are satisfied. However, because FORLOG supports Skolem

constants, this basic mechanism is insufficient. Consider a simple example:

$$\forall s\ P(s) \land Q(s) \supset Z(s) \tag{12}$$

If we know the facts $P(2)$ and $Q(2)$, we know that (12) is satisfied. But if we only know $P(3)$ and $Q(sk1)$, we cannot directly determine if (12) is satisfied. If at a later time the problem solver determines that $sk1 = 3$, then (12) is satisfied with support, $sk1 = 3$, $P(3)$ and $Q(sk1)$. Hence, we must supplement the simple RETE implementation with an *equality system* whose responsibility is to determine whether antecedent patterns are satisfied through equality information such as $sk1 = 3$.

In both FORLOG and the CA language, a detached consequent is always converted into a collection of atomic formulas before being asserted into the database. This policy serves to simplify the subsequent task of determining which implications are triggered by these assertions.

In the CA language, this process is straightforward, since we only have universal quantification. Again we need only consider the DNF form. Each disjunct of the detached consequent will be a conjunction of ground atomic formulas since the literals will be completely instantiated with constants from the antecedent satisfaction. Each of these ground facts produces a single consumer. There is a complication when we have more than one disjunct, since we may not know which of the disjuncts are true. Consider the following example:

$$\forall w\ adult(w) \supset woman(w) \lor man(w) \tag{13}$$

If we know $adult(Chris)$, we cannot determine which disjunct is true, so we must *assume* both are true. In other words we apply the following axiom:

$$a \supset b \lor c \ \vdash\ a \ \supset\ \begin{bmatrix} Choose\{B, C\} \\ a \land B \supset b \\ a \land C \supset c \end{bmatrix} \tag{14}$$

where upper case letters denote new assumptions and $Choose\{B, C\}$ informs the ATMS that $B \lor C$ is true. Hence, we create a new assumption for each disjunct and, since the antecedent is satisfied, put two consumers on the agenda:

	Assertion	Justification
1)	$woman(Chris)$	$(A\,B)$
2)	$man(Chris)$	$(A\,C)$,

(15)

where A is the ATMS-node of $adult(Chris)$ and B stands for "assume that Chris is a woman" and C stands for "assume that Chris is a man."

In FORLOG, detaching consequents is significantly more complex because of the use of existential variables. There is insufficient space to describe how all consequents in FORLOG are detached, we will simply follow the *append* example introduced in section B. (see [Dietterich, Corpron and Flann, 1987]).

When we detach the consequent of the append example (4), we first try to determine which of the disjuncts is true. To do this we apply the following axiom:

$$a \lor b \vdash (\neg a \supset b) \land (\neg b \supset a). \tag{16}$$

If we know one branch of a disjunction is false—the other must be true. By applying this axiom, FORLOG can avoid the unnecessary generate and test involved in applying (14). In step (9) of the example, we determined that the first branch of the disjunction was true and formed the consumer given in step (10). In step (6) the second disjunct was ruled in. Here we first perform two unifications, one of which produced the consumer (7) and then instantiated the append form to obtain the consumer (8).

To perform these unifications and determine the disjunct, FORLOG applies the following axioms: If f and g are distinct constructors, then the following *identity theory* must hold:

$$\begin{gathered} f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m) \\ x_1 \neq y_1 \lor \ldots \lor x_n \neq y_n \supset f(x_1, \ldots, x_n) \neq f(y_1, \ldots, y_m) \\ f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \supset x_1 = y_1 \land \ldots \land x_n = y_n \\ f(x) \neq x \quad f(x) \text{ is any term in which } x \text{ is free} \end{gathered} \tag{17}$$

In other words, the entities denoted by two constructors are equal if and only if all of their components are equal (unifiable).

In general, detaching a consequent in FORLOG involves a complex sequence of actions. In the *append* example, we first determine the branch. Then we may perform some unifications and, depending upon the unifications, create Skolem constants. Finally we produce consumers. Rather than determine which steps to perform each time we detach this consequent, we "compile" the *append* implication into a form that will automatically perform these actions each time we detach. This form is described in the next section.

B. FORLOG implementation

FORLOG implications are compiled into a data-flow like language that has much in common with the RETE network [Forgy, 1982] and the Warren Abstract Machine (WAM, [Warren, 1983]). In general, an implication is compiled into a network. The roots are input patterns that are satisfied by facts in the data base, while the leaves specify consumers to be created. This is illustrated by the network constructed for *append* given in Figure 2[2]. Execution consists of passing jobs down the "wires" from top to bottom. Jobs get created at the top when input patterns are satisfied by the global database, and get "consumed" at the leaves when they are used to create consumers. The consumers created are then run by the consumer architecture and satisfy further input patterns, thus generating new jobs. Hence, FORLOG is implemented as a data flow machine whose network is "sliced." Each slice represents a global communication, via the database, between output and input nodes in the network.

To illustrate this data-flow language, we continue with the *append* example presented in Section B. At step (8) when

(APPEND NIL (CONS 2 (CONS 3 NIL)) SK2)

[2] Universal variables are denoted by $ prefixes, existential variables are denoted by # prefixes. The basic constructor in FORLOG is the function CONS. Finally, # denotes anything.

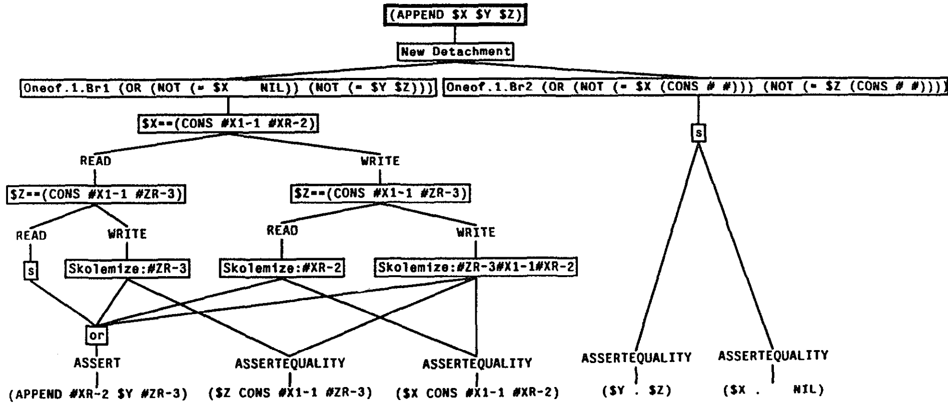

Figure 2: Data-flow implementation of FORLOG *Append*

is asserted, the pattern (APPEND $X $Y $Z) is satisfied, creating the following data-flow job:

bindings : (($X . NIL)($Y . (CONS 2 (CONS 3 NIL)))
($Z . SK2))
support : (D), (18)

where D is the ATMS-node of (8). This job passes through the New Detachment node (which will be discussed later) to the Oneof nodes. The oneof nodes allow us to simply determine at run time the correct disjunct. To do this, we instantiate the expressions in the nodes with the bindings on the job, and apply the identity theory (17). In this example, the resulting expression in the right-most oneof node is

(OR (NOT (= NIL (CONS # #)))
(NOT (= SK2 (CONS # #)))), (19)

which is true, because NIL does not equal a constructor. Hence, this job is passed down the right-most branch. When the job reaches the leaves, we create a consumer that asserts the equality between the binding on $Y and the binding on $Z, corresponding to step (10) above.

To demonstrate how unifications are compiled and run, we follow the first *append* goal assertion (5):

(APPEND (CONS 1 NIL) (CONS 2 (CONS 3 NIL)) SK1).

In this case, the left-most Oneof node is satisfied through

(OR (NOT (= (CONS 1 NIL) NIL))
(NOT (= (CONS 2 (CONS 3 NIL)) SK1))), (20)

and the data-flow job passes down the network to the left. Recall from the definition of append (4) that the second disjunct involves unification of both $X and $Z. Each unification can be used in two ways: in *read mode* or in *write mode* [Warren 1983]. Read mode corresponds to unifying the pattern with *existing* structure, for example, in the first unify node $X,

(= (CONS 1 NIL) (CONS #X1-1 #XR-2)).

Write mode corresponds to constructing new structure, for example, in the second unify node $Z,

(= SK1 (CONS 1 #ZR-3).

The compiler enumerates the possible combinations and constructs the decision tree illustrated in Figure 2. In this case, the job passes down the decision tree to the Skolemize #XR-2 node, where the new Skolem constant is generated and added to the bindings of the data-flow job. Finally the job is propagated to both the leaves, leading to the consumers given in (7) and (8).

Note how the network is shared between the different detachments. To ensure detachments are separated from each other, the New Detachment node generates a unique name that is associated with each data-flow job that passes through.

This example does not demonstrate how FORLOG handles the case when we are unable to determine the disjunct. For example, if we assert

(APPEND SK3 SK4 (CONS 1 NIL)) (21)

and evaluate the instantiated expressions of the Oneof nodes, we are unable to determine directly which disjunct is true. There is not enough information available to *completely* evaluate these equalities. FORLOG therefore *partially* evaluates them and sets up "listeners" that will trigger when the evaluation can proceed further. In other words, we are implementing the equality axiom of substituting equals for equals by rewriting the *patterns* rather than rewriting the facts. For example, consider the first disjunct of the right-most oneof node following (21):

(NOT (= SK3 (CONS # #))).

The partial evaluator translates this (through the use of the equality axiom) into:

(AND (= SK3 $1) (NOT (= $1 (CONS # #)))).

The first clause acts as a "listener" for any object that is asserted equal to SK3, while the second clause tests whether the object is not a constructor. This process is implemented by simply extending the data-flow network at run time. The listener becomes a new input pattern that connects to the right-most oneof node.

If we later determine a value for SK3,

(= SK3 NIL) (22)

The listener is satisfied, and a job created and passed down

the network, eventually producing the following consumer:

Assertion	Justification	
(= SK4 (CONS 1 NIL))	$(C\ D)$,	(23)

where C is the ATMS-node of (21) and D is the ATMS-node of (22).

This value of SK3 ((22) above) could come directly from an assertion or from equality reasoning. For example, the system may first determine that another Skolem constant, say SK10. is equal to NIL. then determine that SK3 equals SK10. Hence, to ensure that point 1 in Section III. is satisfied, the supporting equality system needs to apply the reflexive and transitive laws of equality.

The problem may be so unconstrained that values for SK3 and SK4 are not available. In this case, FORLOG enumerates both branches by applying the method presented in (14). Hence, FORLOG can exploit the ATMS to explore alternative solutions in any order.

IV. Conclusion

FORLOG is a forward chaining logic programming system that combines the flexible search and assumption handling of the ATMS with the clean denotational semantics, expressive power, and reversibility of logic programming.

This was achieved by extending deKleer's simple CA language to include much of full, first order logic and by augmenting the problem solving architecture so that Skolem constants can be supported.

FORLOG is completely implemented and running in Interlisp-D. Problems solved include the bulk of the Prolog examples given in the early chapters of [Sterling and Shapiro, 1986] and some simple constraint propagation problems such as those in [Steele, 1980]. In addition, a small expert system for designing cloning expriments has been constructed in FORLOG.

In this paper we have used a rather pathological, but well known example—append. FORLOG is currently being applied to solving more complex problems including mechanical design problems. In this domain, FORLOG's ability to both manipulate partial designs (through the use of Skolem constants) and to reason simultaneously with multiple contradictory designs, gives it a significant advantage over other logic programming systems [Dietterich and Ullman, 1987].

Acknowledgments

This research was supported in part by the National Science Foundation under grants DMC-8514949 and IST-8519926 and by a contract from Tektronix, Inc. We thank both Colin Gerety, who implemented the ATMS, and the students of the spring 1986 Non-monotonic Reasoning class who used FORLOG for their projects. We also thank Caroline Koff and Ritchey Ruff for their valuable comments on earlier drafts of this paper.

References

[Clark 1978] Negation as Failure. In *Logic and Databases,* H. Gallaire and J. Minker Eds. Plenum Press, New York, pp. 293-322.

[Clocksin, and Mellish, 1984] *Programming in Prolog.* Berlin: Springer-Verlag.

[Corpron, 1987] Disjunctions in Forward-Chaining Logiç Programming. Rep. No. 87-30-1. Computer Science Department, Oregon State University.

[Corpron, Dietterich, and Flann, 1987] Forthcoming. Viewing Forward Chaining as Backward Chaining.

[deKleer, 1986a] An Assumption-based TMS. *Artificial Intelligence,* 28 (2) pp. 127-162.

[deKleer, 1986b] Extending the ATMS. *Artificial Intelligence,* 28 (2) pp. 163-196.

[deKleer, 1986c] Problem-solving with the ATMS. *Artificial Intelligence,* 28 (2) pp. 197-224.

[Dietterich and Ullman, 1987] FORLOG: A Logic-based Architecture for Design. Rep. No. 86-30-8. Computer Science Department, Oregon State University.

[Dietterich, Corpron and Flann, 1987] Forthcoming. Forward chaining Logic Programming in FORLOG.

[Forgy, 1982] RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence,* 19.

[Steele, 1980] The definition and implementation of a computer programming language based on constraints. Doctoral dissertation. Massachusetts Institute of Technology.

[Sterling, and Shapiro, 1986] *The Art of Prolog,* MIT Press, Cambridge, Mass.

[Warren, 1983] An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Stanford, October 1983.