Adding Forward Chaining and Truth Maintenance to Prolog

Tim Finin
Rich Fritzson
Dave Matuszek

Paoli Research Center
Unisys Corporation
Paoli, Pennsylvania

Contact: Paoli Research Center, Unisys, PO Box 517, Paoli, PA 19301. finin@prc.unisys.com (215-648-7446); fritzson@prc.unisys.com (215-648-7606); dave@prc.unisys.com (215-648-7489).

Topic: Principles, knowledge representation and reasoning tools.

Abstract: This paper describes P_{fc} , a simple package which supplies a *forward chaining* facility in Prolog. P_{fc} is intended to be used in conjunction with ordinary Prolog programs, allowing the programmer to decide whether to encode a rule as a forward-chaining P_{fc} rule or a backward chaining Prolog one. Like other logic programming languages, P_{fc} programs have a declarative interpretation as well as clear and predictable procedural one. A truth maintenance system is built into P_{fc} system which maintains consistency as well as makes derivations available for applications. Finally, P_{fc} is designed to be practical, being relatively efficient and fairly unobtrusive.

Status: Research tool.

Domain: This paper describes a domain-independant tool.

Language: Prolog.

Effort: approximately 1/2 person year.

1 Introduction

Prolog, like most logic programming languages, offers backward chaining as the only reasoning scheme. It is well known that sound and complete reasoning systems can be built using either exclusive backward chaining or exclusive forward chaining [19]. Thus, this is not a theoretical problem. It is also well understood how to "implement" forward reasoning using a backward chaining system and vice versa. Thus, this need not be a practical problem. In fact, many of the logic-based languages developed for AI applications [14, 7, 20, 11] allow one to build systems with both forward and backward chaining rules.

There are, however, some interesting and important issues which need to be addresses in order to provide the Prolog programmer with a practical, efficient, and well integrated facility for forward chaining. This paper describes such a facility, P_{fc}, which we have implemented in standard Prolog.

The P_{fc} system is a package that provides a forward reasoning capability to be used together with conventional Prolog programs. The P_{fc} inference rules are Prolog terms which are asserted as facts into the regular Prolog database. For example, Figure 1 shows a file of P_{fc} rules and facts which are appropriate for the ubiquitous kinship domain.

In the next section of this paper we will review the nature of forward chaining, motivations for using it and some of the issues one must address in a forward chaining system. We will then give an informal, example-based presentation of the P_{fc} language in the third section. The fourth section will give some more detailed examples of the use of P_{fc}. Some of the implementation details are described in the fifth section. This paper concludes with a final section which summarizes our experiences and plans for the future.

2 Forward Chaining

It is well known that one can define sound and complete inferencing systems which rely solely on either forward or backward reasoning. This is discussed in introductory AI texts [19]. It is equally well understood that there are good reasons why one might wish to rely on one or the other or a mixed strategy in practice [15, 23]. This section reviews some of the reasons why one might want to apply certain knowledge in the form of forward-chaining rules and looks at some of the issues surrounding the design of a forward chaining facility. In particular, we will discuss the importance of a *truth maintenance system* in a forward chaining system.

```
spouse(X,Y) <=> spouse(Y,X).
spouse(X,Y),gender(X,G),{otherGender(G,G2)}=>gender(Y,G2).
gender(P,male) <=> male(P).
gender(P,female) <=> female(P).
parent(X,Y),female(X) <=> mother(X,Y).
parent(X,Y),parent(Y,Z) => grandparent(X,Z).
grandparent(X,Y),male(X) <=> grandfather(X,Y).
grandparent(X,Y),female(X) <=> grandmother(X,Y).
mother(Ma,Kid),parent(Kid,GKid) => grandmother(Ma,GKid).
grandparent(X,Y),female(X) <=> grandmother(X,Y).
parent(X,Y),male(X) <=> father(X,Y).
mother(Ma,X),mother(Ma,Y),{X\==Y} => sibling(X,Y).
```

Figure 1: **Examples of P_{fc} Rules which represent common kinship relations**

2.1 Characterization of Forward Chaining

The terms *forward* and *backward* chaining are used somewhat loosely within the AI community, covering such disparate things as OPS5-like production systems to resolution-based theorem provers. In general, when one speaks of a forward chaining rule, one has in mind a rule of the form:

$$P_1P_2\dots P_n \rightarrow Q_1Q_2\dots Q_n$$

Informally, the P_i on the *left hand side* (lhs) of the rule form a set of conditions which, if satisfied, enable the rule for firing. When fired, the Q_i on the rule's *right hand side* (rhs) specify a set of actions which are to be carried out. Typically, the conditions correspond to the presence of assertions in a database of facts. A condition is satisfied if it matches some particular fact in this database. There is more variability on the meaning and character of an action in the rhs of a rule. In some systems, such as OPS5 [10, 17], these can be arbitrary evaluable expressions. In others, such as MRS [7], the Q_i are propositions which are to be added to the fact database. Even in this latter case, systems differ as to whether the newly derived facts are persistent and are recorded into the global database [7, 14] or are held in a temporary extension to the permanent database (as in a list) [18].

Some of the ways in which the meaning of a forward chaining rule can vary are:

- Are the constituents of the rule's rhs *actions* to be performed or *propositions* to be added to the database?
- When the addition of a new fact to the database triggers some rules, should all of the rules be run or just one? If more than one is run, should the order be important?
- Should retracting database assertions be allowed?
- If retractions are allowed, how should they effect partially triggered rules?
- If retractions are allowed, should we perform "truth maintenance"?

Given that we are trying to provide a forward chaining facility for Prolog, some of these choices are fairly clear cut. The particulars of the package that we have designed are described in the sections to follow.

2.2 Motivations for Forward Chaining

There are a number of situations in which a forward chaining control strategy is preferred over a backward chaining one. We will briefly mention the major ones. We will assume that we are discussing a forward chaining system in which the rhs's of rules specify necessarily true facts which are to be asserted into the global database and remain there until explicitly retracted.

The usual reasons for choosing a forward reasoning strategy stem from the particular nature of the problem being solved. These include:

- *space/time tradeoffs.* If a problem involves solving the same goal frequently, it may be more efficient to express the rules for deducing that goal as forward chaining rules. This reduces the computation while increasing the need for static memory.
- *the shape of the inferential space.* Forward chaining tends to involve less search than backward chaining when the ratio of premises to conclusions is high.
- *avoiding long deductive chains.* Forward chaining is useful for avoiding long (or even infinite) deductive chains.
- *drawing all possible inferences.* Many problems require one to draw all possible inferences from a set of axioms. Forward chaining is an efficient way to do this.
- *alerting.* A related situation is one in which we want to ensure that certain deductions are made *as soon as possible*, e.g. for purposes of alerting or monitoring.

There are other reasons to consider using a forward chaining control structure, one of which we will briefly mention. It is sometimes very convenient to construct a logic program in which a subset of the database acts as the interface between two independent and "concurrent" subsystems. As an example, consider designing a system in which one subsystem maintains a model of a set of objects and their locations on a plane and another is responsible for maintaining a graphical display of these objects. One way to (loosely) couple these two subsystems is to have the first make database assertions which represent the properties of the objects and their locations. The second subsystem has a set of forward chaining rules which are triggered by the first subsystem's assertions and cause the display to be updated. This is similar to the notion of an *active value* found in many Lisp-based systems (e.g. Loops [2], KEE [1]). This was one of the chief ways in which the forward chaining (thante) rules of MICRO-PLANNER [22] were used.

2.3 Truth Maintenance

A forward chaining facility in a logic-oriented, rule-based system (as opposed to a production-rule oriented system like OPS5) has a special need for a *truth maintenance system* (TMS). Although a TMS can be used with a deductive language using any sort of reasoning strategy, most of the work in TMS systems is associated with forward reasoning [6, 5, 3, 16, 13].

There are really two closely related reasons for this need: one *"external"* and the other *"internal"*. The first (the "external") is that we want, in general, to keep the database consistent. Consider a situation in which a forward chaining rule has made an inference based on the state of the database at some time and thereby adds a new fact to the database. If the database changes at some later time such that the earlier inference is no longer valid, then the conclusion may have to be withdrawn and the assertion removed from the database. This is not a problem in a pure backward-chaining system since conclusions are always drawn (and re-drawn, if necessary) with respect to the current state of the database.

The second ("internal") need for a TMS has to do with the particular implementational strategy we have employed. A rule with a conjunction on its lhs (i) can be rewritten as shown below (ii):

```
(i) P, Q, R => S
(ii) P => (Q => (R => S))
```

That is, as a rule with a atomic lhs (P) which, when triggered, adds a new rule which monitors for the remaining conjuncts from the original rule. In such a scheme, it is important to keep track of these *derived* rules which represent partially instantiated or triggered rules. If a database assertion supporting a partially triggered rule is erased, then the rule need to be killed. This ensures that a rule will fire only if *all* of its required conditions are simultaneously true.

A truth maintenance system is useful for reasons besides keeping the knowledge base consistent. A TMS typically records the proof derivations of all of the facts in the database. This is useful in a variety of ways, such as generating explanations, abductive reasoning and debugging.

3 The P_{fc} language

This section describes P_{fc} . We will start by introducing the language informally through a series of examples drawn from the domain of kinship relations. This will be followed by an example and a description of some of the details of its current implementation.

Overview

The P_{fc} package allows one to define forward chaining rules and to add ordinary Prolog assertions into the database in such a way as to trigger any of the P_{fc} rules that are satisfied. An example of a simple P_{fc} rule is:

```
gender(P,male) => male(P)
```

This rule states that when a fact unifying with $gender(P, male)$ is added to the database, then the fact $male(P)$ is true. If this fact is not already in the database, it will be added. In any case, a record will be made that the validity of the fact $male(P)$ depends, in part, on the validity of this forward chaining rule and the fact which triggered it. To make the example concrete, if we add $gender(john, male)$, then the fact $male(john)$ will be added to the database unless it was already there.

In order to make this work, it is necessary to use the predicate *add/1* rather than *assert/1* in order to assert P_{fc} rules and any facts which might unify with a goal in the lhs of a P_{fc} rule.

Compound Rules. A slightly more complex rule is one in which the rule's left hand side is a conjunction or disjunction of conditions:

```
parent(X,Y), female(X) => mother(X,Y)
mother(X,Y); father(X,Y) => parent(X,Y)
```

The first rule has the effect of adding the assertion $mother(X,Y)$ to the database whenever $parent(X,Y)$ and $female(X)$ are simultaneously true for some X and Y. Again, a record will be kept that indicates that any fact $mother(X,Y)$ added by the application of this rule is justified by the rule and the two triggering facts. If any one of these three clauses is removed from the database, then all facts solely dependent on them will also be removed. Similarly, the second example rule derives the parent relationship whenever either the mother relationship or the father relationship is known.

In fact, the lhs of a P_{fc} rule can be an arbitrary conjunction or disjunction of facts. P_{fc} handles a rule like (i) by putting it into conjunctive normal form, resulting in the two rules given in (ii) below:

```
 (i) P, (Q;R), S => T
(ii) P,Q,S => T
     P,R,S => T
```

Bi-conditionals. P_{fc} has a limited ability to express bi-conditional rules, such as:

```
mother(P1,P2) <=> parent(P1,P2), female(P1).
```

In particular, adding a rule of the form `P<=>Q` is the equivalent to adding the two rules `P=>Q` and `Q=>P`. The limitations on the use of bi-conditional rules stem from the restrictions that the two derived rules be valid horn clauses. This is discussed in a later section.

Conditioned Rules. It is sometimes necessary to add some further condition on a rule. Consider the following definition of sibling: "Two people are siblings if they have the same mother and the same father. No one can be his own sibling." This definition could be realized by the following P_{fc} rule

```
mother(Ma,P1), mother(Ma,P2), {P1\==P2},
father(Pa,P1), father(Pa,P2)
 => sibling(P1,P2).
```

Here we must add a condition to the lhs of the rule which states the the variables $P1$ and $P2$ must not unify. This is effected by enclosing an arbitrary Prolog goal in braces. When the goals to the left of such a bracketed condition have been fulfilled, then it will be executed. If it can be satisfied, then the rule will remain active, otherwise it will be terminated.

Negation. We sometimes want to draw an inference from the absence of some knowledge. For example, we might wish to encode the default rule that a person is assumed to be male unless we have evidence to the contrary:

```
person(P), ~female(P) => male(P).
```

A lhs term preceded by a $\sim$ is satisfied only if *no* fact in the database unifies with it. Again, the P_{fc} system records a justification for the conclusion which, in this case, states that it depends on the absence of the contradictory evidence.

As a slightly more complicated example, consider a rule which states that we should assume that the parents of a person are married unless we know otherwise. Knowing otherwise might consist of either knowing that one of them is married to a yet another person or knowing that they are divorced. We might try to encode this as follows:

```
parent(P1,X), parent(P2,X), {P1\==P2},
~divorced(P1,P2),
~spouse(P1,P3), {P3\==P2},
~spouse(P2,P4), {P4\==P1}
  =>
spouse(P1,P2).
```

Unfortunately, this won't work. The problem is that the conjoined condition `~spouse(P1,P3), {P3\==P2}` does not mean what we indended it to — that there is no $P3$ distinct from $P2$ that is the spouse of $P1$. Instead, it means that $P1$ is not married to any $P3$. We need a way to move the qualification `P3\==P2` inside the scope of the negation. To achieve this, we introduce the notion of a qualified goal. A lhs term P/C, where P is a positive atomic condition, is true only if there is a database fact unifying with P and condition C is satisfiable. Similarly, a lhs term $\sim P/C$, where P is a positive atomic condition, is true only if there is no database fact unifying with P for which condition C is satisfiable. Our rule can now be expressed as follows:

```
parent(P1,X), parent(P2,X)/(P1\==P2),
~divorced(P1,P2),
~spouse(P1,P3)/(P3\==P2),
~spouse(P2,P4)/(P4\==P1)
  =>
spouse(P1,P2).
```

Procedural Interpretation. Note that the procedural interpretation of a P_{fc} rule is that the conditions in the lhs are checked *from left to right.* One advantage to this is that the programmer can chose an order to the conditions in a rule to minimize the number of partial instantiations. Another advantage is that it allows us to write rules like the following:

```
at(Obj,Loc1),at(Obj,Loc2)/{Loc1\==Loc2}
   => {remove(at(Obj,Loc1))}.
```

Although the declarative reading of this rule can be questioned, its procedural interpretation is clear and useful — "If an object is known to be at location *Loc*1 and an assertion is added that it is at some location *Loc*2, distinct from *Loc*1, then the assertion that it is at *Loc*1 should be removed."

The Right Hand Side of a Rule. The examples seen so far have shown a rules rhs as a single proposition to be "added" to the database. The rhs of a P_{fc} rule has some richness as well. The rhs of a rule is a conjunction of facts to be "added" to the database and terms enclosed in brackets which represent conditions/actions which are executed. As a simple example, consider the conclusions we might draw upon learning that one person is the mother of another:

```
mother(X,Y) => female(X), parent(X,Y), adult(X).
```

As another example, consider a rule which detects bigamists and sends an appropriate warning to the proper authorities:

```
spouse(X,Y), spouse(X,Z), {Y\==Z} =>
   bigamist(X),
   {format("~N~w is a bigamist, married
      to both ~w and ~w~n",[X,Y,Z])}.
```

Each element in the rhs of a rule is processed from left to right — assertions being added to the database with appropriate justification and actions being executed. If an action can not be satisfied, the rest of the rhs is not processed.

We would like to allow rules to be expressed as bi-conditional in so far a possible. Thus, an element in the lhs of a rule should have an appropriate meaning on the rhs as well. What meaning should be assigned to the conditional fact construction (e.g. P/Q) which can occur in a rule's lhs? Such a term in the rhs of a rule is interpreted as a *conditioned assertion*. Thus the assertion P/Q will match a condition $P\prime$ in the lhs of a rule only if P and $P\prime$ unify and the condition Q is satisfiable. For example, consider the rules that says that an object being located at one place is reason to believe that it is not at any other place:

```
at(X,L1) => not(at(X,L2))/L2\==L1
```

Note that a *conditioned assertion* is essentially a Horn clause. We would express this fact in Prolog as the backward chaining rule:

```
not(at(X,L2)) :- at(X,L1),L1\==L2.
```

The difference is, of course, that the addition of such a conditioned assertion will trigger forward chaining whereas the assertion of a new backward chaining rule will not.

The Truth Maintenance System

As discussed in the previous section, a forward reasoning system has special needs for some kind of *truth maintenance system.* The P_{fc} system has a rather straightforward TMS system which records justifications for each fact deduced by a P_{fc} rule. Whenever a fact is removed from the database, any justifications in which it plays a part are also removed. The facts that were justified by a removed justification are checked to see if they are still supported by some other justifications. If they are not, then those facts are also removed.

Such a TMS system can be relatively expensive to use and is not needed for many applications. Consequently, its use and nature are optional in P_{fc} and are controlled by the predicate $pfcTmsMode/1$. There are three possible cases:

- $pfcTmsMode(full)$ - The fact is removed unless it has *well founded support* (WFS). A fact has WFS if it is a given or is supported by the *user* or by a justification all of whose justificees have WFS. This is the most expensive mode, since determining if a fact has WFS requires detecting local cycles (see [15] for an introduction).
- $pfcTmsMode(local)$ - The fact is removed only if it has no supporting justifications.
- $pfcTmsMode(none)$ - The fact is never removed.

A fact is considered to be a *given* if it is found in the database with no visible means of support. That is, if P_{fc} discovers an assertion in the database that can take part in a forward reasoning step, and that assertion is not supported by either the user or a forward deduction, then a note is added that the assertion is assumed to be a given. This adds additional flexibility in interfacing systems employing P_{fc} to other Prolog applications. A fact is supported by the user if it was directly asserted into the database via an explicit call to the add/1 predicate.

For some applications, it is useful to be able to justify actions performed in the rhs of a rule. To allow this, P_{fc} supports the idea of declaring certain actions to be *undoable* and provides the user with a way of specifying methods to undo those actions. Whenever an action is executed in the rhs of a rule and that action is undoable, then a record is made of the justification for that action. If that justification is later invalidated (e.g. through the retraction of one of its justificees) then the support is checked for the action in the same way as it would be for an assertion. If the action does not have independent support, then P_{fc} tries each of the methods it knows to undo the action until one of them succeeds.

In fact, in P_{fc} , one declares an action as undoable just by defining a method to accomplish the undoing. This is done via the predicate $pfcUndo/2$. The predicate $pfcUndo(A1, A2)$ is true if executing $A2$ is a possible way to undo the execution of $A1$. For example, we might want to couple an assertional representation of a set of graph nodes with a display of them through the use of P_{fc} rules:

```
at(N,XY) => {displayNode(N,XY)}.
arc(N1,N2) => {displayArc(N1,N2}.

pfcUndo(displayNode(N,XY),eraseNode(N,XY)).
pfcUndo(displayArc(N1,N2),eraseArc(N1,N2)).
```

Limitations

The P_{fc} system has several limitations, most of which it inherits from its Prolog roots. One of the more obvious of these is that P_{fc} rules must be expressible as a set of horn clauses. The practical effect is that the rhs of a rule must be a conjunction of terms which are either assertions to be added to the database or actions to be executed. Negated assertions and disjunctions are not permitted, making the following rules ill-formed:

```
parent(X,Y) <=> mother(X,Y);father(X,Y)
male(X) <=> ~female(X)
```

Another restrictions is that all variables in a P_{fc} rule have implicit universal quantification. As a result, any variables in the rhs of a rule which remain uninstantiated when the lhs has been fully satisfied retain their universal quantification. This prevents us from using a rule like

```
father(X,Y), parent(Y,Z)  <=> grandfather(X,Z).
```

with the desired results. If we do add this rule and assert *grandfather(john,mary)*, then P_{fc} will add the two independent assertions *father(john,_)* (i.e. "John is the father of everyone") and *parent(_,mary)* (i.e. "Everyone is Mary's parent").

A final problem is associated with the use of the Prolog database. Assertions containing variables actually contain "copies" of the variables. Thus, when the conjunction

```
father(X,Y), parent(Y,Z) => grandfather(X,Z).
parent(X,Y), male(X) <=> father(X,Y).
parent(X,Y) <=> child(Y,X).
=> male(tom).
=> parent(tom,tim).
=> child(clare,tim).
=> father(tim,peter).
```

Figure 2: **A simple P_{fc} program in the kinship domain**

predicate abbreviations:
p=parent, f=father, m=male, gf=grandfather, c=child, s=fcSupports
deduced facts:

```
f(tom,tim).
c(tim,tom).
p(tim,clare).
p(tim,peter).
```

triggers:

```
1. pt(f(A,B),pt(p(B,C),rhs([gf(A,C)]))).
2. pt(p(A,B),pt(m(A),rhs([f(A,B)]))).
3. pt(f(A,B),rhs([p(A,B),m(A)])).
...
11. pt(p(clare,A),rhs([gf(tim,A)])).
```

tms relations:

```
1. s((user,user),(f(A,B),p(B,C)=>gf(A,C))).
2. s(((f(A,B),p(B,C)=>gf(A,C)),user),
     pt(f(A,B),pt(p(B,C),rhs([gf(A,C)])))).
3. s((user,user),(p(A,B),m(A)<=>f(A,B))).
...
34. s((f(tim,clare),pt(f(tim,clare),
                        rhs([p(tim,clare),m(tim)]))),
      m(tim)).
35. s((m(tim),pt(m(tim),rhs([f(tim,peter)]))),
      f(tim,peter)).
```

Figure 3: **The network of assertions produced by the simple kinship program**

```
add(father(adam,X)), X=able
```

is evaluated, the assertion `father(adam,_G032)` is added to the database, where _G032 is a new variable which is distinct from X. As a consequence, it is never unified with *able*.

4 Examples

This section gives two examples. The first uses the simple kinship domain to show the network of TMS assertions generated from a small set of assertions. Figure 2 shows a simple P_{fc} program and Figure 3 shows a part of the resulting network of assertions (we have abbreviated the names of the predicates and constants to make the depiction more compact.

Our second example shows how P_{fc} can be used to provide the representation and reasoning core of a simple diagnostic application. Figure 4 shows a standard diagnostic problem that has been used widely in the literature [8, 12, 4, 21]. The problem is to determine which components are faulty, given the description of the structure of the circuit and the observed input and output values. The general approach to this kind of problem is to represent the circuit (i.e. the devices, their behavior, and the connections) in the form of logical assertions and/or rules. The representation indicates (implicitly or explicitly) that some of the facts about the circuit are to be taken as assumptions (e.g. that a device behaves as intended or that an observation is correct).

If all of the information about the circuit (structure, device behavior and observations) is consistent, then there is no evidence of a problem and no diagnosis is needed. If there is an inconsistency, however, then one or more of the assumptions must be incorrect. The diagnostic problem, then, can be expressed in terms of finding "minimal" sets of assumptions to retract such that the resulting set of known facts and assumptions is consistent (see [9] for an overview of this approach).

Although P_{fc} was not specifically designed to solve diagnostic problems, it turns out to provide some of the necessary representation and reasoning capabilities. The P_{fc} system supplies a good way to describe a set of facts and to draw all possible conclusions from them (the deductive closure) and to record the derivations of these facts. The first capability is just what is needed to *simulate* the behavior of the circuit to be diagnosed and to identify the *conflicts* that arise. The second capability allows the derivations of these conflicts to be explored, looking for a set of assumptions to be retracted.

The predicates to represent such circuits are given in Figure 5. In this scheme, a device is represented by a symbol. To indicate that a device, d, is a kind of *adder*, we can add the assertion *isa(d,adder)*. The first P_{fc} rule in Figure 4 will then conclude that d exhibits the behavior of an adder (i.e. adding the assertion *behave(d,adder)*) unless it is known that d is defective (i.e. unless there is a fact matching *faulty(d)*). When the fact that d behaves like an adder is asserted, the fifth rule in this figure will add the constraints (expressed as P_{fc} rules) which relate d's inputs and output.

5 Implementation

This section briefly describes the current implementation of P_{fc}. The basic user predicates are *add/1* and *rem/1*. The *add/1* predicate adds a new P_{fc} fact or rule to the database, triggering any forward chaining. Adding a new rule involves putting the rule's lhs into a modified conjunctive normal form and then adding one or more *triggers* to the database. Conceptually, a trigger represents a demon which monitors the database, watching for the addition or removal of an assertion which can unify with its *head*. A trigger also has a *condition* which, if satisfiable, will enable the "evaluation" of its *body*. For example, the rule

```
father(X,Y),parent(Y,Z)=>grandfather(X,Z).
```

results in the following trigger being asserted into the database:

```
pt(father(A,B),
   true,
   pt(parent(B,C),
      true,
      rhs([grandfather(A,C)]))).
```

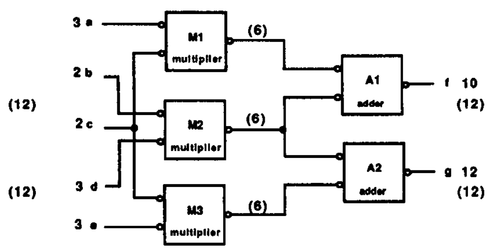

Figure 4: **A simple circuit to be diagnosed**

```
% Devices behave as they should unless they are faulty.
isa(X,Class), ~faulty(X) => behave(X,Class).

% a wire equates the values at its two ends.
wire(T1,T2) => (val(T1,V) <=> val(T2,V)).

% It is a conflict if a terminal has two different values.
val(T,V1), val(T,V2)/{\+V1=V2} => conflict(T).

% assume an observation is true.
observed(P), ~false_observation(P) => P.

% an adder's behaviour
behave(X,adder) =>
  (val(in1(X),I1),val(in2(X),I2) => {O is I1+I2},val(out(X),O)),
  (val(in1(X),I1),val(out(X),O) => {I2 is O-I1},val(in2(X),I2)),
  (val(in2(X),I2),val(out(X),O) => {I1 is O-I2},val(in1(X),I1)).

% a multiplier's behaviour.
behave(X,multiplier) =>
  (val(in1(X),I1),val(in2(X),I2) => {O is I1*I2},val(out(X),O)),
  (val(in1(X),I1),val(out(X),O) => {I2 is O/I1},val(in2(X),I2)),
  (val(in2(X),I2),val(out(X),O) => {I1 is O/I2},val(in1(X),I1)).

% a gizmo is the standard example circuit.
isa(X,gizmo) =>
   isa(m1(X),multiplier),
   isa(m2(X),multiplier),
   isa(m3(X),multiplier),
   isa(a1(X),adder),
   isa(a2(X),adder),
   wire(out(m1(X)),in1(a1(X))),
   wire(out(m2(X)),in2(a1(X))),
   wire(out(m2(X)),in1(a2(X))),
   wire(out(m3(X)),in2(a2(X))).
```

Figure 5: P_{fc} **rules which simulate the behavior of simple circuits composed of adders and multipliers**

Whenever a fact is added to the database (for the first time) all positive triggers with unifying heads are collected and fired. Firing a trigger means ensuring that its condition is satisfied and processing the body. The body can be another trigger, a conditional body, a "cut point", or the rule's rhs.

When the body of a trigger is another trigger, it is asserted into the database with a note that it's support comes from the initial trigger and the unifying fact. Thus, in the above example, when *father(tom,tim)* is asserted, the trigger

```
pt(parent(tim,C),
   true,
   rhs([grandfather(tom,C)]))
```

is added to the database with support coming from the original trigger and fact.

An item in the lhs of a rule can be an arbitrary condition wrapped in braces, as in:

```
age(P1,A1), age(P2,A2), {A1>A2} => older(P1,P2).
```

This provides additional flexibility in mixing forward and backward reasoning and also makes the semantics of bi-conditional rules sensible.

We are experimenting with a technique for pruning the tree of triggers which grows from a rule and a stream of facts which is being added to the database. This is analogous to the use of the *cut* operation in Prolog and other logic programming languages. For example, consider a rule which encodes the knowledge that a person is a parent if they have offspring. We could write this in P_{fc} as:

```
person(P),parent(P,_) => isParent(P).
```

However, this rule is somewhat redundant in that it records multiple justifications for the *isParent* conclusion. That is if a person has six children, then there will be six justifications for the conclusion. In many applications, it is desirable to "prune" away the other justifications, an operation similar to the "cut" in logic programming languages. In P_{fc} the "!" symbol represents such a pruning operation. We can write our rule as:

```
person(P),parent(P,_),! => isParent(P).
```

Whenever the "!" is encountered in a rule instance, all ancestor triggers "frozen". This effectively blocks any justifications beyond the first. If the first justification is removed by the tms system, the effective triggers will be "thawed".

Finally, the trigger which represents the last condition in a rule will have the rule's rhs as its body. Similarly, whenever a positive trigger is added to the database, it is "fired" for each extant fact in the database with which it unifies. Consider the following rule which contains a negated fact in the lhs:

```
parent(P1,K),spouse(P1,P2),~parent(P2,K)
  => stepParent(P2,K).
```

This rule would generate the following trigger:

```
pt(parent,
   true,
   pt(spouse(P1,P2),
```

```
        true,
        nt(parent(P2,K),
           true,
           rhs[stepParent(P2,K)])))
```

The *nt/3* term represents a *negative trigger* which is immediately satisfied if there is no unifying fact in the database. Whenever a fact is removed from the database, all negative triggers with unifying heads are gathered and, if their conditions are satisfiable, fired. Conversely, whenever a fact is added to the database, a search is made for justifications which include a negative trigger whose head unifies with the newly added fact. Any such justifications are then removed.

The support for conclusions is recorded by the *fcSupport/2* predicate. It has one of the following forms:

- *fcSupport((user,user),X)* - where X is a user asserted rule or fact.
- *fcSupport((Rule,user),Trigger)* - where Rule is user-asserted rule and Trigger is one of the resulting initial triggers.
- *fcSupport((Fact,Trigger),X)* - where Fact is an atomic fact, Trigger is a positive or negative trigger and X is a resulting fact or another trigger.

These assertions are hidden from the user in a shadow database.

Other predicates exist for finding the immediate facts and rules which support a given clause and for finding the set of "user asserted" facts and rules which support a clause. These can be used to construct the possible P_{fc} derivations of a clause.

6 Conclusions

This paper has described P_{fc} a forward chaining facility for Prolog. P_{fc} is intended to be used in conjunction with ordinary Prolog programs, allowing the programmer to decide whether to encode a rule as a forward-chaining P_{fc} rule or a backward chaining Prolog one. Like other logic programming languages, P_{fc} programs have a declarative interpretation as well as clear and predictable procedural one. A truth maintenance system is built into P_{fc} system which maintains consistency as well as makes derivations available for applications. Finally, P_{fc} is designed to be practical, being relatively efficient and fairly unobtrusive.

We have begun to experiment with P_{fc} are expecting to use it in several Prolog-based applications requiring a forward reasoning facility. There are a number of issues which we intend to examine in the near future. These include exploring additional ways to control forward reasoning; devloping techniques for the optimization and compilation of P_{fc} programs; and exploring the opportunities for the parallel execution of a "pure" subset of P_{fc}

In summary, we have found that the P_{fc} system effectively extends Prolog to enable the use of a mixed backward and forward reasoning strategy. This is done in a way that maintains the advantages of using Prolog (as opposed to a more general logic-based AI language) — simplicity, speed and portability.

References

[1] *KEE Reference Manual.* Intellicorp, level 3.0 edition, 1986.

[2] D. G. Bobrow and M. Stefik. *The Loops Manual.* Technical report KB-VLSI-81-13, Xerox PARC, 1981.

[3] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence,* 28:127–162, 1986.

[4] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence,* 32(1):97 - 130, 1987.

[5] Jon Doyle. The ins and outs of reason maintenance. In 8^{th} *International Conference on Artificial Intelligence,* pages 349–351, 1983.

[6] Jon Doyle. A truth maintenance system. *Artificial Intelligence,* 12(3):231–272, 1979.

[7] M. Genesereth et. al. *MRS Manual.* Technical Report, Stanford University, 1983.

[8] Randall Davis et. al. Diagnosis based on description of structure and function. In *Proc. National Conf. on Artificial Intelligence,* AAAI, CMU, Pittsburgh PA, August 1982.

[9] Tim Finin and Gary Morris. Abductive reasoning in multiple fault diagnosis. *Artificial Intelligence Review,* 3(2), 1989.

[10] C. L. Forgy. *The OPS5 User's manual.* Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, 1981.

[11] Rich Fritzson and Tim Finin. *Protem - An Integrated Expert Systems Tool.* Technical Report 84, Logic Based Systems, Paoli Research Center, Unisys, 1988.

[12] M. R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence,* 24:411–436, 1984.

[13] David A. McAllester. *Reasoning Utility Package User's Manual.* Technical Report, MIT Artificial Intelligence Laboratory, April 1982.

[14] Drew McDermott. *DUCK: A Lisp-Based Deductive System.* Technical Report, Computer Science, Yale University, 1983.

[15] Drew McDermott and Eugene Charniak. *Introduction to Artificial Intelligence.* Addison Wesley, 1985.

[16] Drew McDermott and Jon Doyle. Non-monotonic logic I. *Artificial Intelligence,* 13(1):41–72, 1980.

[17] Dennis Merritt. Forward chaining in Prolog. *AI Expert,* November 1986.

[18] Paul Morris. A forward chaining problem solver. *Logic Programming Newsletter,* Autumn 1981.

[19] Nils Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co., Palo Alto, California, 1980.

[20] Charles J. Petrie and Michael N. Huhns. *Controlling Forward Rule Inferences.* Technical Report ACA-AI-012-88, MCC, January 1988.

[21] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence,* 32(1):57 - 96, 1987.

[22] Gerald J. Sussman, Terry Winograd, and Eugene Charniak. *Microplanner Reference Manual.* Technical Report AIM 203a, MIT Artificial Intelligence Laboratory, 1971.

[23] Richard Treitel and Michael Genesereth. Choosing directions for rules. *Journal of Automated Reasoning,* 3(4):395-432, Dec. 1987.