Theories and Proofs in Fault Diagnosis

Ilyas Cicekli

Dept. of Comp. Eng. and Info. Sc., Bilkent University,
06533 Bilkent, Ankara, TURKEY,
ilyas@cs.bilkent.edu.tr

Abstract. This paper illustrates how theories (contexts), fail branches, and the ability to control the construction of proofs in MetaProlog play an important role in the expression of the fault diagnosis problem. These facilities of MetaProlog make it easier to represent digital circuits and the fault diagnosis algorithm on them. MetaProlog theories are used both in the representation of digital circuits and in the implementation of the fault diagnosis algorithm. Fail branches and the ability to control their construction play a key role during the construction of hypothesises to explain the fault in a given faulty circuit.

1 Introduction

Meta-level facilities in logic programming languages provide explicit representation of contexts (theories), statements (clauses), derivability relationship between theories and goals, and proofs. This explicit representation of meta-level objects and control knowledge may improve the expressive power of the language and help to shrink the search space of a goal by avoiding unnecessary searches.

Many systems having some kind of meta-level facility are presented in the literature [13]. Weyhrauch's FOL system [18] builds up contexts (theories) by declaring predicates, functions, constants and variables, and defining axioms. In that system, theorems are proved with respect to the axioms of a context and proofs are recorded. In the OMEGA system [2], a metalanguage defines the syntax of expressions and statements, viewpoints describe sets of assumptions, and the consequence concept formalizes derivability relationship between statements and viewpoints. The system developed by Lamma et al. [10] for the contextual logic programming [11] represents a set of Prolog clauses as a unit, and an ordered set of units as a context. Nadathur et al. [12] create a new context adding clauses in an implication goal to the current context in their system. Some other researchers in the logic programming community have sought meta-level facilities in meta-interpreters [14–16] based on Prolog. Even standard Prolog [6] has some meta-level facilities. The predicates *assert* and *retract* add and remove clauses from a system-wide database by destroying the old version of that database. The meta predicate *call* tries to prove an explicitly given goal with respect to the single system-wide database. There are no notions of contexts in standard Prolog.

MetaProlog is a meta-level extension of Prolog which is evolved from the research of Bowen and Kowalski [3,4]. In MetaProlog, theories are made explicit so that they can be manipulated just as other data objects in the system. Once theories are made explicit, deductions are made from these theories instead of a single system-wide database. The basic two-argument *demo* predicate in MetaProlog is used to represent the derivability relation between an explicitly represented theory and goal. Another meta-level facility in MetaProlog is dynamically-constructed proof trees. They are collected by the system when a goal is proved with respect to a theory by using the three-argument version of *demo* predicate. A given partially instantiated proof of a goal when the deduction of that goal is started may shrink the search space of that goal.

We implemented a compiler-based MetaProlog system [7,8] for efficient implementation of theories and derivability relation. This compiler-based MetaProlog system supports multiple theories and a fast context switching among the theories in MetaProlog. Since MetaProlog is an extension of Prolog, the Warren Abstract Machine [1,17], which is used in the implementation of Prolog, is extended to get efficient implementation meta-level facilities and this extension is called the Abstract MetaProlog Engine [8].

There can be many applications of meta-level facilities in a logic programming language. An obvious application of proofs is the explanation facility of an expert system. Collected proofs can be used to give justifications about the behavior of a rule based expert system. Sterling describes a meta-level architecture for expert systems in [16]. In [9], Eshghi shows how to use meta-level knowledge in a fault finding problem in logic circuits. Bowen [5] describes how to use meta-level programming techniques in knowledge representation.

In this paper, we chose the fault diagnosis problem as an application to demonstrate how multiple theories and fail branches in MetaProlog play a key role in the representation of this problem. This problem is problem chosen, because we use the most of the meta-level facilities in MetaProlog in its representation. Dynamically created multiple theories are used in the representation of logic circuits, hypothesises representing faulty circuits and the problem itself. Fail branches are used to construct the set of hypotheses describing possible faults in the given faulty circuit.

The next section explains the representation of theories and how they are created in the MetaProlog system. Section 3 explains derivability relations, creation of proofs and fail branches, and how the creation of proofs are controlled in the MetaProlog system. Section 4 illustrates how these meta-level facilities are used in the representation of the fault diagnosis problem.

2 MetaProlog Theories

Theories are the meta-level objects which are addressed firstly in many meta-level systems. They are made explicit in these meta-level systems so that they can be manipulated just as other data objects. Since they are explicitly represented, we can reason about them or we can discuss their characteristics. Since explicit

representations of theories and statements are available, the provability relation between them can also be defined explicitly.

In Prolog, there is only one theory, and all goals are proved with respect to this single theory. On the other hand, there can be more than one theory in MetaProlog at a certain time, so that a goal can be proved with respect to any of them. The same goal can also be proved with respect to a different theory in the MetaProlog system.

Since there is a single implicitly represented database in Prolog, ad hoc methods are used when there is a need to update this database. The builtin predicates *assert* and *retract* update the Prolog database to create a new version of this database by destroying the old version in favor of the new version. On the other hand, we do not need to destroy an old theory when we create a new one from that theory in the MetaProlog system.

Theories of the MetaProlog system are organized in a tree whose root is a distinguished theory, the ***base theory***. The base theory consists of all builtin predicates, and all other theories in the system are its descendants; i.e., all builtin predicates in the base theory can be accessed from all other theories in the system.

A new theory is created from an old theory by adding or dropping some clauses. The new theory inherits all the procedures of the old theory except for procedures explicitly modified during its creation. The system can still access both the new theory and the old theory. The following builtin predicates are used to create new theories in the MetaProlog system:

addto(OldTheory, Clauses, NewTheory)

dropfrom(OldTheory, Clauses, NewTheory)

The given clauses are added to (dropped from) the given old theory to create a new theory by the predicate *addto* (*dropfrom*). The variable *NewTheory* is bound to the internal representation of the new theory after the execution of one of these commands. Assume that *p* is a procedure in *NewTheory*. The clauses of *p* are exactly the same as the clauses of *p* in *OldTheory*, if *p* does not contain any clause in *Clauses*. Otherwise, the clauses of *p* in *NewTheory* consist of the clauses in *OldTheory* and *Clauses* which belongs to *p* if *NewTheory* is created by the *addto* predicate. If *NewTheory* is created by the *dropfrom* predicate, the clauses of *p* contains all clauses of *p* in *OldTheory* except the clauses which appear in *Clauses*.

The first argument of the *addto (dropfrom)* predicate is a theory (a variable bound to the internal representation of that theory), the second argument is a list of clauses, and the third argument must be an unbound variable which is going to be bound to the internal representation of the new theory after the successful execution of the *addto (dropfrom)* predicate. Both predicates create a completely new theory with a unique theory identifier in its internal representation. This means that any two theories with two different internal representations are not unifiable in our system even though they may contain exactly the same clauses. In fact, this is the reason why the last argument of these predicates must be

an unbound variable. Two theories can be unifiable only if they have the same internal representations.

3 Derivability Predicates in MetaProlog

The basic derivability relation in MetaProlog is represented by a two-argument *demo* predicate between an explicitly represented theory and a goal. The basic *demo(Theory,Goal)* predicate holds iff *Goal* is provable in *Theory*. This predicate is used to check whether a goal is provable in a theory which is currently available in the system.

The first argument of the *demo* is normally a variable which is bound to the internal representation of a theory. The second argument is a regular Prolog goal. If the given goal is provable in the given theory, the two-argument *demo* predicate succeeds; otherwise it fails. If there are more than one solution, we can get all solutions one by one by backtracking to that *demo* predicate.

In the MetaProlog system, we not only prove a goal with respect to a theory, but also can collect its proof. The proof of a goal is collected by a three-argument *demo(Theory,Goal,proof(Proof))* predicate. The variable *Proof* is normally an unbound variable before the three-argument *demo* predicate is submitted, and that variable is bound the proof of *Goal* in *Theory* after the successful execution of the *demo* predicate. The more details about derivability relations including three-argument *demo* predicate and their implementation can be found in [8].

The three-argument *demo* predicate can also be submitted with a partially instantiated proof. In this case, the *demo* predicate tries to find a solution whose proof can be unifiable with the given partial proof. After a successful execution, the partial proof is completed. By giving a partial proof, the search space of a goal can be shrunk since the system may not need to search all parts of its search space.

The structure of the proof of a goal G in the MetaProlog system is a list whose head is an instance of G, and whose tail is the list of the proofs of the subgoals of the clause whose head is unified with the goal G. For example, let us assume that the variable T1 is bound to the internal representation of the theory containing the following clauses.

```
p(X,Y) :- q(X), r(Y).        s(1).
q(X) :- s(X).                r(a).
```

After the execution of *demo(T1,p(X,Y),proof(P))*, the variable P is bound to the following term:

```
[p(1,a),[q(1),[s(1)]],[r(a)]]
```

Proofs are just success branches in a search tree. In the MetaProlog system, we can also collect fail branches of a search tree. When the following three-argument demo predicate is executed in the MetaProlog system, *Branch* is bound to the leftmost branch of the search tree of *Goal* relative to *Theory*.

demo(Theory,Goal,branch(Branch))

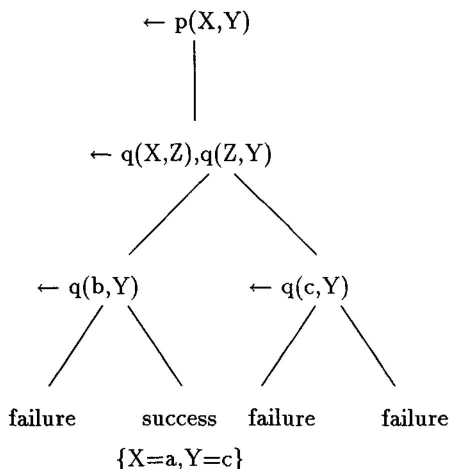
a. A Trivial Theory T

p(X,Y) :- q(X,Y).

q(a,b).

q(b,c).

b. The Search Tree of p(X,Y)



c. Branches of The Search Tree

1. [p(a,Y),[q(a,b)],[q(b,Y),fail]]
2. [p(a,c),[q(a,b)],[q(b,c)]]
3. [p(b,Y),[q(b,c)],[q(c,Y),fail]]
4. [p(b,Y),[q(b,c)],[q(c,Y),fail]]

Fig. 1. A Trivial Theory and Its Search Tree

Backtracking into this demo predicate will cause *Branch* to be bound to successive branches of the search tree. This branch can be a success branch (proof) or a fail branch of the search tree.

In Figure 1, a trivial theory T and the search tree of the goal p(X,Y) relative to theory T are given. In that search tree, there are one success branch and three fail branches. After the execution of *demo(T,p(X,Y),branch(Branch))*, the variable *Branch* is bound to the first branch of the search tree in Figure 1. We can get other branches by backtracking to the *demo* predicate.

Each fail branch has exactly one atomic fail subbranch. An atomic fail subbranch is a list whose head is a subgoal and its tail is the list *[fail]*. For example, the atomic fail subbranch of the first branch in Figure 1 is the following term:

[q(b,Y),fail]

An atomic fail subbranch separates a fail branch into two parts. The first part is the collected part of the fail branch, and the second part is the uncollected part of the fail branch. Even though fail branches are not completely collected, their collected parts are enough to give the reason of that failure. The collected part will reflect all unifications occurred before the failure, and the atomic fail subbranch will reflect the exact location of that failure.

Although branches (proofs or fail branches) are useful in many applications, all details of branches may be unnecessary in some cases. We should not pay extra cost to collect these unnecessary parts of branches in those cases. In the MetaProlog system, certain subbranches of a branch can be skipped by using

a four-argument demo predicate instead of a three-argument demo predicate. The fourth argument of this demo predicate contains a list of procedures whose branches are skipped during the execution of the given goal.

4 Fault Diagnosis in Digital Circuits

In this section, we describe a MetaProlog program which tries to find a fault in a given faulty digital circuit. The fault diagnosis algorithm given in this section is based on the ideas of Esghi (cf. [9]). We will assume that there is a single faulty gate in the given faulty circuit in the form of a gate sticking at zero or one. Although this program is designed to find the fault in digital circuits with a single faulty gate, it can easily be extended for digital circuits with multiple faulty gates.

Section 4.1 describes how a digital circuit is represented in MetaProlog. The description given in Section 4.1 can be used to represent the topological description of both normal and faulty circuits. In Section 4.2, the fault diagnosis algorithm and its implementation in MetaProlog are described.

4.1 Digital Circuit Description

We have to describe a digital circuit in some sort of predicate calculus formalism. A circuit will be represented by a MetaProlog theory which contains its topological description in the form of facts and rules. The theory will be organized in such a way that hypotheses describing faulty circuits can be created from it by simply adding a fact indicating that one of the gates is stuck at zero or one.

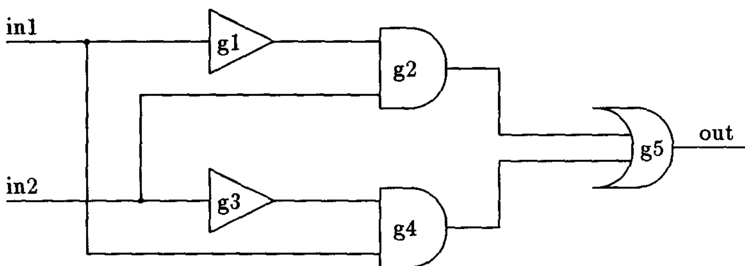


Fig. 2. An Exor Circuit

For the purposes of this simple example, we will consider the exor circuit given in Figure 2. There are five gates in that exor circuit and they are labeled as g1···g5. A faulty exor circuit will have one of its five gates stuck at one or zero. The exor circuit has one output and two input lines, and its input lines are labeled as *in1* and *in2*.

```
circuit(in(In1,In2), out(Out)) :-
    gate(g1, not(in1), In1, _, G1Out),
    gate(g2, and(in2,g1), In2, G1Out, G2Out),
    gate(g3, not(in2), In2, _, G3Out),
    gate(g4, and(in1,g3), In1, G3Out, G4Out),
    gate(g5, or(g2,g4), G2Out, G4Out, Out).

gate(G, _, _, _, Out) :- stuckAt(G, At), !, Out = At.
gate(_, and(L1,L2), X, Y, Z) :- andTable(X, Y, Z), !.
gate(_, or(L1,L2), X, Y, Z) :- orTable(X, Y, Z), !.
gate(_, not(L1), X, _, Z) :- notTable(X, Z), !.

getInput(in(In1,In2)) :- lowHigh(In1), lowHigh(In2).

lowHigh(0).      notTable(0,1).      andTable(0,0,0).      orTable(0,0,0).
lowHigh(1).      notTable(1,0).      andTable(0,1,0).      orTable(0,1,1).
                                     andTable(1,0,0).      orTable(1,0,1).
                                     andTable(1,1,1).      orTable(1,1,1).
```

Fig. 3. Theory *exor* Representing Exor Circuit

The MetaProlog theory *exor* given in Figure 3 represents the exor circuit given in Figure 2. The theory *exor* contains truth tables for *not*, *and* and *or* gates in addition to the topological description of the exor circuit. The theory *exor* could have inherited truth tables from one of its ancestors; but for simplicity reasons, we put truth tables together with the circuit description into a single theory. The theory has also the predicate *getInput* to create a possible input for the exor circuit.

The topological description of a circuit is represented by a call to the predicate *circuit* which has two arguments. The first argument is a term holding input lines of the circuit, and the second one is the output of the circuit. Each gate in the circuit is represented by the predicate *gate* whose first argument holds the name of a gate. The second argument is a term representing the type of the gate, and the names of its input lines. The last three arguments of that predicate denote the inputs and the output of the gate in consideration. Each gate takes an input line or the output of another gate as its input. The output of a circuit is the output of one of its gates. In theory *exor*, the output of the gate *g5* is also the output of the exor circuit.

The significance of the first clause of the predicate *gate* is that first we check whether a given gate is stuck at zero or one before we look at its truth table for its behavior. In other words, normally a gate behaves as it is described in its truth table unless it is a faulty gate. This clause makes it easier to represent faulty circuits in addition to normal circuits with no fault in the form of the theory given in Figure 3.

Since the theory *exor* in Figure 3 does not contain a fact *stuckAt(G,At)*, it represents a normal exor circuit without a faulty gate. We can create a theory

representing a faulty exor circuit whose gate *G* is stuck at value *At* by adding a fact *stuckAt(G,At)* to the theory *exor*. For example, the following *addto* statement creates theory *FaultyExor* which represents a faulty exor circuit whose gate *g2* is stuck at zero.

addto(< *theoryexor* >,stuckAt(g2,0),FaultyExor)

We will create theories representing faulty circuits when we create a hypothesis to explain the fault in a given faulty circuit.

The predicate *circuit* in the theory *exor* can be used to simulate the action of the exor circuit in Figure 2. For example, to simulate the action of the exor circuit when its input lines are respectively 1 and 0, we would run the MetaProlog goal

demo(< *theoryexor* >,circuit(in(0,1),out(Out)))

which would be solved yielding the output value Out=1.

4.2 Fault Diagnosis

Our problem is that we have to find the faulty gate in a given faulty circuit from the given circuit description and a faulty input-output pair. Although the algorithm given here is designed for circuits with a single faulty gate, it can be easily extended for circuits with multiple faulty gates by modifying the hypothesis generation.

The algorithm presented here relies heavily on the ability to manipulate and create theories in MetaProlog. In this algorithm, we will get a theory *Circuit* which describes the circuit under diagnosis and a faulty input-output pair *(InF, OutF)* for the actual circuit represented by the theory *Circuit* for which the output *OutF* is a faulty output. We will also get another theory representing the physical faulty circuit. The main task of this algorithm is to infer a new theory *FaultyDescription* from the given theory *Circuit* such that this new theory will correctly simulate the input-output pair *(InF, OutF)*. This new theory will also describe the faulty behavior in the given circuit due to a single gate stuck at zero or one.

The basic fault diagnosis algorithm given in Figure 4 has two major parts. The first part is Step 1 in which a set *FAULTS* of theories is created. This set includes all possible theories which simulate the faulty input-output pair *(InF, OutF)*. Steps 2-6 of the algorithm constitute a standard test and eliminate loop which is used to choose the theory which correctly describes the faulty circuit from the set created in Step 1.

Step 1 of the fault diagnosis algorithm in Figure 4 is implemented in MetaProlog by the following procedure *possibleFaults* which takes a theory describing a circuit and a faulty input-output pair for that circuit as input, and returns a set of theories in which every theory correctly simulates the given faulty input-output pair.

1. From the circuit description *Circuit* and the faulty input-output pair *(InF,OutF)*, construct a set *FAULTS* of theories such that every theory in that set correctly simulates the faulty input-output pair.
2. If the cardinality of the set *FAULTS* is 1, the set *FAULTS* contains the theory correctly describing the faulty circuit. Stop and output the result.
3. Choose two distinct theories *Fi* and *Fj* in *FAULTS*, and construct, if possible, a discriminating input *InD* which distinguishes *Fi* and *Fj*. If this is impossible, stop and output *FAULTS* which contains more than one theory which describes the faulty circuit. Otherwise, go to Step 4.
4. Apply the input *InD* to the given physical faulty circuit to the resulting output *OutD*.
5. Delete all *Fi* which cannot simulate the input-output pair *(InD,OutD)* from the set *FAULTS*.
6. Go to Step 2.

Fig. 4. Fault Diagnosis Algorithm

```
possibleFaults(Circuit, InF, OutF, Faults) :-
    demo(Circuit, circuit(InF,OutF), branch(Branch), skip(gate/5)),
    member([Gate,[fail]], Branch),
    getFaults(Gate, Branch, Circuit, Faults).
```

This procedure constructs the set of theories in which every theory correctly simulates the faulty input-output pair by using a failed branch and a heuristic deduction method. Normally, when we run the following goal

```
demo(Circuit,circuit(InF,OutF))
```

it will fail since (InF,OutF) is a faulty input-output pair for the correct circuit. On the other hand, the goal

```
demo(Circuit,circuit(InF,OutF),branch(Branch))
```

will succeed by binding the variable *Branch* to a failed branch. We can get all failed branches of the search tree by running the goal above recursively. But we are not interested in all branches and their complete details, we are only interested in which subgoals of the predicate *circuit* succeeded and which one failed. For these reasons, the procedure *possibleFaults* calls the following subgoal

```
demo(Circuit,circuit(InF,OutF),branch(Branch), skip(gate/5))
```

to skip the proof of subgoals *gate*. In this case, the fail branch we get will not contain proof details of these subgoals. In this fail branch, there will be a success branch for a subgoal *gate* indicating that output behavior of that gate for its input is correct based on its truth table, or a fail branch for it indicating that the output behavior of that gate for its input is faulty. Another fact about this fail branch is that it will contain a single failed gate. For example, if we run the goal above for the theory *exor* in Figure 3 and the faulty input-output pair (in(1,1),out(1)), the variable *Branch* will be bound to the following term.

```
[ circuit(in(1,1),out(1)),
    [ gate(g1,not(in1),1,_,0) ],
    [ gate(g2,and(g1,in2),0,1,0) ],
    [ gate(g3,not(in2),1,_,0) ],
    [ gate(g4,and(in1,g3),1,0,0) ],
    [ gate(g5,or(g2,g4),0,0,1), [fail] ] ]
```

The term above is a failed branch of the goal *circuit* in the theory *exor* for the faulty input-output pair (in(1,1),out(1)). In this failed branch, the gate *g5* is a failed gate since 1 is a faulty output for an *or* gate when its inputs are 0.

Later, the procedure *possibleFaults* chooses the failed gate in the fail branch which has a single failed gate. Then, it calls the procedure *getFaults* to construct the hypothesis set of theories which correctly simulate the faulty input-output pair. The procedure *getFaults* is represented in MetaProlog by the following clause.

```
getFaults(Gate, Branch, Circuit, [Fault | Faults]) :-
    Gate = gate(G,GType,GIn1,GIn2,GOut),
    addto(Circuit, stuckAt(G,GOut), Fault),
    getFaultyInputs(Gate, Branch, Circuit, Faults).
```

The procedure *getFaults* simply starts from the failed gate in the fail branch to construct the hypothesis set of theories. A gate fails if its input-output pair is a faulty one based on its truth table. For example, the gate *g5* which is an *or* gate fails since the output of an *or* gate cannot be 1 when its inputs are 0. So, either gate *g5* is stuck at one or one of its input lines is not zero. The procedure *getFaults* constructs the hypothesis set by first assuming the failed gate is stuck at its faulty output and then calls the procedure *getFaultyInputs* given in Figure 5 to find out the possible faulty input lines for that failed gate.

The procedure *getFaultyInputs* finds out which input line of a failed gate can be faulty. We only check input lines which are outputs of another gate since we assumed that only gates can be faulty in a circuit. Each clause of the procedure *getFaultyInputs* in Figure 5 represents a faulty input-output pair for *or*, *and*, and *not* gates. Since we assumed that there is a single faulty gate in a given faulty circuit, we do not consider all possible faulty input-output pairs for all gates. For example, the faulty input-out pair ((1,1),0)) for an *or* gate is intentionally not included in Figure 5, since it requires that both inputs of *or* gate must be faulty.[1] If we want to update our algorithm so that it can find out faults in circuits with more than one faulty gate, we only have to update procedures *getFaults* and *getFaultyInputs* to satisfy our goals.

The top level of the fault diagnosis algorithm is represented by the following procedure *findFault* which takes theories describing normal and faulty circuits, and a faulty input-output pair as input, and prints out an explanation for the fault in the faulty circuit.

[1] We also assume that an output of a gate can be used as an input for only one gate. If an output of a gate can be the input of more than one gate, we should change the procedure *getFaultyInputs* to accommodate this fact.

```
getFaultyInputs(gate(G,or(L1,L2),0,0,1), Branch, Circuit, Faults) :-
  ( member([gate(L1,L1Type,L1In1,L1In2,0)], Branch), !,
    getFaults(gate(L1,L1Type,L1In1,L1In2,1), Branch, Circuit, Faults1);
    Faults1 = [] ),
  ( member([gate(L2,L2Type,L2In1,L2In2,0)], Branch), !,
    getFaults(gate(L2,L2Type,L2In1,L2In2,1), Branch, Circuit, Faults2);
    Faults2 = [] ),
  append(Faults1, Faults2, Faults).
getFaultyInputs(gate(G,or(L1,L2),0,1,0), Branch, Circuit, Faults) :-
  member([gate(L2,L2Type,L2In1,L2In2,1)], Branch), !,
  getFaults(gate(L2,L2Type,L2In1,L2In2,0), Branch, Circuit, Faults).
getFaultyInputs(gate(G,or(L1,L2),1,0,0), Branch, Circuit, Faults) :-
  member([gate(L1,L1Type,L1In1,L1In2,1)], Branch), !,
  getFaults(gate(L1,L1Type,L1In1,L1In2,0), Branch, Circuit, Faults).

getFaultyInputs(gate(G,and(L1,L2),0,1,1), Branch, Circuit, Faults) :-
  member([gate(L1,L1Type,L1In1,L1In2,0)], Branch), !,
  getFaults(gate(L1,L1Type,L1In1,L1In2,1), Branch, Circuit, Faults).
getFaultyInputs(gate(G,and(L1,L2),1,0,1), Branch, Circuit, Faults) :-
  member([gate(L2,L2Type,L2In1,L2In2,0)], Branch), !,
  getFaults(gate(L2,L2Type,L2In1,L2In2,1), Branch, Circuit, Faults).
getFaultyInputs(gate(G,and(L1,L2),1,1,0), Branch, Circuit, Faults) :-
  ( member([gate(L1,L1Type,L1In1,L1In2,1)], Branch),
    getFaults(gate(L1,L1Type,L1In1,L1In2,0), Branch, Circuit, Faults1);
    Faults1 = [] ),
  ( member([gate(L2,L2Type,L2In1,L2In2,1)], Branch), !,
    getFaults(gate(L2,L2Type,L2In1,L2In2,0), Branch, Circuit, Faults2);
    Faults2 = [] ),
  append(Faults1, Faults2, Faults).

getFaultyInputs(gate(G,not(L1),0,_,0), Branch, Circuit, Faults) :-
  member([gate(L1,L1Type,L1In1,L1In2,0)], Branch), !,
  getFaults(gate(L1,L1Type,L1In1,L1In2,1), Branch, Circuit, Faults).
getFaultyInputs(gate(G,not(L1),1,_,1), Branch, Circuit, Faults) :-
  member([gate(L1,L1Type,L1In1,L1In2,1)], Branch), !,
  getFaults(gate(L1,L1Type,L1In1,L1In2,0), Branch, Circuit, Faults).

getFaultyInputs(Gate, Branch, Circuit, []) :- !.
```

Fig. 5. Finding Faulty Inputs for A Gate

```
findFault(Circuit, FaultyCircuit, InF, OutF) :-
    possibleFaults(Circuit, InF, OutF, Faults), !,
    filterFaults(Faults, Fault, FaultyCircuit), !,
    printFault(Fault).
```

This procedure first constructs the set of all possible theories which simulate the given faulty input-output pair by calling the procedure *possibleFaults*. Then, it

calls the procedure *filterFaults* to choose the theory which correctly describes the given faulty circuit from the set created by the procedure *possibleFaults*. The procedure *printFault* which takes the set of theories correctly describing the faulty circuit, and prints out which gate is faulty in the given faulty circuit is implemented as follows.

```
printFault([Fault]) :- !,
    demo(Fault, stuckAt(G,At)),
    nl, write(G), write(' is stuck at '), write(At).
printFault([Fault | Faults]) :-
    demo(Fault, stuckAt(G,At)),
    nl, write(G), write(' is stuck at '), write(At), write(' or'),
    printFault(Faults).
```

It simply demonstrates which gate is faulty in the theory that correctly describes the faulty circuit, and prints out this faulty gate. If there is more than one theory correctly describing the faulty circuit, this fact is also printed out by this printing routine.

The MetaProlog procedures given in Figure 6 implement Steps 2-6 of the fault diagnosis algorithm in Figure 4. The procedure *filterFaults* in Figure 6 first chooses two distinct theories from the set created by the procedure *possibleFaults* and an input which distinguishes those two theories. Then, it applies that input to the faulty circuit to get its behavior on that input. Later, it deletes all theories whose behaviors differ from the behavior of the faulty circuit on the distinguishing input. This filtering operation continues until it is not possible to choose two distinct theories and a distinguishing input on them. In that case, all theories left in the set *Faults* correctly describe the faulty circuit.

The input-output pair (in(0,0),out(1)) is a faulty input-output pair for the theory *exor*. The procedure *possibleFaults* will construct a set containing three theories. These theories represent faulty exor circuits whose gates *g2*, *g4* and *g5* are stuck at 1, respectively. These three theories correctly simulate the faulty input-output pair above. Unfortunately, there is no input distinguishing any two of these three theories. So, any of these three gates in the exor circuit can be faulty, and we cannot determine which one of them. The procedure *filterFaults* will recognize this fact, and return the set containing these three theories as output.

Now, let assume that we have a faulty exor circuit whose gate *g3* (cf. Figure 2) is stuck at zero, and a faulty input-output pair (in(1,0),out(0)). The procedure *possibleFaults* will generate a set containing theories F1, F2, F3 representing exor circuits whose gates *g3*, *g4* and *g5* are stuck at zero, respectively. The procedure *filterFaults* will find a distinguishing input in(0,1) for theories F1 and F3. Since the output behavior will be different on this input from the behavior of our original faulty circuit on the same input, the theory F3 will be eliminated from the hypothesis set by leaving theories F1 and F2 in the set. Unfortunately, again there will not be any distinguishing input for these two theories, so the output of the procedure *filterFaults* will be the set containing these two theories.

```
filterFaults(Faults, Fault, FaultyCircuit) :-
    chooseTwoDistinctFaults(Faults,F1,F2,Input,Output1,Output2), !,
    demo(FaultyCircuit, circuit(Input,Output)),
    ( Output = Output1, !,
            deleteFaults(Faults, NewFaults, Input, Output2),
            filterFaults(NewFaults, Fault, FaultyCircuit) ;
      Output = Output2, !,
            deleteFaults(Faults, NewFaults, Input, Output1),
            filterFaults(NewFaults, Fault, FaultyCircuit) ).
filterFaults(Fault, Fault, FaultyCircuit).

chooseTwoDistinctFaults(Faults,F1,F2,Input,Output1,Output2) :-
    chooseTwoFaults(Faults, F1, F2),
    demo(F1, getInput(Input)),
    demo(F1, circuit(Input,Output1)),
    demo(F2, circuit(Input,Output2)),
    Output1 \= Output2.

chooseTwoFaults([F1 | Faults], F1, F2) :- member(F2, Faults).
chooseTwoFaults([F | Faults], F1, F2) :- chooseTwoFaults(Faults, F1, F2).

deleteFaults([F | Faults], NewFaults, Input, Output) :-
    demo(F, circuit(Input,Output)), !,
    deleteFaults(Faults, NewFaults, Input, Output).
deleteFaults([F | Faults], [F | NewFaults], Input, Output) :-
    deleteFaults(Faults, NewFaults, Input, Output), !.
deleteFaults([], [], Input, Output) :- !.
```

Fig. 6. Filtering Possible Faults

5 Conclusion

In this paper, we illustrated how meta-level facilities in MetaProlog such as theories and fail branches played a key role in the representation of the fault diagnosis problem. These meta-level facilities improve the expressive power of the MetaProlog programming language so that the problems, where these facilities are necessary, can naturally be represented by the tools in MetaProlog. The languages without this kind of facilities such as Prolog can only represent those problems using adhoc methods.

The MetaProlog system, as a programming tool, is also useful for many other AI and non-AI applications where meta-level facilities such as contexts and proofs are necessary. For example, the proofs in MetaProlog can also be used in explanation facilities of expert systems. A proof can be collected when a goal is proved by a *demo* predicate, and that proof can be used to justify the results of that goal. Besides, the inheritance mechanism in MetaProlog theories can also be useful for the representation of the objects in the object-oriented programming paradigm.

References

1. Aït-Kaci, H., *Warren's Abstract Machine: A Tutorial Reconstruction*, The MIT Press, Cambridge, 1991.
2. Attardi, G., and Simi, M., Metalanguage and Reasoning Across Viewpoints, in: *Proc. of the 6th ECAI*, Pisa, Italy, 1984.
3. Bowen, K.A., and Kowalski, R.A., Amalgamating Language and Metalanguage in Logic Programming, in: *Logic Programming*, Clark, K., and Tarnlund, S.-A. (eds.), Academic Press, London, 1982, pp. 153-173.
4. Bowen, K.A., and Weinberg, W., A Meta-Level Extension of Prolog, in: *Proc. of the 1985 Symp. on Logic Programming*, IEEE Computer Society Press, 1985, pp. 48-53.
5. Bowen, K.A., A Meta-Level Programming and Knowledge Representation, *New Generation Computing 3:359-383*, 1985.
6. Bratko, I., *PROLOG Programming For Artificial Intelligence, 2nd Edition*, Addison-Wesley, New York, 1990.
7. Cicekli, I., Design and Implementation of An Abstract MetaProlog Engine for MetaProlog, in: *Meta-Programming in Logic Programming*, Abramson, H., and Rogers, M.H. (eds.), The MIT Press, Cambridge, 1989, pp. 417-434.
8. Cicekli, I., *Abstract MetaProlog Engine*, *Journal of Logic Programming 34(3):169-200*, 1998.
9. Eshghi, K., Application of Meta-Language Programming to Fault Finding in Logic Circuits, in: *Proc. of the 1st Int. Conf. on Logic Programming*, Marseille, 1982.
10. Lamma, E., Mello, P, and Natali, A., An Extended Warren Abstract Machine for The Execution of Structured Logic Programs, *Journal of Logic Programming 14:187-222*, 1992.
11. Montiero, L., and Porto, A., Contextual Logic Programming, in: *Proc. of the 6th Int. Conf. on Logic Programming*, The MIT Press, 1989, pp. 284-302.
12. Nadathur, G., Jayaraman, B., and Kwon, K., Scoping Constructs in Logic Programming: Implementation Problems and Their Solution, *Journal of Logic Programming 25:119-161*, 1995.
13. des Rivieres, J., Meta-Level Facilities in Logic-Based Computational Systems, in: *Proc. of The Workshop on Meta-Level Architectures and Reflection*, Alghero-Sardinia, Italy, 1986.
14. Safra, M. and Shapiro, E., Meta-Interpreters for Real, in: *Concurrent Prolog, Vol 2*, Shapiro, E. (ed.), The MIT Press, Cambridge, 1987, pp. 166-179.
15. Sterling, L.S., Meta-Interpreters: The Flavors of Logic Programming?, in: *Proc. of Workshop on Deductive Databases and Logic Programming*, Washington D.C., 1986, pp. 163-175.
16. Sterling, L.S., A Meta-Level Architecture for Expert System, in: *Meta-Level Architectures and Reflection*, Maes, R., and Nardi, D. (eds.), North Holland, 1988.
17. Warren, D.H.D., An Abstract Prolog Instruction Set, SRI Technical Report 309, 1983.
18. Weyhrauch, R.W., Prolegomena to A Theory of Mechanized Formal Reasoning, *Artificial Intelligence 13:133-170*, 1980.