ICOT Technical Report: TR-143

TR-143

An Algorithm For Finding A Query Which Discriminates Competing Hypotheses

by
Hirohisa Seki and Akikazu Takeuchi

October, 1985

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

An Algorithm For Finding A Query Which Discriminates Competing Hypotheses

Hirohisa Seki and Akikazu Takeuchi

ICOT Research Center,
Institute for New Generation Computer Technology,
Mita-Kokusai Bldg. 21F, 1-2-28, Mita,
Minato-ku, Tokyo 108, JAPAN

[Abstract] This paper presents a logical approach to the problem of hypothesis selection. In hypothesis formation such as diagnosis or inductive inference, we construct a hypothesis which is expected to give a consistent explanation of observations on some object domain. Suppose that more than one plausible hypotheses can cover the given observations and that each of the hypotheses is mutually incompatible with another. Then we try to identify which hypothesis is a "better" one, by finding another observation discriminating hypotheses obtained so far. In this paper, we adopt Popper's principle as the criteria for hypothesis selection and present an algorithmic method to find a "crucial" query which discriminates a more appropriate hypothesis among the competing ones.

1. Introduction

In the formation of scientific theory, we try to find a hypothesis which is capable of giving a consistent explanation of given observations on some object domain, by performing experiments and refining the hypothesis, if necessary. The typical examples are found in fault/medical diagnoses or inductive inference. In such problems, it often happens that more than one plausible hypotheses can be considered and each of the hypotheses is mutually incompatible with another. Then we try to identify which hypothesis is a more appropriate one. This is a classical problem in scientific theory formation (e.g., [1]). In [1], Popper gave several criteria that we should be inclined to say that a theory T1 corresponds "better" to the given facts than the other theory T2. The followings are some of his criteria.

(a) T1 explains more facts than T2.
(b) T1 has passed test which T2 has failed to pass.

Existing expert systems for diagnosis such as MYCIN give multiple hypotheses to given observations. To each hypothesis, the certainty factor is attached, which indicates its plausibility.

In this paper, we present a logical approach to give a method for discriminating a more appropriate hypothesis among the competing hypotheses. In our framework which lies on the same lines with Theorist [2] or MIS [3], observations and hypotheses are expressed with logical formulas. A hypothesis explains observations means that they are logical consequences of the hypothesis. Suppose that there are two hypotheses both of which are capable of explaining given observations and that there exist a literal (we call it a "crucial" literal) which is a logical consequence of only one of those hypotheses. Then, by performing an experiment to decide the truth value of that crucial literal, we can know which one is a "better" hypothesis, in the above-mentioned sense of Popper. In this paper, we give an algorithmic method to find such a

"crucial" literal.

In the next section, we summarize some preliminary materials and introduce terminologies. In section 3, we prove a basic theorem which gives a basis to an algorithm for finding a crucial query which discriminates a preferable one among competing hypotheses. In section 4, we show a more efficient algorithm which uses a divide-and-query algorithm. In section 5, we give some examples to explain how our algorithm finds a crucial literal.

2. Preliminaries

In this section, we introduce some terminologies and notations.
A hypothesis is given as a finite set of definite clauses of the form:

A <- L1,..., Ln

where A is an atom and L1, ..., Ln are positive literals.

An observation (or fact) is expressed with a literal. A hypothesis H explains (or supports) an observation O iff there exists an SLD-refutation of H U { <- O } via some computation rule. A hypothesis H explains observations Obs iff H explains each one of Obs. In this case, H is said to be a hypothesis for Obs.

An integrity constraint is given as a goal, i.e., a clause of the form :

<- L1,..., Ln

where L1,..., Ln are positive literals. The integrity constraint expresses a situation which should not happen. A hypothesis H is said to violate an integrity constraint IC iff there exists an SLD-refutation of H U {IC} via some computation rule ; otherwise, we say that H satisfies IC. A hypothesis H1 is said to be incompatible with a hypothesis H2 with respect to an integrity constraint IC iff H1 U H2 violates IC.

Let P be a set of definite clauses and G be a goal of the form : <- A1,..., Am. Assume that there exists an SLD-refutation of P U { G } via some computation rule R and θ is its R-computed answer substitution for P U { G }. Using the substitution θ, a proof tree for P U {G} is defined as follows :

(a) the root is Gθ.
(b) the root has k descendents A1θ,....,Amθ.
(c) Let A be a node in the tree which is not the root. Suppose that A' <- B1,...,Bn is a clause in P and Aθ = A'θ. Then node A has n descendents B1θ,...,Bnθ. If n=0, then A has a descendent "true".
(d) Nodes which are "true" have no descendents.

The degree of a proof tree is the number of nodes in the tree. The depth of a proof tree is the maximal length of paths from the root to its leaves. A node is called a node of level k if, from the root to the node, there exists a path whose length is k. We denote by tr(N) a subtree whose root is N.

Let S be a subset S of P. In a proof tree, if a node N has descendents which are derived by using a clause C belonging to S, then we say that node N is an S-node. If L is a node and a subtree tr(L) consists of only S-nodes, then it is called an S-subtree. Furthermore, if L is a logical consequence of S, then we say that L has an S-subtree.

3. Crucial Literal And An Algorithm For Finding It

Let H1 and H2 be hypotheses both of which are capable of explaining given observations Obs, and let IC be an integrity constraint which hypotheses should satisfy. Assume that each of H1 and H2 satisfies IC, but H1 is incompatible with H2 with respect to IC. Then, the next step is to decide which one is a more appropriate hypothesis for Obs by performing another experiment.

A literal L is called to be a "crucial literal" with respect to H1 and H2 if either H1 or H2 (but not both) explains L. When there exists such a crucial literal and the user tells the truth value of that literal, then it gives a clue for discriminating a more desirable hypothesis. That is, assume that Hi explains L but Hj doesn't (i,j = 1,2 and i =\= j). If the user tells that L is true, then it is known that Hj is too weak a hypothesis for Obs U {L} and Hi is preferable to Hj : if the user tells that L is false, then Hi is an incorrect hypothesis for Obs U {L} and it should be modified, whereas Hj is still a correct hypothesis for Obs U {L}.

Now we show an algorithmic method to find such a crucial literal. At first, we prove a theorem which gives the basis of the algorithm. Before that, we need the following lemma.

[Lemma 3.1]

Let N be a node in a proof tree H1 U H2 U { IC }. Assume that (1) tr(N) is neither an H1-subtree nor an H2-subtree, (2) N is Hc-node (either c=1 or i=2) and (3) it has descendents Dn(n=1,...,d, d>=1). If each tr(Dn) is either an H1-subtree or an H2-subtree, then one of the followings holds :

(a) Among D1,...,and Dd, there is a crucial literal with respect to H1 and H2.
(b) Hc |- N.

(proof)

The set of subtrees {tr(D1),...,tr(Dn)} is partitioned into two disjoint subsets (possibly empty) P1 and P2, where Pk (k = 1,2) is the set of a subtree which is an Hk-subtree, respectively. From the assumption (1) and (2) above, there exit the following two cases :

(case 1) neither P1 nor P2 is empty.
(case 2) Pc is an empty set.

In (case 1), let Pj be {tr(Dj1),...,tr(Djdj)} (j =\= c, dj >= 1) and for each l = 1,...,dj, we check to see whether Hc |- Djl or not. If, for some l, it doesn't hold, such Djl gives a crucial literal with respect to H1 and H2. Hence, case (a) holds. Otherwise, each Djl has also its proof consisting of an Hc-subtree, so we can replace each tr(Djl) by its corresponding Hc-subtree, which makes tr(N) an Hc-subtree. So, case (b) holds.

On the other hand, in (case 2), Pj (j =\= c) becomes the set {tr(D1),...,tr(Dd)}. Then, for each l (l =1,...,d), we check whether Hc |-\- Dl, then Dl gives a crucial literal with respect to H1 and H2, so case (a) holds. Otherwise, since each Dl has also its proof consisting of Hc-subtree, so we replace each tr(Dl) by its corresponding Hc-subtree, which makes tr(N) an Hc-subtree. Hence, case (b) holds. Q.E.D.

[Theorem 3.1]
Let H1 and H2 be hypotheses which satisfy the above-mentioned conditions. Then, in a proof tree for H1 U H2 U { IC }, there exists a node which corresponds a crucial literal with respect to H1 and H2.

(proof)
Assume that IC be of the form : <- C1,...Cn. From the assumption that H1 U H2 violates IC, there exists an SLD-refutation of H1 U H2 U { IC } via some computation rule R. Let θ be its R-computed substitution. Hence, a proof tree for H1 U H2 U { IC } actually exists and we denote it by T. Consider the following two cases :

(case 1)
For all j, if tr(Cjθ) is either an H1-subtree or an H2-subtree, then among C1θ,...,Cnθ, there exists a crucial literal with respect to H1 and H2, because both H1 and H2 satisfy IC.

(case 2)
Otherwise, for some j, tr(Cjθ) is neither an H1-subtree nor an H2-subtree. Then, we examine its nodes in tr(Cjθ) one by one according to their levels, from leaves upwardly to its root Cjθ. Let d be the depth of tr(Cjθ).
At first, for each node L of level d (which means that L is a leaf "true"), it clearly holds that H1 |- L and H2 |- L. That is, tr(L) is both an H1-subtree and an H2-subtree.
Next, consider a node N of level d-k (k=1,...,d). Assume that N is an Hc-node (either c= 1 or c= 2) and that, for every descendent D of N, tr(D) is either an H1-subtree or an H2-subtree. Then, there exist the following two cases :
(a) tr(D) is either an H1-subtree or an H2-subtree.
(b) tr(D) is neither an H1-subtree nor an H2-subtree.

In case (b), from [Lemma 3.1], it follows that if,among the descendents of N, there exists no crucial literal with respect to H1 and H2, then N has an Hi-subtree. Hence, from induction, unless there is no crucial literal with respect to H1 and H2 in tr(Cjθ), then Cjθ has either an H1-subtree or an H2-subtree.

Likewise, applying the same discussion to other literals Cjθ (j = 1,...,n), it follows that, unless there exists a crucial literal with respect to H1 and H2 in some tr(Cjθ), each Cjθ has either an H1-subtree or an H2-subtree, which is exactly the same as in (case 1). Q.E.D.

The above-mentioned theorem and lemma immediately suggest a naive algorithm for finding a crucial literal.

(step 0) j := 0.
(step 1) j := j + 1. If j = n, then go to (step 4). Otherwise, check whether tr(Cjθ) is either an H1-subtree or an H2-subtree. If so, go to (step 1).
(step 2) Otherwise, k := d, where d is the depth of tr(Cjθ).
(step 3) If k= -1, then go to (step 1). Otherwise, for each node of level k, check whether it has either an H1-subtree or an H2-subtree. If the node has neither an H1-subtree nor H2-subtree, then a crucial literal is found among the descendents of that node. Otherwise, k := k - 1 and go to (step 3).
(step 4) Find a crucial literal among C1θ,...,and Cnθ, by checking whether each Cjθ (j=1,...,n) has either an H1-subtree or an

H2-subtree.

The above algorithm is, however, apparently inefficient, because it might examine thoroughly all the nodes one by one in a proof tree. Hence, in the next section, we present a more efficient algorithm.

4. Divide-and-Query Algorithm For Finding A Crucial Query

In this section, we present a more efficient algorithm for finding a crucial query. The algorithm employs a divide-and-query algorithm which was used in [3]. First we define the weight of a node in a proof tree.

Let M be a subset of the nodes in a proof tree for H1 ∪ H2 ∪ IC. The weight of a node N modulo M is defined as follows. If N is in M, then its weight is 0. If N is a leaf then its weight is 0. Otherwise, the weight of N is 1 plus the sum of the weight modulo M of its sons.

The definition of the middle node of a proof tree is the same as that of Shapiro, i.e., it is the leftmost heaviest node in the tree whose weight modulo M is $=< \lceil w/2 \rceil$, where w is the weight of the original tree. In order to find a middle node of a given proof tree, we use the procedure fpm (see [3]) which computes the middle node and its weight. Using this procedure, our divide-and-query algorithm for finding a crucial literal is shown below (for comparison, we describe our algorithm following Shapiro's).

[A Divide-and-query algorithm for finding a crucial literal]
Input : hypothesis H1, H2, and an integrity constraint IC, which satisfy the conditions mentioned in [Theorem 3.1]
Output: a crucial literal with respect to H1 and H2

Algorithm : Let M be an empty set. Simulate the execution of IC that returns its proof tree T, computing w, the weight modulo M of the computation tree. Then call a recursive procedure fp with T, w and M.

The procedure fp, on input T, w, and M, operates as follows. First, it applies the procedure fpm, which finds the heaviest node N in T whose weight wq modulo M is less than or equal to $\lceil w/2 \rceil$. It then checks whether N is a logical consequence of either H1 or H2.

(i) if both H1 |- N and H2 |- N, then fp calls itself recursively with T, w-wq, and M ∪ {N}.
(ii) if Hi |- N and Hj |-\- N (i,j = 1,2, i =\= j), then fp returns N.
(iii) if neither H1 |- N nor H2 |- N, then fp calls itself recursively with tr(N), wq and M.

The main difference between the above algorithm and that of Shapiro is as follows : in his algorithm, each time fpm is called, computing the middle node in a proof tree, then its truth value is given by an oracle query, while in our algorithm, instead of oracle queries, it is checks to see whether the middle node is a logical consequence of either H1 or H2. The correctness of the above algorithm is established by the following theorem.

[Theorem 4.1]
Using the condition of Theorem 3.1, the above algorithm always terminates and returns a crucial literal with respect to H1 and H2.

(proof)
Assume that N is not a crucial literal, i.e., fp is called in either

case (i) or case (iii) in the above. In case (i), there exists at least one crucial literal in T but not in tr(N), because, if we assume the contrary, either H1 or H2 violates IC, which contradicts the assumption of the theorem. As for case (iii), the subtree tr(N) satisfies the condition of Theorem 3.1. Hence, there exists a crucial literal in tr(N). Note that every time fp is called, the input value of the weight strictly decreases. Since the existence of a crucial literal is guaranteed by Theorem 3.1, fp must be called in case (ii), which is to be proved. Q.E.D.

As for the computation-complexity of the algorithm, the same properties as Shapiro's (see Lemma. 3.6 in [3]). For example, the number of the calling of fp is at most b log n, where n is the degree of the initial input proof tree and b is the maximal branching of the proof tree.

5. Examples

In this section, we give some examples to show how the above-mentioned algorithm finds a crucial literal with respect to two competing hypotheses.

[Example 1]

The first example is a problem of the circuit diagnosis which is adopted from [4]. Consider the following circuit diagram C0 (Fig. 1).

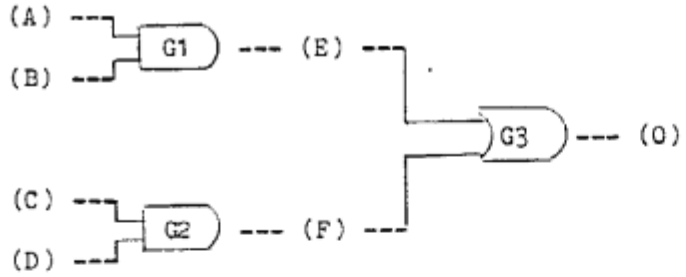

Fig. 1 --- an example of a circuit diagram

Fig. 1 shows a simple circuit diagram, where (A),(B),(C),(D), (E), (F) and (O) are nodes, G1 and G2 are AND-gates and G3 is an OR-gate. Nodes (A), (B), (C) and (D) are the input nodes, (O) is the output node of the above circuit C0.

A node has its state which is either "low" or "high". When a node N is in a state S, then it is represented with a unit clause : node(N,S).

An AND-gate is represented as a unit clause : and_gate(G,I1,I2,Og), where G is the name of an AND-gate, I1 and I2 are the state of its input nodes and Og is the state of its output node. Hence, and_gate(G,I1,I2,Og) holds iff (i) G is an AND-gate, (ii) Og=high when both I1 and I2 are high and (iii) Og=low when either I1 or I2 is low. Using this predicate, node (E) and (F) are defined as follows :

```
node(e, E) <- and_gate(g1,A,B,E), node(a,A), node(b,B).        (1)
```

```
node(f, E) <- and_gate(g1,C,D,F), node(c,C), node(d,D).          (2)
```

Similarly, an OR-gate is represented as a unit clause : or_gate(G,I1,I2,Og) and node (O) is defined as follows :

```
node(o, O) <- or_gate(g3,E,F,O), node(e,E), node(f,F).           (3)
```

Now, assume that the states of the input nodes are given as follows :

```
node(a, high).
node(b, high).
node(c, high).                                                   (4)
node(d, low).
```

We denote by H0 the definitions of nodes (1),(2),(3) and (4), together with the definitions of predicates "and_gate" and "or_gate".

If, for the given input (4), the state of the output node(O) is low, then it means that there is something wrong in the above circuit. Eshghi [4] gives the following two hypotheses for this faulty output.

Hypothesis 1 :

the gate G1 is faulty and its output is stuck at low.
So, in place of (1), the following holds :

```
node(e, E) <- fault1(g1,A,B,E), node(a,A), node(b,B).
fault1(g1,X,Y,low).                                              (2)'
```

Hypothesis 2 :

the gate G3 is faulty and its output is stuck at low.
So, in place of (2), the following holds :

```
node(o, O) <- fault2(g3,E,F,O), node(e,E), node(f,F).
fault2(g3,X,Y,low).                                              (3)'
```

Let H1 be H0 - {(2)} U {(2)'} (i.e., H0 except that the definition of node (E) is given in (2)' instead of (2)) and H2 be H0 - {(3)} U {(3)'}. Furthermore, let Obs be {node(o,low)}. Then, clearly, both H1 and H2 explain Obs. As an integrity constraint, we give the following goal formula IC :

```
<-  node(X,high), node(X,low).
```

which means that, if there exists a node which has the state high and low simultaneously, then it is a contradiction. It is easily checked that H1,H2 and IC satisfy the condition of [Theorem 3.1] and a proof tree of H1 U H2 U { IC } is as follows (for simplicity, leaves "true" are omitted).

```
node(o, high), node(o,low)
   node(o, high)
        or_gate(g3,high,low,high)
        node(e,high)
                and_gate(g1,high,high,high)
                        node(a,high)
                        node(b,high)
        node(f,low)
                and_gate(g2,high,low,low)
                        node(c,high)
                        node(d,low)
   node(o,low)
        or_gate(g3,low,low,low)
        node(e,low)
                fault1(g1,_,_,low)
                node(a,high)
                node(b,high)
```

```
node(f,low)
        and_gate(g2,high,low,low)
        node(c,high)
        node(d,low)
```

From the above proof tree, our algorithm finds a crucial literal node(e,high), which is a logical consequence of Hypothesis 2, though it cannot be derived from Hypothesis 1.

In [4], in order to select a better hypothesis between Hypothesis 1 and Hypothesis 2, Eshghi gives a method of devising discriminating input values of nodes (A),...,(D) which will give different outcomes on the node (O), depending on which hypothesis is assumed to be a correct explanation of the faulty circuit. That is, by simulating the circuit behaviors derived from Hypothesis 1 and from Hypothesis 2 respectively, if, for the same input values, the output state from one hypothesis is different from that of the another, then such input values are the discriminating input values.

On the contrary, in our approach, we assume that states of all nodes in the circuit are observable by performing experiments. Without changing initial values of the input nodes, we find a node which has different states, depending on which hypothesis is assumed to be correct. From the standpoint of computation-complexity, our divide-and-query algorithm is more efficient than Eshghi's method, because the latter method might examine exhaustively all states of nodes in the circuit. Furthermore, we believe that there are some cases where the information obtained from the crucial literal will be used for finding discriminating input values without exhaustive search of input values.

[Example 2]

The second example is the one from Shapiro's model inference system [3]. Assume that we are now synthesizing a "qsort" program and that as facts of qsort, for example, the followings are given.

```
qsort([1],[1]) is true,
qsort([2,1],[1,2]) is true,
qsort([2,1,3],[2,1,3]) is false.
```

Let these facts be the observations Obs of qsort. Furthermore, suppose that we have the following two candidate programs for "qsort" P1 and P2, both of which explains the given facts above.

```
P1 :
   qsort([], []).
   qsort([X|Xs], Result)
        <-  partition(Xs, X, Lo, Hi),
            qsort(Lo, Sorted_Lo),
            qsort(Hi, Sorted_Hi),
            append(Sorted_Lo, [X|Sorted_Hi], Result).
   partition([X|Xs], Crit, Lo, [X|Hi])
        <-  Crit >= X,
            partition(Xs, Crit, Lo, Hi).
   partition([X|Xs], Crit, [X|Lo], Hi)
        <-  X =< Crit,
            partition(Xs, Crit, Lo, Hi).
   partition([], Crit, [], []).
   append([X|Xs], Ys, [X|Zs])
        <-  append(Xs, Ys, Zs).
   append([],Ys,Ys).
```

P2 :
the same as P1 except the definition of a predicate "partition".

```
split([X|Xs], Crit, [X|Lo], Hi)
     <-  X =< Crit,
         split(Xs, Crit, Lo, Hi).
split([X|Xs], Crit, Lo, [X|Hi])
     <-  Crit =< X,
         split(Xs, Crit, Lo, Hi).
split([], Crit, [], []).
```

It is easily checked that both P1 and P2 satisfy Obs. Actually P2 is a correct program for "qsort", while P1 is a buggy program. As an integrity constraint for "qsort", we consider the following goal IC :

```
<-   qsort(X, Y), not_ordered(Y).
```

where a predicate "not_ordered" is defined as follows :

```
not_ordered([A,B|X]) <- A>B.
not_ordered([A,B|X])
     <-  A =< B, not_ordered([B|X]).
```

Let Pi (i=1,2) be a hypothesis Hi. Then H1,H2 and IC satisfy the condition of [Theorem 3.1]. As a proof tree of H1 U H2 U { IC }, consider the following formula in which variables X and Y occurring in IC are instantiated into [2,1,3].

```
qsort([2,1,3],[2,1,3]), not_ordered([2,1,3])
   qsort([2,1,3],[2,1,3])
        partition([1,3],2,[],[1,3])
                2>1
                partition([3],2,[],[3])
                        2<3
                        partition([],2,[],[])
        qsort([],[])
        qsort([1,3],[1,3])
                partition([3],1,[],[3])
                        1<3
                        partition([],1,[],[])
                qsort([],[])
                qsort([3],[3])
                        partition([],3,[],[])
                        qsort([],[])
                        qsort([],[])
                        append([],[3],[3])
        append([],[2,1,3],[2,1,3])
   not_ordered([2,1,3])
        2>1
```

From the above proof tree, our algorithm finds a crucial literal qsort([1,3],[1,3]), which is a logical consequence of Hypothesis 2, though it cannot be derived from Hypothesis 1. Since qsort([1,3],[1,3]) is a true fact, we know that Hypothesis 1 should be refined.

In Shapiro's Model Inference System, a theory is identified incrementally by the repeated process of refining its hypothesis and refuting that with crucial experiments. A single hypothesis is assumed to cover given observations. When an oracle gives another fact for which the

current hypothesis is too weak, then it is replaced by a strengthened one. If we suppose the situation where multiple hypotheses for given facts are considered during the theory identification, then a crucial literal obtained by our algorithm gives the information which fact is necessary for refining or refuting hypotheses. Hence, it will sometimes save insignificant facts given by the oracle.

6. Concluding Remarks

This paper has presented a logical method to discriminate a more appropriate hypothesis among competing hypotheses. Under the condition of [Theorem 3.1], our algorithm finds a crucial literal which gives a clue to select a "better" hypothesis. The correctness of the algorithm has proved in [Theorem 4.1] and its validity is exemplified by several examples.

We believe that the algorithm described here will be used effectively in the scientific theory formation (such as diagnosis, inductive inference), where generally multiple (possibly still weak) hypotheses can be considered to cover currently known facts and, by performing experiments, we try to find a most appropriate one among them. In such a situation, a clue for refining those hypotheses will be found by our algorithm, although each one of the hypotheses itself might be not correct yet.

The algorithm described here has been implemented in DEC-10 prolog on DEC-2065. In our implementation, we have utilized Shapiro's diagnosis programs [3] with minor changes. In [5], Takeuchi has implemented a divide-and-query algorithm in GHC which has reduced some computations involved in the original Shapiro's algorithm.

[Acknowledgments]

The authors appreciate Kazuhiro Fuchi(Director of ICOT) and Koichi Furukawa (Chief of 1st laboratory) for the chance of doing this research. We are greatly indebted to Randy Goebel, who gave valuable suggestions to our current work while his visit to ICOT. We would like to thank Susumu Kunifuji and other members of ICOT 1st laboratory for their useful discussions.

[Reference]

[1] Popper, K.R., "Conjectures and Refutations : The Growth of Scientific Knowledge", Harper Torchbooks, 1965.
[2] Poole,D., Aleliunas, R., and Goebel,R., "Theorist : a logical reasoning system for defaults and diagnosis", submitted to "Knowledge Representation", IEEE Press, in preparation.
[3] Shapiro,E.Y., "Algorithmic Program Debugging," The MIT Press, 1982.
[4] Eshghi, K., "Application of meta-language programming to fault finding in logic circuits", First International Logic Programming Conference, 1982, pp. 240-246.
[5] Takeuchi, A., a private memo in ICOT, 1985.