Interactive Diagnosis and Testing of Logic Programs

Tamás Horváth (e-mail: h158hor@ella.hu)
Tibor Gyimóthy (e-mail: h42gyi@ella.hu)
Zoltán Alexin (e-mail: h157ale@ella.hu)
Ferenc Kocsis (e-mail: h155koc@ella.hu)

Research Group on the Theory of Automata
Hungarian Academy of Sciences, H-6720 Szeged
Aradi vértanúk tere 1., Hungary

Abstract: In this paper a method (called IDT) is presented which combines Shapiro's Interactive Diagnosis Algorithm with the Category Partition Testing Method. This method can be used both in the debugging and in the testing of Prolog programs. The basic idea of IDT is that the test database prepared by the Category Partition Testing can be used to reduce the amount of user interaction during the debugging. In the IDT method the diagnosis process effects the testing hence IDT can be considered as an integrated debugging and testing method.

Keywords: logic programs, category partition testing method, algorithmic debugging

1 Introduction

The automatic program debugging technique introduced by Shapiro [SHA83] can isolate an erroneous procedure, given a program and an input on which it behaves incorrectly. Shapiro's model has been applied to Prolog programs to diagnose the following three types of errors: termination with incorrect output, termination with missing output, and nontermination. A major drawback of this debugging method is the great number of queries to the user for the correctness of intermediate results of procedure calls. Thus, an important improvement would be to minimize the amount of information the user needs to supply for algorithm to diagnose the error.

An algorithmic debugging method GADT (Generalized Algorithmic Debugging and Testing) for imperative languages is discussed in [FRI91]. A major improvement in bug-localization process is demonstrated in GADT by combining the Category Partition Testing Method (CPM) [OST88] with the algorithm introduced in [SHA83]. The main concept of the improvement is the following: during the debugging of a program the user has to answer many "difficult" questions. If this program has been already tested by CPM, the test results for the procedures of the program can be used in the debugging process.

Of course, during the testing the programs cannot be tested with all possible properties and values of the input parameters. Hence, the first task of the tester in CPM is to define the critical properties of parameters (called *categories*) of the procedures. These categories can be divided into classes (called *choices*) presuming that the behavior of the elements of one choice is identical from the point of view of the test process. From the specification of choices *test frames* are generated. A test frame contains exactly one choice from each category. The behavior of a test frame from the point of view of the testing process can be represented by an arbitrary element of it (called *test case*). Running the given procedures on the corresponding test cases a *test database* can be created. An item of this test database contains the

*** This work is supported by OTKA-501 and ESPRIT BRA 6020.**

identifier of the test frame, the concrete input parameters (test case), the output values and the evaluation result of this test frame (*yes*/*no*/*undefined*).

In the debugging process the concrete values of the parameters of a procedure are given. By determining the corresponding test frame to a given input the test database can be checked with the selected frame. In the case of 'good' tests report the debugger skips to the next procedure without query.

In this paper we present a method (called *IDT*) which combines Shapiro's Interactive Diagnosis Algorithm with CPM. This method can be used both in the debugging and testing of Prolog programs. The basic idea of the IDT system is similar to GADT e.g., by using the test database prepared by the CPM the number of user interaction can be reduced during the debugging. Shapiro's diagnosis algorithm presented in [SHA83] traverses the refutation tree of a program and asks the user about the expected behavior of each clause (which can be considered as a procedure). The user has to give yes or no answer and the algorithm isolates an error inside a certain procedure body.

Shapiro suggests that the number of queries can be reduced by storing the answers to previous queries. However, the probability that the same query for a procedure is used several times during the debugging is very small. In the IDT method we also store the queries, but each query represents a (equivalence) class of the possible inputs. These classes are defined during the CPM testing process. A query can be represented by a triple <p, x, y>, where p is a procedure and x and y vectors over some domain D, such that p on x returns y. For a query <p, x, y> a test frame t is identified on the basis of the CPM test specification of procedure p. This identification can be done automatically by using predefined *searching functions* in the test specification for the inputs and classes. When these searching functions are not given new queries are generated to the user to determine the correct test frame. Usually it is much easier to answer these queries than to the original ones. After the test frame t is identified to a query <p, x, y> the system is looking for a stored query <p, x', y'> which represents the test frame t. If the procedure p was previously tested by such query <p, x', y'> then the result of this query is used to answer the query <p, x, y>. Of course sometimes the assumption that queries <p, x', y'>, <p, x, y> give the same result may be incorrect. The 'visible' effect of such a false assumption may be that a bug could not be localized using IDT method. In this case we use Shapiro's original algorithm to identify the incorrect procedure. Assume that the query <p, x, y> was used to isolate the incorrect procedure p and this procedure was tested with the query <p, x', y'>. Suppose that <p, x, y> and <p, x', y'> belong to the same test frame t and the result of the query <p, x', y'> is *yes* but the result of the query <p, x, y> is *no*. In this case the partition specification for the procedure p during CPM testing was not correct. Therefore the IDT method prepares a new partition for the inputs of the procedure p. In this new partition the inputs x and x' represent different test frames. Hence, the IDT method can be considered as an improvement of CPM algorithm presented in [OST88].

The main differences between the GADT and the IDT methods are that the GADT has been applied to imperative languages while IDT to Prolog and in the IDT the diagnosis process effects testing. Hence IDT can be considered as an integrated debugger and testing method.

In the rest of this paper we firsts give a brief overview of Shapiro's algorithm in Section 2. Section 3 contains a formal description of the Category Partition Method. In Section 4 we describe the IDT method and an example for this method is presented in Section 5. Finally in Section 6 the conclusion and some remarks for future works are described.

2 Shapiro's single-stepping diagnosis algorithm

In this section we give a short overview of an algorithm presented in [SHA83] and recall definitions related to this algorithm. Shapiro's single-stepping algorithm can isolate an erroneous procedure (clause), given a program and an input on which it behaves incorrectly. This algorithm traverses the refutation tree of a program and asks the user about the expected behavior of each clause. The user has to give a *yes* or *no* answer and the bug inside a certain procedure is identified.

2.1 Definition: A *logic program* **P** is a finite set of definite clauses (the clause $A \leftarrow B_1, \ldots, B_n$ is definite iff all B's are atoms, $n \geq 0$).

2.2 Definition: Let C denote the clause $A \leftarrow B_1, \ldots, B_n$ ($n \geq 0$). Then *head*(C) denotes A and *body*(C) is the set $\{B_1, \ldots, B_n\}$.

2.3 Definition: Let **P** be a logic program, M an interpretation of **P**, A' a ground atom and $A \leftarrow B_1, \ldots, B_n$ an arbitrary clause in **P**. We say that $A \leftarrow B_1, \ldots, B_n$ *covers* A' in M iff there is a substitution θ such that $A\theta = A'$ and for all i ($1 \leq i \leq n$) $B_i\theta \in M$.

2.4 Definition: An arbitrary clause $p \in$ **P** is *correct* in M iff all ground atoms covered by p are in M. Otherwise we say that p is incorrect in M.

2.5 Definition: Let p be an arbitrary clause in **P** that terminates on some input x and returns y as output. Then the *top level trace* of the triple <p, x, y> is a finite (possibly empty) ordered set $\{\langle p_1, x_1, y_1\rangle, \langle p_2, x_2, y_2\rangle, \ldots, \langle p_n, x_n, y_n\rangle\}$. Where p on input x calls first p_1 with input x_1, that returns y_1 as output then p_2 with x_2, ... , and so on. Finally p call p_n on input x_n which returns y_n and p returns y.

2.6 Definition: A *partial computation tree* of **P** is an ordered tree. Every node in this tree is labeled with some triple <q, u, v>. The set of the direct descendants of an inner node is a legal top level trace of this node. A T tree is a *complete computation tree* of **P** if it is a partial computation tree and all leaves in T are empty sets.

In the following we suppose that the program **P** is free of side-effects. Let p be an arbitrary clause in **P** that terminates on input x and returns y as output such that $\langle p, x, y\rangle \notin M$. It means that **P** has at least one incorrect clause. Then for finding the incorrect clause that causes the error the computation tree rooted by <p, x, y> is traversed in a postorder manner. During the traversing of the tree at each <q, u, v> node a membership question is issued. Let us suppose that the first *false* answer is received at the node <q, u, v>. Let the direct descendants of <q, u, v> be $\langle q_1, u_1, v_1\rangle, \ldots, \langle q_m, u_m, v_m\rangle$. Since we used postorder strategy for all i ($1 \leq i \leq m$) it holds that $\langle q_i, u_i, v_i\rangle \in M$. From this it follows that the clause $q \leftarrow q_1, \ldots, q_m$ covers the triple <q, u, v> which is not in M. The algorithm stops at the node <q, u, v> and returns the clause instance $\langle q, u, v\rangle \leftarrow \langle q_1, u_1, v_1\rangle, \ldots, \langle q_m, u_m, v_m\rangle$. By this method the erroneous clause can always be identified assuming that the answers to the membership questions are correct.

A query-optimal modified version of this method is called divide-and-query. We demonstrate the behavior of the single-stepping method through a small example [SHA83].

2.7 Example: a simple Prolog program for sorting a list.

```
insert(X,[Y|U],[Y|V])    :- Y < X , insert(X,U,V).
insert(X,[Y|U],[X,Y|U]) :- X ≤ Y.
insert(X,[],[X]).

sort([],[])    :- .
sort([X|Y],Z) :- sort(Y,U),insert(X,U,Z).
```

The above Prolog program contains two procedures (*insert and sort*). The procedure *insert* puts a new element into the list — given as the second argument — and returns the result in the third argument. The procedure *sort* sorts the tail of the input list and then inserts the head of the input list into this list.

This program works as follows:

?: sort([-2,-4,3,-6,2,-3],X).

X = [-6,-4,-3,-2,2,3].

For the purposes to demonstrate Shapiro's algorithm let us make a bug in the above program: change in the first line the call **insert(X,U,V)** to **insert(Y,U,V)**. It is a case of simple mistyping.

```
insert(X,[Y|U],[Y|V]) :- Y < X , insert(Y,U,V).
```

Calling the sort with the argument [6,-2,4] it returns X = [-2,-2,4] as a result which is obviously wrong. Let us try to find the wrong procedure using the single-stepping algorithm:

This algorithm works as follows: recursively interprets the call structure of Prolog. It performs the Prolog evaluation from bottom up and each step it asks the user whether the newly determined goal is *true* or *false*. The call tree looks as follows:

```
sort([6,-2,4],[-2,-2,4])
        sort([-2,4],[-2,4])
                sort([4],[4])
                        sort([],[])
                        insert(4,[],[4])
                insert(-2,[4],[-2,4])
                        -2 ≤ 4
        insert(6,[-2,4],[-2,-2,4])
                6 > -2
                insert(-2,[4],[-2,4])
                        -2 ≤ 4
```

The questions, the user should answer are the following:

Query: Is it ok?	*sort([],[])*	yes
Query: Is it ok?	*insert(4,[],[4])*	yes
Query: Is it ok?	*sort([4],[4])*	yes
Query: Is it ok?	*insert(-2,[4],[-2,4])*	yes
Query: Is it ok?	*sort([-2,4],[-2,4])*	yes
Query: Is it ok?	*insert(-2,[4],[-2,4])*	yes
Query: Is it ok?	*insert(6,[-2,4],[-2,-2,4])*	**no**

After seven questions the false procedure call has been identified. The wrong clause instance is:
insert(6,[-2,4],[-2,-2,4]) ← -2 < 6, insert(-2,[-2,4],[-2,-2,4]).

3. Category Partition Testing Method

An informal description of the Category Partition Testing Method can be found in [OST88]. In this section we give a formalization of this functional testing method. During the process of functional testing, the programs (procedures) cannot be tested with all possible properties of the input parameters. Hence, the tester's first task is to define the critical properties of parameters. These critical properties - called categories - are investigated in the testing process.

The categories can be divided into classes - called choices - presuming that the behavior of the elements of one choice is identical from the point of view of the testing.

If the categories and choices for a program have been defined, then all the possible test frames can be generated. A test frame contains exactly one choice from each category.

In general, there are many superfluous frames among the generated test frames. These frames can be eliminated by associating selector expressions with the choices. A choice can be made in a test frame if the selector expression associated with the choice is true. The selector expressions contain property names. A property name is also associated with a choice and can be considered as a logical variable. The value of this variable is true if the given frame contains that choice. In example 3.1 we give simple CPM specification for the clause 'insert' and the generated test frames.

3.1 Example: The category-partition specification for 'insert':

```
Test specification: insert
Category: number_of_elements

   Choice: d0: {(X,[]) | X ∈ D}
      property zero

   Choice: d1: {(X,[Y]) | X, Y ∈ D}

   Choice: d2: {(X,[Y|Z]) | X, Y ∈ D, Z ∈ D+}

Category: first_element

   Choice: e0: {(X,[]) | X ∈ D}
      if zero

   Choice: e1: {(X,[Y|Z]) | X, Y ∈ D, Z ∈ D* and X ≤ Y}
      if not zero

   Choice: e2: {(X,[Y|Z]) | X, Y ∈ D, Z ∈ D* and X > Y}
      if not zero

End of specification
```

The generated test frames are: $\{(d_0,e_0), (d_1,e_1), (d_1,e_2), (d_2,e_1), (d_2,e_2)\}$

In the following we give a short formal description of the Category Partition Method in logic programming environment. Let $\mathbf{P} = \{p_1, \ldots, p_n\}$ denote a logic program, where p_i $(1 \le i \le n)$ is a clause (procedure) in the program. We assume, that the domain of the interpretation is the set D. Then every clause p in $\mathbf{P}$ can be considered as a mapping from D^n into D^m, where n and m are the input and the output arities of p, respectively.

3.1 Definition: Let p an element of $\mathbf{P}$ and n the input arity of p. An equivalence relation over D^n is called a *category* of p.

3.2 Definition: Let p an element of $\mathbf{P}$ and c a category of p. An equivalence class of c is called *choice of c*.

Every choice of a category determines a class of D^n. In the CPM the user can define a choice using an expression (searching function) that depends on a subset of the input variables. By these expressions the choice an arbitrary element of D^n belongs to can be automatically determined. We assign to every procedure a finite set of categories and each category can be considered as a finite set of choices.

3.3 Definition: Let p be an arbitrary clause in $\mathbf{P}$ and let $\Sigma_p = \{\sigma_1, \ldots, \sigma_{pk}\}$ denote the set of the categories of the clause p where $\sigma_i = \{c_1, \ldots, c_{in}\}$ (c's are the choices). Σ_p is a *test specification of the clause* p. A *test specification of the program* $\mathbf{P}$ is the set $\Sigma_{\mathbf{P}} = \{ \Sigma_p \mid p \in \mathbf{P}\}$.

3.4 Definition: Let p be a clause in **P**, and $\Sigma_p = \{\sigma_1 , ... , \sigma_{pk}\}$ a test specification of p. Let $F(\Sigma_p)$ denote a subset of the Cartesian-product σ_1 x ... x σ_{pk}. $F(\Sigma_p)$ is called the *set of test frames* of p and we suppose that $F(\Sigma_p)$ covers the total D^n. The set $F(\Sigma_p)$ is a classification of D^n therefore $F(\Sigma_p)$ can be considered as an equivalence relation over D^n. An element in $F(\Sigma_p)$ is called a *test frame* of p. The set of test frames of the program **P** is the set $F(\Sigma_{\mathbf{P}}) = \{ F(\Sigma_p) \mid p \in \mathbf{P} \}$.

3.5 Definition: Let p be a clause in **P**, $\Sigma_p = \{\sigma_1, ... , \sigma_{pk}\}$ a test specification of p, and $f = (t_1, ..., t_{pk})$ a test frame of p. A test case of f is an element $(d_1 , ... , d_{pk})$, where d_i is in the class (choice) t_i $(1 \le i \le p_k)$. (Of course if f denotes an empty subset of D^n then such a test case does not exist.)

A test frame can be considered as an element of the products of equivalence relations over D^n. The Category Partition Method checks the behavior of p for a given test frame by investigating the result of p on a representative element (test case) of this frame. We assume that the set of test frames is partially evaluated i.e. there are test frames which are not evaluated yet (this is the usual situation in practice). Therefore we have to order an evaluation function to the set of the frames which maps the test frames into the set {*true*, *false*, *undefined*}. The T_GEN [SZU91] system based on the category partition method works as follows: the user has to define for every procedure a set of categories. Every category has to be an equivalence relation. The user can mark the important test frames, which he wants to test. Then the system generates test cases automatically to these test frames, and calls the procedure with the test cases as input. The user as the ground oracle has to decide the truth of an output. The test frame, its test case, the output values and the evaluation is stored into a database. The evaluation of a test case is a function, see below. We assume, that every test frame has at most one test case.

3.6 Definition: Let p be a procedure of **P**, and Σ_p a test specification of p. Then a *CPM-testing of p* by Σ_p is a mapping φ_p: $F(\Sigma_p) \rightarrow$ {*true*, *false*, *undefined*}. Let f be an arbitrary test frame of p, and t a test case of f. Then $\varphi_p(f)$ is *true/false* iff we applied p to t and the output was *true/false* by the ground oracle. Otherwise $\varphi_p(f)$ is *undefined*. (if f denotes an empty set then $\varphi_p(f) = true$).

3.7 Definition: Let f be an arbitrary test frame of p over Σ_p. We say, that the CPM-testing of f is *consistent* iff $\varphi_p(f)$ is independent of the choice of its test case.

3.8 Definition: Let p be an arbitrary procedure in **P** and Σ_p a test specification of p. The CPM-testing of p is *consistent* iff it is consistent for every test frame of p over Σ_p. The CPM-testing of **P** is *consistent* iff every p in **P** is consistent.

3.9 Definition: Let p an arbitrary procedure (clause) in **P** and let Σ_p an arbitrary test specification of **P**. Let χ_p : $D^n \rightarrow F(\Sigma_p)$ mapping, such that $\chi_p(x) = f$ iff f is the corresponding test frame of x. From the previous definitions it follows that χ_p is a function. (i.e. χ_p is the *searching function*.)

3.10 Definition: A *CPM test configuration* of the program **P** over the set of test frames $F(\Sigma_{\mathbf{p}})$ is a finite set of five-tuples $T(F(\Sigma_{\mathbf{p}})) = \{ (p, f, i, o, e) \mid p \in \mathbf{P} , f \in F(\Sigma_p) , i$ is a test case of f, o is the output of clause p on input i and $e \in$ {*true*, *false*, *undefined*} is the evaluation of the test frame f (e.g. $e = \varphi_p(f)$) }. A CPM test configuration can be considered as a test database. For a clause $p \in \mathbf{P}$ the CPM test configuration of the clause p contains all elements of $T(F(\Sigma_{\mathbf{p}}))$ such that the first component is p.

3.11. Definition: Let σ be an equivalence relation over D^n and let x be an arbitrary element of D^n. Then $\sigma[x]$ denotes the class of σ containing element x. Let σ_1 and σ_2 be two equivalence relations over D^n. We say that $\sigma_2 \ge \sigma_1$ iff from $\sigma_2[a] = \sigma_2[b]$ it follows that $\sigma_1[a] = \sigma_1[b]$ for any a, b $\in D^n$. Let $F(\Sigma_p)$ and $F(\Sigma'_p)$ be two sets of test frames of an arbitrary clause p in **P.** We say that the CPM test configuration of clause p $T(F(\Sigma'_p))$ is a *refinement of T(F(Σp))* iff $F(\Sigma'_p) \ge F(\Sigma_p)$ (i.e. $\forall \pi \in F(\Sigma_p)$ $\exists \rho \in F(\Sigma'_p)$ such that

$\rho \geq \pi$). For a program **P** a CPM test configuration of **P** $T(F(\Sigma'_{\mathbf{p}}))$ is a *refinement of* $T(F(\Sigma_{\mathbf{p}}))$ iff $T(F(\Sigma'_p))$ is a refinement of $T(F(\Sigma_p))$ for all $p \in$ **P.**

If T_1 and T_2 are two CPM test configurations of the same program **P** and T_2 is a refinement of T_1 then the cardinality of the inconsistent test frames of T_2 is less or equal than of T_1.

4 The Interactive Diagnosis and Testing Algorithm

In Section 2 the single-stepping method for algorithmic debugging of logic programs has been briefly summarized. This algorithm can be applied only for those clauses that terminate on some input and result in an incorrect output. For these inputs the algorithm traverses the computation tree rooted by <p, x, y> in postorder manner. At each node it generates a membership question. It stops at the first node <q, u, v> which is not in the interpretation M. The number of the queries depends on the position of the incorrect clause instance inside the computation tree. In order to reduce the number of the queries an extended method called divide-and-query has been proposed. We present our algorithm as a modification of the single-stepping method but it can be used with the divide-and-query method as well.

We assume that there exists a CPM test configuration for the program P i.e. for all $p \in P$ a set of test frames is defined and each test frame may have an evaluation. In the process of algorithmic debugging we use the assumption that the given CPM test configuration is consistent. From this assumption it follows that if a test frame is already evaluated then this evaluation is independent of the test case chosen. This is only a hypothesis of the user which may be false. Therefore we should take into account that there are inconsistent test frames in the CPM test configuration of **P**. If a referred test frame is not evaluated (i.e. φ is *undefined* for this test frame) we automatically generate a membership question.

Let $p \in$ **P** be a clause and let us suppose that $\langle p, x, y\rangle \notin M$ for some x and y and the complete computation tree rooted by <p, x, y> is finite. Similarly to the single-stepping method our algorithm walks this tree in postorder manner. Let us suppose that we have to answer the query <q, u, v> is in M or not, where <q, u, v> is a node in the tree. We examine whether the test frame belonging to q and u is already evaluated (i.e. $\varphi_q(\chi_q(u))$ is not *undefined*). If this test frame has already been evaluated then the result of the evaluation is used to answer the membership query.

Since our hypothesis for the consistency of the test frames is not proved therefore the result given by our algorithm should be verified. On one hand it is possible that we will not find the false clause. On the other hand, if a false clause has been identified by the IDT algorithm it may occur that the test frame belonging to the head of the clause is inconsistent or in the body of the clause instance there is an atom whose test frame is inconsistent. This latter case means that the clause we found is not the deepest incorrect clause in the tree. For resolving this inconsistency the original single-stepping method is invoked. We store all questions and answers in a test database (CPM test configuration) therefore the number of queries the user has to answer is less or equal (in the worst case) than in the original single-stepping method. In the case of consistent test frames the number of queries may be significantly decreased. If we find an inconsistent test frame during the debugging then the user has to modify the CPM test configuration. More exactly the user has to define a refinement for the CPM test configuration such that the new configuration will be consistent regarding the known test cases.

4.1 Algorithm for scanning the computation tree by IDT:

Input: <p, x, y> not in M,
Output: q false clause instance and f Boolean.

```
procedure IDT_Debug (<p,x,y>,q,f) ;
begin
  let T := { <p1,x1,y1>, <p2,x2,y2>, ... , <pn,xn,yn>}
                        be the top level trace of <p,x,y>.
  f := false ;
  i := 1 ;
  while (i ≤ |T|) and (not f) do
  begin
    IDT_Debug ( <pi,xi,yi>,q,f ) ;
    inc(i) ;
  end ;
  if not f then
  begin
    if φp(χp(x)) = undefined then ans := Query (<p,x,y>) ;
    f := φp(χp(x)) ;
    if not f then
    begin
      f := true ;
      q := <p,x,y> ← <p1,x1,y1>,<p2,x2,y2>,...,<pn,xn,yn>;
    end ;
  end ;
end ;
```

Let us suppose that $\langle p, x, y\rangle \notin M$ for some $p \in \mathbf{P}$, $x \in D^n$ and $y \in D^m$. The Algorithm 4.1 gets the triple $\langle p, x, y\rangle$ as an input and returns in f a Boolean value which shows whether the procedure found the incorrect clause instance. If this value is *true* then q will contain this false clause instance. Let us suppose that $\{\langle p_1, x_1, y_1\rangle, \langle p_2, x_2, y_2\rangle, \ldots, \langle p_n, x_n, y_n\rangle\}$ is the top level trace of $\langle p, x, y\rangle$. The algorithm recursively calls itself on the elements of the top level trace until it finds a false clause (i.e. f = *false*). If for all i ($1 \le i \le n$) the evaluation of the test frame belonging to $\langle p_i, x_i, y_i\rangle$ is *true* then let the test frame belonging to $\langle p, x, y\rangle$ be examined. If this test frame is not evaluated yet (i.e. $\varphi_p(\chi_p(x)) =$ *undefined*) then the **Query(<p,x,y>)** is asked (see later). Otherwise (i.e. $\varphi_p(\chi_p(x)) \neq$ *undefined*) the membership question is not printed out but it is answered with the evaluation of the test frame. If $\varphi_p(\chi_p(x)) =$ *false* then the algorithm stops and returns the corresponding instance of the clause p in variable q.

4.2 Algorithm for finding false procedure by IDT:

Input: <p, x, y> not in M,
Output: r false clause instance.

```
procedure Main_IDT_Debug (<p,x,y>,r) ;
begin
  IDT_Debug (<p,x,y>,r,f) ;
  if f then
  begin
    let r = <q,u,v> ← <q1,u1,v1>,<q2,u2,v2>,...,<qm,um,vm>
    ans := Query (<q,u,v>) ;
    if ans = false then
    begin
      ans := Query_Top_Level_Trace (r);
      if ans = true then
        return (r)
      else
      begin
        Modified_Single_Step (<q,u,v>,r,f);
        return (r);
      end ;
    end ;
  end ;
  Modified_Single_Step (<p,x,y>,r,f);
  return (r) ;
end ;
```

Let us suppose that the Algorithm 4.1 has found the clause instance $\langle q, u, v\rangle \leftarrow \langle q_1, u_1, v_1\rangle$, $\langle q_2, u_2, v_2\rangle$,..., $\langle q_m, u_m, v_m\rangle$. Then we should examine the following cases.

1. A membership question for <q, u, v> should be asked. If this concrete question has already occurred then the result is retrieved from the test database. For this purpose the function **Used_Test_Case** is invoked. If *true* was answered then the test frame belonging to p and x is inconsistent so a call of **Modified_Single_Step** is necessary.

2. If in 1 there was no inconsistency then for all elements of the top level trace of <q, u, v> the Query($\langle q_i, u_i, v_i\rangle$) should be issued. If all elements in the top level trace are in the interpretation M then the false clause instance is $\langle q, u, v\rangle \leftarrow \langle q_1, u_1, v_1\rangle$, $\langle q_2, u_2, v_2\rangle$,..., $\langle q_m, u_m, v_m\rangle$. Otherwise if there is an element $\langle q_i, u_i, v_i\rangle$ in the top level trace whose test frame is inconsistent then the **Modified_Single_Step** should be called on the triple <q, u, v> since we know that the error is in the computation tree rooted by <q, u, v>.

If the algorithm did not find a false clause then there is at least one node (the root surely) in the computation tree of <p, x, y> whose test frame is inconsistent.

The above Algorithm 4.2 handles these cases.

4.3 Utilities for finding a false procedure by IDT:

Input: clause instance $\langle q, u, v\rangle \leftarrow \langle q_1, u_1, v_1\rangle, \langle q_2, u_2, v_2\rangle$,..., $\langle q_m, u_m, v_m\rangle$,
Output: a Boolean value.

```
function Query_Top_Level_Trace (c) : boolean ;
begin
  Let c := <q,u,v> ← <q1,u1,v1>,<q2,u2,v2>,...,<qm,um,vm>;
```

```
    f := true ;
    i := 1 ;
    while (i ≤ |body(c)|) and (f) do
    begin
      f := Query(<q_i,u_i,v_i>) ;
      inc(i) ;
    end ;
    return (f) ;
  end ;
```

Query_Top_Level_Trace gets a clause instance as input. It returns *true* if for all i ($1 \le i \le m$) $<q_i, u_i, v_i>$ is in M.

Input: the triple <p, x, y>,
Output: a Boolean value.

```
  function Query (<p,x,y>) : boolean ;
  begin
    if Used_Test_Case(p,x) then  return( φ_p(χ_p(x)) ) ;
    f := φ_p(χ_p(x)) ;
    ans := Is_In_Interpretation (<p,x,y>) ;
    if (f ≠ undefined) and (f ≠ ans) then
      divide test frame χ_p(x) into two parts consistently.
    Store (<p,x,y>,ans) ;
  end ;
```

The function Query receives a triple <p, x, y> as input and returns a logical value. If there is a test case in the database which equals <p, x, y> then it returns the stored answer. Otherwise the question <p, x, y> ∈ M is asked from the user. If the test frame belonging to p and x is not evaluated yet then the answer is stored into this frame. In the other case we compare the answer with the evaluation of the frame. If these values are different then the test frame is inconsistent therefore the user is requested to refine the CPM test configuration consistently with the known set of test cases. Naturally all questions and answers are stored.

4.4 The modified single-stepping algorithm:

Input: <p, x, y> not in M,
Output: false clause instance q and Boolean f.

```
  procedure Modified_Single_Step (<p,x,y>,q,f);
  begin
    let T := { <p_1,x_1,y_1>, <p_2,x_2,y_2>, ... , <p_n,x_n,y_n>}
        be the top level trace of <p,x,y>.
    f := false ;
    i := 1 ;
    while (i ≤ |T|) and (not f) do
    begin
      Modified_Single_Step (<p_i,x_i,y_i>,q,f);
      inc(i) ;
    end ;
    if not f then
    begin
      f := not Query (<p,x,y>) ;
      if f = true then  q := <p,x,y> ←
  <p_1,x_1,y_1>,<p_2,x_2,y_2>,...,<p_n,x_n,y_n> ;
    end ;
  end ;
```

Algorithm 4.4 differs from the original single-stepping method [SHA83] in that it uses the procedure **Query** which loops back to the CPM test configuration.

Let us suppose that <p, x, y> is not in the interpretation M. Let |N(<p, x, y>)| denote the cardinality of the nodes of the complete computation tree rooted at <p, x, y> (it is supposed to be finite). Then $0 \le n \le |N(<p, x, y>)|$ where n is the number of queries the user should answer. The best case is the following: the Algorithm 4.1 found exactly the false clause. Let $<q, u, v> \leftarrow <q_1, u_1, v_1>, <q_2, u_2, v_2>, ..., <q_m, u_m, v_m>$ be this clause. In Algorithm 4.2 we have to call the function Query m+1 times. If all of these queries occurred in the database then it terminates without any questions. The worst cases are: either the Algorithm 4.1 did not find the false clause or it finds one and the test frame belonging to its head is inconsistent. Both cases are caused by inconsistent test frames. In these cases we should call the original single-stepping algorithm.

5 Example

In this section we propose a detailed example on testing a sorting program. Let the program **P** be the following[SHA83]:

```
insert(X,[Y|U],[Y|V])    :- Y < X , insert(Y,U,V).
insert(X,[Y|U],[X,Y|U]) :- X ≤ Y.
insert(X,[],[X]).

sort([],[])    :- .
sort([X|Y],Z) :- sort(Y,U),insert(X,U,Z).
```

$\Sigma_{sort} = \{\sigma_{sort}\}$.
σ_{sort}.= $\{c_0, c_1, c_2\}$.
c_0 = {the empty list}
c_1 = {all lists that contain exactly one element}
c_2 = {all lists that contain more than one element}

$\Sigma_{insert} = \{\sigma_{insert1}, \sigma_{insert2}\}$.
$\sigma_{insert1} = \{d_0, d_1, d_2\}$.
$d_0 = \{(X,[]) \mid X \in D\}$
$d_1 = \{(X,[Y]) \mid X, Y \in D\}$
$d_2 = \{(X,[Y|Z]) \mid X, Y \in D, Z \in D^+\}$

$\sigma_{insert2} = \{e_0, e_1, e_2\}$.
$e_0 = \{(X,[]) \mid X \in D\}$,
$e_1 = \{(X,[Y|Z]) \mid X, Y \in D, Z \in D^*$ and $X \le Y\}$
$e_2 = \{(X,[Y|Z]) \mid X, Y \in D, Z \in D^*$ and $X > Y\}$

It is obvious that the category σ_{sort} is an equivalence relation over the domain D^*, and similarly $\sigma_{insert1}$, $\sigma_{insert2}$ are also equivalence relations on D x D^*. The set of test frames $F(\Sigma_{\mathbf{P}}) = F(\Sigma_{sort}) \cup F(\Sigma_{insert})$ are defined as follows:

$F(\Sigma_{sort}) = \{c_0, c_1, c_2\}$
$F(\Sigma_{insert}) = \{(d_0, e_0), (d_1, e_1), (d_1, e_2), (d_2, e_1), (d_2, e_2)\}$

For these frames the following test cases are generated:

c_0	[]
c_1	[6]
c_2	[1,4,7]

(d_0, e_0)	(6,[])
(d_1, e_1)	(4,[20])
(d_1, e_2)	(10,[6])
(d_2, e_1)	(7,[10,5,1])
(d_2, e_2)	(15,[9,8])

Here is the content of the test database:

$T(F(\Sigma_{\mathbf{P}}))$ =

{ (**sort**, c_0, [], [], *true*),
(**sort**, c_1, [6], [6], *true*),
(**sort**, c_2, [1,4,7], [1,4,7], *true*),
(**insert**, (d_0, e_0), (6,[]), [6], *true*),
(**insert**, (d_1, e_1), (4,[20]), [4,20], *true*),
(**insert**, (d_1, e_2), (10,[6]), [6,6], *false*)
(**insert**, (d_2, e_1), (7,[10,5,1]), [7,10,5,1], *true*),
(**insert**, (d_2, e_2), (15,[9,8]), [9,8,8], *false*)
}

The sort program terminates with an incorrect output on the list [6,-2,4]. We call the Algorithm 4.2 on *<sort,[6,-2,4],[-2,-2,4]>*. The first step is to call the *IDT_Debug (<sort,[6,-2,4],[-2,-2,4]>,r,f)*. IDT_Debug works on this input as follows:

It calls the function **`Query`** with the following arguments but the answers are taken from the CPM test configuration.

<sort, [], []>	$\chi_{sort}(\ [\ ]\ ) = c_0$	answer = $\varphi_{sort}(c_0)$ = *true*
<insert, (4, []), [4]>	$\chi_{insert}(\ (4, [\ ])\ ) = (d_0, e_0)$	answer = $\varphi_{insert}((d_0, e_0))$ = *true*
<sort, [4], [4]>	$\chi_{sort}(\ [4]\ ) = c_1$	answer = $\varphi_{sort}(c_1)$ = *true*
<insert, (-2, [4]), [-2, 4]>	$\chi_{insert}(\ (-2, [4])\ ) = (d_1, e_1)$	answer = $\varphi_{insert}((d_1, e_1))$ = *true*
<sort, [-2, 4], [-2, 4]>	$\chi_{sort}(\ [-2, 4]\ ) = c_2$	answer = $\varphi_{sort}(c_2)$ = *true*
<insert, (-2, [4]), [-2, 4]>	$\chi_{insert}(\ (-2, [4])\ ) = (d_1, e_1)$	answer = $\varphi_{insert}((d_1, e_1))$ = *true*
<insert,(6, [-2, 4]), [-2, -2, 4]>	$\chi_{insert}(\ (6, [-2, 4])\ ) = (d_2, e_2)$	answer = $\varphi_{insert}((d_2, e_2))$ = *false*

The returned clause instance is *insert(6,[-2,4],[-2,-2,4]) ← -2 < 6, insert(-2,[-2,4],[-2,-2,4])*. First we should ask the membership question:

? *insert(6,[-2,4],[-2,-2,4])* ∈ M *no*.

Since the answer was *no* we inspect all atoms in the body:

? *insert(-2,[-2,4],[-2,-2,4])* ∈ M *true*.

The Algorithm 4.2 terminates successfully and returns in r the clause instance **insert(6,[-2,4],[-2,-2,4]) ← -2 < 6, insert(-2,[-2,4],[-2,-2,4])** which is exactly the false clause.

6 Summary

In this paper we have presented an overview of an Interactive Diagnosis and Testing method for logic programs. The basic idea of this approach is that we try to give an integration of the debugging and testing phases of the software development process. It means that the IDT algorithm introduced in this paper uses the test results of a program during the bug localization process to reduce the number of user interaction. In addition, if during the IDT algorithm an inconsistent test frame is identified then a refinement of the given test specification is prepared.

On the basis of the IDT method a prototype system (called *IDTS*) is under development. A small prototype for CPM test generator in SB-Prolog environment is implemented. The Shapiro's algorithms for automatic program debugging are under implementation. After the complete implementation we will integrate the testing and debugging programs and we will prepare the system IDTS.

We believe that the IDT method can be applied in the process of Interactive Inductive Logic Programming. We try to investigate this topic in the future.

7 References

[FRI91] Peter Fritzson, Tibor Gyimóthy, Mariam Kamkar, Nahid Shahmeri Generalized Algorithmic Debugging and Testing. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada June 26-28, 1991. pp. 317-326.

[OST88] Thomas J. Ostrand, Marc J. Balker: The Category-Partition Method for Specifying and Generating Functional Tests. CACM 31:6, June 1988. pp. 676-686.

[SHA83] E. Y. Shapiro: Algorithmic Program Debugging MIT Press 1983.

[SHA90] N. Shahmeri, M. Kamkar and P. Fritzson: Semi-automatic Bug Localization in Software Maintenance. San Diego, Nov. 26-29, 1990.

[SZU91] Róbert Szücs, Tibor Gyimóthy: T-GEN: An Extended Version of the Category Partition Testing Method. Proceedings of the Second Symposium on Programming Languages and Software Tools, Pirkkala Finland August 21-23, 1991. pp. 70-77.