Proceedings of the Fourth Symposium on Programming Languages and Software Tools
Visegrád, Hungary, June 9-10, 1995.

IDT: Integrated System for Debugging and Testing Prolog Programs*

Zoltán Alexin[1], Tibor Gyimóthy[2], Gabriella Kókai[3]

[1]Department of Applied Informatics József Attila University
e-mail: alexin@inf.u-szeged.hu

[2]Research Group on the Theory of Automata at the Hungarian Academy of Sciences
e-mail: gyimi@inf.u-szeged.hu

[3]Department of Informatics József Attila University Hungary
e-mail:kokai@inf.u-szeged.hu

Abstract

In this paper the IDT (Interactive Diagnosis and Testing) system is presented which extends the Shapiro's Interactive Diagnosis Algorithms with Category Partition Testing methods. Shapiro's original system demands a lot of user interactions during debugging; the user has to answer large number of queries asked by the system. The results of the debugging are the buggy clause and one of the following three type of errors: termination with incorrect output, termination with missing output and nontermination. The goal of the IDT system is to reduce the number of user interactions by introducing a test-database that based on a Category Partition Testing specification of the target program. The CPM specification gives a classification of the domain of the target program into equivalence classes. After that each class is represented by one of its representative lements. The close relationship between diagnosis algorithm and inductive learning of logic programs provides the possibility to use the IDT system in the learning of logic programs too.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging; D.1.6 [Programming Techniques]: Logic Programming

Key Words: Logic programs, category partition testing methods, learning

1 Introduction

In this paper we present a system (called IDT) which combines Shapiro's Interactive Diagnosis Algorithm [5] with CPM (Category Partition Method) [3]. This system can be used in the testing, debugging and learning of Prolog programs.

* This work is supported by the ESPRIT project BRA 6020 and by the OTKA grant 501

Shapiro's Interactive Diagnosis Algorithm was originally applied to Prolog programs to diagnose the following three types of errors: termination with incorrect output, termination with missing output, and nontermination. The main problem with this diagnosis algorithm is that the user (oracle) has to answer many queries during the diagnosis process.

These questions may be very difficult to answer, therefore the application of this diagnosis method to practical problems is questionable. Shapiro suggests that 'query complexity' of this algorithm can be improved by using test results. However, to our knowledge the IDT system is the first integrated tool for testing, debugging and learning of Prolog programs.

The basic concept of IDT is very similar to GADT (Generalized Algorithmic Debugging and Testing) presented in [1]. In GADT the CPM (Category Partition Testing Method) [3] was combined with an algorithmic debugging method to localize a bug in the program using minimal number of user interactions. In CPM the tester has to define equivalence classes called test frames over the input domain. There is an assumption that the behaviour 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).

The IDT method for logic programs is presented in [2]. This method can be used for both debugging and testing of Prolog programs. The main difference between GADT and IDT is that GADT has been applied to imperative languages and IDT to logic programs.

In the debugging process the concrete values of parameters of procedures are given. By determining the corresponding test frame to a given input the test database can be checked against the selected frame. In the case of successful test the debugger skips to the next procedure without a query to the user.

A further improvement for bug-localization process is presented in [4]. In that paper a method was introduced which combines the IDT with program slicing.

The close relationship between diagnosis algorithm and inductive learning of logic programs has been reported in [5]. Therefore we can apply the IDT system not only in the debugging and testing but in the learning of logic programs as well.

In the paper first we give a description of the IDT system. Section 4 contains the inductive inference algorithm which synthesizes logic programs from positive and negative samples. The algorithm combines the diagnosis algorithm with CPM methods. Finally, in Section 5 some remarks for the future work is given.

2 Description of the IDT system.

In this section first we give a short overview of CPM and the implementation of this method in Prolog environment. After that the basic idea of Shapiro's single stepping diagnosis algorithm is presented. The last part of this section contains the description of the combination of CPM with diagnosis algorithm.

2.1 Category Partition Testing Method

The CPM method has been defined in [3]. A formal description of this method can be found in [2]. 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 2.1 we give simple CPM specification for the predicate member and the generated test frames, are also presented.

Example 2.1 The category-partition specification for the predicate **member**:

member(X,[X|_]).
member(X,[Y|Z]) :- member(X,Z).

Test specification: member
Category: number_of_the_elements
empty: **property** empty
{The second argument is an empty list}
one: **property** one
{The second argument has exactly one element}
more: **property** more
{The second argument has more than one element}
Category: position_in_the_list
first: **if not** empty
{The first argument and the first element of the second argument are the same.}
last: **if** more
{The first argument and the last element of the second argument are the same}
inside: **if** more
{The first argument can be found in the second argument but it is neither the first nor the last element of the list}
none: **if not** empty
{The first argument can not be found in the second argument}
End of specification

The generated test frames are: {(empty,_), (one,first), (one,none), (more,first), (more,last), (more,inside), (more,none)}

In Example 2.1 the keywords of the specification are in bold. There are two categories (number_of_the_elements, position_in_the_list) and seven choices. There are three choices in the first category (empty, one, more) and four are in the second (first, last, inside, none). The choices in the first category have property names and the choices in the second category have selector expressions. The text enclosed in braces {,} are comments.

From this CPM specification the following test frames are generated: {(empty,_), (one, first), (one, none), (more, first), (more, last), (more, inside), (more, none)}. For example the test frame (more, first) denotes those inputs where the second argument has more than one

element, and the first argument is identical with the first element of the second argument. In this example we assume that both arguments of the predicate member are input.

2.2 The implementation of CPM in Prolog

2.2.1 Preparing the initial test database

The test database is generated by a Prolog program. The input of the program is a test specification which is given in a Prolog form. The descriptions are Prolog facts in the following form:

```
choice_of(category_name_1,choice_name_1).
choice_of(category_name_1,choice_name_2).
...
choice_of(category_name_n,choice_name_n1).
choice_of(category_name_n,choice_name_n2).
...
```

In the IDT system we can choose a function from the following menu:

1. Load CPM description.
2. Generate database.
3. Save database.
4. Load database.
5. Searching functions.
6. Listing database.
7. Clear database.
8. Testing.
9. Reset system.
0. Quit.

We explain only the functions 2, 5, 8. The meaning of the other functions is obvious.

2. Generate database

From the loaded CPM description the program generates the initial test database. The elements of test database are Prolog facts in the following form:

```
test (P, A, F, IO, E),
```

where P is the predicate name, A is the predicate arity, F is the test frame (represented by a list of corresponding choices), IO is the list of input and output values of representing elements, E is the evaluation of the test frame e.g.:

```
test(member,2,[more,first],[],undefined).
```

Every fact begins with a functor test. It follows the name of the predicate member which will be tested, and its arity 2. The third parameter is a test frame [more first]. The next parameter contains the list of pairs of input/output parameters on which this clause will be tested (initially it is an empty list). The last argument of the item will be the evaluation of the test (initally undefined). The test database is manipulated by the Prolog system predicates retract and assert.

5. Searching functions.

In the IDT system searching functions have been introduced to decide which test frame contains the given input. For every choice name a searching function can be defined. If the searching function is not defined the selection of the proper testframe is perfomed via menu. The searching functions are Prolog predicates. For example whether an element is the last element of the list can be decided with a help of these two clauses:

```
last(X,[Y|Ys]):- X \== Y, last(X,Ys).
last(X,[X]).
```

8. Testing.

At the beginning of testing the database does not contain any test case. The user can choose a test frame and can give a representing element of this frame. It means that the user types in the values of the input variables. Then it can be decided whether the evaluation that takes the input and returns the output is correct or not. The user need not to fill in the full database because there are some items which are not relevant for the testing. The test database contains every proper combination of choices. In Example 2.2 we give a test database

Example 2.2 The database for member:

Test frames:

```
test(member,2,[empty,_],[],undefined).
test(member,2,[one,first],[([a,[a]],[a,[a]])],true).
test(member,2,[one,none],[([a,[b]],[a,[b]])],false).
test(member,2,[more,first],[],undefined).
test(member,2,[more,last],[([b,[a,b]],[b,[a,b]])],true).
test(member,2,[more,inside],[],undefined).
test(member,2,[more, none],[],undefined).
```

Searching functions:

```
empty(_,[]).
one(_,[_]).
more(_,[In|_]).
first(In,[In|_]).
last(X,[Y|Ys]):- X \== Y, last(X,Ys).
last(X,[X]).
inside(X,Y):- member(X,Y), \+ first(X,Y), \+ last(X,Y).
none(X,Y):- \+ member(X,Y).
```

2.2.2 Searching in the test database

During the debugging we need the evaluation of some test frames. The *$deb_geteval/5* predicate can return the CPM evaluation of an arbitrary input. First it finds the test frame which the given input belongs to then it returns the evaluation of this test frame.

An input is in a test frame iff all searching functions, associated with the choices of the test frame return true for this input.

Algorithm 1 Searching in the test database

Input: the procedure P, its arity A and the input I.
Output: the test frame Fe which contains this input and the evaluation value E of this test frame.
Algorithm: find all test frames in the test database then selects those test frames for which all searching functions return true.

The part of the program which makes this identification can be seen in Example 2.3.

Example 2.3 Searching in the database

```
$deb_geteval'(P,A,I,E,Fe)            :-   findall(Fr,test(P,A,Fr,_,_),S),
                                          '$deb_solve0'(S,P,A,I,E,Fe).

'$deb_solve0'([],_,_,_, [], []).
'$deb_solve0'([Fr|R],P,A,I,Ee,Fe)    :-   '$deb_solve0'(R,P,A,I,EeRem,FeRem),
                                          ('$deb_gettestframe'(P,A,Fr,I) ->
                                          (Z=[Fr|FeRem],test(P,A,Fr,_,E), Y = [E|EeRem]);
                                          (Z=FeRem, Y=EeRem)),
                                          Ee=Y, Fe=Z.
'$deb_gettestframe'(_,_,[],_).
'$deb_gettestframe'(P,A,[C|CRem],I) :-    \+ undefined_search(C),
                                          append([A,P], I, I1),
                                          T =.. [C|I1],
                                          (T),
                                          '$deb_gettestframe'(P,A,CRem,I).
```

The program is organised as follows. The *$deb_geteval* is the top level clause. The *findall* predicate constructs the set of frames which contains the procedure P with arity A. The predicate *$deb_solve0* takes each choice from the set S. The *$deb_getestframe* examines the elements of the choice list.

2.2.3 The modification of the database

If the program finds the test frame which contains the input parameter and its evaluation is undefined then it modifies the test database in the following manner.

Algorithm 2 The modification of test database

Input: the fact P and the predicate name F of this fact with its arity, A, the input I and the frame C.
Output: the new evaluation S.
Algorithm: the program solves the fact and asks the user if the output is correct or not. There are two cases depending on the length of input/output list. If it is empty the program retracts the item and asserts a new item where the new input/output list has one element and the evaluation will be the answer of the user. In the other case the new input/output pair will be inserted into the input/output list.

That part of the program which makes this retract-assert modification is listed below:

Example 2.4 Modifying the database

```
'$deb_ask_for_solution'(P)              :-  nl, write(' Is it ok ['), write(P),
                                            write('] (y/n) '),flush_output,
                                            '$deb_readln'([V|_]),
                                            '$deb_falseanswer'(V)fail;
                                            (ground(P) →true).
'$deb_bagof0'(X,P,S)                    :-  bagof(X, P, S), !, true ; S = [].
'$deb_ask_for_solutions'(P,S)           :-  '$deb_bagof0'(P, '$deb_ask_for_solution'(P), S),
                                            assert((sol(P,S))).
'$deb_ask_for_test'(P,F,A,I,C,S)        :-  '$deb_bagof0'(P, '$deb_ask_for_solution'(P), S),
                                            P =.. [_|O], '$deb_retractassert'(F,A,C,I,O,S).
'$deb_retractassert'(F,A,C,I,O,S)       :-  test(F,A,C,[],_),!,
                                            retract((test(F,A,C,[],_))),
                                            (S = [] →
                                            assert((test(F,A,C,[(I,O)],incorrect)));
                                            assert((test(F,A,C,[(I,O)],correct))) ).
'$deb_retractassert'(F,A,C,I,O,[S])     :-  test(F,A,C,In,_),
                                            non_member((I,O), In),
                                            retract((test(F,A,C,In,_))),
                                            (S = incorrect →
                                            assert((test(F,A,C,[(I,O)|In],incorrect)));
                                            assert((test(F,A,C,[(I,O)|In],correct)))); true.
```

The *$deb_ask_for_test* is the top level clause in this part of the program. The procedure *$deb_ask_for_solution* asks the user whether the output correct or not. Depending on the answer and the length of the input/output list *'$deb_retractassert* is called with different arguments.

3 Shapiro's PDS (Program Diagnosis System)

3.1 Shapiro's Single Stepping Algorithm

In this section we give a short overview of an algorithm presented in [5] 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.

3.1.1 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,* $n \geq 0$*).*

3.1.2 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* $\{B_1, \ldots, B_n\}$*.*

3.1.3 Definition: *The M interpretation of a logic program* **P** *is the set of all facts on which* **P** *refutes [5]*

3.1.4 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* $(1 \le i \le n)$ $B_i\Theta \in M$.

3.1.5 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.*

3.1.6 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* $\langle p, x, y \rangle$ *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 calls* p_n *on input* x_n *which returns* y_n *and p returns y.*

3.1.7 Definition: *A partial computation tree of* **P** *is an ordered tree. Every node in this tree is labeled with some triple* $\langle q, u, v \rangle$. *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 <p, x, y> is not in 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 $(1 \le i \le m)$ 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 the clause member but in this case it is a wrong version.

Example 3.1 The performance of single stepping algorithm for a buggy version of member (called member_ver):

```
member_ver(X,[X|_]).
member_ver(X,[Y|Z]):- member_ver(Y,Z).

| ?- fp(member_ver(x, [a,a,a,a,a,b] ),X).
Is it ok [member_ver(a, [a,b] )] (y/n) y
Is it ok [member_ver (a, [a, a, b] ) ] (y/n) y
Is it ok [member_ver (a, [a, a, a,b] ) ] (y/n) y
Is it ok [member_ver(a,[a,a,a,a,b])] (y/n) y
Is it ok [member_ver(x,[a,a,a,a,a,b])] (y/n) n
X = (member_ver(x, [a,a,a,a,a,b] ) :-member_ver(a, [a,a,a,a,b] ) ) ?
yes
```

3.2 Modified Single Stepping with Category Partition Method

Now we are ready to modify the Shapiro's Single Stepping method. The idea of modification is that the algorithm does not ask the user about the correctness of each resolved goal if it finds the evaluation of this goal in the database for test frames. The original description of the algorithm and the program written in Prolog can be found in [5]. Now we present only the modified part of the program:

Example 3.2 The modified single stepping

```
'$deb_querytest'(exist,P,V) :-    issystem(P) →( (P)→ V== true ; V == false) ;
                                  solutions(P,S) →
                                  (member(P,S)→V == true ; V == false) ;
                                  functor(P,F,A), P =.. [F|I],
                                  ('$deb_iscpm'(F,A)→
                                   '$deb_geteval'(F,A,I,E,Fr),
                                  (E \== [] →
                                  '$deb_insert_testdatabase'(P,F,A,I,Fr,E,V), !;
                                  '$deb_handle_menu'(P,F,A,I,V));
                                  (fact(P,true), V == true ;
                                  '$deb_ask_for_solutions'(P,S) →
                                  (S == [] → V == false ;
                                  (member(P,S), V == true)))).
'$deb_fpx'((A,B),X)       :-      !, '$deb_fpx'(A,Xa),
                                  (Xa == ok → '$deb_fpx'(B,X) ; X = Xa).
'$deb_fpx'(A,X)           :-      issystem(A)→X = ok, (A) ;
                                  clause(A,B),
                                  '$deb_fpx'(B, Xb),
                                  (Xb \== ok →X = Xb ;
                                  '$deb_querytest'(forall,A,true) →
                                  X = ok ; X = (A :- B)).
```

The above clause *$deb_fpx* traverses the computation-tree in postorder manner and at each step it consults with the test database. If the test database contains the evaluation of the corresponding test frame then this evaluation is taken and the algorithm goes on. If it does not contain such evaluation then the oracle is asked whether the inspected node is correct or not. The *$deb_querytest* predicate consults with the test database and with the oracle. To get the evaluation from the test database it calls the *$deb_geteval* that was presented above. The advantage of our method can be seen if we run it on the predicate member_ver described above

Example 3.3 The advantage of modified single stepping algorithm.

```
| ?- fpx(member_ver(x, [a,a,a,a,a,b] ),X).
Is it ok [member_ver(a,[a,b])] (y/n) y
Is it ok [member_ver(x,[a,a,a,a,a,b])] (y/n) n
X = (member_ver(x,[a,a,a,a,a,b]):-member_ver(a,[a,a,a,a,b])) ?
yes
```

In this example we supposed that the system uses the test database presented in Example 2.2. Therefore the IDT system first gives a question for the input (a,[a,b]) because the test frame (more, first) was undefined (see Example 2.2). However after the answer yes the evaluation of this test frame changed to true. After that the system does not give more questions to this frame. The state of the test database after running the Example 3.3 can be seen below.

```
| ?- listing(test/5)

test(member_ver, 2,[empty, []], [ ( [a, [] ], [a, [] ] ) ], false).
test(member_ver, 2,[one, first], [ ( [a, [a] ], [a, [a] ] ) ], true).
test(member_ver, 2,[one,none], [ ( [a, [b] ], [a, [b] ] ) ], false).
test(member_ver, 2,[more, last], [ ( [b, [a,b] ], [b, [a,b] ] ) ], true).
test(member_ver, 2,[more, inside], [], undefined).
test(member_ver, 2,[more, first], [ ( [a, [a,b] ], [a, [a,b] ] ) ], true).
test(member_ver, 2,[more,none], [ ( [x, [a,a,a,a,a,b] ], [x, [a,a,a,a,a,b] ] ) ], false).
yes
```

4 The improved inductive program synthesis

In this section we apply the CPM method for improving the inductive program synthesis algorithm published in [5]. A detailed description of the concept and implementation (MIS Model Inference System) of this algorithm can be found in [5]. Now we give an informal description which is helps to understand our improvement.

4.1 The inductive inference algorithm of Shapiro

A presentation of a program P is a (possibly infinite) sequence of input/output samples of P in which every input in the domain of P eventually appears. Assume that the algorithm is given an initial program (normally the empty program) and a presentation of some target program. The inference algorithm reads the samples one at a time, and performs modifications to the initial program as it is necessary. The inference algorithm is said to identify the target program in the limit, if eventually there comes a time when the modifications it perform results in a program with the same input/output behavior as the target program, and it does not modify this program afterwards.

Note that within this definition, a inference algorithm based on patching alone will not identify a program in the limit if the initial program behaves incorrectly on an infinite number of inputs. The algorithm that has a fixed initial program (say, the empty program) is called an inductive inference algorithm.

4.2 The algorithm for inductive program synthesis

Algorithm 3 Inductive program synthesis

Given: a (possibly infinite) ordered set of clauses **L**,
an oracle for an interpretation **M**,
an oracle for a well-founded ordering > on the domain of **L**,
and a definition of **X**, the parameterized interpretation.

Input: a (possibly infinite) list of facts about **M**.
Output: a sequence of programs $\mathbf{P_1}$, $\mathbf{P_2}$,... in L each of which is totally correct with respect to the known facts.

Algorithm:

set **P** to be the empty program
let the set of marked clauses be empty.
repeat
read the next fact.
repeat
if the program **P** fails on a goal known to be true
then find a true goal **A** uncovered by **P**
search for an unmarked clause **p** in **L** the covers **A** in **X**;
add **p** to **P**.
if the program **P** succeeds on a goal known to be false
then detect a false clause **p** in **P**
remove **p** from **P** and mark it.
until the program **P** is totally correct
with respect to the known facts
output **P**.
until no facts left to read.

if the depth of a computation of **P** on some goal **A** exceeds h(**A**), then apply the stack overflow diagnosis algorithm, which either
detects a clause **p** in **P** that is diverging with respect to > and **M**, or a clause **p** in **P** that is false in **M**;
remove **p** from **P**, mark it, and restart the computation on **A**.

4.3 The modified MIS

if the program **P** succeeds on a goal known to be false
then detect a false clause **p** in **P**
remove **p** from **P** and mark it.

We modified the part of the Algorithm 3 listed above. Originally for detecting the false clause in the logic program the Shapiro's single stepping algorithm was used. Instead of this we introduced the modified single stepping algorithm (see Section 3.2) to avoid the questions which must be answered by the user.

Initially we have some knowledge about the behaviour of the program to be learned and the initial test database. All the evaluations in the test database are undefined.

We use in every step of the learning algorithm those frames of test database that are valid. We mean those frames that are consistent with some positive samples. If the algorithm produces a new program which is correct for every sample and we want to learn a new sample with the help of the inference system we must qualify again our test database.

The advantage of our modified system can be seen for example in the learning of member. The Algorithm 3 finds the clause by using the refinement operator

```
member(X,[Y|Z]) : - member (Y,Z).
```

This clause is of course wrong. Let member(x,[a,a,a,a,b]) be one of the negative samples. If the inferred program would accept it the program have to be corrected. The inference system asks questions (see Example 3.1) to find the false clause. Our modification decreases the number of questions asked. In the example above we saved four questions. In the case of more samples more questions might be saved.

5 Conclusion

In this paper a system has been presented which combines the Category Partition Testing with the algorithmic debugging techniques introduced in [5]. A similar method is presented in [1] to diagnose imperative programs but as far as we know the IDT system presented in this paper is unique in the context of logic programming. This integrated tool can be used in the testing, debugging and learning of Prolog programs.

The Category Partition Testing Method in Prolog environment have already been implemented. The integration of CPM with the single stepping and divide and query diagnosis algorithms are also implemented. The integration of CPM with the other two diagnosis algorithms and the MIS is currently under development.

References

[1] 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.

[2] Tamás Horváth, Tibor Gyimóthy, Zoltán Alexin, Ferenc Kocsis: Interactive Diagnosis and Testing of Logic Programs. Proceedings of the Third Symposium on programming languages and Software Tools Ed: Mati Tombak Kääriku Estonia, August 23. 1993.

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

[4] Jukka Paakki, Tibor Gyimóthy, Tamás Horváth: An Integrated Method for Algorithmic Debugging of Logic Programs. in manuscript

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

Postal Addresses

Zoltán Alexin
Department of Applied Informatics
József Attila University
H-6720 Szged Árpád tér 2

Tibor Gyimóthy
Research Group on the Theory of
Automata at the Hungarian Academy of Sciences
H-6720 Szeged, Aradi vétanu tere 1.

Gabriella Kókai
Department of Informatics
József Attila University Hungary
H-6720 Szged Árpád tér 2