# Debugging and Testing of Prolog Programs

Gabriella Kókai[1], László Harmath[1] and Tibor Gyimóthy[2]

[1] Institute of Informatics József Attila University
Árpád tér 2, H-6720 Szeged Hungary
e-mail: kokai,harmat@inf.u-szeged.hu

[2] Research Group on Artificial Intelligence
Hungarian Academy of Sciences,
Aradi vértanuk tere 1, H-6720 Szeged Hungary
e-mail: gyimi@inf.u-szeged.hu.

**Abstract.** In this paper the IDTS (Integrated Debugging, Testing and Slicing), method is presented for the algorithmic debugging and functional testing of Prolog programs. This method integrates Shapiro's Interactive Diagnosis Algorithm with the Category-Partition Testing Method and a slicing technique. Shapiro's original method demands a lot of user interactions during the debugging process. The IDTS method can avoid irrelevant questions to the user by categorizing input parameters, and match them against test cases and test database. In addition, a slicing method is used to compute which parts of the program are relevant for the search. The IDTS method has since been used in a large ECG classifier program and in an interactive learner. [1]

## 1 Introduction

In this paper we present a method, called IDTS (Integrated Debugging, Testing and Slicing) for the algorithmic debugging and functional testing of Prolog programs. This method not only integrates Shapiro's Interactive Diagnosis Algorithm [12] with the Category Partition Testing Method (CPM) [10] but a common slicing technique [11],[5]. IDTS can improve the efficiency of the algorithmic debugging method for bug localization by using given test specifications and test results. The method can avoid irrelevant questions to the user by first categorizing input parameters, and then matching these against test cases in the test database. In addition, a slicing method is employed that is based on a program dependence graph to compute which parts of the program are relevant in the search, thus further improving bug localization. In the method presented the test database is modified during the debugging of a program and the test specification is improved. In this way our technique can reasonably be considered as an *integrated method for debugging and functional testing of logic programs.*

The basic concept behind IDTS is similar to the GADT (Generalized Algorithmic Debugging and Testing) system given in [4] [7]. The main difference

[1] This work was supported by the grant AMFK 193/96

between the GADT and the IDTS is that the GADT has been applied to imperative languages and the IDTS to logic programs.

As a large scale application the IDTS method has been used in an ECG waveform classifier method called PECG [8].

The method has been also integrated in an interactive Inductive Logic Programming [9] learner called IMPUT [1].

In the remainder of this paper we furnish a brief overview of the IDTS method. Later short example is presented in Section 3. Then finally in Section 4 a summary and comments on future work are given.

### 1.1 The IDTS System for Prolog programs

IDTS combines Shapiro's algorithmic debugging technique [12] with the CPM functional testing method [6] and also a program slicing technique [11] to make it an even more advanced debugging system. The overall structure of the system more or less the following:

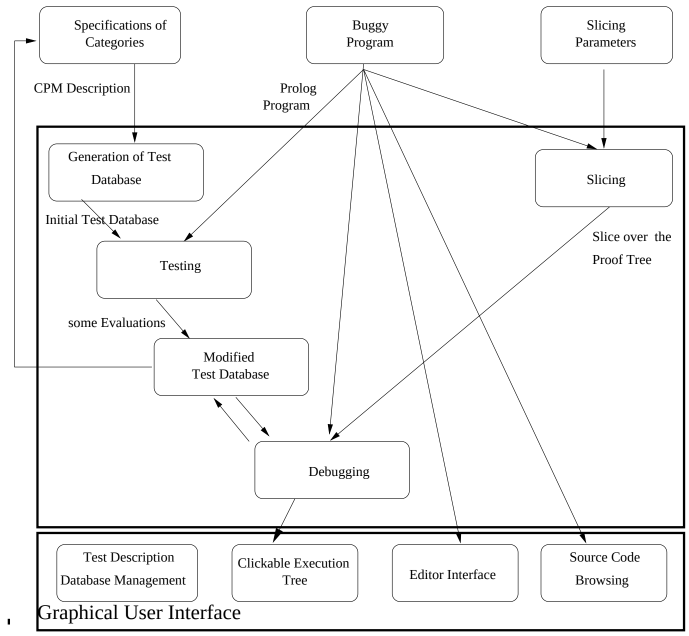


**Fig. 1.** The structure of the IDTS system

The algorithmic program debugging method introduced by Shapiro can isolate an erroneous procedure if a program and an input on which it behaves incorrectly is given. Shapiro's model was originally 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 approach is the great number of queries put to the user about the correctness of intermediate results of clause calls.

The main idea behind IDTS is the following. During the debugging of a program the user has to answer many difficult questions. If the program has already been tested, the test results of the procedures of the program can be directly applied to the debugging process without consulting the user. In addition the slicing method computes the relevant part of the proof tree.

### 1.2 Shapiro's Method for Algorithmic Debugging

Shapiro's algorithms can isolate any erroneous clause, given a program and an input on which it behaving incorrectly. The algorithm traverses the proof tree of a program in different ways and ask the user about the expected behaviour of each resolved goal.

The *bottom-up* method traverses the proof tree in postorder manner and asks the *oracle* about the correctness of the computed values of the nodes. If the result at a node is incorrect and all sons of this node are evaluated correctly results the algorithm identifies the clause applied to this node as a *buggy* one. The query complexity of this method is linear in the size of the tree.

The second method investigates the nodes in a *top-down* manner. If the result computed at a node is evaluated correctly by the *oracle* then the algorithm does not visit the nodes inside the sub-tree. Using this approach the query complexity can be reduced to a linear dependence in the depth of the proof tree.

The most efficient technique is the *divide-and-query* strategy which requires a number of queries logarithmic in the size of the proof tree. The divide-and-query algorithm splits the proof tree into two approximately equal parts, and makes a query for the node at the splitting point. If this node gives an incorrect evaluation the algorithm goes on recursively to the sub-tree associated with this node. If the node's answer is correct its sub tree is removed from the tree and a new mid-point is computed.

### 1.3 The CPM Testing Method

The CPM method for imperative programming languages has been defined in [10], with an extended version presented in [4]. A formal description of the CPM for logic programs can also be found in [6]. Informally this method can be outlined as follows: During the functional testing the programs (procedures) cannot be tested for all possible properties of their parameters. Hence the tester's first task is to define the critical properties (*categories*) of the input parameters which will be investigated in the testing process.

The categories are divided into disjoint classes called *choices* which presume that all the elements within a choice have an identical observable behaviour with respect to correctness. Once the categories and choices for a program have been defined all the possible *test frames* can be generated, each test frame covering exactly one choice from each category. In general there may be many superfluous test frames among the generated ones, namely property combinations that have no real relevance. These frames can be eliminated by associating *selector expressions* with the choices.

### 1.4 The Slicing Method

The program slicing part of IDTS is based on the *annotation* inference technique of [2]. Using this technique an annotating specification of directionality (*input, output*) can be automatically generated for the functional part of a logic program. The user may define the slicing point like the suspected *buggy* position in the program to start the slicing process.

From an annotation a *dependence graph* is constructed for the logic program [3]. A proof (refutation) tree is produced for a buggy program and using the dependence graph the tree is sliced, the segments having no obvious influence on the visible symptom of a bug being removed. The algorithmic debugger traverses the sliced proof tree only, thus concentrating on the suspect part of the program. The annotation of the program is helpful for preparing the test database as well, where the user may provide test cases on input arguments of the annotated program.

During the debugging the IDTS system does not investigate any node of a proof tree which is not in the sub tree determined by the slice of the *buggy* position.

### 1.5 The graphical user interface

The GUI supports the following activities

- *clickable execution trees:* In each step of the debugging process the GUI labels one node of the proof tree, the intended behaviour of which the debugger needs to know to carry on with the debugging process. Nodes can be hidden and extended. The user can answer by pressing one of the shown options *correct*, *incorrect* or *cancel.*
- *source code browsing*: The *static-call graph* shows which predicates could be invoked by a given procedure. With the help of this window a static program structure is displayable to a complement the dynamic proof tree window. *The information-retrieval display* shows the line number and the clauses matching the pattern selected by the user.
- *editor interface:* By pressing a mouse button over a node of the proof tree an editor window opens and displays the part of the Prolog program which corresponds to the node, the source code beeing editable also.
- *test description and database management:* All data is utilised by IDTS can be stored and loaded with a few mouse clicks.

## 2 Small Example in IDTS

In this section we demonstrate the operation of IDTS in finding a false clause in a *buggy program.*

### 2.1 The program compute

To illustrate how our integrated debugger works, a program *compute* (borrowed from [11] and listed in Appendix) was chosen which computes the difference between the lists in the predicate *complement* and the sum of their elements by the predicate *sigma*. Let us, say, introduce the following buggy version of clause *delete*:

```
(C8) delete(X,[Y|Z],[X|U]) :- delete(X,Z,U).
```

instead of the original clause

```
(C8) delete(X,[Y|Z],[Y|U]) :- delete(X,Z,U).
```

The CPM description for the predicates *member* and *delete* is listed the Appendix 4.2. The description begins with the line *type(predicate_name(...))* which defines whether the parameters of the predicate name are printed or not during the debugging sessions. The category-partition specification for predicate *member* contains two categories *number_of_elements* and *relation*. The first category is divided into three and the second into four disjoint classes. For example, category *relation* expresses the view that the number in the first argument of *member* is greater, smaller or equal to the first element of the second argument. The fourth choice *none* is used if the list in the second argument is empty. In each case searching functions are simple Prolog facts.

### 2.2 The questions asked in the debugging process

The most complex part of the system is the algorithmic debugging phase. Let us suppose that the top-down traversal algorithm is selected to demonstrate the usual questions asked by the system. In our example the system first builds up the proof tree (see in Figure 2) for the goal:

```
compute([1,2,3],[3],C,S).
```

If Shapiro's top-down method is employed (without CPM and slicing) the following questions are invoked to identify the buggy clause *delete*.

```
Is the goal sigma([1,2,3],[3],9) (y/n)? y
Is the goal complement([1,2,3],[3],[3,3]) (y/n)? n
Is the goal member(3,[1,2,3]) (y/n)? y
Is the goal delete(3,[1,2,3],[3,3]) (y/n)? n
Is the goal delete(3,[2,3],[3]) (y/n)? n
Is the goal delete(3,[3],[]) (y/n)? y
The false clause:
delete_ver(3,[2,3],[3]) :- delete_ver(3,[3],[])
```

If Shapiro's method is combined with the CPM technique the following questions will be asked:

```
Is the goal sigma([1,2,3],[3],9) (y/n)? y
Is the goal complement([1,2,3],[3],[3,3]) (y/n)? n
Is the goal member(3,[1,2,3]) (y/n)? y
Is the goal delete(3,[1,2,3],[3,3]) (y/n)? n
Is the goal delete(3,[3],[]) (y/n)? y
```

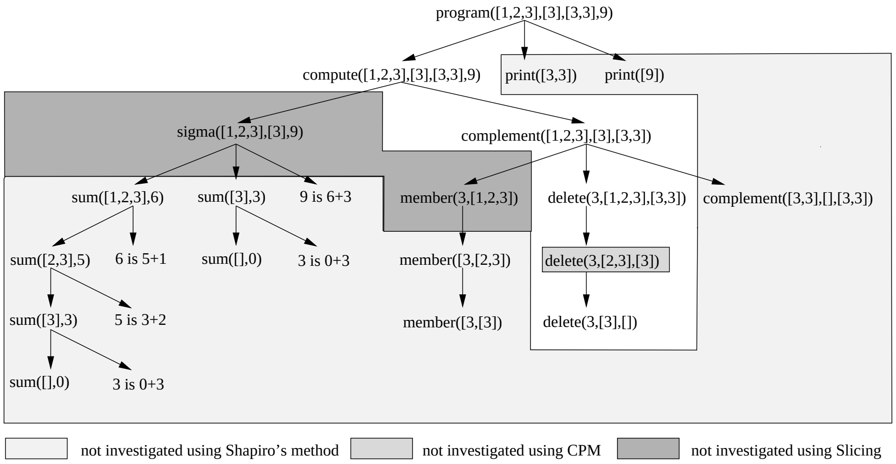


**Fig. 2.** The proof tree of the example

But, if the slicing method is used the following three questions are enough to identify the bug:

```
Is the goal complement([1,2,3],[3],[3,3]) (y/n)? n
Is the goal delete(3,[1,2,3],[3,3]) (y/n)? n
Is the goal delete(3,[3],[]) (y/n)? y
```

## 3 Conclusion, Further Work

In this paper a method has been presented which combines Category Partition Testing, the program slicing method with the algorithmic debugging techniques introduced in [12]. A similar method was presented in [4] to diagnose imperative programs but as far as is known the IDTS method presented in this paper is unique in the context of logic programming. This integrated system can be used in the testing and debugging of Prolog programs.

The first version of the IDTS method has been fully implemented. However, due to the poor implementation technique the computation of dependence graph required for slicing is very slow, hence we have embarked on a new implementation of this part. [2]

## References

1. Alexin, Z., Gyimóthy, T., Boström, H.: Integrating *Algorithmic Debugging and Unfolding Transformation an Interactive Learner* In: Proceedings of the 12th

[2] The implementation language currently is SICStus Prolog Version 3.0 #3 or Quintus Prolog Version 3.2 running on a SUN SPARCStation.

European Conference on Artificial Intelligence ECAI-96 ed. Wolfgang Wahlster, Budapest, Hungary (1996) 403–407 John Wiley & Son's Ltd. 1996.
2. Boye J., Paakki J., Maluszyński J.:*Synthesis of Directionality Information for Functional Logic Programs.* In: Proc. 3rd Int. Workshop on Static Analysis, Padova, 1993. LNCS 724, Springer-Verlag, 1993, 165-177
3. Deransart P., Maluszyński J.: *Relating Logic Programs and Attribute Grammars* In: Journal of Logic Programming 2, 1985, 119-156.
4. Fritzson, P., Gyimóthy, T., Kamkar, M., Shahmeri, N.: *Generalized Algorithmic Debugging and Testing* In: Proceedings of ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario 1991. ACM SIGPLAN Notices 26, 6, (1991) 317–326
5. Gyimóthy T., Paakki J.:*Static Slicing of Logic Programs.* In: Proceedings of Automated and Algorithmic Debugging ed. Ducassé, M. Sain-Malo, France. (1995) 65-77
6. Horváth, T., Gyimóthy, T., Alexin, T., Kocsis, F.: *Interactive Diagnosis and Testing Logic Programs* In: Proceedings of the Third Symposium on Programming Languages and Software Tools, ed. Mati Tombak, Kääriku Estonia August 23-24. (1993) 34–46
7. Kamkar M.: *Interprocedural Dynamic Slicing with Applications to Debugging and Testing* Linköping Studies in Science and Technology - Dissertations No. 297, Department of Computer and Information Science, Linköping University, 1993.
8. Kókai, G., Alexin, Z., Gyimóthy, T.: *Analyzing and Learning ECG Waveforms* In: Proc. of The Sixth International Workshop on Inductive Logic Programming (ILP'96)Stockholm, Sweden 28-30 August, 1996 152-171
9. Muggleton S. (ed.): *Inductive Logic Programming* Academic Press, 1992.
10. Ostrand, T. J., Balker, M. J.: *The Category-Partition Method for Specifying and Generating Functional Tests* In: CACM 31:6 June (1988) 676–686
11. Paakki, J., Gyimóthy, T., Horváth T.: *Effective Algorithmic Debugging for Inductive Logic Programming* In: Proc of the Fourth International Workshop on Inductive Logic Programming (ILP-94) Bad Honnef/Bonn Germany September 12-14 (1994) 175-194
12. Shapiro, E. Y.: *Algorithmic Program Debugging* MIT Press (1983)

# 4 Appendix

## 4.1 The compute program

```
program(L1,L2,C,S)     :- compute(L1,L2,C,S), print(C), print(S).
compute(L1,L2,C,S)     :-  sigma(L1,L2,S), complement(L1,L2,C).
complement(L,[],L).
complement(L,[H|T],U) :- member(H,L), delete(H,L,L1), complement(L1,T,U).
member(X,[X|T]).
member(X,[Y|T])        :- member(X,T).
delete(X,[X|T],T).
delete(X,[Y|Z],[X|U]) :- delete(X,Z,U).
sigma(L1,L2,S)         :- sum(L1,S1), sum(L2,S2), S is S1+S2.
sum([],0).
sum([X|T],S)           :- sum(T,S1), S is S1+X.
```

### 4.2 Test description for the compute program

```
type(member(+,+)).
type(delete(+,+,+)).
%       first category for member/2 : number_of_elements
 choice_of(member,2, number_of_elements, empty).
 choice_of(member,2, number_of_elements, one).
 choice_of(member,2, number_of_elements, more).
%       second category for member/2 : relation
 choice_of(member,2, relation, none).
 choice_of(member,2, relation, greater).
 choice_of(member,2, relation, less).
 choice_of(member,2, relation, equal).
%       first category for delete/3 : number_of_elements
 choice_of(delete,3, number_of_elements, empty).
 choice_of(delete,3, number_of_elements, one).
 choice_of(delete,3, number_of_elements, more).
%       second category for member/3 : membership
 choice_of(delete,3, relation, none).
 choice_of(delete,3, relation, greater).
 choice_of(delete,3, relation, less).
 choice_of(delete,3, relation, equal).
%       searching functions:
 empty(member,2,_,[]).
 one(member,2,_,X)                :- length(X,L), L ==1.
 more(member,2,_,Y)               :- length(Y,L), L >1.
 none(member, 2, _, []).
 greater(member,2,X,[Y|_])        :- X > Y.
 less(member,2,X,[Y|_])           :- X < Y.
 equal(member,2,X,[Y|_])          :- X = Y.
 empty(delete,3,_,[],_).
 one(delete,3,_,X,_)              :- length(X,L), L ==1.
 more(delete,3,_,X,_)             :- length(X,L), L >1.
 none(delete, 3, _,[],_).
 greater(delete, 3, X,[Y|_],_)    :- X > Y.
 less(delete, 3, X,[Y|_],_)       :- X < Y.
 equal(delete,3, X,[Y|_],_)       :- X = Y.
%       properties:
 p_empty(member, 2, empty, X)     :- member(empty, X).
 p_empty_del(delete, 3, empty, X) :- member(empty, X).
%       selectors expressions:
 cond(member, 2, none, X)     :- !, p_empty(member, 2, _, X).
 cond(member, 2, greater, X)  :- !, \+ p_empty(member, 2, _, X).
 cond(member, 2, less,    X)  :- !, \+ p_empty(member, 2, _, X).
 cond(member, 2, equal,   X)  :- !, \+ p_empty(member, 2, _, X).
 cond( delete, 3, none, X)    :- !, p_empty_del(delete, 3, _, X).
 cond( delete, 3, greater, X) :- !, \+ p_empty_del(delete, 3,_, X).
 cond( delete, 3, less,    X) :- !, \+ p_empty_del(delete, 3, _, X).
 cond( delete, 3, equal,   X) :- !, \+ p_empty_del(delete, 3,_, X).
```