GIDTS - A Graphical Programming Environment for Prolog

Gabriella Kókai, Jörg Nilson, Christian Niss
kokai@informatik.uni-erlangen.de
nilson@dreisoft.de
cnniss@cip.informatik.uni-erlangen.de
Department of Computer Science, Programming Languages
Friedrich-Alexander-University of Erlangen-Nürnberg
Martensstr. 3, D-91058 Erlangen, Germany

Abstract

This paper puts forward the Graphical Interactive Diagnosing, Testing and Slicing System (*GIDTS*) which is a graphical programming environment for PROLOG programs. The *IDTS*part of the system integrates Shapiro's Interactive Diagnosis Algorithm with the Category Partition Testing Method (*CPM*) and a slicing technique performing the *algorithmic debugging* and *functional testing* of PROLOG programs. The integration of *IDTS* with a graphical user interface (*GUI*) supports the whole functionality of *IDTS* and provides a user-friendly environment giving the user more information on the state of the debugging process. *GIDTS* extends *IDTS* to a complete programming environment. It allows one to handle the debugging of complex programs using the *extended* syntax and semantics of PROLOG in a very flexible way. A static code diagnosis has also been implemented. In addition *GIDTS* supports debugging-directed editing of the source program, and a quick source code navigation via any of the tools (for example: the debugger, the static call graph and the information retriever). All these features are supported by the graphical user interface.

1 Introduction

This paper presents a Graphical Interactive Diagnosing, Testing and Slicing System (*GIDTS*) which integrates an Interactive Diagnosing, Testing and Slicing System (*IDTS*) algorithmic debugger module with a modular graphical user interface. Furthermore *GIDTS* extends the functionality of *IDTS* in several ways. Together they form a graphical programming environment for PROLOG. The system combines an Interactive Diagnosing, Testing and Slicing System (*IDTS*) algorithmic debugger module with a modular graphical user interface. The *IDTS* ([KHG97]) system was developed for algorithmic debugging and functional testing of PROLOG programs. This system integrates Shapiro's Interactive Diagnosis Algorithm [shapiro] with the Category Partition Testing Method (*CPM* [cpm]) and a slicing technique [slice]. *IDTS* can improve the efficiency of the algorithmic debugging method for bug localization by using test specifications and test results. The method helps avoid irrelevant questions to the user by categorizing input parameters, then matches these against test cases in the test database. In addition a slicing method is used, based on program dependency graphs, to compute those parts of the program relevant to the search, thus further improving bug localization.

Initially *IDTS* hat a simple textual user interface. In this paper we present a graphical programming environment for PROLOG named *GIDTS*. The *GUI* part of *GIDTS* assists the user with the identification of buggy clauses in a PROLOG program. By using this interface the functions of the *IDTS* can easily be invoked. Furthermore the user can directly follow the debugging process using the graphical representation of the *extended proof tree* (proof tree which contains the *redo* branches of the execution of a given program). Similar user interfaces for procedural languages have been proposed and (at least partially) implemented many times before (for example [WF93]) but as far as we know this is the first full featured programming environment for Prolog.

GIDTS allows one to handle the developing of complex programs using the *full* syntax and semantics properties of PROLOG in a very flexible way. A static code analyser has also been implemented. In addition *GIDTS* supports debugging directed editing of the source program, and a quick source code navigation via any of the tools (for example: the debugger, the static call graph and the information retriever).

Other debugging techniques and tools of PROLOG programs were developed ([Lo88], [DNM88], [TakShi], [Per86] and [NDZ89]). The main disadvantage of these systems is that they cannot handle the *full* PROLOG syntax and semantics (using for example `repeat`, `if` and definite clause grammar) and they support the debugging task only with a textual user interface.

In the remainder of this paper a brief overview from other systems developed for PROLOG programs is given in Section 2. It follows introduction to *IDTS* in Section 3. Section 4 then contains a description of the implemented *GUI*. The extended features of *GIDTS* are discussed in Section 5. Section 6 contains same detail about the current implementation. Finally a summary concludes the paper.

2 Related Work

As we mentioned research on this topic started with the development of Shapiro's APD-System ([shapiro]). Many other debugging techniques and tools of PROLOG programs were developed. Some of them are based on Shapiro's algorithm (APROPOS [Lo88], and ADAss [DNM88]) or used other methods (PRESET [TakShi], RD [Per86], NU-PROLOG debugging environment [NDZ89]). The big deficiencies of the existing debugging tools are:

- They use only the proof tree. It means that they can not find the error if it is situated in the *redo* branch of the derivation of the given goal.
- They do not have any graphical user interface.
- They cannot handle the full PROLOG and the extra-logical features.

It means that these systems cannot be applied for *realistic* programs. A critical overview about the systems developed for testing and debugging of PROLOG programs can be found in [Kok98].

3 The *IDTS* System for PROLOG programs

In this Section we give a short overview of the *IDTS* system. The *IDTS* system ([KHG97]) combines Shapiro's algorithmic debugging technique [shapiro], the *CPM* functional testing method [cpm] and program slicing techniques [slice] to facilitate advanced debugging.

3.1 Shapiro's Method for Algorithmic Debugging

The algorithmic program debugging method introduced by Shapiro can isolate an erroneous procedure if a program together with an input leading to an incorrect result is given. Shapiro's model was originally applied to PROLOG programs to diagnose the three types of errors: termination with incorrect output, termination with missing output and non-termination. A major drawback of this debugging method is the great number of queries to be answered by the user on the correctness of intermediate results of clause calls.

Shapiro's algorithms traverse the proof tree of a program in various ways (*bottom-up*, *top-down* and *divide and query*) asking the user about the expected behaviour of each resolved goal.

3.2 The *CPM* Testing Method

The *CPM* method for imperative programming languages has been defined in [cpm]. A formal description of the *CPM* for logic programs can be found in [idt:1]. Informally the method can be described as follows. During functional testing, the programs (procedures) cannot be tested for all possible properties of their parameters because there are too many. Hence the tester's first task is to define those critical properties (*categories*) of the input parameters which are to be investigated during the testing process.

The categories are divided into disjoint classes — called *choices* — in which we presume that all the elements within a choice have 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 covers exactly one choice from each category. Intuitively, a test frame expresses a certain syntactic property of input elements with the same semantic behavior, expressed as a combination of input properties. Hence, a test frame covers exactly one choice from each category.

In general, there may be many superfluous test frames among the generated ones, that is, property combinations that are not relevant. These frames can be eliminated by associating *selector expressions* with the choices.

3.3 The Slicing Method

The program slicing part of *IDTS* is based on the *annotation* inference technique [BPM93b]. 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, the *incorrect* position in the program to start the slicing process.

From an annotation of the program a *dependency graph* is constructed for the logic program [DM85]. A proof (refutation) tree is produced for a buggy program and using the dependency graph the tree is sliced. The parts having no 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 used also for preparing the test database; the user may provide test cases on input arguments of the annotated program.

This slicing method combines static and dynamic slicing features [Kol90]. This means, that a program dependency graph is *statically* constructed for a given logic program. This graph is then used to build up dependency graphs for the proof trees. The nodes of the dependency graph of a given proof tree contain the argument positions in the tree [slice] and the dependencies among these positions come from the *statically* computed program dependency graph. The computation of a slice starts from an argument position of the proof tree and this slice determines the argument positions which may effect the values of the starting point (backward slice). In our method we used a modified version of the [slice] method. If the annotation is incorrect the program stops, displays an error message and does not generate the dependency graph.

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

3.4 *IDTS* — Integration of Shapiro's Method with *CPM* and Slicing

The idea behind the integrated method is that the algorithm does not ask the user about the correctness of a resolved goal if it finds the evaluation of this goal in the test database or if the sliced extended proof tree does not contain this clause. If the system does not find the error then the exhaustive single-stepping algorithm is invoked. More about the *IDTS* system can be read in [KHG97].

4 The Graphical User Interface of *GIDTS*

In this Section the main features of the *GUI* developed will be explained. The *New Graphical User Interface Developer's Environment* (*NGUIDE*, [NGUIDE]) was used for creating the user interface.

4.1 Clickable Proof Trees

The derivation of goals leads to the construction of extended proof trees which are displayed on the screen. This tree also contains the *redo* branches of the derivation of the given goal. The extended proof trees are the most advanced graphical parts of the *GUI*. At the beginning of each debugging process the proof tree must be constructed before debugging can start. The time and memory usage of realistic computations is shown in Table 1. The *GUI* builds up the graphical representation of the tree from a structure generated by *IDTS*. The correct graphical display is achieved by the facilities available to *NGUIDE*.

Example	size of the derivation tree	additional memory usage		time usage	
		Prolog	GUI	Prolog	GUI
isort	170 nodes	448 KB	5 MB	2 sec	16 sec
sieve	610 nodes	960 KB	13 MB	3 sec	62 sec
ECG	747 nodes	1664 KB	48 MB	48 sec	134 sec

Table 1: Time and memory usage of computations in GIDTS (measured on a Sparc Ultra 1 with 128MB RAM); Example description with the used goals:

isort: Simple insert sort – isort ([1, 2, 3, ..., 14, 15, 0], L).

sieve: Eratosthenes' sieve – primes(30,L).

ECG: ECG analysis – main(...).

The *GUI* supports the following activities:

- At each step of the debugging process the *GUI* labels one node of the extended proof tree, the intended behaviour of which the debugger needs to know to proceed with the debugging process.
- The user can focus on the bug location. The user interface supports the selection of the debugging points and the selection of the exact location at which the slicing process is to be started.
- Nodes can be displayed in various formats which differ in the way they consume space on the display. This is a very useful feature of the *GUI* because predicates may use a lot of parameters.
- Any sub-tree can be hidden and hidden sub-trees can be extended again.

In the debugging process the task of the user is to answer straightforward *yes-no* questions. Every question represents a specific node in the extended proof tree of the program. A visualization of the question context is useful for making correct decisions because the user's answer has an effect on the further execution of the debugging process.

The node in question is highlighted and the user can answer by pressing one of the presented buttons *correct*, *incorrect* or *cancel*. To further help the user in the decision-making process *GIDTS* provides a flexible filtering of the questions presented. This could be used for additional graphical displays. For more details see Section 5.3. The debugging process may be continued until the error is located. Additionally the user has always the possibility of exiting from the debugging process altogether.

The following problems were solved by implementing the clickable extended proof trees:

- The original textual user interface of *IDTS* did not identify the nodes of the extended proof tree uniquely. Now the nodes are marked by unique paths through the tree.
- The input of some debugging parameters (debugging and slicing points) in the textual user interface is complicated. The *GUI* supports entering this data by clicking on the appropriate nodes in the extended proof tree.

4.2 Source Code Browsing

The only type of tree discussed so far is the extended proof tree which focuses on the execution of the program. However, some other structures can also be displayed in tree form. For example, *static call graphs* that show which predicates are callable by a given predicate may be displayed in a separate window to complement the dynamic extended proof tree window. More textual displays such as sorted lists can also be useful, especially if information about textual data (e.g. PROLOG predicates) needs to be retrieved. One major aspect of information retrieval is the ability to restrict the quantity of the data displayed. For instance, the display may be restricted to predicates with a given arity, to predicates which match a particular name pattern, or to predicates for which facts are defined.

4.3 Editor Interface

By pressing a mouse button over a node of the extended proof tree a popup menu comes up. The user can then choose between displaying the definition of the current predicate or changing the format of how nodes are displayed after copying the current predicate to the clipboard. If the first item is chosen an editor window opens and displays that part of the PROLOG program which corresponds to the node, after which the source code can be edited. After saving the changes in the program the debugger can be invoked again to find other errors in the source program.

Because *emacs* is a well-known and often used text editor in the Unix environment, an interface for this editor was developed. As an alternative the editor gadget of *NGUIDE* might be used instead.

5 GIDTS extensions to IDTS

GIDTS extends the functionality of *IDTS* in several ways. It introduces an editor interface described in Section 4.3. Five additional extensions are explained in this Section; session management, reuse of test databases, flexible visualization of the debugging process, powerful features for retrieval of information related to the program, and a static code analysis not just for *pure* PROLOG programs.

5.1 Session Management

Since large programs are generally not debugged in one session, it would be useful, to save the current state of the debugging process so that work can be continued at another time with this state restored. Another useful feature is that a reference test database can be assembled.

Sessions are managed in conjunction with the whole programming environment. All data which belongs to a particular session can be stored and loaded with a few mouse clicks. This data includes the name of the current PROLOG program, name of the current *CPM* description file, name of the current test database (automatically derived from the session name), current traversing mode (bottom-up, top-down, divide-and-query), the current debugging point action, and the current debugging method (original Shapiro, *CPM*, slicing or all three together).

5.2 Reuse of Test Databases

During the program debugging sessions the same test frames are performed again and again. To reduce these tests it may be helpful to load formerly applied test databases again. But *IDTS* cannot use such test databases without validation. The current version of the program may differ from the version for which the test database was developed. The test results need to be verified. We have to distinguish several possible situations:

- If a test frame was untested, it is left untested.
- If a test frame had an evaluation, one or more test cases are given. These test cases are automatically tested again and the result of this repeated test is compared:
  - If the result is the same as in the former test, the evaluation of this test frame is left unchanged.
  - If the test case contains no variable and the result is different, the evaluation is negated. If the evaluation is incorrect a warning is displayed.
  - If the test case contains at least one variable and the result is different, the user is asked whether the new result is correct or not. The user's answer then becomes the evaluation of the test frame.

It is quite clear from the above that only matching test databases can be reused. If the verification process leads to different evaluations in the same test frame, the test description is inconsistent and the user has to correct the test description. The user has to initiate the reuse of test databases manually.

5.3 Filtered Visualizing of the Debugging Process

As we remarked in Section 4.1 the extended proof tree is displayed during the whole debugging process. The user has the option of varying the size of the tree nodes in several ways. But sometimes this is insufficient. It could be that the predicates have some simple and short parameters (such as numbers) and some complex and long parameters (for example the input and output list of DCG [DCG] clauses). *IDTS* allows a simple filtering mechanism for parameters. But the user can only really choose between showing a parameter and hiding it completely. In contrast *GIDTS* supports external filters. The filtering program has to be named in the test specification. and is an external program which must be operable in the underlying operating system environment.

GIDTS initiates the filter at the beginning of the debugging process. Then every question asked by the system is sent to this filter. The filter then processes the input and generally generates a different output. Only this output is displayed as a question in the debugging window of *GIDTS*.

The filter can process its given input in almost any way desired. In one it can compress the representation of determined parameters. In another it can use the input for further visualization. For example consider a complex grammar which can recognize heart diseases from a textual representation of an ECG [ecg]. To debug this grammar, you will be confronted with a virtually incomprehensible set of questions. Even professional PROLOG programmers and experienced cardiologists can hardly interpret this textual representation. You need further visualization to assist you in your task. A filter process which draws the current part of the ECG on the screen might be very helpful indeed.

5.4 Information Retrieval

Debugging is an important part of a programming environment. But a real programming environment is much more than this. *GIDTS* also supports several ways of retrieving information from PROLOG programs. It provides two basic tools; a retriever and static call graphs. Both were briefly discussed in Section 4.2. The data needed for the *GUI* of these tools is generated via an integrated PROLOG parser. This parser constructs an internal data structure with all information that is needed for effective and efficient search in PROLOG programs.

5.5 Static Code Analysis of PROLOG Programs

GIDTS is able to find errors in a program containing the following:

- Declarations
- Control facilities: if (−>), or (;), repeat
- Definite Clause Grammars
- Built-In Predicates

The main task of the static code analysis module is to search for errors which can be statically recognized. Before the execution of the program these errors were already detected. They may be one of the following:

- undefined clause
- improper arity
- irreducible goal

GIDTS checks each goal which is not a system predicate in the body of every clause. If a literal on the right side of a clause cannot be unified with a head of a clause in the program the program delivers *fail*. The system shows each recognized error and the user can correct it using the editor interface (see Section 4.3).

6 Implementation issues

The main programming tool (in addition to the parser generator *bison* [bison]) used in the implementation of *GIDTS* was the New Graphical User Interface Developer's Environment (*NGUIDE*), an environment for building graphical user interfaces ([NGUIDE]).

The *GUI* needs to communicate with the PROLOG interpreter and also with *IDTS*. This was solved by using a separate terminal connection line, a *tty* (see [UNIX]). The advantage of this approach is that the textual user interface of *IDTS* can still be used without the *GUI*. With the *tty* the GUI can emulate a human user working with the textual user interface of *IDTS*.

In addition to the above the output and input aspects of *IDTS* are done via the optional *socket* (see [UNIX]) communication line, so that the *GUI* won't be disturbed by output or input actions of the debugged program. Nevertheless this is only an option for *IDTS*, so we could debug and use *IDTS* even without a *GUI*.

The communication part from the *GUI* to *IDTS* is synchronous, but in the opposite direction the communication is asynchronous so that the *GUI* will not be blocked by a possibly slow PROLOG interpreter. This is important because graphical user interfaces should be able to react to user input at any time.

In each case all parts of the *GUI* are integrated into one process system consisting of two operating system processes, the PROLOG interpreter, and the *GUI*. Both parts interact in such a manner that the user will recognize only one process running under a homogeneous graphical user interface.

7 Conclusions and Further Work

The *IDTS* system (implemented in PROLOG)[1] improves Shapiro's original algorithmic debugging method by reducing the number of questions posed to the user. It combines the Category Partition Testing Method and program slicing approach with the algorithmic debugging technique.

In this paper we described *GIDTS* (implemented in *C++*)[2], an extended *IDTS* with a full featured programming environment. We consider a user-friendly graphical interface to be of great importance for (successfully) using the *GIDTS* system. Therefore a graphical user interface provides additional information that makes it easier for the user to answer

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

[2] The currently used compiler is gcc 2.8.0.

the computer's questions and to control the debugging process. *GIDTS* assists in finding errors in PROLOG programs using the full syntax and semantics of PROLOG. A static code analyser was also implemented. In addition to debugging and testing *GIDTS* supports debugging directed editing of the source program, and a quick source code navigation with any of its tools.

At present the user has to define the syntactically and semantically correct *CPM* description. It will be an improvement to develop a tool which is an extension to *GIDTS* which assists the entering of the test description using a syntax directed editor. This tool can also be augmented with a library module to manage typical test cases. Additionally we are planning a special support for test description files (consistency checks, more intuitive way of definition).

Another important topic is to make the whole debugging process more dynamic, i.e. the user need not specify all debugging parameters before debugging is started. The user ought to be be able to change most of the debugging parameters during the debugging process. For example he should be able to set and clear debugging points or change the traversing method.

References

[BPM93b] 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

[DNM88] Drabent, W., Nadjm-Tehrani, S., Małuszyński, J.: *Algorithmic Debugging with Assertions*, In Proc Meta-Programming in Logic Programming, 1988, 375-378,

[DM85] Deransart P., Maluszyński J.: *Relating Logic Programs and Attribute Grammars*. Journal of Logic Programming 2, 1985, 119-156

[bison] Hansen, W. J.: *Andrew Bison Version A. 2.6*. CMU (1995).

[idt:1] Horváth, T., Gyimóthy, T., Alexin, T., Kocsis, F.: *Interactive Diagnosis and Testing Logic Programs*. Proceedings of the Third Symposium on Programming Languages and Software Tools, ed. Mati Tombak, Kääriku Estonia August 23-24. (1993) 34–46

[Kol90] Korel B., Laski J.:*Dynamic Slicing of Computer Programs* In: The Journal of Systems and Software 13, 3, 1990, 187-195.

[ecg] Kókai, G., Alexin, Z., Gyimóthy, T.: *Classifying ECG Waveforms in Prolog.* Proc. of the Fourth International Conference on the Practical Application of Prolog, London, UK (1996), 173-201.

[KHG97] Kókai, G., Harmath, L., Gyimóthy, T.: *Algorithmic Debugging and Testing of Prolog Programs* ICLP '97 The Fourteenth International Conference on Logic Programming, Eighth Workshop on Logic Programming Environments Leuven, Belgium, 8-12 July 1997, 14-21

[Kok98] Kókai, G.:*Error Diagnosis in Prolog Programs, A Critical View*, In Proc Conference of PhD Students on Computer Sciences, July 1998

[Lo88] Chee-Kit Looi: *Analysing Novices' Programs in Prolog Intelligent Teaching System*, In Proc of the European Conference on Artificial Intelligence, Munich 1988, 314-319,

[NDZ89] Naish, L., Dart, P. W., Zobel J.: *The NU-Prolog debugging environment*. In Proc of the Sixth International Conference on Logic Programming, Lisboa, Portugal, June 1989, 521–536

[NGUIDE] Nilson, J.: *NGUIDE — ein System zur Erstellung grafischer Benutzeroberflächen*. Ph.D. thesis, Department of Computer Science – Programming Languages, University Erlangen Nürnberg, 1998.

[cpm] Ostrand, T. J., Balker, M. J.: *The Category-Partition Method for Specifying and Generating*. Functional Tests CACM 31:6 June (1988) 676–686

[slice] Paakki, J., Gyimóthy, T., Horváth T.: *Effective Algorithmic Debugging for Inductive Logic Programming*. Proc of the Fourth International Workshop on Inductive Logic Programming (ILP-94) Bad Honnef/Bonn Germany September 12-14 (1994) 175-194

[DCG] Pereira, F. C. N., Warren, D. H. D.: *Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks.* Artificial Intelligence 13 (1980), 231-278.

[Per86] Pereira, L. M. : *Rational Debugging in Logic Programming* In Proc of the Third Logic Programming Conference, London, England, July 1986, 203-210

[shapiro] Shapiro, E. Y.: *Algorithmic Program Debugging* MIT Press (1983)

[TakShi] Takahashi, H., Shibayama, E.: *PRESET - A Debugging Environment for Prolog* , Proc of the International Conference on Logic Programming, Tokyo, 1985

[UNIX] Stevens, W. R.: *Advanced programming in the UNIX environment*. Addison Wesley (1994)

[WF93] Rickard Westman and Peter Fritzson. Graphical User Interfaces for Algorithmic Debugging". In Proceedings of Automated and Algorithmic Debugging, Linkping, Sweden, May 1993. LNCS 749, Springer-Verlag.