TCPW/CW/K.INT.

BIM-PROLOG

Joint Project between
BIM
and
Department of Computer Science
Katholieke Universiteit LEUVEN

Sponsored by DPWB/SPPS
under grant nr KBAR/SOFT/1

A Debugging System for Prolog

by
Raf VENKEN *

Internal Report
BIM-prolog IR9

November 1984

* BIM
Kwikstraat 4
B-3Ø78 Everberg Belgium
tel. +32 2 759 59 25

** Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 2ØØA
B-3Ø3Ø Heverlee Belgium
tel. +32 16 2Ø Ø6 56

DPWB = Diensten van de eerste minister : Programmatie van het Wetenschapsbeleid.
SPPS = Services du premier ministre : Programmation de la Politique Scientifique.

A Debugging System for Prolog

by

Raf Venken

Belgian Institute of Management
Kwikstraat 4
3078 Everberg
Belgium

1.0 INTRODUCTION

Until recently, logic programming and more specifically Prolog, was mostly used in the context of artificial intelligence research for the development of experimental systems or for basic research. On one hand a lot of research was done on the development of new logic programming languages (e.g. parallellism, interaction with functional languages, control features, relations with database theory) and on the efficient implementation of them, on the other hand these languages were used for building experimental systems in the 'classical' artificial intelligence domains (e.g. vision, natural language understanding, robotics). However, since the Japanese Fifth Generation Computer Project announced Prolog as one of the building blocks of future generation computer systems, the interest in logic programming has exploded.

For the moment a lot of research or industrial institutes are investigating the usability of Prolog in the development of complex systems (complex in structure or complex in the AI-like features they use). Although not being a perfect logic programming language, Prolog has proven its qualities as implementation language for AI-applications such as expert systems or natural language systems. However the main criticisms on account of Prolog, as a language for programming real-life applications remain : it has no (or a very poor) programming environment and there exists no methodology (and associated tools) for the development of complex systems in Prolog.

In this paper we will concentrate on the programming environment of Prolog : more specifically we will tackle the problem of designing a debugging system for Prolog programs. A debugger is a very (if not the most) important module of the programming environment of any high or low level programming language : it is the quality of the debugger which is decisive for the overall appreciation of a language.

In the next chapter we describe the nature and the structure of a possible programming environment for Prolog. Then we shortly survey the literature concerning Prolog debugging systems. Finally we introduce a new approach concerning the debugging of Prolog programs, which tries to unify the existing approaches.

2.0 THE PROLOG PROGRAMMING ENVIRONMENT

Figure 1 depicts a possible scenario for the development of Prolog programs using a programming environment.

```
           Designer
               |
               | program specification
               |
        _______________
       |               |
       |    Program    |
       |  Generators   |
       |_______________|
               |
               | Prolog code
               |
        _______________             _______________
       |               |           |               |
User -->|    Editor     | <---------> |   Debugger    |
       |_______________|           |_______________|
               |                          |
               | Prolog code              |
               |                          |
        _______________                   |
       |               |                  |
       |    Partial    |                  |
       |  Evaluation   |                  |
       |_______________|                  |
               |                          |
               | Prolog code              |
               |                          |
        _______________             _______________
       |               |           |               |
       |  Compilation  |           |               |
       | Query Optimis.| <---------> |               |
       |_______________|           |   Relational  |
               |                   |     Data      |
               | Object Code       |     Base      |
               |                   |    System     |
        _______________            |               |
       |               |           |               |
       | Prolog System | <---------> |               |
       |_______________|           |_______________|
```

Figure 1

The program generator could be the final tool in an integrated software technology laboratory. E.g a lot of research is going on in the definition of conceptual modelling languages which could serve as a very high level description formalism of applications. This description of applications could serve as the basis of a semi-automatic conversion to programs written in high level languages like Prolog. For more details on this matter see [Greenspan84] and references listed there.

A Prolog oriented editor is the first building block of a sophisticated programming environment. It serves as the interface between the programmer and the Prolog system (including the environment). Language dedicated editors can be classified in two categories concerning the mode of operation. One category contains the editors working in a guiding mode : this means a.o that the editor guides (or sometimes imposes) the way of writing a program, this may be useful for novice programmers, but unacceptable for experienced programmers who developed an own personal strategy of writing programs in Prolog. The second category of editors are characterized by a free style of editing : at first sight one can not distinguish them from normal text editors, however they provide some language dependent features, e.g. checking of syntax of entered program (on demand or on exit), semantical checking like dataflow or type checking (see [Bruynooghe82] for more details on this kind of program checking), searching for language specific constructs, generation of intermediary code. One could see the debugging system, the partial evaluation system and the compiler as specialised functions of the editor, although for simplicity we will treat them separately while at the implementation level the different modules should be interconnected closely, admitting especially the flexible interplay between the editor and the debugger.

Since the topic of this paper is the debugging system, we will not go into detail on this module at this stage. Although one can notice we make a clear distinction between the Prolog system and the debugging system. Both systems should have the same external behaviour when executing programs, although the debugger needs to have an extended bookkeeping (as we shall see later) while the Prolog system, for the sake of performance and robustness, tries to optimise, i.e. to keep only that information absolutely necessary for further calculation.

Partial evaluation is a semi-automatic optimisation technique which tries to apply at compile-time the mechanisms a Prolog system applies to a Prolog program at run-time as much as possible. A Prolog program is converted to an equivalent Prolog program, which is partially evaluated and which, ran by the Prolog system has the same external effect, but needs less logical inferences, the apparent performance of the system thus increases. For more details (a.o. the impact of partial evaluation on query optimisation) see [Venken84].

The compiler and the Prolog system have to be considered together : there are two extreme ways, the first being a Prolog system which executes the Prolog source directly, the other being a compiler from Prolog to assembler. Between these two we see the whole range of compilation to intermediary codes and interpreters for those codes. Finaly, for real-life applications it is needed to have a flexible interface to an existing relational database.

3.0 A SURVEY OF DEBUGGING SYSTEMS FOR PROLOG

3.1 The Problems When Debugging Prolog Programs

Prolog is a very high level programming language : its notation is very compact and the semantics are very powerful, there are no declarations, its syntax is very simple and uniform and it is an ideal tool for quick (and dirty) programming and complex systems written in Prolog are extremely hard to debug, when no precautions on style or methodology were taken. In this section we give some features of Prolog which demand for a sophisticated debugging system.

The Prolog system builders sometimes make the life of the Prolog programmer very hard. The distinction between values and variables is rather arbitrary and differs substantially from system to system, in different systems variables start with a higher case letter, an underscore, an asterisk, a lower case letter, ... This not only decreases the portability of a Prolog program, but confuses the Prolog programmer. A standardisation effort would be desirable. Any way, since there are no variable declarations and a variable is only a typing error away from a value, one can easily produce very invisible bugs.

The basic concepts of logic programming are very clean and straightforward, Prolog however has been 'enhanced' for efficiency and functionality reasons with some extra-logical features (cut, assert, retract, metacall, ...) which mutilate the declarative reading of Prolog programs, make the language more difficult to master and the debugging of Prolog programs a hard job.

The main reason however that makes Prolog hard to debug is its complete lack of redundancy concerning the notation. A small typographical error does often not result in compile-time or run-time diagnostics, but simply gives another, unintended, meaning to the program. Where the semantics of Prolog are called very high level, the syntax in comparison is very poor and below the current standard of software engineering (the only advantage is the decrease of clerical work in writing programs, but at what price ...). There are no declarations for variables, nor for procedures, the number of arguments, the expected types and the parameter passing mechanism.

3.2 The Execution Trace

To monitor the execution of a Prolog program the first Prolog systems provided the programmer with a tracing facility which gave information about the procedures which were being called during execution. This trace contains all necessary information to be able to follow the execution of a program, but the amount of displayed information rapidly grows with the size of the program and since, it was not at all structured, impossible to master.

3.3 The Procedure Box

Most popular Prolog dialects use as debugging tool a trace package based on a boxes model for the execution of Prolog (see [Byrd 80]). A Procedure is represented as a box with 2 entrances and 2 exits, an entrance for a 'call' to the procedure and one for a 'redo' on backtracking, two exits one for success and one for failure. The programmer can now place spy-points on the four gates of the different boxes which will trigger the trace package, interactively he can skip spy-points or cause particular calls to fail or be retried.

In essence this approach remained an execution trace, where the user could monitor the amount of information to be displayed on the screen. The other main inconvenients however remained : the tracing is closely bound to the execution, you cannot go back in your trace without restarting the complete trace again (very inconvenient, especially for bugs at the 'end' of a huge program). The failure of a call to a procedure can have a multitude of reasons, these are all covered by the unique 'fail'-comment, which is not very helpfull for fixing a possible bug.

3.4 Algorithmic Program Debugging

The main inconvenience of the tracing way of debugging lies in the fact that it compares with a linear search for the bug. It is in [Shapiro 82] that a method was given to speed up this search : the divide-and-query algorithm. If a bug has been encountered the programmer can ask the debugging-algorithm to split the prooftree in two equal parts ; the user can then check (by looking at the output of the first part) whether the bug is in the first or the second part. The prooftree of the buggy part is then split up recursively and so forth. The search time for a bug is reduced to log(n) like a binary search. An algorithm has been presented for finite failure, incomplete solution and infinite failure.

This method relies on the single assignment structure of Prolog and the lack of global variables : each part of the prooftree is independent of the other. However real-life Prolog supports also global variables (assert-retract) which are incompatible with the method, as are all other kinds of side-effects.

3.5 Rational Program Debugging

A similar method is under investigation in [Pereira 84]. This method uses information about Prolog data term dependency on derivation goals (which is also used in selective backtracking Prolog) for a better debugging. It automates the reasoning required to pinpoint errors, where the programmer only has to answer questions about the intended program semantics. The method exploits both the declarative and operational semantics of Prolog. The main advantage to the previous method is that it can jump over subderivations irrelevant to the investigated bug and so speed up the search process and reduce the number of questions to be answered by the programmer.

3.6 Ameliorations Of The Previous Methods

In the literature some ameliorations have been presented, mainly on the boxes-model (see the programmer's manuals of the respective Prolog systems and [Eisenstadt 84]) or on Shapiro's work ([Plaisted 84]). They enhance the methods mainly concerning the user interface and optimise the space and time requirements for execution of the algorithms. But the principles remain the same.

4.0 A UNIFIED APPROACH

All the methods as summarized in the previous section don't differ that much, since they all try to represent the same information to the user only in a different way. The first two methods are execution bound : the information is displayed while the program is executed, and impose a linear search upon the user. Shapiro's algorithmic debugging imposes a binary search strategy through the prooftree, while the relational debugging optimises the search by skipping over parts of the prooftree irrelevant to the bug. The information used is merely syntactic and the methods apply only to 'pure' Prolog, i.e. if one does not use global variables (assert, retract) or other side-effects.

To our view it is possible to gather the different approaches in one debugging system and doing so combine their advantages, moreover we will develop strategies to locate bugs involving non-logic Prolog features like 'cut', 'retract', 'assert', etc.

4.1 The Basis Of The Approach

The basis of an implementation gathering all methods consists of a Prolog interpreter or meta-interpreter with an extended bookkeeping facility which keeps track in all details of the invocation of the different clauses and of the binding of the variables. Part of this information is in the run-time structures of any Prolog system, but where a run-time Prolog tries to minimise the amount of information to be kept in core, a debugging system must keep all information possibly relevant to a debugging strategy. Therefore we propose to keep the complete profftree in the datastructures of the debugging system. Only this garatuees us to be able to track down any possible bug in a Prolog program using any possible strategy.

It is clear that the information to be kept in core will increase significantly with the size of the program, but the complexity of debugging does so too. In any case is amodular approach to writing programs and debugging them advisable. The inefficiency, eventually brought in through modularity can be eliminated after the debugging phase by partial evaluation as demonstrated in [Venken 84].

4.2 An Example Debugging Strategy

There are different kinds of bugs possible in a Prolog program which cause different kinds of unexpected program behaviour. This unexpected program behaviour are the symptons of the bugs and each sympton asks for a different strategy to track down the bug. Some symptons are : finite failure (the program fails to find enough solutions), infinite failure (the program goes in an endless loop), wrong result (exact number of solutions, but wrong values), too many solutions (duplicate or wrong results), unexpected side-effects and any possible combination of the previous.

We now give a possible strategy for tracking down the bug in the case of a wrong solution, which can be implemented using the complete prooftree which can be obtained by executing the buggy program only once.

A.

The Prolog program is executed by the debugging system, this permits the construction of the internal data structures containing the information concerning the prooftree of the executed program.

B.

The user identifies a wrong solution to the program he executed, with a pointing device he points to the term containing the wrong value.

C.

This pointing to a wrong term permits the system to analyse which part of the prooftree is relevant to the obtained wrong value, using the dependencies between the variables.

D.

The 'previous' call relevant to the obtaining of the wrong result is considered next. If this call is not admissible (it doesn't have the correct type of arguments), the user points to a wrong term in it and the strategy continues with phase C.

E.

The call under inspection, as it would look after unification with the head of the clause used to generate the wrong solution, is analysed. If the result is not acceptable then the clause head is wrong since it produced a wrong binding or matched a call it should not. In the case of a unit clause, a wrong unit clause is detected.

F.

If the call is not solvable (according to the expected behaviour of the program) then the body of the clause is wrong. The user points to the wrong term, now it can be analysed whether this term was passed to the clause via the call (the clause is then wrong since it accepted a wrong input) or actually produced by the clause by executing the body.

G.

The body is analysed next, if one of the calls should have failed (which it didn't) then this call is taken as focus for step A. If all should succeed (as they did) then the clause is in contradiction with the semantics you wanted to attribute with it.

This is only an example strategy a user could follow to track down a possible bug in a program. To our opinion, this strategy or similar ones for different symptons, should not be automized in a debugging system, but facilities should be implemented that would enable the user to analyse the behaviour of his program according to his own strategy. The implementation of a debugging system then consists of providing the user with a flexible and intelligent interface to the information contained in the datastructures of the debugging system and facilities to move the focus through the prooftree or to mark subtrees which are dependent of a term.

One can note that in the example debugging strategy the focus of attention traverses the prooftree in 'reverse' order, the variables in a call can get a value through unification with the heading of a clause or through execution of the body, the different stages through which a call goes should be separated one from the other. This must enable the user to decide for what reason a call fails (no unification, failing call in the body) and when it succeeds where the values for the variables were produced.

4.3 The Interface

The most important part of the debugging system is to our view the interface between the internal datastructures en the user of the debugging system. The most appropriate way to start with a debugging session is to have the information (the prooftree of the executed program) represented in a graphical way. The user should be provided with easy means to traverse this prooftree using a pointing device or simple control keys. Zooming on particular points of interest should be made possible in order for the user to distinguish between input or output values in a call, to see the subsequent results after unification and after execution of each subgoal. Reference must be made to the initial source code of the program under investigation to enable the user to see which particular clause was selected during execution of a certain call, also the same variable names should be used as in the intial source code since they normally have a mnemotechnical meaning.

The complexity of the interface suggests to use state of the art display devices like bitmap screens with multiple window facilities and pointing devices. Multiple windows are needed to represent at the same time the graphical representation of the prooftree, the source code under investigation and eventually a menu with the different manipulation or investigation possibilities.

4.4 Side-effects

One thing which is very difficult to debug using the 'classical' strategies as represented in section 2 is any program using side-effects (being it input-output on screens or files, assert or retract in internal or external memory and other non-logic Prolog features). On the other hand, side-effects are the most important aspects of a real application : the aim of any program is to have some side-effect (on database or the user's terminal). Moreover, it is nearly always through a side-effect that a user detects the existence of a bug (wrong value output on the screen, inconsistent database, etc).

In order to treat side-effects in a debugging system one is obliged to keep the complete prooftree in the internal datastructures (this means the or- and the and-trees). For pure Prolog programs one could contend to only keep the branch from the root to one of the (eventually wrong) solutions to be able to implement a debugging strategy, whenever one needs something to know about branch which failed (or a previous solution) this can savely be reexecuted to show the user the eventual reasons for failure or success. Apart from the fact that reexecution may involve considerable time-overheads, this cannot work for programs involving side-effects : an assert would be executed twice, ... By keeping the entire prooftree (eventually abreviated for already debugged parts) one can circumvent this problem, all information concerning failed branches remains available for investigation and no reexecution of side-effects is to feared.

To be able now to debug the impact of the program on the environment of the Prolog system, i.e. the produced side-effects, we propose to keep snapshots of the environment on secondary storage, taken at regular intervals. The snapshots refer to particular nodes in the prooftree, and can be used to show the user the status of the environment (e.g. the screen as it was during execution) at any point of the execution using the prooftree (which serves as a logging file) to construct any intermediate situation. The environment of a real application program usually involves the terminal screen (eventually graphics), the internal database (global variables), the external database. This snapshot technique combined with the logging as provided by the complete prooftree permits a flexibility of movement through the prooftree which is not possible in any other debugging system proposed or implemented yet.

5.0 DEVELOPMENTS

We are currently working on the implementation of a so-called Professional Prolog system including a state-of-the-art compiler and interpreter (initially for UNIX systems) and a Prolog programming environment.

The environment includes a Prolog oriented editor with incremental data- and typechecking, a partial evaluation system ([Venken 84]), an integration with an existing commercial database system and the here introduced debugging system. The debugging system will be designed to work with advanced screens as available on the SUN workstations which run UNIX 4.2. A first version of the system should be available summer '85.

6.0 CONCLUSION

We proposed a new approach to the debugging of Prolog programs which permits the application of all strategies and techniques as introduced in the Prolog literature. This new approach permits not only more flexibility for the user in designing more appropriate debugging techniques for pure Prolog programs, but offers facilities which permit the debugging of non-logic features of Prolog. Advantage is taken of state-of-the-art technology concerning display screens which permit multi-windowing and pointing.

7.0 ACKNOWLEDGEMENTS

This work has been done under the research contract KBAR/SOFT/1 sponsored by the DPWB (Diensten voor de Programmatie van het Wetemschapsbeleid).

8.0 REFERENCES

[Bruynooghe 82]

Bruynooghe M., Adding Redundancy to obtain more reliable and readable Prolog Programs, Proceeding of the First Int. Logic Programming Conference, Marseille, 1982.

[Byrd 80]

Byrd L., Understanding the Control Flow of Prolog Programs, Proceedings of the Logic Programming Workshop, Debrecen, Hungary, 1980.

[Eisenstadt 84]

Eisenstadt E., A Powerful Prolog Trace Package, Proceedings of ECAI '84, Pisa, 1984

[Greenspan 84]

Greenspan, S.JJ., Requirements Modeling : A Knowledge Representation Approach to Software Requirements Definition, University of Toronto, Technical Report CSRG-155, 1984.

[Pereira 84]

Pereira L.M., Rational Debugging of Logic Programs, Universidade Nova de Lisboa, 1984.

[Plaisted 84]

Plaisted D.A., An Efficient Bug Location Algorithm, Proceedings of the Second Int. Logic Programming Conference, Uppsala, Sweden, 1984.

[Shapiro 82]

Shapiro E., Algorithic Program Debugging, Cambridge, MA, MIT Press, 1982.

[Venken 84]

Venken R., A Prolog Metainterpreter for Partial Evaluation and its Application to Source-to-source Transformation and Query Optimisation, Proceedings of ECAI '84, Pisa, 1984.