View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Edinburgh Research Explorer

THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Evaluating Prolog environments

Citation for published version:
Bundy, A, Brna, P & Pain, H 1988, 'Evaluating Prolog environments' Proceedings of the 1988 Alvey Technical Conference.

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Author final version (often known as postprint)

Published In:
Proceedings of the 1988 Alvey Technical Conference

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 20. Feb. 2015

Evaluating Prolog Environments
A Progress Report

Alan Bundy, Helen Pain and Paul Brna

DAI Research Paper No. 351

December, 1987

A short report for the Alvey KBS Club

Department of Artificial Intelligence
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
United Kingdom

© Alan Bundy, Helen Pain and Paul Brna

Evaluating Prolog Environments
A Progress Report

Alan Bundy, Helen Pain and Paul Brna
Department of Artificial Intelligence
University of Edinburgh

December 23, 1987

Abstract

We outline the progress we have made in connection with the Alvey Grant "Evaluating Prolog Environments" (SERC GR/D/44287 and Alvey IKBS 136). This grant runs for three years from 1st November 1985. The grant holders are Professor Alan Bundy and Dr Helen Pain and it employs Dr Paul Brna as a Research Fellow.

1 Background

At the start of the grant period, the Alvey Logic Programming Initiative chose to promote expertise in Prolog. It was felt that Prolog, being a relatively young computing language, had not accumulated the kind of interactive development environment that was commonly found, for instance, in Lisp systems. The provision of such environments was widely seen as essential if the full potential of Prolog was to be realised.

As part of the development of new, sophisticated Prolog environments it is vital to evaluate tools, to identify problems with them, and to draw attention to omissions and needs. In particular, it is known that many tools do not meet user's needs for debugging large programs. Empirical study is seen as vital to discover what kind of bugs occur, how they are currently fixed, and what tools could improve user productivity.

2 Goals

Our main goal is to inform and influence long term activities in the UK Prolog community by the evaluation of extant tools and the identification of problems with them. We also set out to draw attention to omissions and needs.

In particular, our concerns include:

1. Do the user interfaces present consistent stories about how Prolog programs are run and are these stories consistent with available teaching materials?

2. Are these stories consistent with the Logic Programming philosophy of Prolog? Is it possible to maintain an entirely declarative story for the programmer?
3. To what extent do the debugging tools, error messages, etc., help users avoid or cope with the kind of bugs which arise in practice? Do there remain commonly occurring bugs for which existing tools are inadequate?
4. Do the tools help eliminate misconceptions about the Prolog virtual machine?

We had paid attention to the problem of presenting consistent stories (concerning items 1 and 2 above) in previous papers [Bundy 84, Pain & Bundy 87, Bundy et al 85]. This includes an examination of the early Prolog teaching materials to extract the Prolog stories they used and the development of a complete story that covers all the aspects of Prolog in a uniform and coherent manner [Bundy et al 85]. This has had some influence on teaching materials (e.g. 2nd edition of [Clocksin & Mellish 84] and [Eisenstadt & Brayshaw 87]) and tracers (e.g.[Eisenstadt & Brayshaw 86]). Our recent work has mainly focussed on the issues of bug-avoidance and bug-elimination (item 3 above).

In order to confront these issues we set out to evaluate a number of Prolog environments with a view to producing the following:

- Some classifications of Prolog programming errors (bugs) and lists of commonly occurring Prolog bugs.
- An account of bug-avoiding and debugging strategies used on these bugs.
- An account of how various environmental tools are used in these strategies.
- The identification of gaps, i.e. bugs for which no efficient strategies exist and no tools are useful.
- Suggestions about new environmental tools to plug these gaps.
- Suggestions about language improvements/extensions that would avoid certain bugs.
- Suggestions about teaching methods that would enable students to avoid certain bugs.
- An evaluation of how available environments score on the above, e.g. identification of those offering novel, useful tools.

3 Achievements

At this point we have made some progress on all these goals —although much still needs to be done. We have set up a framework for the description of Prolog bugs. This is fundamental as we want both to motivate the construction of new and improved tools based on an analysis of bugs (and program debugging) and to evaluate Prolog environments with respect to the practical requirements of programmers.

This framework —see figure 1— can be viewed in two distinct ways. The first way (program generation), as a means of describing stages through which a programmer goes from the start of coding to the moment that the programmer realises that the program is 'buggy'. The other way (bug detection), as the path by which the programmer might track down the essential cause of the problem.

We now briefly summarise the different layers of description. More detail can be found in [Brna et al 87].

Symptom Description If a programmer believes that something has gone wrong during the execution of some **Prolog** program then there are a limited number of ways of directly describing such evidence —e.g.

- (apparent) non-termination
- exit with operating system error
- generation of Prolog error message
- unexpected generation or failure to generate a side-effect
- unexpected "no" or "yes"
- wrong binding of answer variable

Program Misbehaviour Description The explanation offered for a symptom. The language used is in terms of the flow of control and relates, therefore, to run-time behaviour —e.g.

- the unexpected failure of some clause
- the unwanted success of some clause —e.g. with a cut then causing a subsequent clause not to be called
- a missing clause
- etc.

Program Code Error Description The explanation offered in terms of the code itself. Such a description may suggest what fix might cure the program misbehaviour —e.g.

- there is a missing base case
- there is a clause that should have been deleted
- a test is needed to detect an unusual case

Underlying Misconception Description The fundamental misunderstandings and false beliefs that the programmer may have to overcome in order to come to terms with specific features of the language —e.g.

- recursion is really iteration
- if a subgoal fails then the goal fails

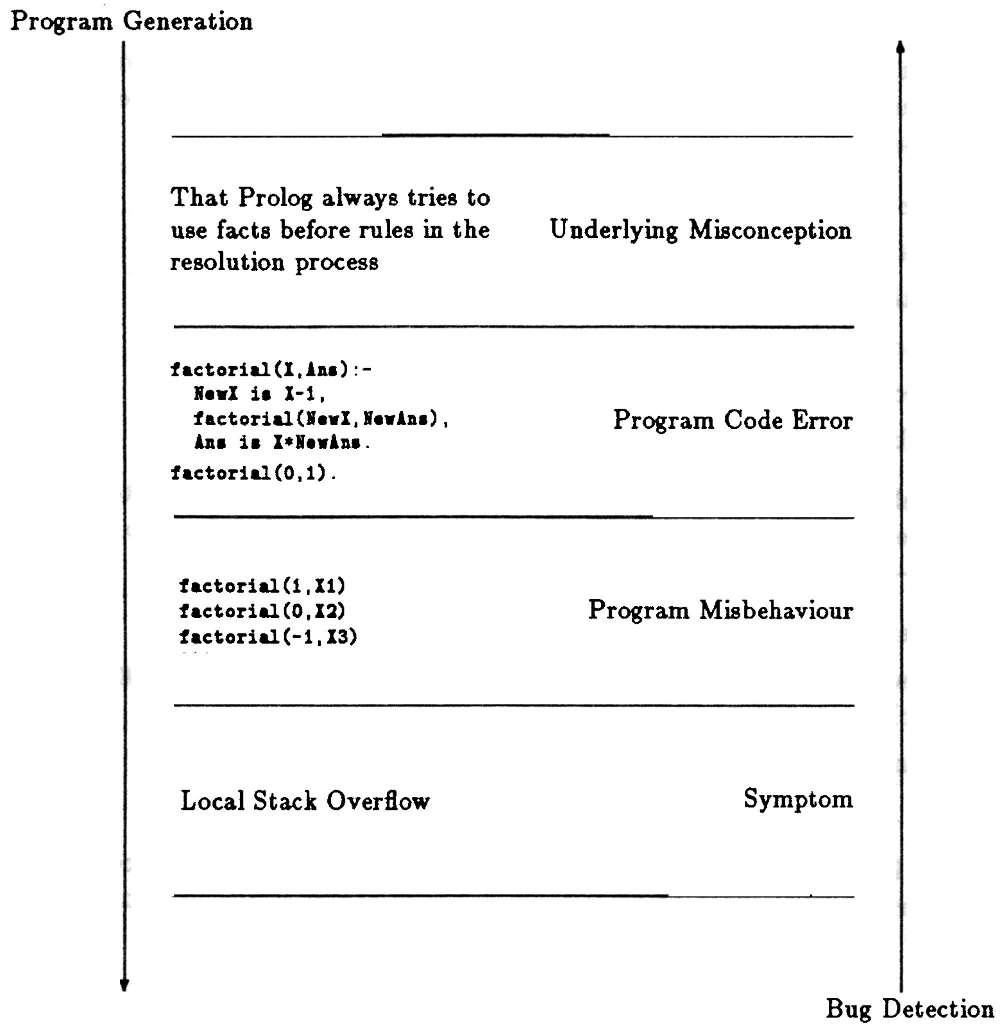

Figure 1: **Four Levels of Description of Prolog Bugs**

- *is/2* is really assignment

To illustrate that Prolog bugs can be classified at a number of levels consider the symptom that a query seems to be taking a long time. The immediate cause of this might be that the program may be in an infinite loop —described at the level of program misbehaviour. This looping might be because of a programming error —for example, that the body of a clause contains a literal identical to the head. And this might be because the programmer has an underlying misconception about recursion. This example illustrates the four levels: symptom, program misbehaviour, program code error and underlying misconception.

We have examined the *Symptom Description* and *Program Misbehaviour Description* levels in some detail. The *Program Code Error Description* level is interesting because it admits of at least two quite different approaches: a *syntactic* classification and a *technique-based* one.

An example of such a programming technique is a *failure driven loop.*

```
PREDICATE(...X1,...X2,...):-
        NON_DETERMINISTIC_PREDICATE(...X1,...X2,...),

        SIDE_EFFECT_PREDICATE(...X1,...X2,...),
        fail.

PREDICATE(...X1,...X2,...).
```

In the above schema, ellipsis is used to indicate missing sub-goals, arguments. Observe that the schema is not sufficient to capture the full range of clauses that exhibit the properties of a failure driven loop. For example, the side-effecting predicate may not require the same list of arguments as the non-deterministic one.

A necessary requirement for a failure driven loop is that it has a clause that always fails. It also should contain at least one non-deterministic literal and one that side-effects. Note that this definition includes some procedural semantics: fail, side-effect, etc. There could be other clauses and literals. The non-deterministic and side-effecting one could be the same or different.

The various ways in which a program can violate the definition of a failure driven loop constitute a technique-oriented bug classification, include:

- No failing clause
- No side-effecting literal
- No non-deterministic literal

A catalogue of techniques are being considered in order to build up other technique-oriented bug classifications [Bundy et al 87]. In particular, the techniques of building up

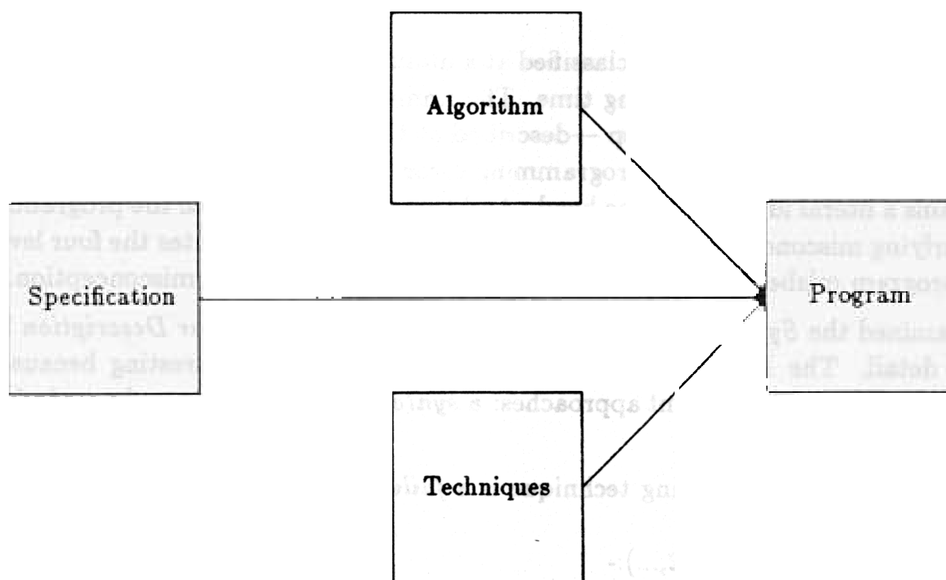

Figure 2: Techniques and Software Production

recursive data structures by pattern matching in the clause head, and building them up in the clause body with an accumulator.

We contrast techniques with both the notion of algorithm and that of programming clichés (see the work on the Programmers' Apprentice [Waters 85]). An algorithm is a language independent procedure which meets some specification, e.g. sorts the element of a list. In contrast, our techniques are language dependent (i.e. Prolog), but specification independent —e.g. the same technique might be used in sorting a list or in finding the maximum of two numbers. Furthermore, a technique might apply to only part of a complete procedure, and many techniques may be combined together in a procedure.

Figure 2 illustrates a formalisation of some aspects of the software production process. Algorithms are selected according to the nature of the specifications. Techniques are not directly related to the specification but they are related to the algorithms selected.

The *Underlying Misconception Level* description has not been developed to any significant extent.

4 Current Status

The next stage in our work requires the evaluation of current Prolog systems and tools. Systems have been selected for a variety of reasons: because they are influential (e.g. DEC-10 Prolog, MicroProlog), because they offer novel tools relating to bug avoidance or debugging (e.g. BIM, ESI Prolog II, NU-Prolog, LPA Sigma Prolog), or because they are

readily widely used (e.g. LPA MacProlog, C-Prolog). Not every important Prolog system is necessarily included. SICSTUS Prolog, for example, is a significant fast, new Prolog system but appears to have few, if any, new tools. We hope to provide an evaluation based on about twenty systems.

So far, we have concentrated on producing a systems/tools survey. Following on from this, our bug description framework will be used as part of the basis of the evaluation of current Prolog systems that we are undertaking. Various identified debugging strategies will also be part of our evaluation framework.

We expect that the bug description system, the list of bug-avoidance strategies and the known debugging strategies will be subject to further revision and development. This necessarily requires that the evaluation evolve. The preliminary set of results, however, are expected by the middle of 1988.

References

[Brna et al 87] P. Brna, A. Bundy, H. Pain, and L. Lynch. Programming tools for Prolog environments. In J. Hallam and C. Mellish, editors, *Advances in Artificial Intelligence*, pages 251–264, Society for the Study of Artificial Intelligence and Simulation of Behaviour, John Wiley and Sons, 1987. Previously, DAI Research Paper No 302.

[Bundy 84] A. Bundy. *What stories should we tell Prolog Students?* Working Paper 156, Dept. of Artificial Intelligence, Edinburgh, 1984.

[Bundy et al 85] A. Bundy, H. Pain, P. Brna, and L. Lynch. *A Proposed Prolog Story.* Research Paper 283, Dept. of Artificial Intelligence, Edinburgh, 1985.

[Bundy et al 87] A. Bundy, P. Brna, B. Smith, T. Dodd, M. Eisenstadt, M. van Someren, and C.K. Looi. Prolog programming techniques. 1987. Paper produced as part of a workshop on Prolog at Abingdon.

[Clocksin & Mellish 84] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, 1984.

[Eisenstadt & Brayshaw 86] M. Eisenstadt and M. Brayshaw. *The Transparent Prolog Machine TPM.* Technical Report 21, Human Cognition Research Laboratory, The Open University, 1986.

[Eisenstadt & Brayshaw 87] M. Eisenstadt and M. Brayshaw. *An Integrated Textbook, Video, and Software Environment for Novice and Expert Prolog Programmers.* Technical Report 23, Human Cognition Research Laboratory, The Open University, 1987.

[Pain & Bundy 87] H. Pain and A. Bundy. What stories should we tell novice PROLOG programmers. In Hawley R., editor, *Artificial Intelligence Programming Environments*, Ellis Horwood, 1987. Also available as DAI research paper 269.

[Waters 85] R.C. Waters. *KBEmacs: A Step Toward the Programmer's Apprentice*. Technical Report 753, MIT Artificial Inteligence Laboratory, 1985.