Programming Tools for Prolog Environments

PROGRAMMING TOOLS FOR PROLOG ENVIRONMENTS

P Brna
A Bundy
H Pain
L Lynch

D.A.I. RESEARCH PAPER NO. 302

Submitted to *AISB-87*.

Programming Tools for Prolog Environments

by
Paul Brna, Alan Bundy, Helen Pain and Liam Lynch

Abstract

The Prolog programmer programs in an environment which provides a number of debugging tools. There is often a mismatch between the way a programmer describes some perceived error and the way in which a debugging tool needs to be used. Worse, there are some problems which existing tools cannot tackle easily —if at all.

The main aim of the work described is to construct a coherent framework on which to base the design of programming tools. This paper describes a particular classification of programming errors. Error classification is then used to provide a natural description of the tools that can, or could, assist the programmer.

Examples are given of useful tools which are not part of well known current Prolog implementations and suggestions are made as to how current tools can be improved to increase their utility.

Keywords

Prolog, Logic Programming, Program Debugging, Teaching Programming, Programming Environments.

1 Bugs and Tools

The sources of Prolog programming errors are numerous. Taylor has six levels at which errors may arise [Taylor & duBoulay 86]. Her analysis is principally concerned with the processes entailed in starting with some problem statement, formalising this problem and eventually producing a Prolog program. Coombs and Stell have investigated the possibility of helping novice programmers detect misconceptions in their understanding of backtracking in Prolog [Coombs & Stell 85]. Other work, by van Someren, has indicated that several programming errors result in programmers trying to write programs by using familiar concepts from some language other than Prolog [vanSomeren 85].

Our current interest, however, lies in the problems that flow from incorrect Prolog programs and the match between the debugging tools provided and the nature of the programming problem.

Our aim is to motivate the construction of new and improved tools based on an analysis of bugs and program debugging. For the moment, the term 'Prolog bug' may be taken to refer to some error that is responsible for the creation of an incorrect Prolog program. This necessarily weak definition is clarified and expanded in what follows.

Current Prolog Tools

The available tools are divided into *dynamic* and *static* ones. Dynamic tools are applied at run time and are usually bundled together inside a trace package. Static ones are applied at consult time[1].

Recent work at the Open University by Eisenstadt and his workers has focussed on the provision of more useful tools [Eisenstadt et al 84,Eisenstadt 84]. In particular, recent work by Eisenstadt has attempted to incorporate both methods of controlling the amount of information revealed by a dynamic tracer and some knowledge about the kinds of error made by programmers [Eisenstadt 85]. Rajan has produced animated tracing tools for novices along with a general set of design principles [Rajan 85].

Other workers have explored Prolog debugging techniques. Shapiro's work is well known [Shapiro 82]. Lloyd is developing an approach based on extending the Prolog syntax [Lloyd 86] while Pereira has extended Shapiro's ideas about *Algorithmic Debugging* [Pereira 86]. Their work is important but it is based on a simple classification of errors which needs further development.

1.2 The Classification of Bugs

The classification and listing of bugs and debugging strategies is of fundamental importance to give a foundation for motivating new, or improved, tools. In this section, the nature of bug classification is discussed together with the methodology for obtaining it.

Different Levels of Classification

Classification can be at a number of levels. For example, the symptoms which are presented to the user, or the underlying causes for these bugs. Note that a particular symptom may be caused by a chain of causes, so that there is a range of causal explanations from shallow to deep. For instance, the symptom might be that a procedure call seems to be taking a long time. The immediate cause of this might be that the program may be in an infinite loop. This 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. These four examples are generalised to the four levels: symptom, program misbehaviour, program code error and underlying misconception. These ideas are illustrated below and defined more carefully in section 2.

A classification at the program code error level might include: missing procedure or clause, multiple copies of procedure or clause etc. At the symptom level there is, for example, non termination, error message, etc.

Note also that the same symptom can arise from different underlying causes, or that the same cause can give rise to different symptoms under different circumstances, so that

[1] Although some tracers provide a post mortem analysis which allows the possibility of easily combining so-called dynamic and static tools

each level will give rise to a different classification. For instance, a procedure call may take a long time because the program is complicated, or looping or inefficient, or because the computer is slow, or has crashed. Misconceptions about recursion can also cause the construction of programs that fail when they they were intended to succeed, or return with the wrong answer, etc.

1.2.2 All Levels Are Potentially Interesting

All these levels are of interest:

- To evaluate dynamic debugging tools it is the symptom and program misbehaviour levels that are of interest. The student who is wondering why his/her procedure call is taking such a long time must chose a tool mainly on the basis of that symptom, although they might also entertain some hypotheses at the program misbehaviour level.
- To evaluate static debugging tools it is the program code error level that is of interest. For instance, it would be reasonable to expect a tool which diagnosed potential infinite loops —for example, by looking for clauses with heads identical to a body literal. It would be nice if such tools were also capable of giving higher level causes and remedial help to remove misconceptions, but this kind of tool requires fundamental research.
- To advise on teaching methods it is necessary to be interested in the program code error and underlying misconception level. For instance, to find more successful ways to teach recursion so that students did not suffer so readily from misconceptions about it.
- To advise on language improvements/extensions it is necessary to be interested in the program code error and the underlying misconception level. For instance, a language might be suggested in which recursive definitions were only accepted if they were shown to be well-founded (i.e. non-looping).

1.2.3 The Consequences

For any given bug it is likely that there is a hierarchy of strategies corresponding to the different causal levels. This might entail a method of teaching that avoids the misconception, a language improvement that makes the bug impossible, a static tool that detects it at edit or compile time, and a dynamic tool that tracks it down at run time. All these are valuable and should be passed on to teachers, language designers, and tool builders. In the short term the low level, symptom oriented solutions will be most valuable, since they will lead to tools that can patch the current badly engineered language and ill-educated programmers. In the longer term the higher level solutions will be more valuable, since they will lead to languages and teaching methods which avoid problems. However, the world being the imperfect place it is, programmers will always make mistakes and there will always be a role for the debugging tools.

2 Classifications of Prolog Bugs

A multi-layered classification of Prolog bugs is described. This involves four layers. These are: symptom, program misbehaviour, program code error and misconception. The latter level is not considered in any great detail but analyses are given for the other three. The program code error level is further expanded by means of two different classification schemes.

A four-level description is provided. Later, each of the four levels are elaborated separately:

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. For example:

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

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. For example, the hypotheses that might be entertained concerning (apparent) non-termination:

- there is a loop
- the computer system is very heavily loaded
- the system has crashed
- the computation takes a very long time

Other potential hypotheses may involve descriptions closely related to the symptom level but at a different level of detail —such as the unwanted success of some subgoal followed by a cut which then causes a subsequent clause not to be used.

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. For example:

- 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. For example:

- recursion is really iteration
- if a subgoal fails then the goal fails
- *is/2* is really assignment
- All arithmetic expressions are automatically evaluated

Each of these four levels of error description is now analysed further.

2.1 Symptom Description

It is supposed that the program under investigation is treated as a black box by the programmer for the purposes of describing symptoms. Now 'the program' may well be a subset of a much larger program but that does not matter[2].

In the broader context of programming environments, the use of various tools will be taken into account —such as trace packages and the cross referencer— but it is argued that they cannot make any contribution to the program's *symptom* description as their role is to open up the 'black box' to the programmer's inspection.

At the symptom description level, the classes of event that can be described permit references to:

- Error messages
  - from Prolog
  - from the operating system
  - from an editor which is running Prolog etc.
- Termination issues —this includes:
  - unexpected 'apparent' failure to terminate
  - unexpected termination —which includes the possibility of terminating the Prolog session
  - termination with an unexpected 'value' —for example, a goal succeeds when it should fail
- Instantiation of Variables which includes:
  - unexpected failure to instantiate a variable
  - unexpected instantiation of a variable
  - a variable is instantiated to an unexpected value

[2]This implies that no reference can be made to the program's internal structure.

- 'visible' side effects
  - unexpected failure to produce a specific effect
  - unexpected production of a side effect
  - production of a side effect with an unexpected value

This analysis has imposed an obvious pattern on the possible outcomes with the exception of the entry for error messages. There is no reason, however, why error messages cannot also be included in the same way although it has been pointed out that an 'unexpected failure to produce an error message' may be a rather unusual event. Nevertheless, in the course of determining that some program functions correctly it might be expected that Prolog ought to produce some error message.

Perhaps it is worth pointing out that this analysis does not provide for completely determined error descriptions. In real life, it is expected that there will be a wide range of symptom descriptions[3]. In particular, symptom descriptions may well be clothed in the terminology imported by the programmer from the nature of the programming task being undertaken. This might include terminology from the domain in which the programming task is located, the data structures and algorithms that the programmer has in mind. It is assumed that it is possible to translate such domain specific descriptions into the appropriate Prolog-specific ones. As far as describing the programmer's expectations, a totally separate effort is required to provide a suitable description language.

Program Misbehaviour

The next stage is to look into the 'black box'.

The Ways in Which a Program May Misbehave

The level of symptom description treats the program as a 'black box' whereas the level of program misbehaviour requires that the black box be opened up.

It is assumed that the programmer knows the names and arities of the various predicates defined. This is consistent with the requirement that each 'black box' has some means of identification or 'handle'.

Whatever is said here must be seen as relative to some Prolog story. A *Prolog Story* is an explanation of the workings of the Prolog interpreter or compiler which a programmer can use to understand and predict the execution of a Prolog program. The basic outline of the 'Proposed Prolog Story' is adopted here [Bundy et al 85].

The program can be 'opened up' to examine the program behaviour as it is executed. The program can be seen as a set of connected 'black' boxes. Examining the program

[3] When programmers comment on their code they often mix symptom description with other types of description. This is not the point at issue.

behaviour incorporates the symptom description behaviour in section 2.1 but includes an extra aspect which can be termed 'flow of control'.

What is needed is some way of capturing *flow* which suggests that *sequences* are of interest. It is suggested that the principle is adopted that sequences are the transitions from one box to the next. This is the principle that only *local* flow should be considered.

The program can be considered as a set of black boxes:

- An expected transition fails to occur
- An unexpected transition occurs
- The expected transition occurs but the instantiation pattern is unexpected

The types of black box are:

- The module —the box consists of an unordered set of predicate definitions. The module can be thought of as a complete program with no undefined predicates. A 'handle' on the box is the principle functor and arity of any procedure visible from 'outside' the module.
- The predicate —the box consists of the (ordered) set of clauses that form the (incomplete) definition of a given predicate. The definition is incomplete if any subgoal depends on any predicate that is not built-in. The handle on the box is the principle functor and its arity.

  A clause —the box is the head and body of a single clause. The handle is the head of the clause.
- An argument of a predicate —the box is the (single) argument while the handle is the name and arity of the argument[4].

The simplest assumption is of the program as a set of clauses. The Byrd box model allows for a program to be a set of predicate definitions but this model has not always been strictly followed by implementors.

2.3 Program Code Error Descriptions

Programming errors can be described at the code level in terms of the syntactic structure of the code itself. For this approach, the principle is adopted that no description must refer to any particular programming techniques being used. For example, consider a faulty version of the standard program for **member/2**:

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

[4]Provided it is not a variable or a constant in which case there are no further boxes to be opened

This could be described by saying that the program code error description is:

missing base case

but this invokes the means for describing some *programming technique* for a simple recursive procedure.

2.3.1 The First Code Level Classification

The program can be divided up in terms of the kinds of 'black box' discussed at the program misbehaviour level. For division by module, there are three basic cases:

- Missing module: a module that was expected to be present was missing. This could be the consequence of failing to load a module or consult a necessary file.
- Extra module: a module is found which was not expected to be present. This can result from failing to edit out a redundant module.
- Wrong module: some expected module has an error description. This means that the program code error description has to be applied recursively to the module.

A corresponding set of possibilities exists for division by predicate.

At the clause level, the program is considered as a set of clauses. The possibilities are:

- Missing clause
- Extra clause
- Wrong clause

It is also necessary to capture the sense of order normally required by the standard Prolog search strategy. This means that there is the extra possibility of

- Wrong clause order

At the clause level, the description of a clause can be taken to involve the subcomponents of the head of the clause and the body of the clause. The classes of error connected with the head are:

- Missing head
- Extra 'head' (this one is effectively a syntax error)
- Wrong head

The *body* of a clause can be handled in a similar manner but with wrong subgoal order as an additional possibility.

It is now necessary to account for errors at the level of a single term or single subgoal. Given some term, the only way there can be a problem is that the term is the wrong term. The categorisation can be further expanded by sub-dividing terms into atoms, variables and compound terms. This, in turn, has to take into account such errors as: missing predicate name, extra predicate name and wrong name as well as errors associated with the argument list.

The classification of the code level described above can be described as 'syntactic'. That is, it did not reflect any declarative or procedural semantics on the part of the programmer, but merely lists the ways in which an actual Prolog program might differ syntactically from some correct program.

The advantages of this classification scheme are:

- it includes all possible bugs
- it is simple and regular
- there are only a finite number of bug types

The disadvantages are:

- it demands a template giving a detailed and inflexible description of the correct program
- in general, there is no way to know what this template is

2.3.2 A Technique Oriented Classification

Other classifications are both possible and desirable. Another basis for classification is now suggested. It is based on the notion of programming techniques.

An example of a programming technique is a *failure driven loop*. Such an analyser which takes a program and tests it to see whether there is a well formed failure driven loop [Lynch 86]. The code contains a definition of what one is and looks for violations of this definition.

For instance, a failure driven loop must have a clause that always fails and that contains at least one non-deterministic literal and one that side-effects. Note that this definition includes some procedural semantics: fail, side-effect, etc. Note that it is not as detailed and inflexible as the template of the syntactic classification. 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, namely:

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

The differences between this technique-oriented classification and the syntactic one are:

- this one does not cover all possible bugs
- there are an indefinite number of techniques, and hence an indefinite number of bug types
- this classification might get arbitrarily complex
- this one requires a flexible definition of a technique rather than a detailed and inflexible template
- one might more readily know what definition was intended than what template was intended.

Various program analysers are being built by Liam Lynch based on both the above classifications [Lynch 86]. His 'tail recursion' analyser uses a fixed template and produces a bug analysis drawn from our syntactic classification. He is now extending this to a syntactic analyser which can take any template and produce an analysis based on the full syntactic classification.

His 'failure driven loop' analyser uses a definition of this technique and produces a technique-oriented bug analysis. He is building similar analysers for other techniques. For instance, a technique-oriented tail recursion analyser, which would contrast with his existing syntactic one and illustrate the difference between the two classification schemata.

The syntactic analyser will have the virtues of generality but the limitations of inflexibility. In the context of an automated programming tutor, such an analyser will be able to criticise any student program, provided the teacher has provided a template but will be liable to reject perfectly good answers just because they use a solution method slightly different from that intended by the teacher.

The technique analysers will have the opposite properties. They will only be able to criticise programs that are intended to be examples of some technique, but will be able to accept a wide range of correct solutions —for example, any correct failure driven loop. However, they will only be able to criticise the program on the basis of its fit to the technique, not on other grounds —such as producing the wrong answer!

Such technique analysers are potentially useful debugging aids in a Prolog environment. The programmer could ask that a program be tested for 'tail-recursiveness' etc. Such analysers fit into the range of other static tools like mode-finders, type-checkers, etc. Note that the syntactic analyser could not be used in this way because there is no

way that users could want their programs checked if they already knew the template they were to fit.

Other techniques are being considered in order to build up other technique-oriented bug classifications. In particular, the techniques of building up recursive data structures by pattern matching in the clause head, and building them up in the clause body with an accumulator.

Other related work at Edinburgh takes a slightly different line. Chee-Kit Looi is seeking to harness a number of methods — such as symbolic evaluation, mode checking, type checking etc. — to build a Prolog Intelligent Tutoring System [Looi 86,Looi & Ross 86].

2.4 Misconception Description

As yet, no attempt has been made to produce some higher level bug classifications in terms of misconceptions. To uncover the deeper bugs we will need to take into account on-going research into the misconceptions of novice Prolog programmers such as that by Taylor and van Someren [Taylor & duBoulay 86,vanSomeren 85]. It is expected that some empirical studies will be necessary to uncover misconceptions and link them with the lower level bugs already identified.

3 Debugging Strategies in Prolog

In previous sections the classification of Prolog bugs was considered at each of four levels: symptom, program misbehaviour, program code error and misconception: Here the beginnings of an investigation into debugging strategies are described in terms of the first two levels of description.

A Prolog programmer's first indication of a run time bug is one of the types of symptom described in section 2. Typically, s/he will then try to identify a bug at the program misbehaviour level to account for this symptom, and hence to a code error which can be corrected[5].

The focus is now the step from a symptom level description to a program misbehaviour level description. For each error type, the question is asked: "Using existing environmental tools, do efficient bug avoiding or debugging strategies exist?" If the answer is no, the next question is "can better dynamic or static tools be suggested?"

Since the immediate symptoms of run time bugs are under investigation then most debugging strategies will involve the tracer and the discussion will be almost exclusively procedural. The role of declarative solutions is in static bug avoiding tools or programming and teaching strategies.

The bugs associated with 'exit with Prolog/operating system error' will not be discussed here.

[5]However, at AAAI-86, an example was given of going straight from symptom to code error. Given an unexpected "no" the programmer would immediately turn on an 'undefined procedure' mode and look for one.

Current Tools for Inspecting Program Behaviour

In some sense there is only one possible tool available for looking at run-time program behaviour —the trace package. Therefore the trace package can be switched on, the execution of some n port model[6] followed or a 'spy' point set etc. A number of other tools such as XREF (cross referencer) and MODGEN (mode generator) are potentially useful —and there has been some consideration as to whether they can be tied into the trace package— but they generally apply to the (static) code rather than the (dynamic) run-time behaviour.

The 'standard' trace package is not a uniform entity. A 'trace package' is a collection of facilities built to form a coherent whole. The tracer provides facilities to enable the programmer to 'walk over' some abstraction of the program's behaviour[7] —often a tree-like representation— and provide various possibilities at each node in the representation. Through an analysis of the ways in which a program can misbehave it is possible to identify (at least some of) the gaps in the provisions for each environment.

(Apparent) Non-Termination

Suppose the program seems to be taking a long time to return. The problem might be:

1. a loop
2. the computer is slow today
3. the computer has died
4. the program just takes a long time

Items two and three can probably be tested by some operating system command. Consideration is now given to testing item one.

In most Prologs the only way to test for non-termination is to interrupt the program run and then use the tracer to look for repeated goals, etc. This is most inefficient. The combination of inefficiency and uncertainty makes this one of the the worst bugs to diagnose.

One exception is Logicware's MProlog[8]. It has a user defined exception handling facility which allows one to recover gracefully from most bugs. This has a default setting to go into a break state at depth 10k. Often, such a symptom suggests a loop. To track it down the programmer can enter the tracer and search around for duplicated goals, etc.

[6] Where n=4 for Quintus, DEC-10 Prolog, n=3 for micro-Prolog and n=7 for Dave Plummer's SODA debugger —citesoda.

[7] Such as the execution AND/OR tree, the AND/OR 'search space' or the Byrd Box execution model

[8] Only an informal investigation has been made of the facilities provided by a number of Prolog implementations. Some of the following is based on discussions with Prolog suppliers at AAAI-86. Further work has yet to be done to detail both strengths and weaknesses of various systems.

The MProlog tracer allows tracing back from this break point. This is useful to find out why Prolog got into the current state.

A useful bug avoiding tool would be a static analyser to detect potential loops in code. Several groups either have or are working on such tools[9]. Such static tools are useful but a dynamic tool can handle situations that the static tool can never detect. A dynamic tool to search for loops at run time would therefore be most useful. For instance, a subsumption checker would go a long way to meeting this need. Subsumption checking is very expensive in general, so one would want to be able to turn it on and off at will, i.e. Prolog should be runnable in subsumption mode. It might be useful to have it on automatically after some prearranged time or at some prearranged point in the program in order not to waste time on checking bits of the search space suspected to be correct. Such a tool would be more useful than the MProlog depth bound, because it would locate the problem more accurately and involve the programmer in less search.

Checking for non-termination is an undecidable problem in general, so no perfect tool could be provided — neither a static nor a dynamic one. Improvements would always be possible and the programmer would always have to be prepared to resort to 'hand' checking of the tracer output. However, most non-terminations are caused by simple goal repetition, so a simple loop checker would go a long way.

3.3 Prolog Error Messages

By its very nature, Prolog provides few error messages. Typically, error messages are mostly for inappropriate arguments to system predicates, e.g. =.., is, etc, or for exhaustion of some system resource, e.g. stack overflow. Some errors are fatal and some not.

For system resource exhaustion errors one is mainly concerned to distinguish program looping from 'genuine' exhaustion. Looping tests were discussed in the last section and will not be repeated here. A major problem with this kind of error is that they are likely to make it difficult to continue to run Prolog and thus to provide any Prolog debugging tools. Something like the MProlog solution seems to be required, i.e. of predicting a problem before the resource is actually exhausted and going into a break to allow investigation.

In many Prologs the standard way to test for system predicate bugs is to discover which system predicate they refer to, and which procedures call this predicate, and then to trace these procedures to try to spot which one is at fault. If many procedures call the offending system predicate, or if any of these procedures is itself called often, then this is a most inefficient debugging strategy.

Since the Prolog interpreter 'knows' exactly which procedure call caused the error message to be printed, it should be simple to arrange for the tracer to be invoked at precisely the right point. In fact, MProlog allows just this but this is not the case with either Quintus Prolog v1.6 or Edinburgh Prolog (NIP).

[9]For example, McCabe and his colleagues incorporate the idea in their proposal to Alvey [McCabe et al 86].

Having found the offending system predicate call, the programmer will then probably want to trace back up the program run, looking for the point at which the trouble started. Most tracers [10] do not currently allow this: 'redoing' only being possible back to where the tracer was invoked. The 'debug' mode in Edinburgh Prolog or Quintus ensures that information is kept which can be used to provide such facilities.

This search backwards for the point of trouble could also be automated, e.g. an uninstantiated input to 'is' might be identified and the tracer requested to backtrack to where this was first introduced. Back-searches for the point of variable introduction or binding is a general requirement.

Side-Effects

Unexpected generation of side-effects is much like system predicate error messages — it is desirable to get into the tracer at the point at which the side effect happened and then root around. The main difference is that Prolog cannot be expected to do this without the programmer specifying which side-effects are regarded as unexpected and unwanted. Some way is needed of stating that writing this term or asserting that one is unwanted and then to be put into the tracer at the point at which these events are happening.

Failure to produce side effects can be tackled initially by spying the procedure which should have produced the side effect. If this fails because this procedure was never called then it is a case such as 'unexpected no' and the remarks in the next section can be adapted.

Unexpected 'No' or 'Yes'

Prolog has been called "the language that likes to say no". 'No' when it was expecting 'yes' or some output binding is certainly one of the most common symptoms of trouble. It is also one of the hardest to debug, since there are so many potential causes and so little evidence to go on.

The normal debugging technique is to use the tracer to compare the actual search space with the one expected. The main goal has failed and one is trying to trace that failure down through the search space to the lowest failing subgoal. This may be caused by:

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

[10] MProlog, again, being a rare exception.

The programmer might suspect some particular procedure to be at fault, e.g. if it has just been added to or edited in a previously correct program. In this case s/he will spy this procedure. Failing such suspicions, the programmer must resort to a more exhaustive search. Most programmers either step through the program top down, or test each subprocedure in turn working bottom up, or try some middle out variant.

Exactly where the tracer should initially enter in the execution history is a difficult problem. One of the simplest possibilities is to enter the tracer at the point at which the first goal failed in the chain of most recently failing goals. Either this goal should not have failed and selective backwards tracing can now be attempted to pinpoint why, or this goal should never have been called and selective backwards tracing can now be used to discover how this part of the search space was reached. Note that this technique requires the ability to re-enter the tracer after a top level call has been exited — no Prolog system currently provides such an ability [11] but it should not be hard to implement provided one is prepared to go into a special mode before making the call.

Unexpected 'yes' when 'no' was expected is very rare but can be dealt with in a similar way to the above.

3.6 Wrong Binding

The wrong binding of an answer variable is currently similar to the previous case of unexpected 'no'. However, the additional information implicit in the erroneous binding gives us much more to go on. The tracer could be made to back up in turn through each unification which contributed to the binding. At each stage it is known what the binding was before the procedure call and what new instantiations were made. The user can then choose whether to go on up or root around at this point. Note that this requires the ability (originally suggested in the 'error messages' section) to trace variable bindings, as well as the ability (originally suggested in the 'unexpected no' section) to trace back into a terminated program.

It has been suggested that answer variables could be traced forwards instead of backwards. This avoids the problem of tracing backwards and re-entering terminated programs, but it is less efficient as a debugging strategy. Forwards tracing forces the programmer to enter and search bits of the search tree which will ultimately fail, whereas backwards tracing will enter only the ultimately successful branch of the tree.

This completes the survey of the symptom level bugs descriptions. Now, attention is given to the second level: program misbehaviour.

3.7 Program Misbehaviour

Debugging at the program misbehaviour level consists mainly of tracking the immediate cause of the symptom to its 'initial' cause[12]. When the 'initial' cause is found it can

[11] To the best of our knowledge.

[12] The term 'initial' is odd because, as will be seen, it is initial only at the program misbehaviour level and itself has a cause at the code level. That is why it is in, and will remain in, scare quotes.

be identified because its cause can be found in some corresponding code level bug. For instance, the immediate cause may be some Prolog error message. This might be caused by an unbound variable, caused by a clause being unexpectedly called, caused by an earlier clause unexpectedly failing, caused by its head not matching the goal. This last is the 'initial' cause at the program misbehaviour level because its cause is a code level bug, say a mistyped function name.

It is assumed that the test for an 'initial' cause can be done by inspection of the code corresponding to the erring clause, i.e. without any special tools beyond the ability to access source code from the tracer [13] However, the tracer might be augmented to assist in the tracking of the causal chain between immediate and 'initial'.

Most such augmentations have already been mentioned above:

- the ability to search for nested identical goals or, more generally, subsumption
- the ability to trace backwards ('redo') from a break point
- the ability to trace back into a terminated program
- the ability to name a variable and have the tracer look for its last binding or its introduction
- the ability to drop into the tracer at the point where an error message or unwanted side effect were generated

The reason that most of these augmentations have already been considered is that, for that most part, the program misbehaviour bug classification echoes the symptom level one but within the search space rather than external to it. The main exception is unexpected calling or non-calling of a clause, but this does not seem to require any augmentations or tools beyond those discussed above.

4 Conclusion

Examination of available debugging strategies for each of the bug types at the symptom level has been a fruitful activity in terms of revealing the shortcomings of existing Prolog environments and suggesting improvements. The normal debugging strategies available in DEC-10/Quintus type environments are very inefficient for many bug types, although the situation is a little better in some other Prologs, e.g. ESI's Prolog-2 and especially Logicware's MProlog.

A little thought suggests some dramatic improvements over the current situation: in the main improvements to the tracer to make it more selective and to enable kinds of tracing not currently allowed. These suggestions are listed at the end of the previous section. These could be implemented, although some of them would be costly in resources and one would want to switch them on specially rather than always pay the overhead.

[13] Sadly lacking in most tracers. A rare exception is Dave Plummer's SODA debugger [Plummer 85].

The proposal by McCabe, Wilk, Thwaites and Ramsay indicates an intention to provide tools to do a post mortem analysis of the stack which would make implementation of some of the above ideas feasible [McCabe et al 86].

The debugging strategies used by Prolog experts for each Prolog bug symptom and for bugs at other levels will be investigated further. More ideas will emerge as further Prolog bug types are considered [14] and further relations between them. More work is especially needed on the code level and static tools, and about declarative classifications and tools.

Much work is still needed to develop the classification of program code errors. In the near future, there will be a more thorough investigation of the range of programming techniques.

Acknowledgements

This research was supported by Alvey Grant number GR/D/44287. We thank the other members of the Mathematical Reasoning Group and the Programming Support Group at Edinburgh for numerous conversations and useful feedback. Our thanks for conversations at the AAAI-86 exhibition with: Jonathan Grayson and Alex Goodall (Expert Systems International), Lew Baxter (Logicware), and Fred Malouf and Bill Kornfeld (Quintus).

[14] It is suspected that it may be necessary to refine even the symptom level further.

References

[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.

[Coombs & Stell 85] M. J. Coombs and J. G. Stell. *A Model for Debugging PROLOG by Symbolic Execution: The Separation of Specification and Procedure*. Research Report MMIGR137, Department of Computer Science, University of Strathclyde, 1985.

[Eisenstadt 84] M. Eisenstadt. A powerful Prolog trace package. In T. O'Shea, editor, *ECAI-84: Advances in Artificial Intelligence*, Elsevier Science Publishers, 1984.

[Eisenstadt 85] M. Eisenstadt. Retrospective zooming: a knowledge based tracking and debugging methodology for logic programming. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, 1985.

[Eisenstadt et al 84] M. Eisenstadt, T. Hasemer, and F. Kriwaczek. An improved user interface for prolog. 1984. INTERACT-84, IFIP conference on Human-Computer Interaction.

[Lloyd 86] J.W. Lloyd. *Declarative Error Diagnosis*. Technical Report 86/3, Department of Computer Science, University of Melbourne, 1986.

[Looi & Ross 86] C.K. Looi and P.M. Ross. Automatic program debugging for a Prolog Intelligent Tutoring System. 1986. Forthcoming.

[Looi 86] C.K. Looi. *Automatic Program Debugging for a Prolog Intelligent Teaching System —A Thesis Proposal*. Discussion Paper 30, Dept. of Artificial Intelligence, Edinburgh, 1986.

[Lynch 86] L. Lynch. *A Thesis Proposal for a Computer Progam to Teach Prolog*. Discussion Paper 10, Dept. of Artificial Intelligence, Edinburgh, 1986.

[McCabe et al 86] F. McCabe, G. Thwaites, A. Ramsay, and P.F. Wilk. Logic Programming Environment. 1986. A Submission to the Alvey IKBS Directorate.

[Pereira 86] L.M. Pereira. Rational debugging in logic programming. In E. Shapiro, editor, *Third International Conference on Logic Programming*, Springer Verlag, 1986. Lecture Notes in Computer Science No. 225.

[Plummer 85] D. Plummer. *SODA: Screen Oriented Debugging Aid.* Blue Book Note 260, Dept. of Artificial Intelligence, Edinburgh, 1985.

[Rajan 85] T. Rajan. *APT: The Design of Animated Tracing Tools for Novice Programmers.* Technical Report 15, HCRL, Open University, March 1985.

[Shapiro 82] E. Y. Shapiro. Algorithmic program diagnosis. *Association for Computing Machinery*, 299–308, June 1982.

[Taylor & duBoulay 86] J. Taylor and J.B.H. du Boulay. Why novices may find learning Prolog hard. In J. Rutkowska and C. Crook, editors, *The Child and the Computer: Issues for Developmental Psychology*, Wiley, 1986. forthcoming.

[vanSomeren 85] M. W. van Someren. *Beginners Problems in Learning Prolog.* Memorandum 54, Department of Experimental Psychology, University of Amsterdam, 1985.