A Framework for the Principled Debugging of Prolog Programs: How to Debug Non-Terminating Programs

A Framework for the Principled Debugging of Prolog Programs:
How to Debug Non-Terminating Programs

Paul Brna, Alan Bundy and Helen Pain

DAI RESEARCH PAPER NO. 472

March 1990

To appear in the Proceedings of the ALP-UK 90 Conference on Logic Programming

Department of Artificial Intelligence
University of Edinburgh

A Framework for the Principled Debugging of Prolog Programs: How to Debug Non-Terminating Programs

Paul Brna, Alan Bundy and Helen Pain

Abstract

The search for better Prolog debugging environments has taken a number of different paths of which three are particularly important: improvements to monitoring tools (notably the Transparent Prolog Machine (Eisenstadt & Brayshaw, 1987)), providing for greater user control over the debugging process (notably as in Opium$^+$ (Ducassé, 1988)), and partially automating the debugging process (notably in (Pereira, 1986; Lloyd, 1986; Pereira & Calejo, 1988; Naish, 1988)).

A serious problem associated with this activity lies in providing a principled conceptual framework within which the programmer can work with a number of different debugging tools. Here, we outline a framework that we have developed for the debugging of Prolog programs. We point out the relationship that holds between this framework and each of these three advances in debugging.

In order to demonstrate how the framework can be used, we explore an issue that has received relatively little attention recently: the run-time detection of programs that do not appear to terminate.

Our analysis of (apparent) non-termination is based on a four level Bug Description Framework that we have developed. This analysis goes further than the consideration of programs that would normally be regarded as 'looping'. We describe a debugging strategy in conjunction with a range of monitoring tools that provide greater assistance than currently found, We indicate the increased efficiency that would be gained through a close-coupling of the program construction and execution phases.

From this analysis, we see that current (non-graphical) debugging tools do not provide the necessary help to deal with the case of (apparent) non-termination. We also note that even a graphical debugger such as TPM (Eisenstadt & Brayshaw, 1987) does not provide all the desired assistance that we would like.

1 Introduction

Recent developments in the provision of tools to aid in debugging Prolog programs have been impressive. We briefly summarise the current situation:

Monitoring: The Transparent Prolog Machine (TPM) is intended to provide a faithful representation of the execution of Prolog programs in terms of an extended AND/OR tree. We will not detail all the different ways in which this system is of use —the interested reader should refer to (Brayshaw & Eisenstadt, 1989) for

further information. It does, however, include a number of aids to navigate through the execution space of a Prolog program —*e.g.* it allows the user to choose between a 'course grained' view and a 'fine grained' one. There is a 'data-flow' representation of the results of unification. The programmer can hide subtrees and has some further control on the amount of detail. A Byrd box debugger is included within the system.

This kind of system has its associated difficulties. The programmer is offered a great deal of choice as to which way he/she should examine program execution. If, however, the programmer is to make full use of such facilities then an overarching debugging framework has the potential to be of great assistance.

Flexibility: The Opium$^+$ debugger provides the user with the greatest degree of control. It allows the programmer to write his/her own debugging aids (in Prolog) —see (Ducassé, 1988) for details. In this way it avoids the problem of prescribing the precise way in which debugging should take place and hands over the decision to the programmer who then has the task of writing the necessary debugging tools in Prolog. This provides for flexibility but evades the issue of the conceptual framework in which debugging takes place. We hope that the designers of Opium$^+$ will continue its development with a view to providing the necessary debugging framework.

Automation: The work on algorithmic debugging, inspired by Shapiro, is valuable because it provided a theoretical basis for the debugging of pure Prolog programs (ones not featuring the use of the cut, assert etc.). Other workers have tried to extend the theory to make it more useful for 'real' Prolog programming —*e.g.* Drabent *et al* (Drabent *et al*, 1988). Apart from various improvements in the efficiency of the diagnosis, we are particularly interested in an approach which respects the cognitive requirements of Prolog programmers. Thus we are interested in Lloyd's top-down diagnosis of wrong answers not because it can be shown to be more efficient that Shapiro's divide and query approach in some circumstances (in terms of queries answered by the programmer/oracle) but because the top-down approach (especially if this preserves the left-to-right order of subgoals) can be argued as providing a better mapping between the programmer's procedural expectations and the queries that the programmer/oracle has to answer (Lloyd, 1986).

Despite these various advances, many researchers have commented upon the difficulties facing the programmer in connection with non-terminating programs (Brough & Hogger, 1986; Covington, 1985; van Gelder, 1987).

In this paper we explore the problem of debugging programs that do not appear to terminate. That is, the programmer expects that the program should terminate but it seems to be taking far too long. The purposes of this exploration are: to outline an approach to the top-down debugging of such programs; to expose problems with current environments; to show how the situation could be improved; and to show how the conceptual framework that we have developed can be used.

Basically, we have the same agenda as Hogger and Brough: how can we avoid creating programs that do not appear to terminate; how can we spot that a program is going to

cause us problems before running it; and how can we pinpoint the reason for the surface manifestation of apparent non-termination.

In line with this, we presume that, ideally, Prolog environments should be augmented with tools for creating programs that can be guaranteed to terminate (as suggested, for example in (Bundy, 1988)) and with tools for detecting non-terminating programs through some form of analysis of the whole program prior to execution. We also know that we will create programs (by accident or by design) that cannot be provided with a guarantee that they will terminate.

For the most part, we assume here that we have reached the stage of running a program which we want to terminate but for which there is no guarantee that it does. We further assume that during the course of testing the program we observe behaviour that makes us suspect, amongst a range of possible explanations, that we might have a non-terminating program.

We observe that this issue has received comparatively little attention outwith the recommendation of providing a standard run-time loop detection system. Certainly few of the researchers interested in developing Shapiro's approach to diagnosis have done more than provide such loop detection.

2 The Debugging Framework

We describe a four level Bug Description Framework as outlined in (Brna *et al*, 1987b). The classification and listing of bugs and debugging strategies is of fundamental importance: to give a foundation for motivating new, or improved, tools; and to provide a framework for integrating tools in a conceptually coherent way.

The particular framework which we have been developing is intended to account primarily for the bugs that are related to the program being developed. This excludes any consideration of bugs that lie outwith a good understanding of Prolog. We illustrate the framework rather than provide a formal definition:

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.

In our framework, the description of the symptom which indicates a buggy program must be an entry in the following table.

Error	Unexpected	Unexpected Failure to Produce	Wrong
Side Effect			
Error Message			
Termination			
Instantiated Variable			

Note that the description of *Unexpected Failure to Produce Termination* in the above table is used for the purposes of uniformity. We regard this as synonymous with *(Apparent) Non-Termination*. *Wrong Termination*, although it looks like a unusual description, covers the cases of *Unexpected Failure of Goal* and *Unexpected Success of Goal*[1].

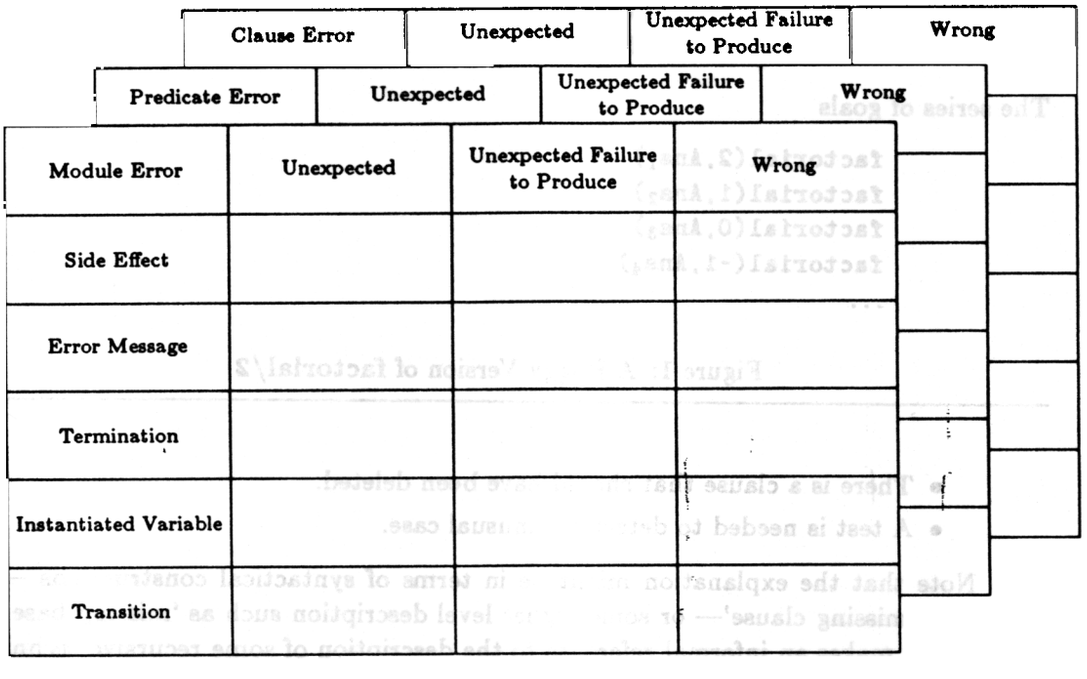
We provide some simple examples.

- Exit with a Prolog error message such as, for example, one caused by an uninstantiated variable in an arithmetic expression. This is an *Unexpected Error Message.*
- Exit to the Prolog top level from an editor (written in Prolog). This is a case of *Unexpected Termination.*
- A Prolog goal succeeds without pretty printing a result —a case of *Unexpected Failure to Produce a Side Effect.*
- A Prolog goal unexpectedly fails —a case of *Wrong Termination.*
- A Prolog goal fails to bind a variable to the desired result —a case of *Unexpected Failure to Produce an Instantiation.*

Program Misbehaviour Description The explanation that is offered for a symptom. The language used is in terms of the flow of control and relates, therefore, to run-time behaviour. At the *Symptom* level, a program is a 'black box'. At the *Program Misbehaviour* level, it is a series of black boxes. Therefore, the detailed analysis of the nature of the *Program Misbehaviour* description level includes the *Symptom* level.

The *Program Misbehaviour* description level also includes a dimension that reflects expectations about control flow and another connected with the granularity of the program —*viz.* whether we look at the program as a set of black box modules, predicates, or clauses. We illustrate with the following three dimensional table.

[1] We admit that some items are less likely to occur than others. The relative frequency of these bugs is somewhat application-dependent.



We also outline the detail present at this level.

- A case of *Wrong Termination* for a specific subgoal of the toplevel query. This illustrates a situation where we can make use of the *Symptom* level descriptions at some finer level of detail. In this case, we have used a level of granularity based on predicates.
- The unexpected failure to make use of some particular clause —perhaps caused by a failure to unify its head with the current, expected goal. This requires a clause-based level of granularity for the description of the problem at the *Program Misbehaviour* level.
- An unexpected sequence of goals —*e.g.* when it is expected that a recursive call of some predicate will terminate only to find that it does not do so. Consider the program for `factorial/2` found in figure 1. This suggests that we have an unexpected sequence of goals (we have to move to the *Program Code Error* description level to provide the standard 'missing base case' explanation for this error).

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.

```
factorial(N,Ans):-
        N1 is N-1,
        factorial(N1,Ans1),
        Ans is N*Ans1.
```

The series of goals

```
factorial(2,Ans1)
factorial(1,Ans2)
factorial(0,Ans3)
factorial(-1,Ans4)
```

Figure 1: A Buggy Version of **factorial/2**

- There is a clause that should have been deleted.
- A test is needed to detect an unusual case.

Note that the explanation might be in terms of syntactical constructions —such as 'a missing clause'— or some higher level description such as 'missing base case' which makes an informal reference to the description of some recursive technique.

We regard such techniques as distinct from algorithms: they have a parallel with the notion of programming plan as used by Spohrer *et al* (Spohrer *et al*, 1985). We give a simple schema (specialised for lists) of the kind of code that is repeated again and again —here, **CDCH** stands for *Constructing Datastructures in the Clause Head*, **ChangeH** indicates the mapping between individual elements of the list datastructure and ...indicates an arbitrary number of further arguments. Both **CDCH** and **ChangeH** are capitalised to indicate that these are variables ranging over predicates.

```
:- mode(CDCH,...+, ...-).
:- mode(ChangeH, ...+, ...-, ...).

CDCH(...[], ...[]).
CDCH(...[H1|T1], ...[H2|T2]):-
        .........
        ChangeH(...H1, ...H2,     )
        .........
        CDCH(...T1, ...T2).
```

We believe such techniques are important because a) there is a great deal of anec-

dotal evidence that Prolog programmers make use of such informal notions (both in debugging programs and in writing programs), b) we can go some way to defining a number of useful techniques (Brna *et al*, 1988; Gegg-Harrison, 1989), and c) there is some leverage to be had by making information obtained from the program creation stage available to the Prolog debugging system.

Underlying Misconception Description The fundamental misunderstandings and false beliefs that the programmer may have to overcome in order to maintain a consistent meaning for the program being developed. We can distinguish a number of different levels at which there might be misconceptions.

- Computation in general
- The underlying operating system
- The specification of the problem
- The Prolog language
- The nature of code which allows for efficient execution
- Program semantics

All these issues are important aspects of the programming process.

We restrict our attention mainly to the problems associated with the step from (assumed) correct specification to correct Prolog code. This simplification is driven by our interest in supporting the activities of experienced programmers. Consequently, we assume that the programmer has no fundamental conceptual difficulties with any of Prolog's features. Therefore, we have mainly considered the problems associated with how the intended semantics may have changed during program development.

We can identify a number of classes of misconception:

- Removing a constraint (possibly partially)
- Imposing an additional constraint
- Maintaining the 'degree of constraint' but swapping (at least) one constraint for another

The kind of constraints to which we refer include: mode information; type information; user-defined operator declarations; and information obtained from the program construction stage.

As an illustration, consider the following program for `factorial/2` which has been written with the intention that the mode of `factorial/2` is given by `factorial(+,-)`:

```
factorial(0,1):- !.
factorial(N,Ans):-
        N1 is N-1,
        factorial(N1,Ans1),
        Ans is N*Ans1.
```

A modified Byrd box model trace for the query `factorial(0,0)` provides us with:

```
call:  factorial(0,Ans1)
call:  factorial(-1,Ans2)
call:  factorial(-2,Ans3)
```

The *Program Code Error* featured in this program can be seen as arising, in part, from the belief that the mode is really `factorial(+,?)`. One possible explanation for this is that the programmer has assumed that the mode could be generalised.

Although we have mainly focussed on the problems facing programmers who have a good model of Prolog execution, we ought to extend the framework to allow for misunderstandings about Prolog. This is because even relatively experienced programmers possess some subtle misunderstandings about Prolog execution. For example, many have problems with how to write code to obtain increased efficiency. This is an important issue which needs further attention.

The approach we are advocating is mainly a descriptive one. However, we also hold a weak hypothesis that states that programmers will progress from a *Symptom* description through to the *Program Code Error* description via the *Program Misbehaviour* description and then, if necessary, on to the *Underlying Misconception* level.

For example, using the program listed in figure 1, we start by obtaining a *Symptom* of *(Apparent) Non-Termination*, we explore the behaviour with the help of a debugger and, perhaps, obtain the *Program Misbehaviour* of *Unexpected Transition* in that we did not expect to find the subgoal `factorial(-1,Ans4)`. Analysing the code results in a *Program Code Error* of, say, *Missing Base Case* in the application of the technique of *Primitive Recursion*. We do not pursue this error to the level of *Underlying Misconception* —except to say that such an error, if not due to some slip (such as forgetfulness), suggests that the programmer has some problem with the nature of computation in general.

Note that, for this paper, we are mainly concerned with the move from the *Symptom* of *(Apparent) Non-Termination* to the level of *Program Misbehaviour*.

3 The Analysis of (Apparent) Non-Termination

Our *Bug Description Framework* provides the framework within which we can develop various debugging strategies to help the programmer move from symptom to cause. We now set out to show how the framework helps us sift through the various possibilities. We have chosen to illustrate how, given the *Symptom* of *(Apparent) Non-Termination*, we can provide some guidance about how we proceed in our search for the root cause of the problem.

First, we remark that the basic symptom can be seen as arising from causes that can either be reduced to *Program Misbehaviour* descriptions or to factors outwith the

correct behaviour of the Prolog system[2] —such as a bug in the Prolog interpreter or in the operating system. We will do no in-depth exploration of the problems associated with an incorrect Prolog interpreter, an incorrect operating system or some bug in the interface. Consequently, the *Program Misbehaviour* descriptions can be seen as concerning the expected behaviour of some program relative to Prolog. We can consider them as relating to various *internal* factors. The other causes can be seen as relating to *external* factors. Our emphasis here is on the *internal* factors.

Given that we have decided to ignore *external* factors, we can turn to consider how we use the information about the symptom that we have detected to drive the search for a description at the *Program Misbehaviour* level.

We show how this search leads to a consideration of the activities of the Prolog Interpreter in building the execution tree. We will take this tree to be an extended AND/OR tree —and, since it therefore makes sense to see the programmer as working with the Transparent Prolog Machine (TPM), we assume the same notation (Eisenstadt & Brayshaw, 1987). We also assume that the programmer examines any side effects and results over a finite period of time via a single window for I/O[3]. Note that the TPM captures some notion of backtracking as further annotations to the tree.

We start from the *Symptom* of *(Apparent) Non-Termination* for some procedure call. Suppose that we now 'pursue' this symptom, with the help of a debugger, through the AND/OR execution tree until we can go no further. We assume that we use a top-down approach: we **skip** over a call. If we decide that we have a bug symptom then we **retry** and **creep** into the body of the procedure.

Let us assume also that the programmer believes that the length of time that he/she has waited is long enough to trigger the necessary description of *(Apparent) Non-Termination*[4].

Scenario 1 We keep on finding a subgoal of the query under current investigation which has the *Program Misbehaviour* of *(Apparent) Non-Termination* but we eventually cannot **creep** into the body of the procedure.

We have reached a leaf of the execution tree[5].

We call this *Suspended Building.*

Scenario 2 We keep on finding a subgoal of the query under current investigation which has the *Program Misbehaviour* of *(Apparent) Non-Termination* and, though we don't know it without some further analysis, we will continue to find this to be the case.

We call this *Malignant Endless Building.*

[2] *Prolog Misbehaviour* descriptions? Also, inter alia, *Operating System Misbehaviour* and *Network Misbehaviour* descriptions.

[3] This situation is only a crude approximation to the circumstances under which debugging takes place.

[4] It is always possible, of course, that the programmer was wrong in ascribing *(Apparent) Non-Termination.*

[5] If a Prolog system made use of modules that were compiled in such a way as to be untraceable then calls to externally visible procedures would, in effect, be system predicates.

Scenario 3 We keep on finding a subgoal of the query under current investigation which has the *Program Misbehaviour* of *(Apparent) Non-Termination* and, though we don't know it without some further analysis, this process will eventually terminate.

We call this *Slow Building*[6].

Scenario 4 We find one subgoal of the query under current investigation fails and, on backtracking, a previous subgoal succeeds. This pattern continues and, thought we don't know it without further analysis, will continue forever because the subgoal that failed will always do so and the subgoal that succeeded on redoing will always do so too.

The subgoal that always succeeds is a case of *Benign Endless Building.*

Scenario 5 We find one subgoal of the query under current investigation has a *Program Misbehaviour* of *Unexpected Failure to Produce a Side Effect.* On further investigation, we note that the side effect was produced but we had not looked in the right place for this.

This is termed *Hidden Building.*

In section 4, we show how these notions are of help in debugging programs featuring *(Apparent) Non-Termination.*

From this partial analysis, we can distinguish two states of affairs concerning the execution tree during the programmer's observations: it can be in the same state throughout; or it can have changed state. We can also distinguish two related states in connection with side effects: they may or may not have occurred as expected.

Note that we should include an aspect that derives from our framework in connection with error messages. We get a modified analysis by considering, for example, whether we get a single error message when we **skip** a procedure call and derive a *Program Misbehaviour* of *(Apparent) Non-Termination* or whether we get an apparently endless stream of such messages when we **skip** the procedure call. We are in the process of extending our analysis to meet this deficiency in the near future.

We have structured this analysis in terms of three categories of low level behaviour: *Suspended Building*, *Hidden Building* and *Unfinished Building.* We have sketched how the various explanations for these kinds of behaviour are derived by considering the *Program Misbehaviour* involved and shown how they are a refinement of our previous classification.

Although we are mainly interested here in the *internal* factors as it it these that we wish to link up to the *Program Misbehaviour* description, we also provide some indication as to how the same basic scheme can be used in terms of the *external* factors. First, however, we consider the *internal* factors and give a more formal description:

Suspended Building The execution tree is not being extended and neither is backtracking taking place. Consequently, no side effect activity is visible. This can only

[6]This is not to say that the call succeeds —it may fail. Also, another subgoal of the top level query might suffer from *Malignant Endless Building.*

occur when some system primitive is being executed and has not terminated —*e.g.* waiting for keyboard input via `read/1`.

Hidden Building The execution tree is being built but some side effect is not visible in the expected place —*e.g.* some use of `write/1` is expected to produce output on the terminal's screen but the output is being redirected to a file.

Unfinished Building The execution tree is being built but Prolog has not finished executing the program in the time during which the program has been observed. All associated side effects are as expected. Common description of this might include looping (endless building) or building very complex data structures (slow building).

Note that the category of *Unfinished Building* permits two subcategories —those of *Endless Building* and of *Slow Building*. If the programmer can show that the program terminates and if we have a case that is definitely that of *Unfinished Building* then we must have a case of *Slow Building*. Unfortunately, the programmer may have high confidence in the program's termination but not be 100% sure. Consequently, a major problem (but, in practice, by no means the only one) is to discriminate between *Slow Building* and *Endless Building*.

Endless Building itself can be further divided in terms of *Malignant Endless Building* and *Benign Endless Building*: we shall discuss this issue further in section 3.4.

There are equivalents for these in terms of external factors:

Suspended Building Since the factors are now external ones, we seek explanations in terms that lie outwith Prolog — *e.g.* the system is dead, or the terminal has been set to take no input.

Hidden Building Again, we can find a corresponding situation in terms of external factors —*e.g.* the terminal has been set to redirect/flush all output.

Unfinished Building This could be due to a variety of causes. The system is heavily laden (sometimes producing an effect equivalent to slow building), or even that there is a bug in the Prolog interpreter (perhaps endless building).

It might be thought that we could construct a decision tree that could be used to guide us through the debugging process. We know, however, that the debugging strategy chosen will depend on contextual information that is not captured by the framework. Consequently, the programmer should be able to choose from a number of *checks* and *investigations* —a subset of which are shown in figure 2[7]. A *check* has a fairly definite test associated with it while an *investigation* is a more open-ended sequence of activities. For example, the check for *Suspended Building* is relatively straightforward but investigating whether or not we have a case of *Endless Building* is much harder.

As we do not explore the *external* factors in great detail, we note briefly that the check for whether the problem is connected with external factors would require us to formulate

[7]This can only be a subset because we have not yet extended our analysis to other symptoms.

an approach which includes checks to see that the computer is still functioning, that the loading is adequately low, that the terminal is functioning properly and so on.

Investigate for External Factors
Check for Suspended Building
Check for Hidden Building
Investigate for Slow Building
Investigate for Endless Building

Figure 2: Different Strategies Available

3.1 Suspended Building

The case of *Suspended Building* due to *internal* factors can be reduced to the *Program Misbehaviour* of *(Apparent) Non-Termination* for a system predicate.

An example would be a use of `skip/1` which might be waiting for a particular character to be input before terminating —but this has not happened. A pathological case occurs (in several well-known Prolog systems) with the built-in predicate `length/2` when it is used to evaluate the length of a circular list —*e.g.*:

```
?- X=[a,b,c|X],length(X,Y)
```

Although the *occurs* check should spot the circularity in the unification associated with `X=[a,b,c|X]`, report this and fail (possibly reporting this as an error), many Prolog systems do not do this. A call to `length/2` then has to evaluate a list which is of infinite length. Therefore, the behaviour of the system predicate can be described as a *Program Misbehaviour* of *(Apparent) Non-Termination.*

The check as to whether we are waiting on the successful termination of a built-in predicate ought to be straightforward —assuming we have ruled out a possible external factor. If we are using the TPM, it is simply a matter of inspecting the display.

3.2 Hidden Building

If we have the *Symptom* of *(Apparent) Non-Termination* then we presumably were not expecting to see any side effects. If we had been expecting to see evidence of side effects then we would have derived the *Symptom* of *Unexpected Failure to Produce a Side Effect* —and we are not directly addressing the issue as to how to debug programs with this symptom.

On the other hand, when we are searching for an explanation for *(Apparent) Non-Termination* then we might be about to examine a procedure that we do associate with side effects. If we look for these and they do not occur where we expect them to occur then we have a *Program Misbehaviour* of *Unexpected Failure to Produce a Side Effect.* We have already pointed out how this can be seen as *Hidden Building* due to *internal* factors.

This would suggest that the programmer did not expect some particular program control choice to be made —*e.g.* the program may either produce a side-effect (or go off and do something else which does not produce a side-effect).

We might think that only a very few system predicates can ultimately be responsible for this. Unfortunately, the predicates that can achieve this effect include many of the side effecting ones —*e.g.* `assert/1`, `record/3`, `retract/1`, `tell/1` etc.

3.3 Unfinished Building

We have already pointed out that this can be seen as either *Slow Building* or *Endless Building.* Discriminating between these two cases is a major problem: if we have a case of *Slow Building* then we may want to know which aspect of the computation is producing the problem.

The main point about *Slow Building* is that the computation will terminate but it is taking a longer time to do so than anticipated. Given that there are no unusual *external* factors, we might have simply failed to appreciate that the datastructures being manipulated would be as complex as they actually are. This raises issues about the anticipated computational complexity of the code.

When we turn to the case of *Endless Building* then we ask whether we want to distinguish between different forms of this. We obtain a natural distinction if we remember that our execution tree is an augmented AND/OR tree. In particular, the annotations which we attach to the tree skeleton allow us to capture the program's backtracking behaviour.

We can distinguish between two extreme cases of *Endless Building*: one case in which the fundamental tree structure is being extended; and another case in which the tree structure is not being extended but further annotations are being added. A purely deterministic program suffering from *Endless Building* would feature the first of these behaviours.

As an example, consider this buggy version of `factorial/2`:

```
factorial(N,Ans):-
        N1 is N-1,
        factorial(N1,Ans1),
        Ans is N*Ans1.
```

If we pose the query `factorial(2,X)` then the execution tree built is infinite and no backtracking takes place.

Now consider a very artificial example that makes use of a *failure driven loop*:

```
echo:-
        repeat,
        read(X),
        write(X),
        nl,
        fail.
```

This program is not necessarily buggy —but if the programmer intended to insert a test for some input term then the program is buggy. We might ascribe a *Program Code Error* of *Missing Test* in the implementation of a *Generate and Test* technique.

A call to `echo/0` does not terminate. We can regard this call as doing no building of the execution tree skeleton after the call to `fail/0`[8]. This is subject to the proviso that `repeat/0` is a system primitive.

This is a specific instance of a much more general scenario which we informally describe by the metaphoric description of 'bouncing around'. Basically we have a recursively defined procedure that terminates but, later, is redone as a consequence of backtracking. The call terminates and the computation eventually backtracks to redo the call whereupon the call terminates —and so on.

This raises an issue as to how many "kinds" of non-termination we really want to distinguish.

3.4 Kinds of Endless Building

The fundamental case is that in which the flow of control enters the Byrd box associated with this procedure and never exits. We could informally regard this as *Malignant Endless Building.*

Another interesting case arises when the program contains a procedure call that always generates a further solution on demand (*i.e.* an infinite generator), and if, for each such solution, there is some later call that will (finitely) fail for that solution. Again, we have *Endless Building* —but this time we will informally call this *Benign Endless Building.* We can see this situation as corresponding to a generalisation of the *failure-driven loop* technique which, if unintended, has resulted in a bug.

There is a third option: we have a procedure call that follows the *Benign Endless Building* behaviour for a few (or even very many) calls but then, eventually, turns into *Malignant Endless Building.*

In short, there are two fundamental ways in which a call can turn out to produce *Endless Building.* Any other non-terminating behaviour may result from the combination of these two fundamental causes.

[8] We appreciate that it is possible to see this differently

4 Debugging an (Apparently) Non-Terminating Program

We now apply a simple top-down debugging strategy which copes with several different *Symptoms* other than the one in which we are interested in here —namely, the *Symptom* of *(Apparent) Non-Termination*. The basic schema is to:

> Turn on the trace
> Issue the goal
> `creep` to examine the subgoals
> `skip` over each subgoal
> If an incorrect result is detected, `retry` the last subgoal
> `creep` to examine the behaviour of the defective subgoal's subgoals
> Repeat the process for the new set of subgoals

This approach is related to the partially automated approach to finding missing answers suggested by Shapiro (Shapiro, 1983) and the approach for tracking down wrong answers outlined by Lloyd (Lloyd, 1986). We explore debugging strategies for programs exhibiting the *Symptom* of *(Apparent) Non-Termination* using a similar top-down approach. This is of interest as other attempts to track down the causes of non-termination have generally examined the goal stack after some arbitrary depth bound has been exceeded.

Our method of presentation is to describe an idealised situation and then relate this to the actual facilities usually provided.

4.1 Suspended Building

If we are using a graphical debugger that shows the execution tree as it is being grown then detecting *(Apparent) Non-Termination* caused by *Suspended Building* is trivial since no tree building activity will be visible —so it will be clear where the execution has stopped.

Note that the graphical debugger must not be a post-mortem one for this to work. The TPM is able to make use of either a post-mortem mode or a run-time mode. Consequently, the TPM is ideal for this problem.

Now consider how we might locate this problem using a standard DEC-10 style debugger. During the execution of the top-level goal, we can apply the following strategy for the standard debugger:

> Raise an interrupt;
> Switch on tracing;
> If no trace output, raise another interrupt; and
> Abort

and we have a *Program Misbehaviour* of *(Apparent) Non-Termination* for a system predicate. Now we have to track down exactly where in the program this is happening as the

debugger gives no help either with identifying the system predicate causing the offence or with the context in which this predicate is called!

Although using the simple top-down approach with the standard debugger does work, the process is extremely slow. This is roughly how it works out: using the top-down approach, we spot which subgoal of the top-level goal caused the problem. We then have to start again as the debugger doesn't offer the normal range of choices (since we are stuck 'inside' a box). Now we go one level deeper to spot the subgoal of the subgoal that causes the problem —and so on. We can keep going but this is really painful.

How can the standard debugger be improved so that we can avoid this tedious problem? In principle, the Prolog system knows which system primitives can cause *Suspended Building*. We need the Prolog system to tell us which is the last 'box' that has been entered. On interrupting execution, we should be able to request this information.

An alternative approach which is more in tune with the top-down approach requires that the debugger can determine that the execution of some of the subgoals of a goal cannot be involved in causing *Suspended Building*. The system could then safely `skip` over them —otherwise, we should require the system to recommend the programmer to `creep` 'into the body' of the relevant goal. Note that this suggests that a programmer be able to communicate which hypothesis is being followed up.

4.2 Hidden Building

As pointed out above, if we seek an explanation for *(Apparent) Non-Termination* then we would probably not expect *Hidden Building* due to *internal* factors. This is because *Hidden Building* carries with it the idea that we are dealing with some form of *Unexpected Failure to Produce a Side Effect.*

On the other hand, the check that something *is* going on would be cheap so that we can at least determine whether or not we are in this situation *if we had the right tools.*

This class of tools is one which we might expect Prolog to possess in order to handle this type of problem. We have already suggested that a view onto the program database is essential for the basic story of how Prolog works (Bundy *et al*, 1985) and have shown how this is necessary for extending the story to cope with side-effects such as those caused by `assert/1` and `record/3` (Brna *et al*, 1987a). We also need to consider views onto all side-effects caused by `write/1`.

The desired approach is to be able to view activity on any output (or input) channel. This would allow the programmer to inspect a window for each open channel (to see what is going on) and to monitor activity on the channel. We can then expect the programmer to be alerted to a channel wanting/receiving data —it is up to the programmer to know which channels should carry which items of data.

Although it is possible to cobble together some partial remedy for this situation, few environments really handle this well. As far as we are aware, Eisenstadt and Brayshaw's TPM does not possess these tools either (Brayshaw & Eisenstadt, 1989). The best we can do is to inspect various aspects of the environment for these side effects —*e.g.* using

`tail -f filename` (in the Unix-like environment) to see how some file is being processed or looking for side effects associated with built-in predicates such as `write/1`. Although the separate I/O window (such as that provided by MacProlog) is an improvement over the standard debugger, we do not see why the programmer should not be provided with monitoring of all channel activity as a default.

4.3 Unfinished Building

We now consider the case of *Unfinished Building.* by considering the ways in which we might detect that we have a case of *Slow Building.*

Now, if we have ruled out all but the case of *Unfinished Building* then finding positive evidence in favour of *Slow Building* is negative evidence in favour of *Endless Building* and vice versa. The check for *Suspended Building* provides very strong positive evidence —as does the check for *Hidden Building.* The investigations that can now be done are theoretically impossible to turn into a completely general decision procedure (*c.f.* the halting problem).

The main point about *Slow Building* is that the computation will terminate but it is taking a longer time to do so than anticipated. Given that there are no unusual *external* factors, we might have simply failed to appreciate that the datastructures being manipulated would be as complex as they actually are. This raises issues about the anticipated computational complexity of the code.

There are various checks that we might do on the computation in order to provide us with some positive evidence in favour of *Slow Building.* Many of these checks can be done during program construction. If we can do these checks within the environment in which program construction is done then so much the better. If the guarantees provided by the checks can be passed reliably to the program execution stage and, therefore, to the debugger, then we should have an easier task in debugging our program. Here, we assume that we have not obtained the necessary guarantees for at least some of the predicates found within the program.

- For some procedures that are recursing, check that some input structures are the anticipated ones: this is connected with type checking
- For some procedures which are recursing, check that the recursion arguments are 'decreasing' in anticipated ways
- For some procedures that are recursing, check that some output structures are being built in the anticipated ways
- For some procedure(s) which is recursing, check that the inputs and outputs are sufficiently well instantiated: this is connected with the expected modes but can be carried further than the standard description (*i.e.* not just +, - and ? but also describing the positions of variables in arbitrary terms)

- For some procedures that are recursing, check that the depth of the recursion is no larger than expected: this is connected to complexity analysis

In the absence of any guarantees that the program will terminate these checks will not provide completely reliable evidence.

We will also briefly examine the basic techniques for looking for positive evidence of *Endless Building*. The received wisdom is to examine the ancestors and look for a repeated goal. The main suggestion here (that we have made before) is to automate the search for these repeated goals. If none are found, then continue. From our point of view this is, however, not quite sufficient as a strategy. This is partly because several different accounts can be given in terms of the *Program Misbehaviour*.

Input Terms are OK We use an example of a very slow sorting program —perms/3— as shown in figure 3.

```
perms(X,Y):-
        perms(X,[],Y)

perms(Inlist,Acc,Outlist):-
        perm(Inlist,Perm),
        \+ memberchk(Perm,Acc),!,
        perms(Inlist,[Perm|Acc],Outlist).
perms(X,Acc,Acc).

perm([],[]).
perm(X,[A|Y]):-
        extract(A,X,B),
        perm(B,Y).

extract(A,[A|Tail],Tail).
extract(A,[H|T],[H|Ans]):-
        extract(A,T,Ans).
```

Figure 3: Code Featuring 'Slow' Building

For example, we can examine the growth of the *accumulator argument* used in perms/3 using the advice package (as shown below).

The advice package was written by Richard O'Keefe in 1984 to provide Interlisp-like advice package facilities. The intention is that it should be used to implement a trace

```
?- advise(perms(_,X,_),call,(write('Call:  '),write(X),nl)).
?- advise(perms(_,X,_),exit,(write('Exit:  '),write(X),nl)).
?- advise(perms(_,X,_),fail,(write('Fail:  '),write(X),nl)).
?- advise(perms(_,X,_),redo,(write('Redo:  '),write(X),nl)).

?- perms([a,b,c,d,e,f],X)

Call: []
Call: [[a,b,c,d,e,f]]
Call: [[a,b,c,d,f,e],[a,b,c,d,e,f]]
Call: [[a,b,c,e,d,f],[a,b,c,d,f,e],[a,b,c,d,e,f]]
Call: [[a,b,c,e,f,d],[a,b,c,e,d,f],[a,b,c,d,f,e],[a,b,c,d,e,f]]
Call: [[a,b,c,f,d,e],[a,b,c,e,f,d],[a,b,c,e,d,f],[a,b,c,d,f,e],[a,b,c,d,e,f]]
```

Figure 4: Watching the Accumulator Grow

package. It allows the programmer to attach output commands and checking commands to the various ports defined by the Byrd box model (O'Keefe, 1984). We show how this can be done in figure 4. We can see how the *accumulator* is growing: as there are no `exits`, `fails` or `redos` we can be sure that the datastructure is being built recursively[9]. Here, we are fairly sure that the structure is being built correctly as we can see that each addition is added at the head of the list and that each is a permutation of the input list and that there are no repetitions —*i.e.* we can check additions against a set of expectations that we have. We can think of this as checking the type we have implicitly assigned to the accumulator.

In general, if we give the accumulator argument a type and we arrange for this type to be checked at run-time then we cannot guarantee that the lack of a type error means that all is going well. All we can guarantee is that if our type description of the *accumulator* argument is broken then we will be told.

In this case, we have some hope of spotting a problem but, in general, we have no hope of looking at some very complex term and reliably saying "yes, that is exactly what the program should produce".

Recursion Argument 'Decreasing' Healthily Examining the recursion argument of `perms/3` cannot be done because there isn't one. Instead, let us consider the program found in figure 5.

[9] We can also see how clumsy it is to provide the necessary specification of what is wanted.

```
intersect([],_ , []).
intersect([Element|Residue], Set, Result) :-
        member(Element, Set), !,
        Result = [Element|Intersection],
        intersect(Residue, Set, Intersection).
intersect([_|Rest], Set, Intersection) :-
        intersect(Rest, Set, Intersection).
```

Figure 5: Code Featuring a Recursion Argument

The predicate `intersect/3` is considered to have mode `intersect(+,+,-)` and the first argument position is the recursion argument. Let us assume that we are finding the intersection of two fairly large sets. Then we might want to engage in a dialogue with Prolog such as found in figure 6. With this trace, we can spot all is well as the numbers output indicate the number of elements in the list which is the input in the recursive argument position.

As long as the programmer has some expectation of how the recursion is supposed to progress, such a tool is very useful. Although we have given a fairly trivial example, the basic idea is useful and can be generalised to constructor functions other than [...|...]. Of course, there are other measures but the commonest ones can be catered for. Another extension would provide for cases where there are two recursion arguments and so on.

Output Structures 'Growing' Well When we know that an argument of some predicate is going to be instantiated more fully then we may want to examine this argument position to see how the growth is going. Unfortunately, this is a hard thing to do using the `advice` package.

Essentially, we want to accumulate a record from information that is not available at call time but is

- Sometimes available after unification of the goal with the head of a clause
- Sometimes only available at the time when the next recursive call is made

We will consider a desirable scenario for the program in figure 7 with the goal:

```
delete(a,[a,s,w,e,r,s,a,a,a,e,r,t,q],X)
```

The output is found in figure 8.

```
?- advise(intersect(X,_,_),call,(length(X,Y),write('Call:  '),write(Y),nl)).
?- advise(intersect(X,_,_),exit,(length(X,Y),write('Exit:  '),write(Y),nl)).
?- advise(intersect(X,_,_),fail,(length(X,Y),write('Fail:  '),write(Y),nl)).
?- advise(intersect(X,_,_),redo,(length(X,Y),write('Redo:  '),write(Y),nl)).

?- intersect([a,b,c,d,e,f],[s,e,r,t,a,g,a,z,s,e],X).

Call: 6
Call: 5
Call: 4
Call: 3
Call: 2
Call: 1
Call: 0
Exit: 1
```

Figure 6: Watching the Recursion Argument Decrease

```
delete(El,[],[]).
delete(El,[El|Rest],Ans):-
        delete(El,Rest,Ans),!.
delete(El,[H|Rest],[H|Ans]):-
        delete(El,Rest,Ans).
```

Figure 7: Code Featuring Output Structures

```
?- watch_grow(delete(_,_,X)).

?- delete(a,[a,s,w,e,r,s,a,a,a,e,r,t,q],X).

_
[s|_]
[s,w|_]
[s,w,e|_]
[s,w,e,r|_]
[s,w,e,r,s|_]
[s,w,e,r,s|_]
[s,w,e,r,s|_]
[s,w,e,r,s|_]
[s,w,e,r,s,e|_]
[s,w,e,r,s,e,r|_]
[s,w,e,r,s,e,r,t|_]
[s,w,e,r,s,e,r,t,q|_]
```

Figure 8:

There are many problems with this kind of output. If backtracking causes some structure building to be undone this can be hard to follow. On the other hand, if we can move away from the teletype model for output to a window model we could at least watch the dynamic growth and contraction of the structure as it is built. In many cases, however, it will not be easy to say that a datastructure is being built correctly.

Groundness of Input and Output Arguments We will not look at an example of this case now. There is a strong parallel with the type checking situation anyway.

Checking the Depth of Recursion Dec-10 Prolog had a feature which allowed the programmer to set an arbitrary number of calls as the maximum allowed before interrupting the computation[10]. This is a very crude way of managing affairs. A less crude method would be to have user-definable bounds on all recursive predicates. This would allow the programmer to represent intuitions about the complexity of the computation. An ambitious extension would be to try to infer the expected depth of the recursion for each recursive predicate and for each combination of inputs. The check, therefore, is to be warned (*e.g.* dumped in the debugger) whenever some threshold is crossed.

Looking for Benign Endless Building If we suspect that we have a case of *Benign Endless Building* then the best solution is a graphical debugger —based on the TPM— with which we are able to watch control 'bouncing around'. This is, howver, not ideal because we would see too much behaviour —so we would prefer much of the irrelevant information to be suppressed.

Turning to the standard debugger, we would like to use the top-down approach to quickly find the lowest root that contains both the over generating procedure call and the goal that keeps on failing.

This can be done by a variant of the standard top-down method (we know here we will have trouble in trying to `skip` over some predicate): switch on the trace, issue the query, `creep` to examine the subgoals, `skip` over each subgoal, if it looks like *(Apparent) Non-Termination* then switch tracing on and go back to the `call` port of the problematic `skipped` call, `creep` to examine the subgoals, `skip` over each subgoal until the failing subgoal found and then `backup` to the last succeeding subgoal.

Now the `backup` command should go back to the `call` port of the most recent box *at the same level in the execution tree* whose call has outstanding alternatives to try. This is, however, still not exactly what we want.

Between the infinite generator and the subgoal that keeps on failing there may be several subgoals that are non-deterministic for the particular inputs provided. Unless we have some intuition about which subgoal is infinitely generating we cannot say whether a given subgoal is the cause of the problem[11].

[10] Other Prolog implementations have taken up this idea. This includes Quintus Prolog and MProlog.

[11] Note that we can expect that only a limited number of such predicates need to be investigated.

This still does not guarantee that we have found the first 'wall'[12] of the two between which the 'control' ball might be bouncing[13]. So we should be able to set up markers which say effectively "unleash the trace for a bit but only report 'bounces' between this subgoal and the failing subgoal. If a bounce doesn't occur then report this as I will want to set up new 'walls'".

We may suspect that we are in this situation —locked forever into the cycle represented informally as "a ball bouncing between two walls"— but we don't know for certain. It may turn out, for example, that there are other 'walls' awaiting being 'set up' once the current 'walls' are breached[14]. For example, suppose we suspect that we have the following situation:

```
goal:-
        subgoal1,
        subgoal2,
        subgoal3,
        subgoal4,
        subgoal5.
```

We assume that we suspect that $\mathbf{subgoal_2}$ is an infinite generator and $\mathbf{subgoal_4}$ is a call that always fails. This may be incorrect in both respects and it may well be that the real combination of infinite generator and failing subgoal is as in:

```
goal:-
        subgoal1,      wall1
        subgoal2,
        subgoal3,
        subgoal4,
        subgoal5.      wall2
```

This does not settle the matter because there may be other walls awaiting being 'set up'. Things can be even worse in that we might have a number of different failing goals and a number of different procedures that are over-generating. Consequently, we need efficient ways of extracting the desired information automatically under programmer control.

We can also expect that some will have been subjected to analysis so that we can be given guarantees that they do terminate. Also, an intelligent debugger could cut out any subgoals that are not defined recursively *for this particular command*. We would like to be able to tell the debugger that we should keep on going back (if there is another subgoal left to go back to).

[12] *i.e.* the infinitely generating procedure call.

[13] The second of these 'walls', the procedure call that keeps on failing, is found inside that failing subgoal but we don't know exactly where yet and, taking the top-down approach, we don't need to know for now.

[14] For example, we might only suspect an infinitely generating predicate —and this suspicion is not well founded.

Looking for Malignant Endless Building If, on the other hand, we are looking for evidence of a procedure that produces the behaviour of *Malignant Endless Building*, then we can find the offending call much more easily using the standard debugger. We can keep on `skipping`, `retrying` and `interrupting` by hand but we would like to be able to automate this and/or be able to attach print statements to the apparent cause of the problem as discussed previously. The graphical debugger again promises much if some way can be found to suppress detail automatically.

If we have a program that possesses a predicate which will at first, generate solutions that keep on being rejected by some call that always fails then things can fall out in one of two main ways (all other things being equal). If we interrupt too early, then there is a hope that we can detect the 'walls' between which the 'control ball' is bouncing. If we wait long enough then we can use a similar method as for detecting the predicate featuring *Malignant Endless Building*.

5 The Bug Description Framework Revisited

We have discussed the debugging of apparently non-terminating programs in terms of the execution tree. This has allowed us to move towards the description of various possible and hypothetical debugging tactics.

We now briefly discuss how our description of *Program Misbehaviours* takes into account the various inputs to a procedure. For instance, suppose we provide a top-level query that has an incorrect parameter value[15]. Alternatively, suppose that a subgoal is provided with an input through the success of a previous subgoal. This suggests that one of three possible events has taken place:

- We have made a mistake in the 'bolting' together of the various parts of the program —*i.e.* the subgoal-about-to-executed is correctly written and everything that has taken place previously is correct but the programmer never intended this sequence of goals to occur in the code. Consequently, we have a *Program Code Error* description.
- We have suddenly realised that our specification for the subgoal-about-to-be-executed is misconceived. This suggests one of the kinds of error that are made at the *Underlying Misconception* level.
- If the subgoal-about-to-be-executed is correct in that no mistake has been made about what sorts of input the procedure should work on then a previous subgoal may have produced some *Program Misbehaviour* connected with instantiation (unexpected instantiation, unexpected failure to instantiate or a wrong instantiation).

  Alternatively, we have taken a wrong path through the search space which could be due to a variety of causes —including a previously undetected case of a procedure being called with the wrong inputs. This might be characterised as an *Unexpected Transition* which is one of the possible *Program Misbehaviour* descriptions.

[15] Pereira and Calejo call such a goal *inadmissible* (Pereira & Calejo, 1988).

As we can see, encountering such a problem, may lead us to investigate for the cause in terms of *Program Misbehaviour*, *Program Code Error* or *Underlying Misconception*. Yet, on the surface, we have what can be regarded as a type error.

We also point out how the behaviour of recursively defined procedures fits into our framework. There are two types of information considered here: the recursion argument(s) and the expected depth of recursion for some recursive procedure(s).

Knowledge about how the recursion argument should decrease is a consequence of the intentions of the programmer at the code writing stage. Knowing that a certain recursion argument should decrease during execution places an expectation on an argument of the recursive call. This appears in our framework at the *Program Misbehaviour* level in roughly the same way that type errors appear.

At first sight, knowing about the expected depth does not appear to fit into our framework at all. It imposes an expectation on some particular recursive call not being found after the expected number of recursive calls. If we are examining the execution of a program that results in a recursive program exceeding the anticipated depth then the occurrence of such a call results in a *Program Misbehaviour* description of an *Unexpected Transition* in that (presumably) the expectation is that termination should have been achieved through calling some different procedure.

6 Other Approaches

We restrict ourselves here to some brief remarks.

- The graphics-based debuggers such as TPM do win in the case of *(Apparent) Non-Termination* of a system predicate but it is extremely painful to use the standard debugger to localise the fault.
- No systems provide much direct support for watching channel activity in order to detect errors described as *Unexpected Failure to Produce a Side Effect*. This includes TPM as well.
- There are few tools that help programmers monitor potential problems connected to the *Slow Building/Endless Building* scenarios. The spy package built by Richard O'Keefe and distributed with the Prolog library is flexible —but is not flexible enough. Dichev and du Boulay's data tracing system is too primitive and aimed at novices (Dichev & duBoulay, 1988). We are not aware of how we could handle this problem in Opium$^+$. The retrospective approaches advocated by Shapiro and many others can only be regarded as a partial solution. As far as we know, TPM cannot easily handle this situation either.
- Watching out for (what is, in effect, unintended) endless failure-driven loops is not supported directly by any system of which we are aware. Again, it may be that Opium$^+$ can be used to construct a useful tool. The TPM is partly useful here.

We note that often systems fail to provide the right 'handles' for debugging: programmers are not able to manipulate debuggers in terms of the strategies that they are following. Instead, programmers are required to encode their strategies to fit the low level facilities provided. We would like to see a more positive approach to controlling debugging within a good conceptual structure.

The work of Shapiro suggests that communicating expectations to the debugger (*i.e.* pursuing wrong answers or missing answers) pays off. Calejo and Pereira have developed a much friendlier dialogue with the programmer allowing for a greater range of expectations to be communicated (Pereira & Calejo, 1989). Other work, such as that by Ducassé (Ducassé, 1988) may yet provide the groundwork for building a far more powerful system for communicating expectations about programs at a level closer to the one at which a programmer is currently working. We have shown here the need both to extend the scope of such work and to take into account the various different levels at which debugging takes place.

7 Conclusions

We summarise our observations about the needs for debugging Prolog programs.

Debugging Prolog programs requires a high level framework which can be used by the programmer to guide his/her debugging. For example, there is a need for a high level interface to O'Keefe's advise package.

- By using the framework, we can reduce some debugging strategies to a level at which the Prolog system ought to be able to offer considerably more help than is typically the case. This required us to introduce the notions of *Suspended Building* etc.
- We have pointed out the problem with finding the root cause of *Suspended Building* and indicated how the standard debugger can be improved at low cost.
- We have indicated how the problem of dealing with *Hidden Building* can be eased by automatically providing each I/O channel with a monitor.
- We have stressed the need for tools: to watch the growth of terms; to watch the behaviour of a term associated with a recursion argument; and to watch the growth of output terms.
- We have suggested an elaboration on the simple idea of a depth limit on the computation by annotating predicates with an expression that provides some notion of the expected computational complexity.
- We have motivated the need for tools to watch for patterns associated with a buggy version of the failure-driven loop —the infinite generate-always fail loop.

In short, we have made a contribution to the methodology of debugging programs and illustrated this with respect to programs that do not appear to terminate. We have also suggested a strategy for debugging based on: seeking to discriminate between *internal* and *external* factors; checking for *Suspended Building* and *Hidden Building*; and discriminating between *Slow Building* and *Endless Building*.

- We have analysed the *Symptom* of *(Apparent) Non-Termination* in terms of a number of factors related to the behaviour of Prolog.
- We have pointed out that, although distinguishing between *Slow Building* and *Endless Building* is a major problem, there are other, non-trivial causes that need methods for debugging them.
- We have indicated a number of debugging tools useful to monitor the execution in ways which are not currently supported in Prolog environments.
- We have sought to show how the four level bug description framework motivates this analysis.

We believe that work on improving the debugging of Prolog programs must be based on more than a collection of low level programming tools. It is essential that we have a principled framework into which the various tools can be located.

Acknowledgements

This research was supported by SERC/Alvey Grant number GR/D/44287. Thanks to Alan Smaill for various discussions and helpful comments.

References

Brayshaw, M. and Eisenstadt, M. (1989). *A Practical Tracer for Prolog.* Technical Report 42, Human Cognition Research Laboratory, The Open University, Submitted to the International Journal of Man-Machine Studies.

Brna, P., Bundy, A., Pain, H. and Lynch, L. (1987a). *Impurities and the Proposed Prolog Story.* Working Paper 212, Department of Artificial Intelligence, Edinburgh.

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

Brna, P., Bundy, A., Dodd, T., Eisenstadt, M., Looi, C.K., Pain, H., Smith, B. and van Someren, M. (1988). *Prolog Programming Techniques.* Research Paper 403, Department of Artificial Intelligence, Edinburgh, To appear in the special issue of Instructional Science on Learning Prolog: Tools and Related Issues.

Brough, D.R. and Hogger, C.J. (1986). *The Treatment of Loops in Logic Programming.* Research Report DoC 86/16, Department of Computing, Imperial College of Science and Technology.

Bundy, A. (1988). *Proposal for a Recursive Techniques Editor for Prolog.* Research Paper 394, Department of Artificial Intelligence, Edinburgh, Submitted to the special issue of Instructional Science on Learning Prolog: Tools and Related Issues.

Bundy, A., Pain, H., Brna, P. and Lynch, L. (1985). *A Proposed Prolog Story.* Research Paper 283, Department of Artificial Intelligence, Edinburgh.

Covington, M. A. (1985). Eliminating unwanted loops in Prolog. *SIGPLAN Notices*, 20(1).

Dichev, C. and du Boulay, B. (1988). *A Data tracing System for Prolog Novices.* Serial No CSRP 113, School of Cognitive Studies, University of Sussex.

Drabent, W., Nadjm-Tehrani, S. and Maluszynski, J. (1988). Algorithmic debugging with assertions. In Lloyd, J.W., (ed.), *Proceedings of the Workshop on Meta Programming in Logic Programming*, pages 365–378, Bristol.

Ducassé, M. (1988). Opium$^+$, a meta-interpreter for Prolog. In Kodratoff, Y., (ed.), *ECAI 88: Proceedings of the 8th European Conference on Artificial Intelligence*, pages 272–277, Pitman Publishing, London.

Eisenstadt, M. and Brayshaw, M. (1987). *TPM Revisited.* Technical Report 21a, Human Cognition Research Laboratory, The Open University, An extended version of technical report 21 to appear in the Journal of Logic Programming.

Gegg-Harrison, T.S. (1989). *Basic Prolog Schemata.* Technical Report CS-1989-20, Department of Computer Science, Duke University.

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

Naish, L. (1988). *Declarative Diagnosis of Missing Answers.* Technical Report 88/9, Department of Computer Science, University of Melbourne.

O'Keefe, R. A. (1984). Advice.pl. Documentation and code provided with the Prolog Library —available from the Artificial Intelligence Applications Institute, Edinburgh University.

Pereira, L.M. and Calejo, M. (1988). A framework for Prolog debugging. In Kowalski, R.A. and Bowen, K.A., (eds.), *Fifth International Conference on Logic Programming*, pages 481–495, MIT Press.

Pereira, L.M. and Calejo, M. (1989). The debugging environment. Chapter 10 of the ALPES final report, Universidade Nova de Lisboa.

Pereira, L.M. (1986). Rational debugging in logic programming. In Shapiro, E., (ed.), *Third International Conference on Logic Programming*, pages 203–210, Springer Verlag, Lecture Notes in Computer Science No. 225.

Shapiro, E. Y. (1983). *Algorithmic Program Debugging*. MIT Press.

Spohrer, J.C., Soloway, E. and Pope, E. (1985). A goal-plan analysis of buggy pascal programs. *Human-Computer Interaction*, (1):163–207.

van Gelder, A. (March 1987). Efficient loop detection in Prolog using the tortoise-and-hare technique. *Journal of Logic Programming*, 4(1):23–32.