An Overview of Prolog Debugging Tools

AN OVERVIEW OF PROLOG DEBUGGING TOOLS

* Paul Brna
** Mike Brayshaw
* Alan Bundy
*** Mark Elsom-Cook
*** Pat Fung
**** Tony Dodd

DAI RESEARCH PAPER NO. 398

Submitted to the special issue of Instructional Science on Learning Prolog: Tools and Related Issues.

* Department of Artificial Intelligence, University of Edinburgh
** Human Cognition Research Laboratory, The Open University
*** Institute of Educational Technology, The Open University
**** Expert Systems International, Oxford

An Overview of Prolog Debugging Tools

Paul Brna[†] Mike Brayshaw[§] Alan Bundy[†] Mark Elsom-Cook[‡]
Pat Fung[‡] Tony Dodd[¶]

Abstract

In this paper we present an overview of the advances in debugging standard Prolog programs. The analysis offered is in terms of a classification of tools that provide different degrees of activity in the debugging process. Other possible dimensions of analysis are also outlined.

1 Introduction

In addition to being a relatively new programming language, Prolog has characteristics that are not present in more purely procedural languages such as BASIC or FORTRAN. The unification and backtracking processes in Prolog give rise to the possibility of an increased confusion about the location of errors. Further, programs can be difficult to comprehend at a functional level because the language does not have the more obvious syntactic markers that are found in, for example, Pascal or LISP. Programs can also be difficult to debug because of semantic errors: because of the logical nature of Prolog, no syntactically correct program would have any error messages associated with it at all except for the need for some of the built-in predicates to impose (non-logical) limitations on the ways in which predicates are used. These aspects must be catered for in debugging systems designed for Prolog.

[†]Department of Artificial Intelligence, University of Edinburgh

[§]Human Cognition Research Laboratory, The Open University

[‡]Institute of Educational Technology, The Open University

[¶]Expert Systems International

In general, debugging is the process by which errors in the program code are detected and fixed. This simple description does not reveal the full complexity of the debugging process. Even though we restrict our attention to standard Prolog, debugging Prolog programs is a complex and composite skill. In describing the skills that programmers need we make use of the descriptive framework developed by Brna *et al* (Brna *et al*, 1987). This provides four levels at which bugs can be classified: the *symptom*, *program misbehaviour*, *program code error* and *misconception* levels:

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.

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.

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.

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 further details, see the paper cited above.

Very briefly, programmers need to:

- Have a thorough grasp of the program's intended semantics.
- Know the various facilities that the programming environment provides.
- Know how to detect the gross *symptoms* of a *program code error*.
- Be able to refine the description of the error's effects in terms of the program's execution —this is the *program misbehaviour* description.

- Be able to abduce sensible *program code error* descriptions from the *program misbehaviour* description.

- Be able to eliminate plausible *program code errors* through sensible selection of discriminatory test cases and through the use of the facilities provided by the programming environment.

Occasionally, they may find it possible to go further and account for certain *program code errors* in terms of various *misconceptions*.

It has been stated that debugging is more akin to an art than a science. We review some aspects of debugging which try to improve the situation in a number of ways.

Improved Monitoring: First, we can seek to improve the facilities for monitoring the execution of a Prolog program. This produces indirect benefits for debugging. Indirect, because the ability to inspect the execution of a program at some level of detail enables programmers to apply their debugging skills. The benefits of improved monitoring facilities might include a decrease in the programmer's memory load, finer grained examination of execution (*e.g.* watching the resolution process at work as in Plummer's Coda (Plummer, 1987) and Rajan's APT (Rajan, 1985)), the ability to focus attention in more efficient ways and the use of different forms of presentation (*e.g.* 2 dimensional forms such as AND/OR trees, Ferguson diagrams (Dewar & Cleary, 1986) and so on). See (Pain & Bundy, 1987) for a review of a number of different ways of presenting the account of how Prolog interprets a program. These accounts are termed *Prolog Stories*.

Automated Search: Next, we can seek to automate the search for specific events in the execution history. This might be done, for example, by augmenting the monitoring tools. For instance, Eisenstadt's PTP system provided the means for searching for *suspicious symptom clusters* (Eisenstadt, 1984).

Annotated Programs: We can also provide the programmer with the means of specifying various pieces of information about a program at edit-time. For example, the programmer might want to declare that certain predicates are intended to be

deterministic. That is, if they succeed at all then they can never be resatisfied. This kind of annotation can provide extra leverage on the debugging process and help with automated search.

Guided Debugging: We can also seek to impose a dialogue upon the debugging task. The programmer and the system cooperate in the tracking down of errors —the rôle of the system is to ask the programmer questions in order to guide the search for program code errors. Generally speaking, the initiative for the dialogue is taken over by the system.

Automatic Debugging and Program Verification: The idea of a completely automatic method of detecting the program code errors (and possibly fixing them) is attractive but critically depends on the system effectively knowing the program specification. Where the complete specification is 'transformed' into a corresponding program then there is a need to debug the program by appealing to the specification. Here, we are interested in *program verification.*

Automatic Debugging and Tutoring Systems: There is also an application in the area of intelligent tutoring systems (ITS) where the goal is to teach novice programmers to write correct programs. In this context, the system can set problems for which there are known solutions. The automatic detection mechanism can be used to point out the errors in the code and check that the correct fixes have been applied. This is not a direct approach to teaching programmers how to debug —if it does teach debugging then there has to be some presentation of the reasoning involved. Systems like PROUST report program code errors in syntactically correct Pascal programs but do not give any explanation of the debugging process (Johnson & Soloway, 1985).

Teaching Debugging Skills: Finally, and somewhat unrelated to the above approaches, we can try to educate programmers in the various debugging skills.

One last comment is needed about the most seductive approach of all: preventing program code errors from arising in the first place. It is often stated that Prolog programs

can be seen as executable specifications. Consequently, if the programmer can get the specification right then there will be no program code errors. Unfortunately, it is not so simple to get the specification right. Even if the specification is correct, Prolog will execute the 'specification' in a way that takes into account information that is not strictly part of the specification. For example, clause ordering, the cut (!) and so on. This is a further factor in the creation of program code errors. The need for debugging tools and skills will not go away so quickly.

The above description (excluding the reference to teaching debugging skills) of debugging has been loosely organised in terms of increasing control by the debugging system —ranging from no control to complete control over the debugging process. There are other aspects which should be noted as significant. In addition to the degree of automation, these include: the extent to which the systems require a knowledge of the programmer's intentions, the mode of display, whether the debugging system is active before, during or after program execution, the reliability of the diagnosis of a program code error, to what extent the programmer can override the debugging system and who drives the dialogue between programmer and system.

It is therefore possible to identify a series of important dimensions along which we may view current debugging tools. Existing approaches have varied both in terms of debugging strategies applied, in the rôle of the user in this process, in flexibility, and in their accuracy.

In the next section we will attempt to deal in some detail with examples of existing systems following the above scheme. The rest of this section features an overview of current trends and contrasts in the area and identifies a number of themes, outlining their relative importance.

There are desirable repercussions of a system that might automatically infer bugs by itself. However, in practice, most systems either require user responses to specific queries (guided debugging) or give the user a powerful tool with which the user may rapidly locate and correct their program errors (automated search and monitoring). Only a few systems operate without interventions from the programmer. Those that do are mainly annotated program systems that try to carry out some form of static analysis prior to

execution — these provide a form of bug prevention.

The systems that require user responses typically try and debug the program while it is executing. The program typically asks for information about the desired behaviour of a particular goal or whether a particular result is correct. Such user querying allows the debugging program to infer intended program behaviour and thereby makes the task of program debugging for the mechanical program much easier. Such an approach requires much more of the user as this may involve answering difficult questions. The task is difficult because the user often finds it hard to infer the context in which the question is being asked by the system. Further, certain systems do not allow the users to change their mind and thus, if an incorrect answer or inputting mistake occurs, the program may well come to the wrong conclusion. The guided search approach also requires user participation and this is often associated with post-mortem debugging. A history of program execution is collected and a series of tools provided with which to examine the observed behaviour. Clearly, for the guided search approach, more information is available through post-mortem examination of program behaviour.

Providing that the debugger is given correct information most current guided debugging systems, if they claim to have detected a bug, are almost certain to have correctly identified the error. Different systems, however, may provide diagnoses of varying precision and, for example, the discovery that a predicate has its arguments the wrong way round may be due to a program code error that lies elsewhere.

Most such systems also have a large computational overhead. This means that they take time and, to be effective, require a large machine. One possible alternative approach, which might prove a source of relief to this problem, is for systems to employ heuristic methods in the hope of providing a series of good guesses as to the cause of the problem — *i.e.* they are automated search systems. Typically this might involve suspicious symptom detectors which try to advise on possible bugs, *e.g.* failure due to wrong arity or no definition. The system identifies things that it thinks may prove to be the source of the error — it is thus fallible.

Such systems by their nature are merely advisory. They do not insist that they have arrived at the correct answer, and that there is no other; at the other end of this dimension

are the systems that, once they believe that they have detected a bug are emphatic about it, and will not let the user proceed unless the bug the system thinks it sees is removed. Such insistency we term *prescriptive insistency*. A second type of insistency we refer to as *dialectic insistency* whereby the system is emphatic in its interactions with the user and doesn't allow the user to influence the course of the debugging session.

Another dimension along which systems vary relates to the type and number of bugs which they attempt to catch. Indeed the type of bug is further restricted in some systems which can only cope with a subset of the language — for example, excluding the metalogical extensions to Prolog.

A final dimension in which we may view the system is in terms of the cognitive load which it places upon the user. This problem is increased when the user is a novice faced with learning a new language. If the debugging tools also require a large amount of effort to learn, then the system may adversely affect program language learning. There is also a need for extra formalisms to enrich the descriptive language within which terms we describe the system. Such formalisms might provide a richer account both of the program's behaviour and about the process of understanding and debugging the program. Inevitably, there is a trade-off between the potential power of a debugging system and the amount of mental effort it takes to learn the system well enough to use it efficiently and effectively.

Improved monitoring methods can pay better dividends than guided (and automated) debugging methods. Certain of the guided debugging methods, particularly those that do a lot of work in order to locate a bug, come up against the problem of computational expense when finding a possible program code error. Given large programs produced by the professional Prolog programmer the cost of finding an erroneous clause may thus become prohibitively expensive. In contrast, although not without its own overheads, tracing can be much more efficient. The argument for tracing would be something like the following "You don't need a program that takes a long time to locate a bug if you can instantly spot the bug from a trace on the screen". However the intelligence in this case is very largely that of the user — deciphering the output of some trace packages is sometimes a job requiring considerable skill in its own right.

There is another use for tracers: that of program comprehension. You can view program debugging as a sub-field of program comprehension, but a program trace can often tell the programmer far more about a program than just a bug's location. This extra understanding, not only about the operational semantics of the program under inspection, but also about the nature and context of a possible error, may, for example, result in the programmer not touching the piece of code associated with the program misbehaviour but, instead, altering some other piece of code in order to stop the program getting into the erroneous state again.

2 Improved Monitoring

Plummer (Plummer, 1987) and Eisenstadt (Eisenstadt, 1984; Eisenstadt, 1985) have independently enhanced the traditional box model tracer (Byrd, 1980) to show details of individual clause head matching. In the normal debugger no information is available about which clauses have unified with the current goal and which have failed. Plummer produced an extension to this traditional model which showed individual clauses as they are attempted or retried on backtracking. Eisenstadt also showed individual clauses being tried, and additionally indicated seven types of clause failure. The technique of retrospective zooming allowed the user selectively to probe the trace, zooming in on suspicious-looking parts of the goal tree. In order to assist the user in spotting these suspicious symptoms, a knowledge-base of *symptom clusters* could identify candidate bugs.

Rajan presented novices with a detailed account of Prolog execution, highlighting unification, backtracking, and variable instantiation (Rajan, 1986; Rajan, 1987). The association between the novice's code and its behaviour was shown in his single-step tracer, by highlighting salient parts of the database at appropriate moments, and by showing instantiations in situ for the code being traced. Rajan showed that novices' performance improved after they were shown demonstrations using his single-stepper compared with a control group.

Dewar and Cleary (Dewar & Cleary, 1986) present a graphical tracer for Prolog based on Ferguson diagrams (see (van Emden, 1984)). Their system, DEWLAP, nicely shows

clause head selection. However the displays rapidly become very complex as all potential matching clause heads are drawn up individually and separately. Although goal-parent relationships are clearly shown, the overall trace sacrifices the unifying structure that an AND/OR tree captures. No distinction is made for system primitives, nor is the 'cut' dealt with explicitly. Lastly, in order to trace even fairly small programs, information away from the centre of focus is physically reduced, thus it is not possible to get a higher level perspective or different conceptual views of the program — only close-up views of small parts of it.

The development of the Transparent Prolog Machine (TPM) by Eisenstadt and Brayshaw (Eisenstadt & Brayshaw, 1986; Eisenstadt & Brayshaw, 1987) can be seen to incorporate many desirable monitoring facilities. The importance of clear and consistent models of program execution are well known both in terms of program language teaching and program debugging(*e.g.* DuBoulay et al, 1981; Rich and Waters, 1986). The use of AND/OR trees as the most effective way of capturing the execution of Prolog in persuasively argued by Bundy et al. (Bundy *et al*, 1985; Pain & Bundy, 1987). Applying these principles to Prolog, the Transparent Prolog Machine was introduced as a medium for visual presentation and animation of Prolog programs.

TPM was conceived as a tool for use by novice and expert Prolog programmers alike (Eisenstadt & Brayshaw, 1988). TPM provides a faithful representation of the inner workings of the Prolog interpreter. The system gives details about clause head matching and variable renaming and deals correctly with the 'cut'. Other meta-logical features like *setof/3* and declarative clause grammars (DCGs) are also supported. The current implementation serves as the uniform basis for textbook diagrams (Eisenstadt, 1988), video animations, and a graphics workstation (Eisenstadt & Brayshaw, 1987).

The workstation implementation has a *replay* facility with buttons for single step forward, single step backward, rewind, and normal play so that code may be observed either 'live', as the program executes, or in 'post-mortem', at the end of execution. A *selective highlight* option allows the user to home in on troublesome code specified in a context-sensitive way. Traditional box model debuggers (Byrd, 1980) are also provided and the implementation is currently being extended to incorporate graphical guided de-

bugging techniques (see section 5). Unlike earlier attempts to provide graphical tracing facilities for Prolog, TPM can deal with extremely large search spaces by viewing the program at different levels of detail. The close-up view, called the AORTA (And/Or Tree Augmented) contains information down to the level of individual variable unification and clause head matching. At a coarser-grained level the user can see the program just in terms of goal outcome. Large traces can be greatly condensed by compressing parts of the trace into user defined 'black-boxes'. Finally the user can define high-level abstract views of the program so that the trace can be viewed at a level that conforms more closely to the users' plans and intentions.

TPM is about to be evaluated for use both by novices and experts. One of the interesting questions that such a study addresses, aside from the obvious one as to the effectiveness of the environment as a whole, is what parts of the trace are critically important to the debugging process. By manipulating the traces themselves, and omitting features like variable re-naming, or detailed views of clause head matching, it may be possible to identify those things that appear to have the greatest effects on learning and debugging. Until such times as an empirical evaluation is complete, it is not possible to provide a formal evaluation of TPM's effectiveness compared with other debugging approaches.

3 Automated Search

Current Prolog tracers give power to the user to step or leap back and forward through the search tree tracking a bug symptom through its causal chain to the source of the problem. The standard on which most such tracers are modelled is the DEC-10 tracer which is based on the Byrd-box model of Prolog program execution —see (Byrd, 1980; Bowen, 1981).

There is little or no automated assistance with this tracking process, but an analysis of Prolog bugs by Brna *et al* (Brna *et al*, 1987) suggests automated assistance that would be helpful. For instance:

1. An automated search for nested identical goals or, more generally, subsumption;

2. The automated location of the last binding or introduction of a named variable;

3. The automated location of where an error message or side-effect were generated;

A further useful requirement is that such systems provide for the ability to re-enter the execution tree for terminated programs.

Item 1 would be useful for finding the cause of apparent non-termination. Item 2 would be useful for finding from where an answer variable got the wrong value. Such automated assistance often requires user input, *e.g.* to request a particular kind of assistance and to name the variable in item 2 or identify the side-effect in item 3[1]. Thereafter, the process would be automatic and would take place during the program run. This user input constitutes a very simple model of the users' expectations *e.g.* that a different value for a variable was expected. Diagnosis would be certain, but the user would be free to react to it as s/he felt appropriate. Assisted tracing could improve the efficiency with which a wide variety of *symptom* level bugs can be located.

Some progress towards these goals has been achieved in Eisenstadt and Brayshaw's Transparent Prolog Machine. TPM has a selective highlight facility which allows the user to highlight those nodes in the graphical trace which have been picked out as being of special interest. Those places where goals with user-specified parents and/or arguments are called may be rapidly located. Particular bindings of variables may be sought out (assisting with item 2 above) or the place where an error message or side-effect occurred easily located (item 3).

The whole process is time sensitive so that the user can either globally view these events or return to a particular moment in the program execution when they occurred. Once at that particular point the user can use the standard TPM replay facility to investigate further. The selective highlight facility identifies features that have happened in the program, thus it can be used both at the end of program execution (thus reinforcing the need for the ability to re-enter the execution tree after program termination) or when live execution of the trace has been temporarily halted.

Current work with TPM includes the development of a powerful built-in debugger,

[1]Simplifying the task of the user in providing such information is an interesting, but doable, HCI task.

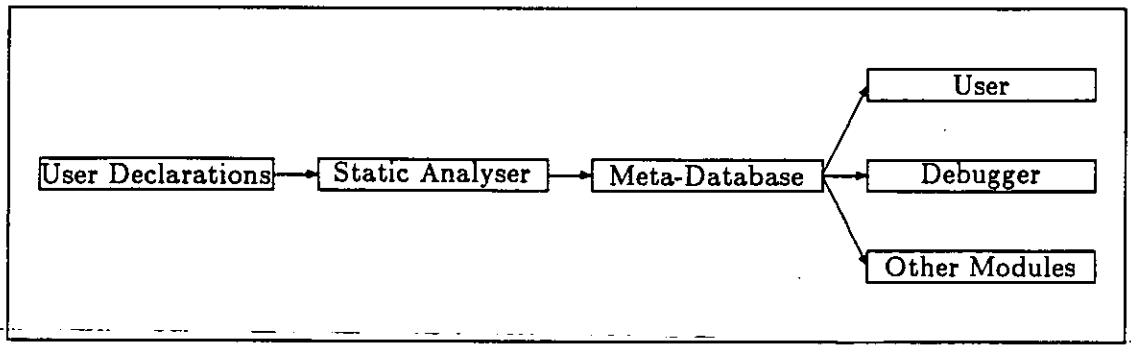
initially based on that of Pereira (Pereira, 1986) (see section 5 for a brief description), but with the intention of extending it with extra heuristic knowledge, so that it can semi-automate the task of bug location. Various bugs may already be picked up by a cliché analyser based on Eisenstadt's work to detect *suspicious symptom clusters* in a post-mortem analysis of an extended trace output (Eisenstadt, 1985).

4 Annotated Programs

We break this discussion into two parts. The first introduces the idea of gathering various kinds of information about the program's intended behaviour into a database. These intentions are supplied explicitly by the programmer at edit-time. The database is known as a *meta-database* because it is a database of facts about the program and its execution. The second section introduces the idea of a programming technique and the ways in which the meta-database approach can be extended.

4.1 The Meta-Database Approach

An approach now being investigated is the maintenance of a meta-database of facts about the program. This may be illustrated as follows:



The user edits a program and includes declarations about predicates, *e.g.*:

- Input modes and types of arguments
- Output modes and types of arguments
- Whether the predicate is deterministic

- Whether the predicate is guaranteed to succeed

The static analyser infers extra declarations and adds them to the meta-database. This can, in turn, be examined by the user and by other parts of the the system including the debugger. It can be used to compile more effective code.

The debugger (which can be the Byrd-box debugger mentioned in section 3, the Transparent Prolog Machine (TPM) mentioned in section 2 or almost any debugger) then monitors these declarations and stops with a message if any of them are violated.

For example:

```
process_row(I,MaxI) :-
        repeat(K),
        process_elt(I,K),
        K>= MaxI.
process_array(MaxJ, MaxI):-
        repeat(K),
        process_row(K,MaxI),
        K=MaxJ.
```

contains a bug. The programmers intention is inconsistent with the declaration.

```
? - deterministic(process_row/2).
```

and the debugger will detect that this has been violated when *process_row/2* resatisfies.

In terms of our earlier analysis of debugging tools:

- Meta-database techniques are semi-automatic. They can function with very little volunteered information but cannot infer large scale plans, at least not as presently implemented.

- Meta-database techniques are independent of display mode

- Meta-database techniques require pre-analysis and dynamic testing but no post-analysis.
- Meta-database techniques use very limited knowledge of programmer intentions. With new kinds of declaration, say of use of specific techniques (see section 4.2), they may use more.
- Meta-database techniques detect errors with complete certainty.
- Meta-database techniques are highly insistent in the prescriptive sense (especially if the compiler produces code relying on the violated declaration).
- Meta-database techniques (usually) deal with predicate-level bugs rather than program-wide ones.
- Meta-database techniques add a small cost to the development cycle if some static checking (such as a style check) is already done.

4.2 Technique-Oriented Debugging Tools

The formal definition of Prolog programming techniques (see (Brna *et al*, 1988)) offers the possibility of debugging tools based on these techniques. Such a tool would check a program to see that the technique was correctly implemented. For instance, a formal definition of the use of accumulators for constructing recursive datastructures would enable the building of a tool which would identify the recursive, accumulator and results arguments of a procedure and check that, among other things, the result was defined in terms of the accumulator in the base clause (see figure 1). If this check failed then the tool would report an error to the user.

Technique-oriented tools require knowledge of which technique the user intended to use, *e.g.* the user would have to explicitly request that a program be tested as realising a technique. The bug report would be limited to all and only the violations of the technique — wider intentions would not be taken into account. The cause of such a violation would normally be evident from the report given about the bug detected. The analysis process would be static and would take place before the program runs. It would be totally

```
:- mode(double,+, +, -)          % from this, the accumulator and result
                                 % arguments can be identified

double([], Acc, Acc).            % the result is defined
                                 % in terms of the accumulator
double([Hl|Tl], Acc, Res):-
        H2 is 2*Hl,
        double(Tl, [H2|Acc], Res).
```

Figure 1: An example of using the accumulator technique

automatic. The computation cost would vary according to the complexity of the analysis to be made and would vary from trivial to arbitrarily expensive. Technique-oriented debugging tools fit well into the framework of the meta-database (see section 4.1); with the user's intended techniques stored as meta-facts and the analysis being merged with the inferences being drawn from other meta-facts.

5 Guided Debugging

Here, we consider the style of debugging derived from the work of Shapiro (Shapiro, 1983). This approach depends crucially on the concept of an (infallible) source of knowledge about both whether a goal should succeed and which variable instantiations should be obtained[2]. This source is known as the *oracle*. The system uses this information to guide the programmer to the program code error.

The guided debugging approach requires, for some terminating query, that the programmer identify whether the result is the expected one. If not, the programmer submits

[2]The original work by Shapiro on *algorithmic debugging* also described an approach to the handling of non-termination.

the query to one of two programs. One of these programs handles the case that a wrong variable instantiation has been made and the other handles missing solutions —that is, some situation where the original query failed when it should have succeeded. Thus errors are presumed to be caused in two different ways (excluding any account as to how non-terminating programs arise) —at least one unsound predicate and at least one incomplete predicate. The programmer has to choose which of these two possibilities to pursue.

A dialogue ensues with the debugging system in relation to a re-execution of the original goal. Note that, as this takes place after the execution of the original goal, it would be difficult to integrate this system into a standard Prolog execution trace package —but see the comment on this in section 3 above. The debugging is expensive in two different ways. Computationally, it requires repeated application according to the number of bugs in the program (each run would provide one bug to be fixed). It also requires the oracle (in the first instance, the programmer) to answer a potentially large number of questions about which goals should succeed and what solutions should hold. The *query complexity* is a measure of how many statements need to be made by the oracle. Efforts are usually made to reduce this cost by storing oracular inputs so that, on reapplications of the debugging process, there is no need to ask the programmer exactly the same question twice. Drabent *et al* have sought to allow the *oracle* to provide more general descriptions of the intended program behaviour during the debugging process (Drabent *et al*, 1988).

For detecting the unsound procedure which caused a wrong variable binding, Shapiro uses a 'divide and query' algorithm. This takes part of the computation and finds a goal such that about half the total number of goals executed during that computation are prior. The oracle is then asked about whether this goal has a correct solution. If not, then the debugging process is applied for a wrong variable to this goal. Otherwise, the 'divide and query' process is repeated on the remainder of the computation. Lloyd has a 'top-down' debugging strategy which is more akin to the normal way of using the standard debugger (Lloyd, 1986). That is, for each goal (query), check whether its subgoals produce wrong variable instantiations. For the first one that does so, repeat the process.

The other possibility is that a predicate is incomplete in that it cannot handle all the

uses to which it is put by the program or programmer. For detecting such an incomplete procedure, Shapiro's system requires that the programmer issue the top level goal and then input the complete set of solutions for each subgoal as it arises.

There is a variant of algorithmic debugging produced by Pereira (Pereira, 1986). Lloyd points out that Pereira's system, known by him as *rational debugging*, suggests an expert system-like approach to debugging (Lloyd, 1986). Pereira achieves some increase in efficiency by keeping information around about term dependencies. For each term, there is an associated tree which indicates which other terms have made a contribution to the structure of the term through unification. The result of applying Shapiro's system is a suspect clause. Pereira's system produces a suspect term —which provides a much finer granularity for debugging.

In all these systems the dialogue is driven by the system. The programmer (oracle) cannot refuse to answer a definite question as the diagnosis depends crucially on the correctness of the programmer's input. If a mistake is made then the whole analysis is invalidated. Providing the means to back up and change an input is not currently a feature of such systems. This is a serious problem in that it is unreasonable to expect the programmer never to make a mistake and hence to know precisely which predicates will succeed or fail for given inputs and with what outputs —especially if the datastructures are very complex.

Thus these debugging systems gradually discover something about the intentions of the programmer during the debugging process. These intentions relate to the success/ failure of specific goals and associated variable bindings. Generally, Shapiro's system cannot handle the cut (!). There are suggestions for handling negation but few ideas for handling other non-logical features. On the other hand, Pereira's system can handle these features —because it uses the procedural semantics of Prolog.

To summarise, the oracle-based guided debugging systems are computationally expensive —but so is debugging with the standard tracer— and very inflexible.

6 Automatic Debugging and Program Verification

In the context of program verification, we need to show that the program meets the specification. As part of this activity we may well have recourse to both static and run-time analysis. At run-time, we require a form of automated debugging to locate the *program code error*. An implication of this approach is that the specification is distinct from the program.

If it is assumed that a Prolog program is a specification then we use this program (that we regard as the specification) and seek to transform it to a more efficient version. Note that, in general, arbitrary Prolog programs are not regarded as formal specifications —partly because there is too much control information embedded in the code.

One recent approach to automatically debugging Prolog programs is based upon Shapiro's *Algorithmic Debugging* (Dershowitz & Lee, 1987). Dershowitz and Lee have built a system, APD, which uses a specification of a program, written in Prolog itself, to generate test cases, locate a bug and correct it. This provides a demonstration that it is possible to debug example Prolog programs for a range of errors.

7 Automatic Debugging and Intelligent Tutoring Systems

We can take advantage of automatic debugging techniques as part of a programme to teach novice programmers to write correct programs. We assume that the idea is to construct an Intelligent Teaching System (ITS).

A novice writes a program. Meanwhile, behind the scene, the tutoring system automatically detects *program code errors* and is then able to suggest possible fixes. We have already pointed out that this is extremely difficult in the general case but that we can take advantage of the methods developed for automatic debugging in the context of teaching

Given that we can locate a program code error then we will want to make use of this information. Analyses of teacher/student dialogues (*e.g.* (Stevens *et al*, 1982)) have shown that errors are frequently the focal points of such dialogues; the teacher going on to infer novice misconceptions from such errors. It thus follows that the location of

program code errors plays a major rôle in tutoring systems. To do this requires that there be an automatic debugger to locate these bugs.

As we have already noted, to do this reliably requires a knowledge of the intended semantics of the program. A tutoring system, however, can be given the ability to set the novice programmer a task for which there is a sufficiently detailed specification. Two approaches being pursued include one which makes use of several sources of information —static analysis, dataflow and Shapiro's guided debugging system— and another which seeks to identify novice's misconceptions —especially in the area of misconceptions about Prolog's flow of control.

Looi describes an automated debugger for Prolog in an intelligent tutoring system context (Looi, 1988). The debugger performs an initial static analysis of the program — checking for things like looping and wrong argument types (Mycroft & O'Keefe, 1984). The system attempts to compare the student program against known library examples, or if this proves unfruitful, checks the buggy code's behaviour against the behaviour of a correct program, using a version of Shapiro's debugging system.

Following an empirical study (Fung *et al*, 1987), Elsom-Cook and Fung (Fung, 1988) have proposed using an adaption of a calculus of communicating systems (CCS) as a formal semantics for describing the behaviour of the Prolog interpreter (Milner, 1980). CCS views a program as a composition of linked communicating statements. By the application of a calculation rule the space of all the possible behaviours of the program may be developed. This however is not necessarily desirable, and thus the space can be pruned by the application of heuristics. These heuristics are at the level of general misconceptions about the language as a whole, they are thus above the level of simply having a library of buggy versions of a program with which to compare the student input. Diagnosis of errors proceeds by the comparison of the modelled incorrect behaviour and a known correct version. The major contributions are therefore that the system doesn't have to rely on a bug library entry for each of the individual bugs contained in a program, but can instead generate these possible bugs itself, using high-level knowledge of the domain. Potentially the system can be generalised to the whole Prolog language. The current implementation works for a subset of variable free Prolog.

7.1 Case Studies in Other Languages

Debugging in Prolog is strongly related to debugging in other languages — the approaches share similar objectives. In this section we take three case studies of debugging systems from other languages and suggest possible areas of cross-fertilisation.

The first is an example of trying to debug a program as late as possible *i.e.* at the time the program is loaded for execution. The last is an example of debugging immediately a deviation from the correct path is made. The middle approach lies somewhere between these two extremes. For general purpose debugging, these systems can only illustrate the various styles of debugging.

It is important to note, however, that Prolog has a series of unique features which may well indicate that it requires special features in its debuggers. The unification and backtracking behaviours of the interpreter are cases in point. They are certainly a source of errors for the novice (Taylor, 1987; Fung *et al*, 1987). Likewise experts can make errors using them, requiring a debugger to be sensitive to such behaviour. For novices also the lack of syntactic cues as compared to other languages may cause additional problems specific to Prolog (Gilmore, in press).

7.1.1 Debugging using PROUST

The central tenet of the PROUST system (Johnson, 1986) is that a (Pascal) program can be described as a set of intentions on the part of the programmer. These intentions are used to guide the production of a hierarchy of plans, which eventually lead to actual code in a programming language.

PROUST takes as input the student's code, and a set of intentions describing what that program should achieve. Using this information the system builds a model of the student in terms of plans and buggy plans which provide a mapping between the intentions and the actual program. The bugs which are needed to produce this model can then be used as input to a program which gives the student advice about the errors which s/he has made. PROUST achieves this mapping using the process of analysis by synthesis. This involves starting from the intentions and, using a library of plans, building every program

which could satisfy those goals. If the program which the student has written is not one of these, then PROUST assumes it must contain errors. At this point it attempts to replace or modify correct plans in the solution using the library of faulty plans. This process continues until PROUST successfully "explains" the errors or decides that it cannot do so.

Analysis by synthesis is a computationally expensive technique. Consequently, PROUST only deals with a small subset of Pascal and can only build models for a small number of problems. For one program it needs something like 100 plans, and it currently knows about 5 programs. PROUST is also limited by the fact that it ultimately represents the program as syntactic units. It has no semantic knowledge of Pascal and cannot perform dataflow analysis. This means that there is a large class of correct versions of any program which it will fail to recognise as correct.

Spohrer, Soloway, and Pope (Spohrer *et al*, 1985) have claimed that these proposed 'programming plans' are a major feature of programming expertise. Gilmore has shown that the content of plans observed in Pascal experts are not the same as those in other languages (BASIC in Gilmore's study, but the same would be true of Prolog) (Gilmore, in press). Thus 'programming plans' will need to be identified in Prolog in order to generalise this technique. More particularly, PROUST operates purely on pattern matching of surface syntactic features. It does no control-flow or data-flow analysis and has a heavy reliability on key-words. The system would have problems spotting recursion or dealing with backtracking in Prolog. Looi has implemented an automated analysis system for simple Prolog programs (Looi & Ross, 1987). The system, like PROUST, uses an intention-based analysis of programs, however Looi keeps a rating of the suitability of each interpretation of a program, and thus, instead of committing to one interpretation as PROUST does, may use alternatives. Ross also argues for the use of plans in a Prolog intelligent tutoring system (Ross, 1987). He argues for the use of a combination of techniques, controlled possibly via a blackboard type architecture, in order to detect correct or buggy plans.

7.1.2 Debugging using GREATERP

GREATERP (Anderson *et al*, 1986) is a tutor for LISP. Based on an empirical study of LISP learners (Anderson *et al*, 1984), the observed student's incorrect behaviour provided the basis for a model of 'bugs' — variations on an 'ideal' student's behaviour. The system attempts to model the student by a technique Anderson called "model tracing". As the student solves a problem the set of rules that they potentially could apply forms a set of predications about the student's possible behaviour. When a buggy rule is chosen the system immediately corrects the student and continues. GREATERP has been used as part of the undergraduate teaching programme at Carnegie-Mellon University for several years. The system is used in conjunction with a textbook that introduces concepts and explains about LISP (Anderson *et al*, 1987). The rôle of the tutor is to monitor the student carrying out the exercises in the book. This combination has proved to be more effective than simply providing the student with a LISP textbook. The model makes major use of Anderson's ACT*-theory of cognition (Anderson, 1983). The tutor "...assumes that productions represent the student's intentions, and that the student's subgoals represent the tutor's subgoals" (Anderson *et al*, 1986, page 843). This involves a big assumption about the psychological validity of the model of the student. In principle the approach could be generalised to Prolog. However it is an open question as to how easy (or even possible) it would be to make the model of student Prolog skill acquisition that the system needs. Without either trying to model Prolog learning after Anderson's LISP modelling (Anderson *et al*, 1984), or some empirical evidence on the relative difficulty and differences between learning to program in LISP and Prolog, it is not possible to answer this question. However, even if it were possible, the target audience for such a system is limited to novices; and even so, the strictly fixed curriculum and inherent inflexibility of the system casts some doubts over its educational desirability.

7.1.3 Debugging using the Programmer's Apprentice

The Programmers Apprentice is a project that aims to produce an intelligent assistant in order to help experts write and maintain programs (Waters, 1985). The system aims to

automate the easier parts of the debugging process whilst leaving the more complicated parts to the expert programmer. Programs are parsed into surface plans from where clichés are used to recognise the function of a particular piece of code. A cliché is some standard way of doing things, *e.g.* the typical way of searching a one dimensional structure is via a 'sequential search' cliché. The emphasis in the Programmers Apprentice, however, is on prevention of bugs. The use of plans helps the user write and structure the programs. Trivial tasks are taken care of by the program, leaving the expert free to concentrate on the more complex aspects of the program. If errors or missing parts of a cliché are spotted by the program, the programmer is informed.

The claim for the Programmers Assistant is that, because the knowledge is represented in the Plan Calculus (Rich, 1981), the approach should generalise across languages. The only modification would involve some operational differences between languages (*e.g.* differences in performing simple I/O). However it is not clear how the existing Plan Calculus could deal with aspects of Prolog behaviour *e.g.* search, backtracking, or the cut. Nondeterminism might also make the correct interpretational behaviour of a predicate harder to specify. Thus it might well prove necessary to developed a modified Plan Calculus. Looi raised two further difficulties (Looi & Ross, 1987). Firstly, different implementations of the same algorithm parse into quite different plan representations if their control and data flow differ. However the clichés should recognise that these two different plans are in fact the same (providing the clichés are smart enough, there may be a way around this problem). Secondly, the recognition of algorithms in buggy student programs would be difficult for the parser in the Programmers Assistant since discrepancies in graph matches would be explained as bugs rather than (possibly) variant, correct solutions.

It has been suggested that the notion of a programmers assistant could be incorporated into Prolog by the use of a techniques oriented editor (Brna *et al*, 1988; Bundy, 1988). The techniques, introduced earlier, could be available to the user *e.g.* via a mouse/menu selection procedure. The user decides how to solve the problem, and how to realise this solution in code. Once this has been done the next step is to choose the appropriate implementation technique and this will be written by the editor. The user can specify multiple techniques *e.g.* in the classical definition of *append/3*, the user could specify that

a recursive clause would have list deconstruction, list accumulator and result techniques as arguments. Parts of the code could still be written by hand, or hand adaptions of techniques produced.

8 Teaching Debugging Skills

One obvious way to improve debugging performance is to teach people how to become better debuggers of programs. However, how to debug a program is also very dependent on what tools are available to do the debugging. This raises two issues, firstly whether we should instead teach people how to use these tools better, and secondly, if so, which tools should we choose. The answer to the problem possibly lies in a better understanding of debugging skill. Currently there is a great deal of effort going into understanding the psychology of programming, however it might be a profitable enterprise to look closely at debugging skill as well. If we could understand how people typically went about a debugging task, we could (a) design better debuggers that more naturally supported and enhanced human performance, (b) explicitly teach debugging skill and strategies, and (c) automate these procedures in order to improve the machine's bug location abilities.

9 The Next Steps

The paper has presented an overview of a series of approaches toward better debugging environments for both novices and experts. However these approaches seem in most cases to be perfectly consistent, and thus could conceivably be coherently integrated into a large scale logic programming environment. It would seem plausible that in systems of the future it would be possible to have meta-database and techniques oriented debuggers operating with improved tracers. Alternative techniques for bug location within a standard debugger should be made available —either entirely automated or operating interactively with the user via an oracle. Thus it would seem that by developing interlinked approaches together it is possible to provide more powerful end-products for the user.

It has been speculated that the size of Prolog programs will increase maybe up to the

extent of one million predicate programs in the next few years. To debug such programs would suggest that there are two ways forward in common with what we have already discussed. Firstly, providing more efficient ways of controlling the amount of information from a trace of the program's execution; and secondly, more powerful and automated aids that assist the user in bug location.

References

Anderson, J.R. (1983). *The Architecture of Cognition.* Harvard University Press.

Anderson, J.R., Farrell, R. and Sauers, R. (1984). Learning to program in lisp. *Cognitive Science*, 8(2):87–129.

Anderson, J.R., Farrell, R. and Sauers, R. (1986). The automated tutoring of introductory computer programming. *Communications of the ACM*, 29(9):842–849.

Anderson, J.R., Corbett, A.T. and Reiser, B.J. (1987). *Essential LISP.* Addison Wesley.

Bowen, D.L., (ed.). (1981). *DECSystem-10 Prolog User's Manual.* Department of Artificial Intelligence, Edinburgh. Available as Occasional Paper No 27.

Brna, P., Bundy, A., Pain, H. and Lynch, L. (1987). 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., Smith, B. and van Someren, M. (1988). *Prolog Programming Techniques.* , Department of Artificial Intelligence, Edinburgh, A forthcoming research paper submitted to the special issue of Instructional Science on Learning Prolog: Tools and Related Issues.

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.

Byrd, L. (1980). Understanding the control flow of prolog programs. In Tarnlund, S., (ed.), *Proceedings of the Logic Programming Workshop*, pages 127–138, Available from Edinburgh as Research Paper 151.

Dershowitz, N. and Lee, Y. (1987). Deductive debugging. In Computer Society Press of the IEEE, (ed.), *Proceedings of the 1987 Logic Programming Symposium*, pages 298–306, IEEE, San Francisco.

Dewar, A.D. and Cleary, J.G. (1986). Graphical display of complex information within a Prolog debugger. *International Journal of Man Machine Studies*, 25:503–511.

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.

Eisenstadt, M. and Brayshaw, M. (1986). *The Transparent Prolog Machine TPM: An Execution Model and Graphical Debugger for Logic Programming.* Technical Report 21, Human Cognition Research Laboratory, The Open University.

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.

Eisenstadt, M. and Brayshaw, M. (1988). An Integrated Textbook, Video, and Software Environment for Novice and Expert Prolog Programmers. In Soloway, E. and Spohrer, J., (eds.), *Studying the Novice Programmer*, LEA.

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

Eisenstadt, M. (1985). Retrospective zooming: a knowledge based tracking and debugging methodology for logic programming. In Joshi, A., (ed.), *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann.

Eisenstadt, M., (ed.). (1988). *Intensive Prolog.* Associate Student Office (Course PD622), The Open University Press.

Fung, P. (1988). *A Formalisation of Novice's Prolog Errors.* CITE Report, Centre for Information Technology in Education, Institute for Educational Technology, The Open University.

Fung, P., DuBoulay, B. and Elsom-Cook, M. (1987). *An Initial Taxonomy of Novices' Misconceptions of the Prolog Interpreter.* CITE Report 27, Centre for Information Technology in Education, Institute for Educational Technology, The Open University.

Gilmore, D.J. (in press). Programming plans and programming expertise. *Quarterly Journal of Experimental Psychology.*

Johnson, W.L. and Soloway, E. (1985). PROUST: knowledge-based program understanding. *IEEE Transactions of Software Engineering*, SE-11(3):267–275.

Johnson, W.L. (1986). *Intention-Based Debugging of Novice Programming Errors.* Pitman.

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

Looi, C.K. and Ross, P. (1987). *Automatic Program Debugging for a Prolog Intelligent Teaching System.* Research Paper 307, Department of Artificial Intelligence, Edinburgh.

Looi, C.K. (1988). APROPOS2: a program analyser for a Prolog Intelligent Teaching System. In Looi, *Proceedings of the International Conference on Intelligent Tutoring Systems-88*, University of Montreal, Previously Research Paper no 377, Department of Artificial Intelligence, Edinburgh University.

Milner, R. (1980). *A Calculus of Communication Systems.* Volume 92, Springer-Verlag, Heidelberg.

Mycroft, A. and O'Keefe, R.A. (August 1984). A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, Earlier version available from Edinburgh as

Research Paper 211.

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

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.

Plummer, D. (1987). *Coda: An Extended Debugger for Prolog.* Technical Report AITR87-54, University of Texas at Austin.

Rajan, T. (1985). *APT: The Design of Animated Tracing Tools for Novice Programmers.* Technical Report 15, HCRL, Open University.

Rajan, T. (1986). *APT: A Principled Design for an Animated View of Program Execution for Novice Programmers.* Technical Report 19, Human Cognition Research Laboratory, The Open University.

Rajan, T. (1987). APT: a principled design of an animated view of program execution for novice programmers. In Bullinger, H.J. and Shakel, B., (eds.), *Human-Computer Interaction — INTERACT'87*, Elsevier Science Publishers B.V. (North Holland).

Rich, C. (June 1981). *Inspection Methods in Programming.* Artificial Intelligence Laboratory Technical Report AI-TR-604, MIT Artificial Intelligence Laboratory.

Ross, P. (1987). Some thoughts on the design of an intelligent teaching system for Prolog. *AISB Quarterly*, Summer(62).

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.

Stevens, A., Collins, A. and Goldin, S.E. (1982). Misconceptions in students' understanding. In Sleeman, D.H. and Brown, J.S., (eds.), *Intelligent Tutoring Systems*, pages 13–49, Academic Press, London.

Taylor, J. (1987). *Programming in Prolog: An In-Depth Study of Problems for Beginners Learning to Program in Prolog.* Unpublished Ph.D. thesis, School of Cognitive Studies, University of Sussex.

van Emden, M.H. (1984). An interpreting algorithm for Prolog programs. In Campbell, J., (ed.), *Implementations of Prolog*, Ellis Horwood, Chichester.

Waters, R.C. (November 1985). The Programmer's Apprentice: a session with KBEmacs. *Transactions on Software Engineering*, SE-11(11).