Declarative Debugging with The Transparent Prolog Machine (TPM)

Mike Brayshaw and Marc Eisenstadt

Human Cognition Research Laboratory
The Open University
Milton Keynes MK7 6AA, UK

ECAI-88 Paper Submission, February, 1988

Abstract: The underlying philosophy of the Transparent Prolog Machine (TPM) is that it should serve as a medium for visualising the execution of Prolog programs in a manner which is rigorously faithful to the internal behaviour of the Prolog interpreter. It is therefore highly biased towards the procedural account of Prolog execution. Although this is useful for certain types of practical program debugging, a true logic programming paradigm demands a more declarative account. Since TPM already incorporates special displays to show unification details and goal outcomes, a straightforward extension enables the user to intervene to say whether a particular unification (or indeed any particular goal outcome) was desirable ('thumbs up') or undesirable ('thumbs down'). This intervention can take place either during 'live' execution, or else retrospectively, after execution is complete. This technique can be combined with a selective highlighting facility which takes the user directly to a particular context-sensitive point in the execution history, and with TPM's 'replay' facility which allows execution to be played forward or backward at different speeds and different grain-sizes of analysis. The combination of techniques enables the user to mix a truly declarative debugging style for clean logic programs with a procedural style for those cases which demand it. The paper presents worked examples of this new mixed paradigm.

Section: Logic Programming

Length: 5000 Words

1 Introduction

The Transparent Prolog Machine (TPM) was introduced in (Eisenstadt and Brayshaw 1986; 1987) as a method of graphically displaying the execution of Prolog programs. The original conception of TPM emphasised fidelity to the underlying workings of the Prolog interpreter. While this provides many convenient conceptual 'hooks' for both novice learners and expert debuggers, it weds TPM overwhelmingly to a procedural account of Prolog execution.

Ideally, logic programs can be looked at declaratively, and divorced from the underlying execution machinery. Our aim in this paper is to show how straightforward extensions to TPM enable it to handle both declarative and procedural debugging styles. The key insights which drive this work are (a) that Prolog programming in practice involves the integration of both declarative and procedural techniques, and (b) user intervention in the form of 'thumbs up' or 'thumbs down' indications at critical points in the TPM animated display can provide vital 'anchor points' from which a declarative analysis of program behaviour can be performed.

Our work draws from the existing work in declarative debugging (Shapiro, 1982; Lloyd, 1986). The use of terms instead of literals as the basis for driving the bug location and the location algorithm are taken from rational debugging (Pereira,1986). Where we differ is in the graphical nature of the interaction and the intergration of such a debugging methodology into a large debugging system. The use of a stored trace history is also used by (Huntback, 1987; Sterling and Shapiro, 1986). In this paper we shall use the term declarative debugging to include both rational and algorithmic approaches.

The next section presents a brief overview of the Transparent Prolog Machine. Subsequent sections introduce the concept of declarative user intervention, and a series of worked examples. We conclude with a summary of the important principles introduced by our 'mixed paradigm' debugging approach.

2 Summary of TPM

2.1 AORTA diagrams

An ordinary node in a traditional and/or tree can be enriched to become a full-fledged 'status box' which concisely reveals the execution history of individual clauses. This simple augmentation, here dubbed the 'AORTA' ('**A**nd/**OR** **T**ree, **A**ugmented') diagram, is the focal point of our graphical debugger. TPM allows both a long-distance view of execution (displaying several thousand nodes and highlighting 'points of interest' at the user's request) and a close-up view, using all of the detailed notation of AORTA diagrams.

To illustrate the close up view consider the following program taken form (Bundy et al. 1986a)..

```
location(Person, Place) :-
        at(Person, Place).
location(Person, Place) :-
        visit(Person, Other),
        location(Other, Place).

at(alan, room19).
at(jane, room54).
at(betty, office).

visit(dave, alan).
visit(jan, betty).
visit(lincoln, dave).
```

If I pose the query **?- location(lincoln,Where)**, the above program succeeds with the instantiation **Where = room19**. Figure 1 shows the AORTA diagram corresponding to the final snapshot of execution.

INSERT FIGURE 1 HERE

Figure 1 AORTA 'snapshot' after processing the query **?- location(lincoln,Where).** The subscript counter is incremented for each new nonground clause.

The large rectangular boxes in figure 1 are called procedure status boxes. The top half of such boxes shows the status of the goal at the time of viewing. A question mark indicates a pending goal; a tick ('check') indicates a successful goal; a cross indicates a failed goal; a tick/cross combination indicates an initial success followed by subsequent failure on backtracking. The lower half of the procedure status box indicates the number of the latest matching clause head. Thus, in the case of the topmost goal **location** the tick in the top half of the box indicates that the goal was successful, and the number 2 below it tells us that was the second clause which succeeded. The small vertical lines dangling beneath each procedure status box are known as 'clause branches', and the square boxes at the end of such lines are 'clause status boxes'. Such boxes use the same question-mark, tick, cross, and tick/cross combination to depict the status of individual clauses. If a given clause head does not unify, then a short horizontal 'dead-end' bar is added instead of a clause status box (examples may be seen under the procedure status boxes for **at** in figure 1). Clause branches correspond to 'or' choices, but are drawn differently from their

traditional counterparts in order to make the processing of individual clauses obvious at a glance. Circular nodes are used to depict system primitives.

To illustrate unification, the relations and arguments next to the top half of each procedure status box depict the state of play when the goal was invoked, whereas the relations and arguments next to the bottom half of each procedure status box depict the matching clause head found in the data base. User-chosen variable names are subscripted automatically to indicate renamed variables. The diagrams use a sideways '=' with arrowheads to show unification. Up arrows indicate output variables; down arrows indicate input variables. Right-angled arrows indicate a variable 'passed across' or shared with a sister goal. Headless arrows indicate directly-matching terms. Often there is a direct visual correspondence between a variable and the arrow showing its instantiation in the diagram. Whenever the correspondence is 'indirect', we place a small lozenge beneath the variable to show its instantiation at the moment of the AORTA snapshot.

2.2 The Long Distance View

The long distance view (LDV) is designed to allow the user to analyse the global behaviour of very large programs. The long distance view (LDV) uses a schematised AND/OR tree in which individual nodes summarise the outcome of a call to a particular procedure. Each node is actually a collapsed procedure status box, showing just the top half of the status box as introduced in above. This collapsed box allows us the luxury of a global presentation without sacrificing the important summary information provided by the AORTA diagrams. In our black and white display we use white to indicate a sucessful node, black to indicate failure, grey shading to indicate success followed by failure on backtracking, and thickenned lines to indicate 'currently pending goal'. Another convention we introduce in the LDV is the 'compressed' node. This appears as a triangle (evocative of a sub-tree shape), and indicates that a given Prolog goal is being treated for the moment as a 'black box' primitive, i.e. a goal whose inner details are not shown at the moment, but which can be expanded later upon request. Figure 2 shows a snapshot taken in the middle of the execution of a moderate size Prolog program. Users of our graphical tracer/debugger implementation have the option of visualising program execution 'retrospectively', i.e. following a behind-the-scenes analysis, or 'live', i.e. interactively, or in a mixed combination of the two styles. In the case of figure 2, the program is being executed 'retrospectively'. This means that it is possible to say 'in advance' (from the user's point of view) precisely which nodes will be traversed. The thin horizontal line indicates a user-defined predicate in the so-called 'pre-ordained execution space', i.e. a call which has not yet happened at the moment the snapshot is taken, but which the program analyser can guarantee will eventually happen. Analogously, a small dot indicates a system primitive which has not yet been executed, but which will eventually be executed.

INSERT FIGURE 2 HERE

Figure 2. A Long Distance View diagram, in the middle of (re-)execution 'replay'..

A detailed account of the notation, and the way it can be used to show complex unification history (including multiple invocations during backtracking) and extra-logical features such as the cut, is presented in Eisenstadt and Brayshaw (1988a).

3. Declarative user intervention: 'Thumbs up'/'Thumbs down'

Declarative debugging allows the user to debug a program without having to know anything about its detailed run-time execution. For declaratively written Prolog programs it is highly desirable to be able to debug them declaratively. Even for programs not

originally conceived declaratively it is still a very powerful debugging technique. For these reasons we wish to incorporate the technique into TPM.

The declarative debugging system is available as an option within TPM . It may be invoked both from within a program that is still being executed 'live' or run on a post-mortem trace of the program. If no guidance is given as to which goal to debug the system will choose the top-level goal of the current query. The user can change this simply by dragging the debugging icon to a particular node, or by selecting a different node in the first place when invoking the declarative debugging system. A third possibility is to use the selective highlight facility of TPM which allows the user very accurately to specify particular goal/argument combinations, and with the aid of this quickly pin-point the start point for the debugging session. The advantage that this flexibility gives us is that the user can use any relevant knowledge about a big program to select candidate goals in the system to debug.

The system uses the intuitive notion of a 'thumbs-up' sign to indicate whether a goal and outcome is correct or not. Further thumbs-up/thumbs-down icons indicate whether a goal is admissable or solvable. The icons are simply interchanged by mouse click. This process is equivalent to the oracle of other declarative debuggers. Wrong solution mode is equivalent to the 'thumbs-down' of a sucessful goal. Incorrect solutions are 'thumbs-down' to an unsuccessful goal. Non-termination isn't addressed here. In a post-mortem trace analysis we have no improvement on depth-bound stack monitoring (Shapiro, 1982) to suggest. However TPM's live mode already assists us in the spotting of non-termination by graphically making the infinite regress apparent, as it happens.

It has been suggested that the number of queries that the oracle makes may be limited by keeping a history of user's oracle responses (e.g. Shapiro, 1982; Lloyd, 1986). However this presupposes that a predicate is used in the same manner in a program wherever it is invoked, and that the program contains no side-effects. For this reason oracle queries are not stored. A second suggestion has been to have another version of the program stored elsewhere which can be used as the source for answering the oracle. Whilst this might be possible for certain scenarios, e.g. fixed curriculum teaching of novices (e.g. Looi, 1987; Anderson et. al., 1985), in the real world, and particular when we are dealing with experts writing novel applications, we believe it is not possible to have a correct version of the program lying around in order to mechanise the oracle.

The debugger searches the tree to find either an 'incorrect' node whose subgoals are all correct (i.e. a wrong solution), or a goal that is the deepest failing subgoal of some other failing goal (i.e. an incorrect solution). Additionally, the program can carry out its search on goals that are still pending. Here, the 'thumbs-up/thumbs-down' symbols refers to whether the goal is admissable, solvable, and whether the goal, as so far executed is ok. The user may additionally 'thumbs-down' click on any lozenge, variable, clause box or data-flow arrow.

A typical sequence for debugging is as follows. The overall execution space is shown using the long distance view (LDV). A 'queried' node is shown in a special 'zoom' viewport in its full (AORTA) status box representation. Beside the status box for any queried node are two things. Firstly, the status box and the corresponding node in the LDV are labelled in order to emphasise the context of the query. Secondly, immediately to the left of the status box is a large thumbs-up icon. This indicates that the outcome of the goal is assumed for the moment to be correct. Beneath this main icon are two 'inferior' thumbs-up/thumbs-down icons, indicating whether the goal is either admissable or solvable. If any 'thumbs-up' assumption is erroneous, the user can simply toggle it to 'thumbs-down' with a mouse-click. After any of these icons have been given the thumbs-down the user then can point to any variable, clause status box, data-flow arrrow or part of a lozenge that is in error, as outlined below.

- *lozenges* Giving a lozenge a 'thumbs-down' is equivalent to indicating an incorrect term. As in (Pereira, 1986) the debugger now uses knowledge of the

dependencies of that term and the operational semantics of Prolog to find another candidate node. If a particular term in a lozenge is chosen, then we use Pereira's algorithm to follow up the term's depemdancies in choosing which node to query next.

- *variables* The user can see variables both in the calling clause and the matching clause head, independent of any instantiations that they might have. By pointing out a variable as incorrect, the user can specify a lexical error either in the goal, or in the matching clause head.
- *clause status box* If a wrong clause outcome is detected, a thumbs down may be given to a particular clause or clauses. For example, a goal may produce the wrong solution, because there is a missing solution to a previous clause. The user can thus immediately point to the failure of an earlier clause as being the source of the error. This unintended failure may be investigated, and no time is wasted debugging a potentially spurious incorrect solution. A missing solution may also result form a earlier clause succeeding unexpectedly, or as a result of a 'cut'. In the case of the 'cut', if this has caused failure of a subgoal and as a result a subsequent clause branch was not explored, then the debugger will query all those subgoals prior to the cut. If one of them succeeds when not expected, the search continues until the deepest incorrectly succeeding goal is found, in which case this item is taken as the bug. If no such goals are at fault, then the 'cut' itself is located as the source of the bug. To indicate that an untried branch was on the intended execution path, the user can indicate 'thumbs-down' on that clause's untried clause branch.
- *data-flow arrows* The user may 'thumbs-down' on a dataflow arrow to indicate that the data-flow itself is incorrect i.e. an input variable should in fact be an output variable, or conversely, an output variable should in fact be an input. This is differnent from indicating that a term is incorrect. Consider an output variable that should have been input. The actual binding that this variable has got may be a plausible value; however this value should have been input, not produced unintentionally as a result of a unification of an incorrectly unbound variable in a subgoal (posssibly in this scenario a test for the intended binding). The bug actually lies in a previous subgoal that failed to produce a binding for this variable. Hence the distinction is this: if the dataflow is incorrect then this should receive the 'thumbs-down', if the dataflow is correct, but the terms that are bound to the variables is incorrect, then the particular variable lozenges should be given the 'thumbs-down'.

The user can thus point to a series of features in the AORTA diagram for a given node and highlight those features that are incorrect. This means that the user can potentially spot more than one bug in a goal. In this case the first error that the user locates is the one that is immediately investigated. However the other bugs are remembered and pushed onto a stack. When a bug is sucessfully located, the user can choose to pop this stack and follow up another error which was spotted 'en route'. These errors may of course be due to the same single bug. If it is the case that the dependencies of an indicated error can be traced back to the same newly located bug, the user is prompted to see if the two errors have the same source and/or are the same. Otherwise the search for the source goal will start from the point at which this new bug was first spotted by the user. The stack of uninvestigated errors grows as and when the user locates multiple bugs. Errors may be popped and discarded if the user wishes no longer to consider them. Alternatively, it is possible that experts may spot a bug before that actual goal in error is located, just because of their knowledge of the program they have written. Instead of carrying on using the debugger they can abort the current bug search at any point. If there are other bugs on the current stack, they can be followed up at this point.

If the user does not indicate any of the lozenges, data-flow arrows, variables, or clause status boxes as in error, and the user has indicated no earlier item to guide the search, then the tracer will follow a top down search similar to that advocated by (Lloyd, 1986). The user can thus merely 'thumbs-up/thumbs-down' the entire status box and the tracer will, albeit more inefficiently, carry on trying to locate the clause in error.

As we have outlined above the declarative debugger will search through the execution tree. However, additionally the user can directly intervene in this process, by explicity moving to some node and carrying on debugging from there. This user-driven direct intervention in the debugger's search results in the previous declarative debugging session being terminated and a new one started at the new node. It clearly must be used carefully, otherwise if randomly employed, the debugger might never converge on a buggy goal! This ability however allows the user to use the other facilities in TPM to explore the trace information and then invoke the declarative debugger at some potentially interesting point. The use of facilities like TPM's 'replay' allows the user to choose at which stage in the history of a program execution they want declaratively to investigate a certain goal. For example, consider programs that change their state by performing side-effects and constantly reinvoking a given goal. By using the replay and/or selective highlighting, we can easily investigate whichever invocation of that goal we wish to inspect more closely and then invoke the declarative debugger on that node. This is a clear example where it is very useful to be able readily to switch between procedural and declarative accounts when debugging.

When a bug is located, the debugger will attempt to say what the bug is. This uses a cliche analysis based upon (Eisenstadt, 1985), but expanded to incorporate the additional symptomatic information as a result of the oracle queries. This currently includes no definition, wrong arity, failure to unify, various failures due to cut (see above), and errors in variable bindings and dataflow, in addition to being able to localise the bug down to a particular predicate and clause.

We shall now consider a concrete example in order to highlight and exemplify the current system.

4. Worked examples

Let us consider the following program. The predicate **explore**[1] is a simple rule-engine, but enhanced to produce both a proof of its derivation and a truth value, **true** or **false**, for that proof.

```
:-  op(850,fx,not).
:-  op(900,xfx,:).
:-  op(870,fx,if).
:-  op(880,xfx,then).
:-  op(800,xfx,was).
:-  op(600,xfx,from).
:-  op(540,xfy,and).
:-  op(550,xfy,or).
:-  op(300,fx,'derived  by').
:-   op(100,xfx,[gives,eats,has,isa]).

explore(Goal,Goal is true was 'found as a fact',true):-
        fact : Goal.  %positive fact the truth value 'true'
explore(Goal,Goal is false was 'found as a fact',false):-
        fact : (not Goal).  % negative fact - truth value 'false'
```

[1] This code is based on that developed by Frank Kriwaczek of Imperial College, for the Open University's 'Intensive Prolog' Course (Eisenstadt,1988), based upon a 'rational reconstruction' of an expert system shell described in Bratko (1986). The bugs have been added by the present authors recalling actual mistakes observed in students.

```
explore(Goal,Goal is TVal was 'derived by' Rule from
          Proof,TVal):-
      Rule:if Cond then Goal,  %Truth of a rule takes the value
      explore(Cond,Proof,TVal).  %of the proof if its conditions
explore(Goal and Goals,Proof and Proofs,true):-
      explore(Goal,Proof,true),  % for a conjunct the proof the
      explore(Goals,Proofs,true).  % conjuction of indiv. proofs
explore(Goal and Goals,Proof,false):-
      explore(Goal,Proof,false),
      explore(Goal,Proof,false).
explore(Goal or Goals,Proof,true):-
      explore(Goal,Proof,true);    % for a disjunct the proof
      explore(Goals,Proof,true).  %that of one of the disjuncts
explore(Goal or Goals,Proof and Proofs,false):-
      explore(Goal,Proof,false);
      explore(Goals,Proofs,false).
```

To this let us add the following simple database of facts and rules:

```
fact: buttercup gives milk.
fact: (not buttercup eats meat).
fact: (not buttercup has hair).

m_rule:  if
                    A has hair
                    or
                    A gives milk
         then
                    A isa mammal.

c_rule:  if
                    A isa mammal
                    and
                    A eats meat
         then
                    A isa carnivore.
```

The program contains a couple of bugs. We shall now work through a session with the debugger to show how these bugs can be located. First observe the following top-level interaction:

```
  ?-explore(buttercup isa carnivore, How, TruthValue).
How = buttercup isa carnivore is false was 'derived by' c_rule from
          buttercup isa mammal is false was 'derived by' m_rule from
              buttercup has hair is false was 'found as a fact' and Proofs6.
TruthValue = false
```

The above output differs from what we expected to see, however. If we now invoke the debugger, the system will show us the LDV of the execution tree, as shown in figure 3, and will default to querying the top goal, as shown in figure 4.

INSERT FIGURES 3 AND 4 HERE

The large ***A*** in the corner shows that this queried node corresponds to the node ***A*** in the LDV above. Next to it is the 'thumbs-up/thumbs-down' icon. It defaults initially to the 'thumbs-up' assumption, but in this instance we can see that this goal is not correct. Although **explore** has succeeded it has produced an incorrect proof of the goal. A mouse click on the 'thumbs-up' icon turns it into a 'thumbs-down' icon, at the moment indicated by the snapshot depicted in figure 4. We need not touch the other two 'thumb icons' since they correctly show that the goal is admissable and solvable. We can now go and point to any parts of the goal, variable bindings, clause boxes, or data-flow arrows which are in

error. As we have already noted, the proof is incorrect, so we mouse click on those parts of the lozenge that are in error. The debugger then looks where this binding was derived from and finds the goal marked ***B*** in the LDV and queries the user as shown in figure 5.

INSERT FIGURE 5 HERE

Figure 5 shows the goal that constructs those parts of the proof that we indicated as being in error, so therefore we have indicated 'thumbs-down' for the goal (as shown) and now we can do the same more specifically for the lozenge as well. The debugger will now bring up the AORTA shown in figure 6 and marked ***C*** , which is correct, so we have indicated 'thumbs-up' for this clause.

INSERT FIGURE 6 HERE

The debugger now correclty identifies that clause 7 of **explore** is incorrect, since we have a case where a goal has correct subgoals, but it itself is not correct. Upon inspection we can see that most probably the user has a confusion about truth tables. To prove the goal **Goal** *or* **Goals** is false you are required to prove both goal **Goal** *and* goal **Goals** are false, not that goal **Goal** *or* goal **Goals** is false. The corrected version looks like the following.

```
explore(Goal or Goals,Proof and Proofs,false):-
        explore(Goal,Proof,false),
        explore(Goal,Proof,false).
```

If we re-run the original query with the modified code however, the program now fails to find a solution. Let us invoke the debugger a second time. This time the LDV is shown in figure 7. The letters next to certain nodes are there to show the correspondence between those nodes and the respective debugger queries.

INSERT FIGURE 7 HERE

The debugger will now try and identify the most deeply incorrectly failing node. It follows (Pereira, 1986) in searching through failed goals and attempts to see if they are admissable or solvable. As we can see from figure 8, after we have identified the top-goal as incorrect as in the previous example, we then search for the next relevant goal to query, that is ***B***. Here this is the first goal with a onetime solving subgoal.. Notice that the intervening goals between the top-goal ***A*** and the candidate ***B*** have ignored as far as the oracle is conserned.

INSERT FIGURE 8 HERE

This goal should have been solvable, so we have given the outcome a 'thumbs down'. We will now investigate the children of this goal. We next choose the failing subgoal, ***C*** and query the user about it, as shown in figure 9.

INSERT FIGURE 9 HERE

The goal labelled ***C*** is inadmissable, so we have updated the icons accordingly. The second argument to **explore** must always be input, so we can now click on the data-flow arrow pointing downwards above **Proof** to indicate that this is incorrect. The debugger will now query goal ***D*** This goal fails on backtracking as we can see in figure 10, however the goal is admissable and solvable and should be true. Therefore we have given the outcome a 'thumbs down'.

INSERT FIGURE 10 HERE

When we have a node like this an additional menu comes up, containing a skeleton status box as shown below. By clicking on any of the clause branches the user is able to toggle through the possible outcomes, i.e. succeed, fail, not unify, suceed but fail on backtracking, or not be attempted, for each of the clauses. In this instance the skeleton status box is shown in figure 11.

INSERT FIGURE 11 HERE

Clause one should have suceeded, whereas clause two should not have been attempted, so we have indicated this. This can now be checked against what actually happened. If this outcome never arose, then we have an incorrect solution and debugging must continue to find the cause. If this scenario did occur, as in this case, then the predicate has behaved correctly Note this is a good example of when it might be useful to use the replay or zoom facility in order to help answer the oracle. If the user did not know the answer to a query or wasn't sure, he or she could easily check it out.

At this point we are able to locate the problem to clause 5 of **explore** , since we have now found a clause which fails, but the oucome of its subgoals is OK. The debugger concludes that clause 5 is in error and either the scoping of the **Proof** variable is incorrect, or else the two **explore** goals must be disjuncts instead of conjuncts in order to produce the incorrect dataflow. The bug is in fact the latter of these two possibilities. If we study the clause in question, you will notice that again the truth values are incorrect. The conjunct of goals **Goal** *and* **Goals** is false if *either* **Goal** or **Goals** is false, not if *both* of them are false, hence the corrected clause should be as follows.

```
explore(Goal and Goals,Proof,false):-
        explore(Goal,Proof,false);
        explore(Goal,Proof,false).
```

An important thing to notice and something that is central to the approach outlined here is that when answering questions the user has the full power of the trace package available at anytime. Hence the context of any oracle query is always immediately available. Further, in order to answer an oracle query, the user can use any of the other facilities of the tracer, e.g. zoom, replay, highlight etc. in order to better understand the question and more easily answer it correctly. The tight integration of the declarative paradigm into the spirit of the rest of the tracing system makes it possible for the user to switch between declarative and procedural views of the program whenever necessary. We believe this may prove to be an essential attribute for any debugging system if it is going to scale up and deal with bugs in truly large logic programs.

5. Conclusions: mixed paradigm debugging

.

We have outlined how the benefits of declarative debugging may be smoothly incorporated into TPM. The integration affords a series of advantages, namely that:

- Declaratively written programs may be debugged declaratively, but the debugger can still capitalise an the clarity of the graphical execution model thus improving the oracle mechanism.
- The 'thumbs-up/thumbs-down' icon provides a powerful yet intuitive method for answering oracle queries.
- Highlighting either lozenges, clause-boxes, variables, or data-flow arrows allows the debugger to choose the most relevant action to take next, dependent upon the specific response.
- Non-declarativley written programs may still be debugged declaratively.

- Multiple bugs may be handled.
- The integration of declarative debugging into TPM allows declarative debugging to be used easily on potentially very large programs.
- The declarative debugging paradigm may be incorporated into a larger tracing and debugging system allowing the user to change at will between the two styles.

Prolog, as the language currently stands, contains a number of features including clause ordering, some of the dirtier uses of the cut, or uses of **assert** and **retract** which are not declarative in nature. Further, many experts prefer, and are very skilled at using procedural tracers in order to locate bugs. However it is undoubtly the case that the language also supports a declarative style of programming. We beleive that it is essential in a usable logic programming environment to support both of these debugging styles and that the very integration described herein makes the notion of mixed paradigm debugging a reality. By supporting the two styles we attempt to provide the user with the best of both worlds, and hope that the uniting of the two approaches leaves the user better equipped to deal with the very bugs that we require a powerful trace package to help us find.

6. References

Anderson, J.R., and Reiser, J.B. The LISP Tutor BYTE:The Small Systems Journal, Vol. 10, No. 4, pp 159-175, April 1985.

Bratko, I. *Prolog programming for artificial Intelligence*. London:Addison-Wesley, 1986.

Eisenstadt, M., Retrospective Zooming: a knowledge based tracing and debugging methodology for logic programming. *Proceedings of the Ninth International Conference on Artificial Intelligence (IJCAI-85)*. Los Angeles: Morgan Kaufmann, 1985.

Eisenstadt, M. (Ed.) *Intensive Prolog.* Associate Student Office (Course PD622), Milton Keynes, U.K.: Open University Press, 1988.

Eisenstadt, M. & Brayshaw, M. The Transparent Prolog Machine (TPM):An execution model and graphical debugger for logic programming, Technical Report 21, Human Cogniton Research Laboratory, November 1986.

Eisenstadt, M., and Brayshaw, M. Graphical debugging with the Transparent Prolog Machine (TPM). *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87).* Los Angeles: Morgan Kaufmann, 1987.

Eisenstadt, M., and Brayshaw, M. The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 1988a (in press).

Huntback, M.M, Algorithmic Parlog Debugging, Department of Computing, Imperial College, January 1987.

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

Looi, C.K., Heuristic Code Matching of Prolog Programs in a Prolog Intelligent Teaching System, Department of AI. University of Edinburgh, July 1987.

Pereira, L.M. Rational Debugging in Logic Programming, *Third Internatinal Conference on Logic Programming*,, Springer-Verlag, pp 203-210, 1986.

Shapiro, E.Y. *Algorithmic Program Debugging* , MIT Press, 1982.

Sterling L., and Shapiro, E.Y. *The Art of Prolog*,, MIT Press, 1986.