Debugging User Conceptions of Interpretation Processes

M. J. Coombs, R. T. Hartley* and J. G. Stell*

Computing Research Laboratory, New Mexico State University
Department of Computer Science, Manchester University, U.K.

ABSTRACT

The use of high level declarative languages ha’s been advocated since they allow problems to be expressed in terms of their domain facts, leaving details of execution to the language interpreter. While this is a significant advantage, it is frequently difficult to learn the procedural constraints imposed by the interpreter. Thus, declarative failures may arise from misunderstanding the implicit procedural content of a program. This paper argues for a constructive approach to identifying poor understanding of procedural interpretation, and presents a prototype diagnostic system for Prolog.

1. Procedural Interference with Problem Specification

Specification ("declarative") languages have arisen out of different considerations from conventional "procedural" languages. Advocates of the former are concerned with the structure of problems, whereas those supporting the latter are concerned with the structure of solutions (Kowalski, 1979). Thus, declarative languages require constructs for problem decomposition and procedural ones for decomposing solutions.

While the declarative approach to programming is attractive, there are difficulties in keeping the concept -pure. First, although applications which fit neatly into the control structure underlying a particular language will be easy to express, It will not be easy to write programs of any complexity outside its scope. To do this, a clear understanding of the constraints Imposed by the inference Wine on the expression of domain facts and rules will be required (see Sauers, 1985 concerning the effects of production system control schemes on knowledge representation). Secondly, where ambiguity exists In the user’s mind over the procedural semantics of the language, It will be difficult for him (or an automated debugging aid) to separate errors In specification from procedural. errors This will reduce the value of the language as a tool for programming with domain knowledge, In which all errors should be explained in terms of an omitted case or an incorrect representation.

The logic language Prolog, for example, solves problems by repeatedly decomposing them into simpler ones which must be finally solved by matching facts in a database. Decomposition is achieved by matching problem statements to rule conclusions, the conditions necessary for those conclusions representing sub-problems. If some sub-problem falls, Prolog backtracks to the last successful solution and seeks some alternative. Although this backtracking is conceptually simple, its effects can be complex and therefore difficult to anticipate.

However, a Prolog program may produce an unexpected result by successfully applying the first of two related rules on backtracking, rather than moving on after failure to apply the second rule. To understand this behaviour, the user must know that all possible instantiations of a given rule are tried before going on to an alternative. Lacking such knowledge, it is unlikely that the user will be able to induce the principle from the program text by symbolic execution because the poor procedural syntax provides few landmarks to help him mentally vest hypotheses concerning backtracking. He may thus try to correct the program by rewriting the rules, so changing the specification, when all that is required is to re-order them to allow the desired rule to be taken first.

Since the text of a declarative language has to carry both specification and procedural functions for the user, errors In programs when viewed as specifications are difficult, to Iodate due to confusion with procedural errors. This difficulty is not found In conventional languages where a separate specification can- exist Independent of the program. This paper thus advocates a programming aid to help Iodate procedural difficulties with declarative languages. (particularly those arising from misconceptions), allowing specification errors to remain for independent treatment.

2. **A Consultant for Debugging User Conceptions of Prolog Processes**

The authors are developing a prototype consultancy, system for debugging user conceptions of Prolog interpretation processes (Coombs and Alty, 1984). Prolog was selected because its predicate logic foundation promised to make it especially suitable for specification programming, yet considerable skill is actually required to master the procedural constraints imposed by the interpreter. Moreover, contrary to our expectations, experts continue to make similar errors to novices, although not with the same frequency.

Many of the major problems of understanding Prolog execution are related to backtracking. Even a simple program of two or three clauses may backtrack in complex ways which, if represented in full, would occupy many pages of trace. Such behaviour may be difficult to predict without a detailed mental model of processing. However, mental execution is difficult to perform accurately, given the lack of syntactic markers in Prolog text to serve as signposts and the need to relate information widely distributed in the sequence of execution events (Green et al., 1981).

During learning, users develop a variety of different conceptions concerning Prolog execution. These conceptions have a procedural component and a memory component. The litter forms a data structure, representing the current state of a problem solution, upon which the procedural component is formulated. In conventional textbook descriptions, the memory, component tends to be an OR or an AND/OR tree of goals. We have found, however, that novice Prolog programmers do not use these simple tree structures (Coombs, 1985; McAllester, 1985). Instead, novices build their procedural models based upon classification of program entities (e.g. goals and clauses) as succeeded, failed and under evaluation. This formally amounts to generating a sequence of mental AND trees, with subtrees added and deleted with satisfaction and failure of goals. This approach results in a different class of procedural problems from those following from textbook descriptions. The two types of misconception described below - "try-once-&-pass" and "redo-body-from-left" - have the same underlying model, that of left-to-right execution of conjunctive goals corresponding to the sequential matching of nodes (goals) at a given level of an AND tree.

"Try-once-&-pass" and "redo-body-from-left", which we will use as examples in the rest of the paper, are the two most common backtracking misconceptions. With the former, a rule is (incorrectly) failed completely after the failure of a single instantiation; with the latter, backtracking into a rule is (incorrectly) seen as proceeding left-to-right, taking the first subgoal to succeed rather than the last. These contrast with the correct procedure, in which all possible instantiations of a rule are tried before progressing to an alternative clause, with the rule subgoals being retried from right-to-left.

For example, with the failure of the goal "c(1)" in figure 1, given the success of "h(1,2" and "i(2)", "try-once-&-pass" would cause the second "b" rule to be tried, and "redo-body-from-left" would cause "d(X)" to be retried in the first "b" clause (followed by "e(2)"). Under correct execution, however, "d" and "e" would be retried from "e(i)"; only after all matches had been made would the second clause be taken.

```
1. a(X):- b(X),c(X).
2. b(X):- d(X),e(X).
3. b(X):- f(X).
4. d(1).
5. d(2).
6. e(1).
7. e(2).
8. f(3).
9. c(3).
```

Figure 1. A simple Prolog program.

The consultant system is designed to be employed when the user has doubts about his view of program execution. The system's task is to explain such user misconceptions by building a modified interpreter (a "inalinterpreter") which reproduces the user's account of execution. This mal-interpreter is constructed by replacing the correct backtracking procedures with incorrect ones, using evidence obtained from comparing the system's account of execution with that of the user.

A typical interaction with the system would be as follows. The user runs his faulted program on the correct interpreter, which generates an internal system trace. Using the same "trace language", the user describes a symbolic execution of the program. The system then seeks discrepancies between the user's trace and the system trace and, requesting the user to give more detailed accounts where necessary, arrives at the identity of misapplied or misunderstood interpreter concept.

3. **The Symbolic Execution Language**

The model is described in a conceptual representation language adapted from the conceptual graphs of Sowa (Sowa, 1984; Hartley, 1985). This is summarised in figure 2 as a procedural net (Sacerdoti 1977). However, in the complete representational scheme, the edges in the network become actors and the nodes become subtypes of the type GOAL. Each actor is triggered independently when all of its inputs concepts are instantiated. The concept graph provides a framework for organizing erroneous interpreter elements (figures 3 and 4)

("mal-rules") and for creating statements for the trace language.

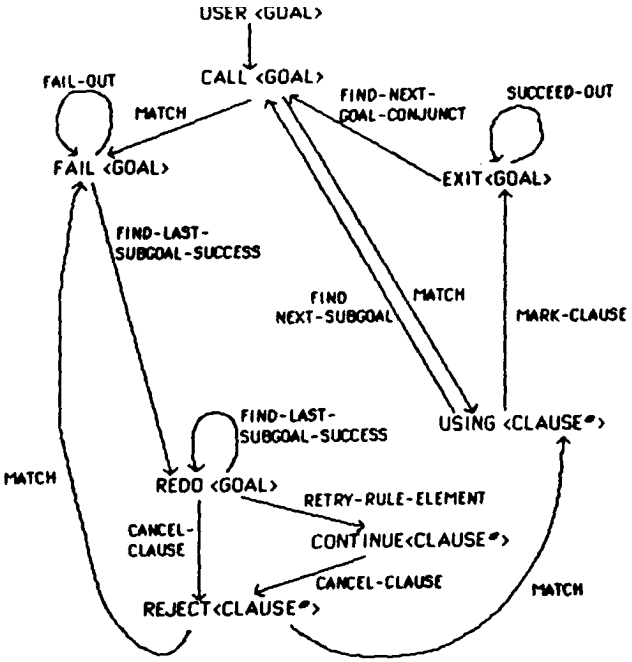

Figure 2. The correct interpreter.

The trace is based upon the familiar box model, with states CALL, EXIT, FAIL and REDO. The model describes a range of operations, the most central of which are the "matching" of goals to clauses and creation of new goals both forwards ("goal-gen") and on backtracking ("back-goal-gen").

4. Bug Diagnosis as Alternative Interpreter Generation

The diagnostic procedure is based on the assumption that discrepancies between the system trace and user trace am generated by discrepancies between the correct interpreter and a hypothetical interpreter. The task of the system is to identify these second discrepancies, which may be seen as user misconceptions concerning Prolog procedural semantics.

These hypothetical interpreters and the correct Prolog interpreter are seen as instantiations of a generic interpreter, organized hierarchically into modules. The generic slots in the interpreters are instantiated by particular procedures for Prolog semantic concepts. A Prolog semantic concept is, for example, "backtracking" or "unification"; these concepts are defined in different ways in actual interpreter modules. In order to simplify our task, we have only fully specified the backtracking module.

```
**********************************************************
NAME: backtrack(ProofTree A

  PARAM: <<clean(X,Y)>
         <back_goal_gen(ProofTree A>>

  BODY: backtrack(ProofTree,X) :-
              back-goal-gen(ProofTree,Y),
              clean(Y,X),
              trace_rep(redo(X)).

**********************************************************
```

Figure 3. The backtracking module.

```
**********************************************************
proof trees are of the form:
          [Goal,Clause_no,List_of_Subgoals]
  e.g. for a in        1. a:-b,c.
                       2. b.
                       3. c:-d.
                       4. d.

       tree is [a,1,[ [b,2,[]] , [c,3,[ [d,4,[]] ]] ] ]
```

```
/* correct rule */

back_goal-gen([Goal,_,Sub],BkGoal) :-
          last(Sub,L),
           back_goal_gen(L,BkGoal),
back_goal_gen([Goal,N,[]], Goal) .
```

```
/* mal-rule  1 - redo-body-from-left
back_goal_gen((Goal,_,Sub],BkGoal)
       Sub == [L,_,[ [BkGoal 1, X,HS ]|T] ] ,
       last([[BkGoal,X,HS] | T], [LG,_ I[] ]).
      tree_unmark(T).
      /* tree_unrnark(T) unmarks all of the
      /* clauses in the list of sub-trees T */
back_goal_gen([Goal,_,Sub),BkGoal) :-
          last(Sub,L),
           back_goal_gen(L,BkGoal)
back_goal_gen([Goal,N,[]],Goal).

**********************************************************
```

Figure 4. An example mal-rule.

The module selection process involves the following stages:

i) locate sequentially first trace line discrepancy;

ii) generate list of possible wrong atomic interpreter modules [using "trace pattern/module" pairs];

ill) run possible interpreters - if no wrong module specified in above list, use the correct module;

iv) select interpreter accounting for largest sec-

tion of incorrect trace;

v) communicate misconception to the user, inviting him to explore it via the tutoring module and to make the appropriate corrections to his symbolic execution;

vi) repeat at 1) - wrong modules may be plugged in where the correct module Is used in the interpreter (we make the simplifying assumption in the prototype that misconceptions are consistent throughout a single symbolic execution).

5. **Am Example of a Debugging Interaction**

The example illustrates the debugging of a symbolic execution which predicts the correct solution for a program run but which the user suspected was faulty. The program is the same as in figure 1. It implements a standard "generate-and-test" sequence, a state generated by the "h" predicate being tested by the "i" predicate, and the result of the "b" predicate being tested by the "c" predicate. A summary of the diagnostic process is given below.

The system first runs the program and generates the trace labeled SY1 (figure 5). The user is then invited to give his account of execution in summary form (trace U1). Taking the two traces, the system first identifies the discrepancy noted by the arrow pointing to "REDO b(X)" in U1 and then seeks to explain it in terms of modifications to the correct interpreter.

Having identified a work area, taken as the discrepant item with a context of one item either side, the system attempts to match the trace pattern to the database of patterns which index the incorrect interpreter rules. Failing to find such a match, the system requests a more detailed account of the work area. Increasing detail is requested until a match is found to one or more inalrules. The more detailed user trace for this example is given by U2.

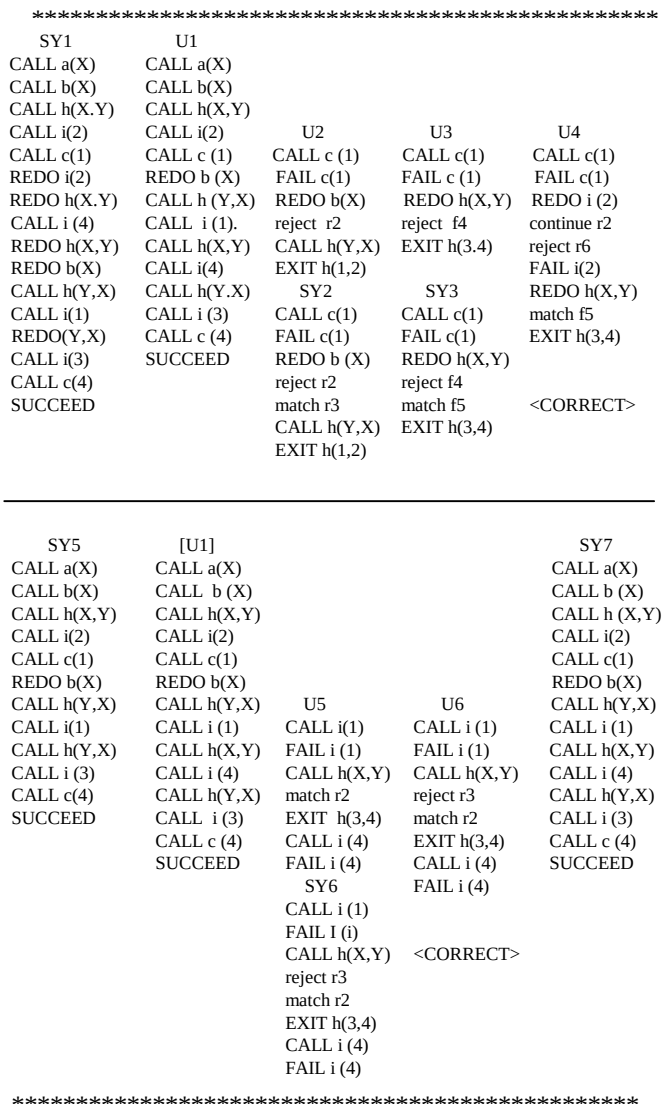

Figure 6. Trace interpretation.

Having identified possible mal-rules (the single rule "try-once-&-pass" in the present case), the Prolog interpreter is modified to include this rule and run on the user's program. The trace generated from this (trace SY2) is compared with the user's trace (U2) and, in the present case, found to give an acceptable match. The nature of the bug "try-once-&-pass" is then communicated to the user, and he is invited to run and inspect traces from tutoring programs designed to highlight differences between the erroneous interpreter rule and the correct rule.

Following the tutoring phase, the user is invited to correct the trace. The users modified trace segment is given in U3, and it may be seen that his account still fails to correspond to the correct system trace (SY1).

The old error proved to hide a further misunderstanding, which must now be identified and corrected. The same procedure is adopted as before, it being found this time that additionally the user does not understand that subgoals am backtracked into from the right (the inalrule "redo-body-from-left"). Tutoring is accordingly undertaken, and the misconceptions within the current work area are corrected. This is demonstrated by the user trace U4.

At this point, the system has identified two mal-rules present within the user's view of Prolog execution and has constructed a mal-interpreter to include these rules. On the assumption that these misconceptions will apply throughout the trace, the system proceeds to seek the next bug. This is achieved by first executing the user's program with the mal-interpreter to generate a further trace (SYS), and then comparing this trace with the user's original account of execution modified by the revised segment. In the example, this identifies a further work area (detailed in U5) which is finally explained by a further mal-rule ("fast_rule_ cycle"), the interpreter for which produces SY6. This is similar to "try-once-&-pass", where a given set of rules are all tried on a single fact, only moving on to the next fact after failure. Tutoring is undertaken for this misconception, which results in a correction to the trace - U6. A further attempt is made to find another work area. In the present -ase, the three misconceptions account for the user trace (compare U1 and SY7), so the consultation is terminated.

6. Conclusions

The present prototype system is able to diagnose 6 backtracking misconceptions. These are all "primitive" misconceptions in that they do not form a part of some error model and are assumed to be composed sequentially within the mal-interpreter. Some experimental work has indicated that this is not necessarily true of Prolog misunderstandings, some being clearly nested in others. Further, it is not clear that users are necessarily consistent in their errors of understanding nor that the failure to generate a trace error implies that the user correctly understands an interpreter concept; the user may have a "loose" concept of the interpretation process which is nevertheless adequate for a particular problem.

These limitations do affect the range of misconceptions capable of being diagnosed and corrected. However, the approach has proved robust on the typical problems presented in basic Prolog courses, which is when it is important that learners should develop a correct image of the interpretation process to employ for symbolic execution. Moreover, the modular approach to building our interpreter makes the progressive improvement or the system relatively easy.

The current system is a prototype and as such contains only a novice conception of Prolog based on the goal structure of successive AND trees (searches through program text). However, when a Prolog user gains experience both his goal structures and their related procedures change. At an advanced stage of learning for example, users employ an OR tree of goal stacks which greatly simplifies the backtracking rule. A realistic consultant would have to be able to represent transitions between such successive conceptualizations. This would require considerable additional knowledge including strategies for reducing the model's complexity.

Although considerable further research is required into the origin of Prolog misconceptions, their diagnosis and correction, we are confident that the present method of analysing, representing and simulating symbolic execution will accommodate the new knowledge.

References

Coombs, M.J. (1985). Internal and external semantics for Prolog: debugging the user interpreter. Invited talk, CRL, NMSU, Las Cruces.

Coombs, M.J. and Alty, J.L. (1984). Expert systems: an alternative paradigm. International Journal of Man-Machine Studies, 20, 2143.

Green, T.R.G., Sime, M.E. and Fitter, M.J. (1081). The art of notation. In M.J. Coombs and J.L. Alty (eds), Computing Skills and the User Interface. London: Academic Press.

Hartley R.T. (1985). Representation of procedural knowledge for expert systems. 2nd. IEEE conference on AI applications.

Kowalski, R. (1983). Logic for Problem Solving. New York: North-Holland.

McAllester, K. (1985). A debugging system for Prolog programs by user query. Technical Memorandum, Computer Science, University of Strathclyde.

Sacerdoti, ED. (1977). A Structure for Plans and Behaviour. New York: Elsevier.

Sauers, R. (1986). Controlling expert systems. In L. Bolc and M.J. Coombs (Eds.), Computer Expert Systems. Springer-Verlag - in production.

Sowa, J.F. (1084). Conceptual Structures. Addison Wesley.