Debugging Prolog Programs Declaratively

Simin Nadjm-Tehrani
Department of Computer and Information Science
Linköping University
S–581 83 Linköping, Sweden
e-mail: simin@ida.LiU.se

Abstract

Traditional declarative debuggers, despite their theoretical attractions are not considered to be practical tools for debugging of many Prolog programs. This is partly due to the fact that the basic declarative debugging system (Shapiro 83) only dealt with pure Prolog programs, and partly due to practical limitations of the suggested methods and algorithms. Our aim has been to study the applicability of declarative debugging methods to a wider range of Prolog programs. Earlier we have reported on improvements to methods and algorithms for declarative debugging of logic programs. In this paper we discuss extention of our algorithms for dealing with some Prolog constructs. In particular, negation of composed goals, unsafe uses of negation, the built-in `setof` predicate, and the if-then-else construct are elaborated upon. We also report on experiments with a sample of non-trivial Prolog programs, using our prototype system. Practical problems such as multiplicity of "irrelevant" user interactions are addressed. Finally, the types of Prolog programs to which the extended declarative debugging techniques are applicable are discussed.

1 Introduction

Debuggers are often cited as examples of meta-programs in Prolog. Declarative debuggers for logic programs, written as logic programs, have been the subject of extensive studies for their soundness and completeness properties (Ferrand 87), (Lloyd 87). These works show the possibility of different representation schemes for object language atoms at meta level - a topic more specifically studied in (Hill and Lloyd 89).

Not surprisingly, however, practical debugging of Prolog programs is persued along totally different lines. The majority of common Prolog systems are equipped with a basic tracing facility. The tracer often presents too much execution information to the user. It follows Prolog execution mechanism and neither supports nor encourages declarative reading of the program under development.

Despite the interest taken in declarative diagnosis by many researchers, the method has not been in wide use as an alternative to the basic tracer. This is partly due to the fact that the basic declarative debugging system (Shapiro 83) only dealt with pure Prolog programs, and partly due to practical limitations of the suggested methods and algorithms.

Our aim has been to study the applicability of declarative debugging methods to a wider range of Prolog programs. This was done at two stages: first, to make the method applicable to non-toy logic programs, the debugging algorithms and the language of communication between the debugger and the programmer were improved. This work has been reported in (Drabent et al. 1989).

That work additionally included preliminary suggestions to extend our debugging methodology for some non-pure features of Prolog. The idea was to isolate those features of Prolog which can not be given any declarative semantics, and incorporate methods for dealing with the remaining Prolog features in a meaningful way.

This second stage of improvement which covers the extentions of declarative debugging methods to a larger subset of Prolog programs is elaborated in this paper. The extentions deal with two types of problems. Some represent extentions to our debugging algorithms in order to deal with more complex language constructs. Among these extentions are treatment of various types of negations for atoms (safe and unsafe uses of negation), negated compound goals, goals using the built-in predicate `setof`, and the if-then-else construct. Some of these extentions to pure Prolog have been discussed (in varying degrees of detail) by other authors. However, since our basic algorithms have a fundamental difference with other reported algorithms in that no instantiations of formulas are required from the user, there is a novelty associated with the discussions in this paper. Adding these features to our system makes a real change to the class of programs covered by these methods, whereas earlier reported works have not been applicable to other than trivial examples.

The second type of extentions concern practical mechanisms which were added to our prototype system in order to further facilitate easier user interaction. These discussions are significant since these extentions too make the meta-program capable of debugging a larger class of Prolog programs.

Finally, we summarize the results of experiments with the prototype system on samples of non-trivial Prolog programs and draw practical conclusions from these excercises.

The structure of the paper is as follows. In section 2 we briefly cover notions from our earlier work which will be referred to in subsequent sections. Section 3 describes the details of our approach for dealing with various types of negated goals. Other extensions made to the basic implementation including the treatment of the built-in predicate `setof` and the if-then-else construct are described in section 4. Section 5 contains a summary of our experiences with the extended system and some practical conclusions. In section 6 we present comparisons with other reported systems and concluding remarks are made in section 7.

2 Preliminary Notions

In (Shapiro 1983) the debugging system is assumed to have access to information about the intended declarative semantics of the program. This information is assumed to be delivered by an *oracle* which is usually the user. The debugger makes queries about certain atomic formulas (atoms) and requires the oracle to classify the atoms as true (false) in the intended model, or satisfiable (unsatisfiable) in the intended model. In some cases the oracle is expected to give (potentially all) ground instances of an atom which are true in the intended model.

In (Drabent et al. 1989) we reported on improvements to the language of communication between the debugger and the oracle. In this section we recall some notions which will be used in the forthcoming sections. Our work concerns cases where logic programs terminate erroneously. In what follows we informally describe our diagnosis algorithms and the data structures which they examine in order to detect the two main types of errors in terminating programs. For a complete and formal description of algorithms and notions see (Drabent et al. 1989). We denote the intended model for a program by I (being a set of ground atoms).

2.1 Diagnosis Algorithms

(a) Incorrectness Diagnosis

Our incorrectness diagnoser examines a (finite) **proof tree** resulting from the successful computation of an atom by a program. Incorrectness arises where the atom in question is not true according to the intended semantics of the program. The algorithm employed queries certain nodes in the proof tree. The queries are called **universal questions** and are of the following type:

> " Is the atom A true in the intended model?" (i.e. are all ground instances of A in I).

A NO answer to this question leads to a recursive call of the algorithm with the queried atom. The output of the algorithm is an incorrect clause instance, i.e. a clause whose head is false in the intended model and whose body atoms are true in the intended model.

(b) Insufficiency Diagnosis

The insufficiency diagnoser examines nodes in a **search forest**. The search forest is a record of computation of the program with an atom as a goal, where some instance of the atom true in the intended model does not have a successful computation. Each tree in the search forest represents top level computations of the atom using one matching clause in the program. The leaves of each tree either represent successful computations of the atom (in which case are not interesting for insufficiency diagnosis), or points in the body of the matching clause where computations reached failure. The internal nodes of each tree represent possible solved instances of atoms in the body of the clause.

The insufficiency diagnosis algorithm identifies an atom *not completely covered* by the program. Informally, this is an atom which has an instance true in the intended model for which no clause in the program can contribute towards successful computation of this instance. Diagnosis is performed by querying certain nodes in the search forest. Failure leaves in the forest are subjected to **existential questions**. These are queries of the type:

> "Is atom A satisfiable in the intended model?" (i.e. is there some ground instance of A in I)

Internal nodes in the forest are subjected to **incompleteness questions**:

> "For atom A, is there an instance of A true in the intended model which is not in the set $\{A_1, \ldots, A_n\}$? " ($A_1, \ldots, A_n$ being instances of A)

A YES answer to any of these questions leads to a recursive call of the algorithm with the queried atom as input. Note that our algorithms no longer require the oracle to provide instances of atoms true in the intended model. Answers to the debugger queries are thus YES/NO answers. However, the queries may still be inconvenient to answer if they are repetitive, similar, or involving large terms. To avoid such sessions, instead of classifying single atoms, we allow the user to optionally describe a property of a larger set of atoms through giving *assertions*.

2.2 Assertions

In (Drabent et al. 1989), we introduced assertions in order to improve the language of communication between the debugger and the programmer. Through giving assertions, which are partial specifications of the intended model, the oracle can now classify possibly infinite sets of atoms. Partial specifications are (meta) logic programs and may be interactively developed during debugging. These optional specifications formalize interesting properties of the intended model. Shapiro oracles classify single atoms as being true (false), or satisfiable (unsatisfiable) in the intended model. Assertions provided by the user can classify *sets of atoms* for having these four properties.

Assertions have the syntax of a clause where the head has one of four fixed unary predicate symbols. The argument to the head is an atom from the language of the program to be debugged. The fixed predicate symbols and the set of (object program) atoms characterized by assertion clauses are given below:

- *true:* is used to characterize a set of atoms true in the intended model
- *false:* is used to characterize a set of atoms false in the intended model
- *posex:* is used to characterize a set of atoms satisfiable in the intended model
- *negex:* is used to characterize a set of atoms unsatisfiable in the intended model

posex (negex) stands for positive existential (negative existential) assertion. An assertion with *posex* appearing in its head describes a set of atoms in the object language (the language of the program P being debugged) for which there exists some instance true in the intended model. Similarly, assertions using *negex* describe a set of atoms in the object language with no instances true in the intended model. Atom A in the language of pogram P is coded as a ground term A' in the language of assertions. Some debugger queries may now be answered automatically in the presence of assertions. For example, the universal question about atom A gets a YES answer if the goal $\leftarrow true(A')$ succeeds with the program containing the assertions, and a NO answer if the goal $\leftarrow false(A')$ or $\leftarrow negex(A')$ succeeds with the assertions. Only if neither of the above three goals succeeds, the user will have to answer the query.

With the above improvements to the basic declarative debugging methods, we have tried to investigate further problems arising due to extention of the declarative framework to a larger subclass of Prolog programs. This is the subject of the forthcoming sections.

3 Debugging Prolog Programs with Negation

Suggestions for extending our framework for debugging normal logic programs were put forward in (Drabent et al. 1989). Essentially, debugging of program P is now done with respect to an intended model I which is supposed to be a model for the completion of P. A correct program is also expected to have sound computations which implies that the use of negation in the computations of the program should be safe or weakly safe (Lloyd 1987).

The extention of declarative debugging to programs using negation was addressed in (Lloyd 1987) in the framework of arbitrary programs. However, two points are worth attention: first, there were no discussions of unsafe uses of negation in that work since NU-Prolog provides a sound implementation for arbitrary programs. An NU-Prolog program either uses negation safely (possibly exploiting a delaying mechanism in the compution rule), or flounders.

Secondly, the algorithms in (Lloyd 87) have one feature in common with the original Shapiro algorithms: requiring the user to instantiate whole formulas[1]. Our methods had to be extended to in both of these directions in order to make declarative debugging of Prolog programs with negation practical and based on a sound methodology.

For Prolog programs run on standard implementations, there is a problem in that safeness is not checked by the system. Hence, a declarative debugging environment for Prolog which incorporates our extended debugging algorithms produces correct results if safeness (or weak safeness) of negation is checked for separately.

In (Drabent et al. 1989) we stipulated that since we perform debugging by examining a record of computation of the program, to check that the program uses negation soundly can be interleaved with producing the record of computation.

[1] Or at best instantiation of atoms depending on how the oracle is implemented. But the implementation of the predicates *valid* and *unsatisfiable* have not been discussed.

The debugging meta-interpreters simply have to check that if a subgoal $\neg A$ fails, A succeeds with the empty substitution (the condition for being weakly safe). A goal which does not pass the test represents an unsafe use of negation on which a sound Prolog implementation would flounder. Our debugging algorithms are not sound when applied to such programs. Therefore, we suggested that debugging is discontinued in these cases and the situation is pointed out to the programmer.

Once the debugger was augmented as above and extensions to deal with queries on negated atoms were added, experiments on Prolog programs using negation were carried out.

These experiments pointed out a hidden assumption in the above method of checking for safeness. The problem arises when the predicate used in a negated atom is defined erroneously in the program. It turns out that the above treatment is not completely satisfactory if safe use of negation is checked while debugging but not while testing the program. We discuss separately the cases for incorrectness and insufficiency.

A literal $\neg A$ is subject to query by the incorrectness algorithm if it appears as a node in some proof tree (say for goal $\leftarrow G$) being examined by the algorithm. Since $\neg A$ has succeeded, A has failed. The weak safeness condition is therefore trivially satisfied and our debugging algorithms can be applied. However, if the program is insufficient for A, the existential question about A is answered YES and a call to the insufficiency diagnoser detects an uncovered atom. Once that error is corrected, the use of $\neg A$ may no longer be weakly safe. This fact may not become obvious to the user if the subsequent execution of goal $\leftarrow G$ (with the unsound Prolog implementation) returns the expected answers. Hence, the debugger contributes to correction of the definition of a predicate, which may appear in an unsafe use of negation in the program, and this may go unnoticed by the programmer.

Now let's consider the situation when the search forest for some goal $\leftarrow G$ contains negative literals. A similar situation arises here. Consider a literal $\neg A$ which is subject to a query by the debugger. Then $\neg A$ is a leaf in the forest (i.e. a failing literal) and the query is an existential question[2].

A YES answer to the existential question about $\neg A$ leads to a call of the incorrectness diagnoser with the atom A (A is false in I). Again, correction of the definition of the predicate appearing in A may result in a subsequent unsafe use of negation. This arises if A is satisfiable but non-valid in the intended model.

The above analysis shows that if the programs to be diagnosed are normal programs (Lloyd 1987), then the following measures can be taken:

1. if the debugging meta-interpreters trap a failing negative literal which uses negation in an unsafe way, the user is warned that the goal would flounder under a sound Prolog implementation, and debugging is discontinued;

2. if an incorrectly succeeding (failing) negative literal contains a satisfiable (non-valid) atom A, the debugger should, as well as finding a cause of insufficiency (incorrectness) in the program with A as goal, warn the user in the cases where the goal $\neg A$ is used in an unsafe manner after correcting the insufficiency (incorrectness).

The second measure (i.e. to check whether $\neg A$ is a floundering goal after correction), can simply be taken by checking the answer to the universal (existential) question about A, whenever A appears in a negative literal and the insufficiency (incorrectness) diagnoser is to be called with A as input. If A is satisfiable but not valid, then with a correct definition for the predicate appearing in A, a sound implementation flounders on goal $\neg A$.

[2] Note that incompleteness questions are asked about internal nodes in the search forest, succeeding negated atoms do not bind any variables, and no incompleteness questions are asked about nodes whose answers include a variant of the questioned node.

Note that the warning about floundering is given to enforce a methodology of programming and despite the fact that the underlying Prolog system accepts unsafe uses of negation. These simple measures would suffice for normal programs run on a standard Prolog system, but unfortunately they are still not sufficient for arbitrary Prolog programs with negation. This is due to the use of negation on compositions of atoms, and the use of the built-in predicate `\+`. These matters are discussed in the following sections.

3.1 Negation of Compound Goals

An arbitrary Prolog program may contain goals which correspond to compound formulas. Compositions of atomic formulas is possible through the use of "`,`" (conjunction), "`;`" (disjunction), and "$\rightarrow$" (the if-then-else construct). The meta-predicate `call/1` further complicates the matter by the possibility of introducing a local cut in the composed goal.

In practice, Prolog programmers may apply negation to arbitrary compound formulas. Further enhancements to the debugger were needed to deal with such programs satisfactorily. The reason is twofold:

Firstly, the questions asked and the assertions stored should be about single atoms and not combined goals (recall that YES/NO answers and assertions specify sets of *atoms* having particular properties with respect to the intended model).

Secondly, warning for subsequent floundering arising from a correction in the program (as discussed in the previous section) is no longer straightforward. These aspects can be best illustrated with examples of goal composition.

Consider a program which uses $\neg C$, where C is a composition of atoms, in a clause body. To illustrate the problems arising and the way the debugger should deal with such negated goals we discuss each case separately:

(a) insufficiency diagnosis

A negated goal appearing in the search forest is subject to a query only if it has failed. Furthermore, debugging proceeds if the goal has passed through the weak safeness test.

The failure of $\neg C$ implies that the composed goal C has succeeded with the empty substitution. Our debugger needs the answer to the universal question about C, and needs to obtain this answer by querying atomic components of C (not questioning the composed goal C). In the debugging algorithms reported in (Lloyd 1987), it is assumed that the oracle (through the definition of the predicate *valid*) can provide instances of formulas true in the intended model. Thus, the user may not only have to provide instances of atoms but perhaps also instances of composed goals. This is a practical problem the solution of which is necessary for convenient user interaction.

Our implementation deals with this problem as follows: in order to obtain the answer to the universal question about C (making queries about its atomic components only), a dummy clause with the composed goal C as body is created. The incorrectness diagnoser is called with the dummy head as input. If the reported incorrect clause by the diagnoser is the dummy clause, then the composed goal is true in the intended model, and the answer to the universal question about C is YES. In this case the original insufficiency diagnosis has to be pursued in another part of the search forest. Otherwise, a genuine incorrect clause is returned by the incorrectness diagnosis.

(b) incorrectness diagnosis

The algorithm examines the nodes of a proof tree. Hence, the nodes may possibly be of the form $\neg C$ where C has failed and may contain some variables. The diagnosis procedure requires the answer to the existential question about C. The answer to this question is YES (leading to a call of the insufficiency diagnoser) if the composed goal is satisfiable.

Again, the answer has to be obtained without questioning the composed goal C. (Note that simply querying the components of the composition does not work: e.g. if C is a conjunction, it is possible that every conjunct is satisfiable but C is not satisfiable.) The algorithms reported in (Lloyd 1987) require the oracle to provide an unsatisfiable instance of a composed goal (through the predicate *unsatisfiable*).

Our algorithms avoid instantiations of goals by the user. But we need the answer to the existential question about a composed goal. To avoid asking a question about the composed goal C, we build the search forest for a dummy atom which matches a dummy clause (head) whose body is the composed goal. If the program is insufficient for some component in C, this will be found using the insufficiency diagnoser with the above mentioned forest (consisting of one tree). Otherwise, the insufficiency diagnoser returns the dummy atom as output. This implies that the composed goal is unsatisfiable. Hence, in the original computation it has failed correctly and the search in the proof tree is continued elsewhere.

In the previous section we discussed the case when an *atom* appearing in the negative literal is satisfiable but not valid, and pointed to the desirability of warning the user about such a use of negation. Similarly, when the composed goal C is non-valid but satisfiable, $\neg C$ is a goal which should flounder.

Warning about such a case may imply having to answer the universal question about each atomic component (e.g. a conjunction is non-valid if some conjunct is non-valid). Therefore, we have made this an option to be exercised by the user. If the user wishes, the debugger can point out an unsafe use of negation in a negative goal which previously succeeded incorrectly. In this case, the user may have to answer some universal questions (see Appendix 1 in (Nadjm-Tehrani 1989)). Note that this option arises only if the negated goal is non-ground at call time. Furthermore, the usual optimizations (Drabent et al. 1989) for making the best use of existing assertions are in force.

3.2 Unsound Use of Negation

Our study of Prolog programs with negation showed that many programs using negation result in a computation which is not weakly safe. In most cases this is because programmers use $\neg A$ with the intended reading $\neg\exists A$. Therefore, the usual Prolog implementations (that do not check for safeness) are in fact sound for this reading. This problem can be illustrated using the following example:

Example
Given the Prolog clause :

$$covers(Tree, Graph) \leftarrow$$
$$\quad not(node(N, Graph), not(node(N, Tree))).$$

The clause is logically equivalent to:

$$covers(Tree, Graph) \leftarrow$$
$$\quad \exists N\, not(node(N, Graph), not(node(N, Tree))).$$

It is obvious that the intended semantics of the predicate $covers/2$ is not the same as the semantics delivered by this clause (since according to this clause a Tree covers a Graph if there exists some node which does not belong to the Graph). What the programmer meant to write was the clause:

$covers(Tree, Graph) \leftarrow$
$\quad not \exists N(node(N, Graph), not(node(N, Tree))).$

But common Prolog syntax does not allow such specific use of quantifiers.

Our debugging algorithms would be not applicable to a program containing the (original) example clause since the meta-interpreter would recognize unsafe use of negation in the clause. Hence, the fact that our approach to negation can not be soundly applied to these uses of negation, reflects the deficiency of the usual Prolog syntax in expressive power: the programmer wishes to write the final clause but is forced to write the first clause in Prolog.

Our experience thus confirms the reported deficiency in Prolog's expressive power which motivated the use of explicit quantifiers in NU-Prolog (Naish 1986). The NUDE debugging environment which is built on top of NU-Prolog is able to debug programs which use the standard quantification as well as programs which use the second, non-standard quantification.

However, most Prolog programmers do not have access to systems which provide for explicit quantification of variables. Programmers using standard Prolog systems ought to be aware of this deficiency and their choice of quantifiers should become explicit to the debugger. A practice recommended by the Prolog community is that the built-in predicate `\+` is used when the users mean $\neg\exists A$, and the library predicate "not" when they mean $\neg A$.

We have assumed the use of this convention by the users and treat the different negation predicates accordingly. In other words, if the built-in predicate `\+` is used, our debugger does not expect weak safeness of computation and does not warn for possible floundering. Moreover, there is a change in the routines for obtaining the answer to the existential question about a negative goal.

Usually the existential question about negation of G is obtained by reversing the answer to the universal question about G. In the case of `\+` G, the existential question and the universal question about the goal coincide, and it is therefore possible to use any of the four assertion types for answering these questions (if assertions about G exist).

For example, let G be $p(x)$. Then the answer to the existential question about $\neg\exists x p(x)$ is YES(NO) if $p(x)$ is unsatisfiable (satisfiable). This results in a higher probability that the existing assertions can answer the question (since the query can be answered by a *posex* assertion as well as a *true* assertion).

The above treatment of the two uses of negation is sound and helpful provided the user chooses the right predicate for her intended quantification. For example, when standard negation is intended and `\+` is used, the possibility of trapping unsafe uses of negation are missed. Alternatively, using not(A) when $\neg\exists A$ is intended, may result in undue complaints by the debugger about violation of the weak safeness condition (see Appendix 1 in (Nadjm-Tehrani 1989)).

4 Other Extensions

In this section we describe the treatment of higher order predicates like *setof*, the if-then-else construct, and certain practical mechanisms to improve the user interactions.

4.1 The Treatment of *setof*

Since `setof` is a higher order built-in predicate, it does not have declarative semantics in the normal sense. That is, its semantics is not described by tuples of terms from the Herbrand universe of the program in which it is used. Nevertheless, it can be thought as having a well-defined intended semantics. For each implementation of `setof`, its semantics can be considered operationally in terms of call-success patterns. In other words, for a given call of the built-in predicate `setof` and a given program, the success patterns for the call are well-defined.

This observation leads to the following treatment: for the call to the predicate `setof(Template, Goal, Set)`, where at call time `Goal` is instantiated to an atom A, and `Template` to a term containing variables appearing in A, the intended success instances of `Set` are implied by the intended (declarative) semantics of the predicate appearing in A. Hence, if the program is not incorrect or insufficient with respect to the atom A, the success instance of `Set` will be computed as expected (since `setof` is a built-in predicate).

Therefore, for any such calls to `setof` (appearing during incorrectness or insufficiency diagnosis), the system should test correctness of the definition of the predicate appearing in A. This should be done by querying the occurring instances of A. In many Prolog implementations the call to `setof` fails where the resulting `Set` is empty. The debugger should therefore check the satisfiability of A (where A has no success instances), or succeeding instances of A (where the call to `setof` succeeds).

In other words, whenever a node containing the predicate `setof` appears in a proof tree or as an internal node in a search forest, the oracle is asked the incompleteness question about A, as well as the universal question about each success instance of A. If any success instance is false in the intended model the system calls the incorrectness diagnoser with that instance of A. If the set of answers for goal A is incomplete with respect to the intended model, the insufficiency diagnoser is called with A. Similarly, if the call to `setof` appears as a leaf in a search forest, the existential question is asked about A whereby a YES answer leads to a recursive call of the insufficiency diagnoser with A.

Unfortunately, in general, favourable answers to these queries are not sufficient to guarantee that the success instance of `Set` is correct according to the intended semantics. In many Prolog systems `setof` can be called with explicit existential quantification of some variables appearing in A. Thus, the success instance of `Set` is dependent on the actual definition of the predicate appearing in A as well as the existential quantification used (see e.g. Quintus Prolog Reference Manual, page 164).

In practice, the programmer may use a different quantification than that needed to characterize the intended instance of `Set`. Also, the choice of `Template` and `Goal` affects the success instances of `Set`. In these cases, our treatment is still sound: either the success instance of `Set` is what was intended by the programmer, in which case the error lies elsewhere and will be diagnosed by queries of other literals, or, the computed `Set` is not the intended set. Then, provided the answers to all remaining queries about other literals are satisfactory, the clause (or the procedure) containing the call to `setof` is reported to be incorrect (or not completely covered).

We consider this treatment as adequate since the exact checking of the `Set` with respect to the given quantification of variables can not be performed using a simple query with an obvious YES/NO answer. However, this basic treatment has to be further extended to deal with arbitrary composed Prolog goals as the second argument to `setof`. This is done in the same way that negation of compound goals are dealt with (as described in section 3.1).

4.2 The if-then-else Construct

Prolog implementations usually allow for a kind of if-then-else construct. The common syntax for such a construct takes the form of the composed goal "$P \rightarrow Q; R$" (if P then Q else R). A shorter format is "$P \rightarrow Q$" (which stands for if P then Q else fail). In order to correctly diagnose programs using these constructs a number of extensions to the diagnoser were necessary. First, the meta-interpreters employed by the diagnoser should explicitly deal with this construct so that the diagnoser has the chance to examine the predicates appearing in P, Q and R and debug them if necessary.

Secondly, the questions asked about atoms appearing in P, Q and R have to be of the right type. We consider each case separately. During incorrectness diagnosis the debugger may be confronted with a fraction of such a construct in the proof tree being examined. Either P and Q have succeeded and should be examined, or P has failed and R succeeded. In the former case the incorrectness diagnoser asks the universal question about P and Q, and makes a recursive call to itself after a NO answer (as usual). In the case where P has failed and R succeeded, the diagnoser has to check that P is not satisfiable in the intended model. Hence, the existential question should be asked about P, and the universal question about R. A call to the insufficiency diagnoser is made in the event of a YES answer to the existential query (i.e. if P was satisfiable but failed).

During insufficiency diagnosis, it is possible that P succeeds and Q appears as an internal node or a leaf in the search forest. In either case no incompleteness questions are asked about P[3]. Q is subjected to the existential query if it appears as a leaf, and to the incompleteness question if it appears as an internal node.

Alternatively, P may appear as a leaf and R as an internal node or a leaf. In this case P is subjected to the existential question and R to the incompleteness question or the existential question as appropriate.

4.3 Delaying

Our experiments have shown that answering incompleteness questions was not a big burden except for those cases where the predicate in question was highly nondeterministic. In these cases, two points are worth mentioning. Firstly, asking the incompleteness questions in our system replaces the need for instantiation of non-ground goals by the user in other systems. When such instantiations are required, the user may potentially have to *provide* all such instances of the relevant goal which are true in I, and is therefore confronted with an even harder task. Secondly, in our basic implementation, the incompleteness questions were delayed until no existential questiona were found to lead to a recursive call of the algorithm. This facility has been further enhanced in the current implementation with the choice of additional delaying which will be described now.

During the experiments with the insufficiency diagnoser on computations with a large search forest, it was obvious that a less rigid order of visiting the nodes in the search forest could be useful. The original version of the debugger had the fixed rule: answering the existential questions using assertions, asking the user to answer remaining existential questions (i.e. those which could not be answered with an existing assertion), and finally asking the incompleteness questions in the fixed order in which they had been collected and stored.

The insufficiency diagnoser needs a YES answer to some query to make a recursive call to itself on the queried atom. Sometimes the query in question is easy to answer but it is preceded by a number of queries on other atoms which are not as easy to answer, and which are irrelevant in the sense that answers to those queries are NO anyway. Examples of difficult questions to answer are existential questions about atoms with large instantiated terms and complicated semantics, or incompleteness questions with several answers and large terms in each answer. This observation led to equipping the debugger with a delaying mechanism whereby the user may postpone answering the difficult questions and proceed with the search until:

[3] Asking the incompleteness question would be inappropriate due to the built-in cut in the construct. However, one could argue in favour of asking the universal question about P in order to detect the cases where the first answer produced by P is incorrect and the bindings produced make Q unsatisfiable.

- the recursive call is made on another atom, in which case the delayed questions will never be asked again,
- or, the recursive call has to be made on some atom subject to a delayed query. In this case, all other questions will get a NO answer, and the user has to answer some difficult queries which were previously delayed.

The addition of the delay mechanism somewhat relaxes the fixed rule for traversing the search forest nodes. It may also be considered as a partial solution to the point raised in the future works section of (Drabent et al. 1989). There, we recognized the case where the user may know that a set of atoms belong to the intended model, and that another set of atoms do not belong to the intended model, but may be unsure about the interpretation for the remaining set of atoms. The delayed queries may be thought to be about atoms falling into this gray area.

4.4 Declarations of Correctness

Already at early stages of experiment, it was known that it is often irritating to answer queries about trivial procedures which are deemed to be correct. This led to extending the system to accommodate a new type of interaction with the user. As well as answering YES/NO to the debugger's various questions, or providing assertions in reply to queries, the user is now allowed to declare certain predicates *correct*. This leads to a treatment of the predicate by the debugger which roughly resembles that of built-in predicates. The subsequent existential and incompleteness questions get an automatic NO answer, and the subsequent universal questions are automatically answered YES.

Our use of correct declarations is slightly different from the heuristic suggested in (Pereira and Calejo 1988). There, it is proposed that a user may begin debugging with a predetermined set of predicates *assumed* correct. It is suggested that the assumption may later be withdrawn if the result of diagnosis is unsound. We encourage declarations about predicates *known* to be correct, and the declarations are made *dynamically* in the course of interaction with the debugger. It is the responsibility of the user to base the correctness declarations on valid grounds. A correct procedure is one which computes all and only the atoms included in its intended model. This requires quasi-correctness of the sub-procedures used by the procedure (in the following sense: if p is a correct procedure then all the sub-procedures used in the definition of p produce all and only correct answers for every possible call to the sub-procedures arising during calls to p).

Obviously, as programs get more complicated it is more difficult to be certain about the correctness of the underlying procedures. Since the aim of these declarations is solely a reduction in unnecessary user interactions, the easiest way to exercise the option arises in response to debugger queries. In other words, correct declarations are encouraged only when an atom is subject to query by the debugger.

Conceptually, we view user's knowledge about correct procedures in a different way from user's assertions. This is both justified and essential. Correct declarations and the stored answers to the incompleteness questions describe properties of the actual program which is under development. They may be rendered invalid once the relevant predicates are modified in the development process. The assertions, on the other hand, describe the properties of the intended model which is (assumed to be) fixed under a development cycle.

An ideal development environment must warn the user if the changes to the program may affect the knowledge stored at earlier debugging scenarios. Our system does not incorporate this facility.

5 Experiments on Sample Programs

The wide range of reported works on debugging of logic programs, and in particular, the numerous enhancements to the basic debugging approach of Shapiro, indicate the desirability of better debugging environments for logic programming systems, and the attractiveness of a declarative component in such environments.

Unfortunately, most published reports can only illustrate simple examples and the practicality of the wide range of ideas put forward has remained largely unexplored.

We aimed to establish the practicality and usefulness of declarative debugging methods once they are sufficiently extended to remedy their obvious shortcomings. We addressed this question by performing three studies:

1. measuring the improvements due to introduction of assertions and an improved insufficiency algorithm;

2. exploring the usefulness of the extended version of the basic system to debug non-trivial Prolog programs (excluding side-effects), or finding the class of programs to which the method could be successfully applied;

3. finding the extent to which the augmented and improved version of algorithmic debugging is helpful while developing "real" Prolog programs.

Preliminary confirmation of the improvements over the basic Shapiro-type debuggers was provided by examining comparative debugging sessions on a range of almost pure logic programs (Drabent et al. 1988). The measure of interest was the number of user queries and in particular, the number of incompleteness questions compared with the number of atoms instantiated by the user.

To answer the second question, a number of non-trivial programs were diagnosed using an implementation which incorporates extensions reported in this paper. The programs diagnosed included both non-trivial correct programs with "artificial" bugs added, and genuine buggy programs which did not use `assert/retract` or the extra-logical predicates (such as `var/1`, as discussed in (Drabent et al. 1989)).

The complete sample consisted of 50 programs around half of which can be considered to be non-trivial examples. Here, by non-trivial examples we mean programs using some Prolog constructs like meta-logicals, cut, compositions of goals and negation. These programs are larger than the programs usually used in publications for illustrative purposes (with a length of one or more pages of Prolog code), and whose bugs may not be easily detected by a quick examination of the code. The sample included examples like an Othello playing program, a program for finding minimal spanning trees of a graph, a solution to the water jug puzzle, an equation solver, and so on.

The overall conclusion here was that finding errors in these programs with our declarative debugger is neater and more convenient than applying other debugging approaches like standard tracing. The actual process of declarative debugging contributes towards formalization and understanding of useful properties of the intended model. This can be of practical help when developing large programs, and of pedagogical importance when teaching logic programming to students. A summary of some practical conclusions of these tests as well as the noted deficiencies of the method are given in the following subsection.

The third question led to another practical consideration: what is a "real" Prolog program? Since the earlier programs subject to test were Prolog programs written in academic environments, an attempt was made to study the collection of "real" programs gathered at a Prolog benchmarking workshop (Kesselman 1988). This was done with a view to establish which types of programs are considered to be serious applications of Prolog. A useful debugging tool should facilitate diagnosis of the buggy versions of such programs under development.

The benchmark programs made up over 36000 lines of Prolog code and covered a wide range of application areas and programming styles. It consisted of 24 programs many of which were spread over a large number of files. Perhaps not surprisingly, many of them, too, originated from academic environments.

A review of the type of programs included, revealed that as few as 9 programs avoided extra-logical constructs to the extent that they could definitely be developed with declarative debugging as a tool. For many other programs, there were large fractions to which declarative debugging could be applied.

Generally speaking, meta-programs like compilers and interpreters (which accounted for 7 programs in the benchmark) invariably used extra-logical predicates and operations on the database. Fractions which interleaved input-output with computation were a common cause for non-applicability of declarative methods in some other programs.

An example of large programs falling outside the framework of declarative debugging is an Othello playing program, fragments of which were successfully subjected to our debugging method (see Appendix 3 in (Nadjm-Tehrani 1989)).

5.1 Debugging Non-trivial Programs

The experiments with the extended debugging system resulted in a number of observations which we summarize below.

Assertions continued to be useful for reducing user interaction in a number of sessions, as well as formalization of interesting but simple properties of the intended model (see Appendices 2 and 3 in (Nadjm-Tehrani 1989)). Likewise, the delaying mechanism and correct declarations reduced the nasty and inessential interactions considerably. Obviously, delaying the answer to a query may lead to queries which would not otherwise be made. The payoff varies from case to case. In particular, there is no gain if a query about the atom which will be subject to a recursive call of the algorithms is initially delayed.

The negative existential assertions were by far the most common types of assertions, which confirms the general belief that many errors are results of, or show up in terms of ill-typed arguments. (Recall that negative existential assertions, characterizing the complement of supersets of the intended model, can be related to type information).

Positive existential assertions were also useful. This often arose in cases where a combination of well-typedness of some argument(s) and other argument(s) being uninstantiated, was a sufficient condition for existence of an instance of the atom in question in the intended model (e.g. see Example 3 in Drabent et al. 1989).

The negative assertions (the *false* predicate) were not used in any of the recorded sessions, since often a stronger property could be specified in terms of negative existential assertions (*negex*).

Another useful observation was that for some programs , where both procedural and non-procedural solutions could be given, the procedural solution was more difficult to debug using declarative methods. It may take a while for the author of the procedural style code to work out the declarative semantics of atoms which are subject to queries by the declarative debugger. This is an obvious statement, but is worth emphasizing because it highlights the pedagogical value of availability of declarative debugging tools in teaching environments.

A related point is the preconditions for willingness of users of systems to resort to declarative debuggers. Our experience with students beginning to learn Prolog (with previous experience in Lisp and Pascal) showed that having to think in terms of models and logical formulas was hard enough at the beginning. To expect them to use the machinery of assertions was not realistic, whereas with suitable guidance they were able to make use of our debugger without the option of assertions. These attempts confirmed the general belief that incorporation of a simple and good user interface plays an important role in a tool's gaining wide acceptance.

6 Comparisons

Earlier, we have made comparatory remarks about various approaches to debugging logic programs (for a more extensive list of references see (Nadjm-Tehrani 1989)). In this section we comment on a number of recently reported Prolog debugging systems to illustrate certain practical criteria. For each system we describe the feature which makes it an interesting comparatory system from our point of view.

6.1 The Debugging Module of Argus/M

The authors of (Maeji and Kanamori 1987) and (Kanamori et al. 1989) have independently come up with many ideas and conclusions which are common with ours. They adopt a sound and complete "top down zooming" algorithm for insufficiency diagnosis which avoids asking for true instances of atoms. But they agree with our stipulation that since the instantiation of atoms by the user results in a smaller search space, the ultimate choice must be the user's.

A nice feature of the work reported in (Kanamori et al. 1989) is the use of (partial) specifications. They use specifications in a similar spirit to ours: it is the user who decides whether specifications are provided and the extent to which they are provided.

Although some of the specification formulas presented in (Kanamori et al. 1989) correspond to our assertions, their system makes a different use of specifications from ours.

The use of assertions in our system is geared towards reducing the frequency of user interactions and formalizing intended program properties. Without assertions our system is a debugging tool which can be used soundly. With assertions, the system can run smoother and diagnosis may be made faster. Note that assertions can be provided interactively and the types of assertions given is guided by the atoms currently under query by the relevant debugging algorithm.

In their system the user decides on the types and extent of specification for each predicate appearing in the program *before* a testing-debugging-modification cycle begins. Specifications are used for three purposes:

1. generating test cases,
2. answering queries about atoms appearing in the computation,
3. instead of (parts of) the actual program in the automatic testing mode.

In what follows we consider each of these uses of specifications in turn. The first use of specifications leads to testing the program on the set of test data which can be generated from the specification formulas. It is well known that adequate generation of test sets according to a particular selection criteria, and checking the reliability and validity of the criteria is a major problem of testing (Goodenough and Gerhart 1975). An automatic generator of test sets which produces sets satisfying a given criteria would be of great help. Of course, checking the adequacy of the criteria can not be automated. But given a criteria, the test set generator must know when to stop.

The Argus/M testing module appears to rely on the user for deciding when to stop, i.e. deciding when the generated test set is adequate for testing according to the intended criteria, and the criteria is not made explicit.

Now, this does not invalidate the approach provided the user is aware of this aspect of the system. In fact, the approach implies that the user has the freedom to "code" the desired criteria into the specification formulas. Then, provided that the system does not loop and testing is performed completely (i.e. the user does not interrupt by stopping), the system tests the program according to the given criteria.

The second use of specifications closely resembles our use of assertions. But as the specifications are not provided in response to debugger queries, and the atoms which would be questioned by the algorithms are not known prior to the debugging session, the extent to which the specification formulas can help to answer debugger queries varies from case to case. Of course, this is not the only use made of specification formulas, but it is possible that the formulas provided are not useful for answering any debugger queries.

The third use of specification formulas arises in the automatic testing mode. The system itself chooses which predicate to select for testing next, and the choice is predefined by caller-callee relationships in the program and the existence of executable specifications for predicates.

The selection rule is such that if a procedure selected for testing calls another procedure which is not yet tested, the latter procedure's executable specification is used instead of the actual program. This use of the specifications is justifiable considering the fixed hierarchy of the predicates to be tested (as defined in page 36 of Kanamori et al. 1989). But it will reduce the transparency of the testing process to the user.

To summarize, the system reported in (Kanamori et al. 1989) is a testing-debugging system which can free the user from some repetitive and troublesome interactions. Its testing component is highly dependent on the user for determination of test set adequacy and the validity of the selection criteria. At bug detection phase, the system relies on declarative knowledge about the intended model, but the class of Prolog programs which can be debugged using this system is not specified. The complications arising from the use of Prolog built-ins are not discussed.

6.2 NUDE: the NU-Prolog Debugging Environment

NUDE is a declarative debugger for NU-Prolog programs currently under development (Naish et al. 1989). The system consists of a collection of programs capable of combining static checks, dynamic declarative debugging, and (a restricted form of) automatic testing.

Obviously, static checkers are not capable of detecting many semantic bugs. Nevertheless, the combination of static checks with dynamic debugging techniques is not only shown to be promising but also is, in our view, an essential part of any practical debugging environment for Prolog programs.

Many Prolog systems already incorporate syntactic checks aiming at detecting mistypes or wrong uses of predicates. One interesting aspect of NUDE is the extension of these checks for detecting loops, floundering and always failing clauses due to ill-typing.

Dynamic debugging in NUDE is declarative. Loops which are not detected by the static checks, are diagnosed using the usual depth-bound interpreter approach (Shapiro 1983). However, insufficiency is detected using a modified version of the Shapiro algorithm, where the user is required to confirm the validity of a succeeding goal's success instances. This is thought to reduce the frequency of asking the user to produce true instances of atoms, but it does not completely eliminate the need to do so (Naish 1988). In this way, missing answers caused by incorrectness are diagnosed by finding the cause of incorrectness first.

Their suggested questioning of the validity of success instances is in line with our observation (Drabent et al. 1988) that since incorrectness diagnosis is easier and produces a more concrete error message, the user should switch to the incorrectness diagnosis algorithm whenever a wrong answer is presented in an incompleteness question. In NUDE, this switching is done automatically at the expense of asking a larger number of universal questions.

The testing component of NUDE provides a restricted form of testing: when invoked with a given goal, it can check for correct behaviour of the goal for all its *known* true, satisfiable and unsatisfiable instances. When invoked to test a procedure, the program is meta-interpreted with a most general instance of the procedure call. Many Prolog programs may loop when tested with such goals. For those which do not loop immediately, the user decides when to stop testing as the (possibly infinite) set of answers to the goal are produced. (In this sense it is similar to the Argus/M system.)

NUDE covers a larger class of Prolog programs than most other reported systems. This is due to the extended syntax and procedural semantics of NU-prolog which ensures sound uses of negation as well as explicit quantifiers. Those uses of negation which cannot be computed safely using the implicit quantification in Prolog can be expressed and implemented soundly using additional quantifiers in NU-Prolog (see Appendix 2 in (Nadjm-Tehrani 1989)). As for the class of programs covered, various complications arising from extra-logical built-ins of NU-prolog and negation of compound goals are not covered by the work reported in (Naish et al. 1989). However, a practical combination with the traditional *spy* facility ensures that the user can declaratively debug those parts of a program to which the method is applicable. The rest of the program has to be checked through the standard tracer.

6.3 The TPM

Another well developed debugging tool is the graphical debugger known as the Transparent Prolog Machine (Eisenstadt and Brayshaw 1988). Two aspects characterize the system: firstly, it relies solely on the procedural semantics of Prolog and therefore represents an alternative strategy which can be contrasted with most other Prolog debuggers. This implies that the system has no restrictions in terms of the class of programs to be diagnosed. All features of Prolog including side-effects can appear in debugged programs.

Secondly, although the user interface and the zooming options available are great improvements to the traditional Prolog tracers, in the last analysis the TPM is a sophisticated trace facility. It does not provide the user with algorithms or methodologies for identifying the bugs. Rather, the user has to use his intimate knowledge about the execution mechanism of Prolog to find out which node in the graphical representation of the execution has misbehaved and has to be zoomed upon.

This system has a number features in common with otherwise declarative debuggers. Its "selective highlight", for example, includes a counterpart to the intelligent backtracking heuristic in (Pereira 1986), which is called retrospective spy in TPM. Similarly, some types of constraints which can be used to specify the node to be highlighted can be compared with our assertions. Likewise, its "compress" facility can be thought as corresponding to our declarations of correctness.

To summarize, the TPM is an example of effective systems for diagnosing full Prolog, and if augmented with declarative components (as suggested in (Eisenstadt and Brayshaw 1988)) it may become a practical debugger for all types of programs and programmer tastes. The example programs given in that paper, although larger and less pure than the examples usually included in published reports, are still far from real Prolog applications. Experience with the extended system for debugging genuine large buggy programs can contribute towards identifying practical approaches to Prolog debugging.

7 Conclusion

We have summarized our positive and negative experiences when applying declarative methods to debugging of Prolog programs. Debugging was performed using an improved and extended version of algorithmic debugging which uses a different insufficiency diagnosis algorithm, and incorporates partial specifications of the intended model.

The problems arising from inconvenient queries and the mechanisms to avoid or alleviate them have been discussed. Also problems arising from common uses of negation which complicate clarity of declarative debugging have been addressed. One possible treatment of the higher order `setof` predicate and a method for dealing with the if-then-else construct were also put forward.

Our tests on non-trivial Prolog programs showed that the extended declarative debugger could successfully be used to find errors otherwise not easily located. This property could not be expected to carry over to arbitrary Prolog programs: there is a class of "real" Prolog programs where the use of extra-logical predicates (like `var/1`) and operations on the data base can not be avoided. Our intuition is that many uses of `var/1` can be replaced by proper use of parameters (where the test operates as checking of a flag). However, a declarative treatment of `var` is only possible if meta-programs using `var` (e.g. compilers) use a proper representation of the object program atoms at meta-level (Hill and Lloyd 1989).

Similarly, it is well known that many uses of `assert` and `retract` may be avoided and replaced by passing over parameters. Of course, this may not be convenient in many applications, but the point is that some version of the these programs can be subjected to declarative debugging. A semi-automatic transformation of a sufficiently tested program may then introduce `assert` and `retract` to store partial results instead of carrying parameters.

However, a more fundamental change is needed for those applications of Prolog where the knowledge represented in the data base has to be updated incrementally in the light of new input. Addition of suitable built-in predicates (for example `insert` and `delete` as in (Hill and Lloyd 1988)) to the underlying Prolog systems may be a starting point for a declarative treatment of such programs.

Finally, the frequency of looping programs, specially with novice programmers, is an important issue which should not be left unexplored. Development of advanced loop checking techniques, and methods for detection of terminating programs are important contributions to the solution of this problem (see e.g. (Apt et al. 1989), (Bezem 1989)). The combination of declarative and operational methods in debugging environments would be one way of tackling this real problem.

Acknowledgements

This work would not be possible without the research environment provided by my colleages in the Logic Programming Laboratory, and my supervisors Jan Małuszyński and Włodek Drabent. In particular, I wish to thank Włodek for numerous discussions about the content of this paper and my implementation efforts. Anna-Maria Emde, Jan Komorowski, Thomas Långbacka, Ulf Nilsson and a number of undergraduate students contributed to my collection of sample programs. This work has been partially supported by the National Swedish Board for Technical Development, project number: 87-02926P.

References

Apt K.R., Bol R.N., Klop J.W., On the Safe Termination of Prolog Programs, in: *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon 1989: 353-368.

Bezem M., Characterizing Termination of Logic Programs with Level Mappings, in *Proceedings of the North American Conference on Logic Programming*, Cleveland 1989: 69-80.

Drabent W., Nadjm-Tehrani S., Małuszyński J., (1988) The Use of Assertions in Algorithmic Debugging, *Proceedings of the FGCS Conference*, Tokyo 1988: 573-581.

Drabent W., Nadjm-Tehrani S., Małuszyński J., (1989) Algorithmic Debugging with Assertions, in: *Meta-programming in Logic Programming* - MIT Press, 1989.

Eisenstadt M., Brayshaw M., The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming, *Journal of Logic Programming*, 5(4), December 1988: 277-342.

Ferrand G., Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method, *Journal of Logic Programming* 1987(4): 177-198.

Goodenough J.B., Gerhart S.L., Towards a Theory of Test Data Selection, IEEE Transactions on Software Engineering, June 1975: 156-173.

Hill P.M., Lloyd J.W., Meta-Programming for Dynamic Knowledge Bases, Technical Report CS-88-18, University of Bristol, December 1988.

Hill P.M., Lloyd J.W., Analysis of Meta-Programs, in:*Meta-programming in Logic Programming* - MIT Press, 1989.

Kanamori T., Kawamura T., Maeji M., Horiuchi K., Logic Program Diagnosis from Specifications, ICOT Technical Report TR-447, February 1989.

Kesselman, K., The Report on the Prolog Benchmarking Workshop, Aerospace Corporation, California, February 1988.

Lloyd J.W., *Foundations of Logic Programming* - Springer Verlag, Second edition, 1987.

Maeji M., Kanamori T., Top-Down Zooming Diagnosis of Logic Programs, ICOT Technical Report TR-290, August 1987.

Nadjm-Tehrani S., Contributions to the Declarative Approach to Debugging Prolog Programs, Licenciate Thesis No. 187, Dept. of Computer and Information Science, Linköping University, 1989.

Naish L., Negation and Quantifiers in NU-Prolog, in: *Proceedings of the 3rd International Conference on Logic Programming*, London 1986, LNCS 225, Springer Verlag : 624-634.

Naish L., Declarative Diagnosis of Missing Answers, Department of Computer Science, University of Melbourne, Technical report 88/9, 1988.

Naish L., Dart P.W., Zobel J., The NU-Prolog Debugging Environment, in: *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon 1989: 521-536.

Pereira L. M., Rational Debugging in Logic Programming, *Proceedings of the 3rd International Conference on Logic Programming*, London 1986, LNCS 225, Springer Verlag : 203-210.

Pereira L. M., Calejo M., A Framework for Prolog Debugging, *Proceedings of the fifth International Conference and Symposium on Logic Programming*, Seattle 1988: 481-495.

Quintus Prolog Reference Manual, version 10, February 1987.

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