PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

THE USE OF ASSERTIONS IN ALGORITHMIC DEBUGGING

Wlodek Drabent

Institute of Computer Science,
Polish Academy of Sciences,
P.O. Box 22,
00-901 Warszawa PKiN, Poland.

Simin Nadjm-Tehrani
Jan Maluszynski

Dept. of Computer and
Information Science,
Linköping University,
581 83 Linköping, Sweden.

ABSTRACT

This paper presents a version of declarative debugging that uses formal descriptions of properties of the intended model of the buggy program. The descriptions are called assertions and can be provided as logic programs. The concept of assertion includes the traditional oracle replies as a special case. Assertions give an approximate specification of the intended model: they need not fully specify the model and may be provided incrementally. Such specifications can significantly reduce the number of user queries. An experimental debugging system incorporating this idea has been implemented. The system uses algorithms which do not require instantiations of atoms by the user, and delays querying the user for as long as possible. In this paper we present the results obtained and the experiences gained while diagnosing a sample of programs using this system.

1 INTRODUCTION

This paper shows how the idea of algorithmic debugging (Shapiro 1983) can be augmented by a concept of assertion that generalizes the oracle replies. A prototype debugging system based on assertions is described and the results concerning its use on a sample of buggy programs are reported. A more comprehensive description of the algorithms used by the system can be found in (Drabent et al. 1988).

Logical foundations of algorithmic debugging can be found in (Ferrand 1987) and (Lloyd 1987). Our basic notions, though slightly different, have been strongly influenced by these papers.

Every (pure) logic program P has a model; see e.g. (Lloyd 1987). P is often considered to be the specification of the least Herbrand model M_P. On the other hand, the program should properly reflect the intentions of the user. These can be thought of as the "intended model" and can be viewed as a subset I_P of the Herbrand base. If I_P differs from M_P the program is erroneous. The program P is said to be

- *incorrect* iff $M_P - I_P \neq \phi$, i.e. iff it specifies some element which is not in the intended model, and
- *insufficient* iff $I_P - M_P \neq \phi$, i.e. iff some elements of the intended model are not specified by the program.

In this paper we do not consider nonterminating programs; we concentrate on tracing incorrectness and insufficiency of a logic program.

The elements of M_P can be computed using SLD - resolution. To discover an error and to localize its cause in the program one has to compare the results of computations, including failures, with the intended model. However, the latter is generally not formalized. To solve this problem Shapiro introduces the concept of oracle (Shapiro 1983). The *ground oracle* decides whether an atom is in the intended model. The *existential oracle* decides whether there is a solution to a given goal and is capable of producing elements of the intended model which are instances of the given goal. In practice, it is the user who answers the questions concerning the intended model.

Shapiro's debugging system acquires knowledge about the intended model through necessary interactions with the oracle. This knowledge consists of:

1. a finite subset of the intended model - YES answers of the ground oracle and the solutions produced by the existential oracle;
2. a finite subset of the complement of the intended model - NO answers of the ground oracle;
3. a finite set of atoms satisfiable in the intended model - YES answers of the existential oracle;
4. a finite set of atoms unsatisfiable in the intended model - NO answers of the existential oracle.

The language of the oracles does not allow to specify infinite subsets of the intended model nor infinite sets of atoms satisfiable in the intended model. The negative answers of the existential oracle are not used for tracing incorrectness although they specify infinite subsets of the complement of the intended model. This language is rather low level - the knowledge about the intended model is communicated in form of examples. There may therefore exist many queries concerning similar atoms.

Shapiro pointed out that incorporation of "constraints and partial specifications" into the algorithmic debugging scheme may reduce the number of interactions with the user (Shapiro 1983, p.79). This idea is formalized in (Drabent et al. 1988) and is explored in this paper. In the approach presented the user is allowed to provide the system with formal specifications of some properties of the intended model. These formal specifications (which are called assertions) may be developed interactively in the debugging process. The diagnosis system uses this knowledge about the intended model to localize errors. Whenever this is not sufficient for evaluation of results of the computation the system queries the user. The answer augments system knowledge about the intended model. This scheme includes as special cases the answers used in Shapiro's system. But generally the language of answers is more powerful. If the user is able to provide the system with some general properties of the intended model, the number of interactions decreases as it will be illustrated in section 4.

Our debugging methodology favours simple specifications and convenient answering procedures. This necessitates employment of algorithms whose questions can be answered with YES or NO. Therefore, we adopt a new algorithm in which the oracle need not provide instances of a given atom that are in the intended model. Instead, the user is expected to recognize the solutions to a goal and to identify a case where some answer is missing. A pragmatic question is whether this type of interaction eases the debugging process as compared with the traditional approach where instantiation is required. The results of the experiment described give a preliminary answer.

The rest of the paper is organized as follows. In Section 2 a language of assertions is introduced as a natural generalization of the language of the oracle and the use of assertions for algorithmic debugging is discussed. Section 3 describes the necessary oracle interactions and gives a brief description of debugging algorithms used in our prototype implementation. In section 4 summarized results from a series of test sessions using our algorithms and the Shapiro algorithms together with a discussion of the results are presented. Section 5 includes comparisons with related work. Conclusions and topics for future research are presented in Section 6. An example debugging session is shown in the appendix.

2 ASSERTIONS

Here we show the extension to the communication language of the algorithmic debugger. In addition to the simple YES and NO answers we provide the user with a possibility to describe some properties of the intended model. For this we introduce assertions as a device to specify (not necessarily finite) sets of (not necessarily ground) atoms of the object language. An obvious choice is to use logic programs to provide executable specifications of such sets using, as much as possible, existing library procedures.

One may argue that assertions can be used for full specification of the intended model. This would amount to giving an alternative correct version of the buggy program (Dershowitz and Lee 1987). Such a solution is completely unrealistic in most cases. It is often suggested that while developing a new version of an existing program the existing version can be used as an oracle (e.g. Sterling and Shapiro, 1986). Taken literally, this idea is also unrealistic because it requires that every procedure of the new program has its counterpart with the same intended meaning in the old program. Therefore our basic pragmatic assumption is that the assertions used for debugging are as simple as possible and only approximate the intended model rather than specify it.

We suggest to use four types of assertions that generalize the four types of answers given by the oracles as discussed in section 1. The assertions specify properties of the intended model (I_P) which are of interest to debugging. Four fixed predicate symbols are used to specify these properties in a logic program denoted by $As(I_P)$.

Note that $As(I_P)$ is a meta-program specifying sets of atoms of the language of object program P. Therefore a suitable representation for the atoms of P as terms of $As(I_P)$ has to be defined. These meta-programming issues are addressed in (Drabent et al. 1988) where we adopt a ground representation (Hill and Lloyd 1988) for the variables occurring in P. If A is an atom with variables $X_1,...,X_n$, then its image A' under our representation scheme is A with X_i substituted by 'VAR'(i) for $1 \leq i \leq n$, where 'VAR' is a functor not used in P.

We now give the definitions of the 4 types of assertions:

Positive assertions. These are used to define sets of (not necessarily ground) atoms valid in the intended model. We specify positive assertions using the predicate symbol *true*. If *true*(A') is a logical consequence of $As(I_P)$ and A' is an image of A then for all substitutions θ, $A\theta \in I_P$.

Example 1.

Consider the intended relation *insert* as in (Shapiro 1983). It includes (as a proper subset) all triples *(X, L, [X|L])* such that *L* is an integer list whose first element is larger than *X*. This property can be formalized as the following assertion:

true(insert(X,[Y|L],[X,Y|L])) ←
integer(X),
integer–list([Y|L]),
X < Y.

(It is assumed that $As(I_P)$ contains procedures with the obvious meaning for the predicate symbols of the body.)

Negative assertions. These are used to specify sets of atoms not valid in the intended model. We specify negative assertions using the predicate symbol *false*. If *false*(A') is a logical consequence of $As(I_P)$ and A' is an image of A then there exists a substitution θ, such that $A\theta \notin I_P$.

Note that YES and NO answers given by the ground oracle can be seen as singleton positive and negative assertions respectively.

Positive existential assertions. These are used to specify sets of atoms satisfiable in the intended model. We define positive existential assertions using the predicate symbol *posex*. If *posex*(A') is a logical consequence of $As(I_P)$ and A' is an image of A, there exists a substitution θ such that $A\theta \in I_P$.

Example 2.

The intended *isort* predicate of (Shapiro 1983) has the property that whenever it is called with the first argument being a list of integers and the second argument being an uninstantiated variable then there exists an instance of this call which is in the intended model. This can be formalized as the following assertion:

posex(isort(X,'VAR'(Y))) ← integer–list(X).

Negative existential assertions. These are used to specify sets of atoms unsatisfiable in the intended model. We define negative existential assertions using the predicate symbol *negex*. If *negex*(A') is a logical consequence of $As(I_P)$ and A' is an image of A then for all substitutions θ, $A\theta \notin I_P$.

Positive (negative) existential assertions generalize YES (NO) answers to Shapiro's existential queries. It is worth noticing that various notions of types for logic programs discussed in the literature e.g. (Zobel 1987), (Mycroft and O'Keefe 1984), (Nilsson 1983), can be seen as negative existential assertions (if an argument in an atom is of a wrong type then the atom should be unsatisfiable).

At every stage of development the program $As(I_P)$ should describe the intended model. A necessary condition for that is that it describes some model. This is not the case if, for example, both *true(A')* and *false(A')* are logical consequences of $As(I_P)$. The responsibility for providing consistent assertions is on the user (as it is the case with Shapiro oracle answers). The implementation discussed in the next section (partially) checks the consistency of $As(I_P)$ (Drabent et.al 1988).

3 DIAGNOSING ERRORS WITH ASSERTIONS

In this section we first set out the questions posed by the diagnosis algorithms and the way they are answered using assertions. Then we outline the principles of the diagnosing algorithms and comment on some design decisions adopted in our prototype system.

(1) Universal questions:

This type of question is asked by the incorrectness diagnoser:

"Is the atomic formula A valid in the intended model?" (i.e. are all its ground instances members of I_P?)

The insufficiency diagnoser requires answers to two additional types of questions:

(2) Existential questions:

"Is A satisfiable in the intended model?" (i.e. is there a ground instance of A which is a member of I_P?)

(3) Incompleteness questions:

The algorithm needs the information whether certain solved goals have produced all the expected answers in the intended model. This is obtained by asking:

"For the atom A, is there an instance $A\theta \in I_P$ such that $A\theta$ is not an instance of some member of the set $\{A\theta_1, \ldots, A\theta_n\}$?" (Substitutions $\theta_1, \ldots, \theta_n$ are (all the) computed answer substitutions for $\leftarrow$A and P).

The system uses the knowledge explicitly represented in $As(I_P)$ for answering the above questions before querying the user. Moreover, some queries to the user may be avoided by exploiting the information that is implicit in the assertions. For instance, it may happen that *true(A')* is a logical consequence of $As(I_P)$ but *posex(A')* is not (where A' is the image of an atom A in the ground representation scheme). However in this case the answer to the existential question for A is YES and querying the user is unnecessary. If A is an instance of some B then also the answer to the existential question for B is YES. Such properties are used by our question answering procedures.

To answer a universal question the system refers to the positive, negative and negative existential assertions of $As(I_P)$. Similarly, the positive existential, negative existential and positive assertions are used to answer an existential question. The details are given elsewhere (Drabent et al.1988).

If a question cannot be answered by referring to $As(I_P)$ then the user is queried. He may choose to answer with YES/NO, or to extend $As(I_P)$ by adding new clauses. The YES/NO answers to the incompleteness questions are to be provided by the user.

The knowledge implied by user YES/NO answers should be accumulated. For universal and existential questions this is done by adding new assertions to $As(I_P)$.

Two diagnosing algorithms are used:

(1) Incorrectness diagnosis

Our incorrectness diagnoser employs a modified version of Shapiro's algorithm (Shapiro 1983). The original algorithm finds an incorrect clause by systematic traversal of a ground proof tree whose root is not in I_P. In actual computations of logic programs the proof trees constructed need not be ground. The algorithm is extended here for nonground trees by exchanging the original ground oracle questions by universal questions (universal questions are a generalization of ground oracle ones since validity of a ground atom means its membership in Ip).

We use the top down version of Shapiro's basic algorithm as presented in (Sterling and Shapiro 1986) with this generalization. The queries posed by the algorithm are dealt with in the manner described above. The input to the algorithm is an atom A for which the program gives a wrong answer (this means a success instance of A which is not valid in I_P). The algorithm returns a (not necessarily ground) instance of a clause in P such that the atoms in the body of the clause are valid in I_P and the head is not.

(2) Insufficiency diagnosis

Our insufficiency diagnoser uses the algorithm formally introduced and proven correct in (Drabent et al. 1988). In contrast to the Shapiro's algorithm it does not require providing correct (i.e. valid in I_P) instances of a given atom. The input to the algorithm is an atom for which the program does not produce all the expected answers in the intended model Ip (and does not loop). As a result it gives a *not completely covered* atom; this means an atom A for which there exists an instance $A\theta$ in I_P such that no clause instance of P with all its body atoms in I_P has $A\theta$ as its head. For a given goal $\leftarrow$B (where B is the input) the algorithm examines the corresponding top-level procedure calls executed by the program. Existential questions are asked about calls that failed and incompleteness questions about calls that succeeded. Then the algorithm is called recursively with an atom for which the corresponding answer is YES. If all the answers are NO then A is returned as a not completely covered atom.

A prototype system employing assertions in the diagnosis of incorrectness and insufficiency has been implemented in Prolog. Our implementation of these algorithms delays queries to the user for as long as possible. For the incorrectness diagnoser, this means that the solved goals at each level of the proof tree are first subjected to assertions. If no assertions detect a false atom, then user queries are made about the remaining atoms at this level.

The insufficiency diagnoser first attempts to use assertions for answering the existential questions for top-level calls of a given goal. Only if this does not determine an atom for which a recursive call of the algorithm should be made, the user is queried, first with the remaining existential questions then with incompleteness questions.

Assertions made during a debugging session can be copied onto a file for future use.

4 EXPERIMENTS WITH THE DIAGNOSIS SYSTEM

In what follows we summarize the results obtained from diagnosis of a sample of programs using our implementation. The aims of the experiments were twofold: to establish the extent to which the use of assertions reduces interactions with the user, and to compare the two different approaches to insufficiency diagnosis: with and without instantiation of the atoms by the user.

To test the first aspect the sample programs were subjected to diagnosis by different systems: our system with some assertions, our system without any assertions, and an implementation of Shapiro algorithms (Sterling and Shapiro 1986). The number of queries made for a given diagnosis task were then compared. Columns 2, 4, and 6 in the table below summarize the results of this test on the program sample.

Note that the (Sterling and Shapiro 1986) programs do not record the results of queries, nor do they always generate the result of calls to built-in predicates. In order to get a fair comparison we have excluded the repetitive questions and the queries about system predicates from the counts of queries when using these programs.

Obviously the number of assertions provided by the user and the extent to which they specify the intended model can considerably affect the number of queries put to the user by our system. The number of assertions is indicated in column 7.

To test the second aspect, i.e. the relative ease with which the queries are answered by the user, we compare the number of instantiations made by the user when using the Shapiro algorithms, with the number of incompleteness questions asked using our system. These figures are presented in columns 3 and 5 of the table and are included in the total number of queries for each system (columns 2 and 4 respectively). The number of incompleteness questions is the same for versions with and without assertions.

In selecting the sample programs we were limited to pure logic programs. This limitation was somewhat

relaxed by inclusion of programs which make use of some Prolog built-in predicates (arithmetic, etc.). Such predicates are assumed correct and are not subject to examination by the diagnosing algorithms. The diagnosis was also unaffected by one safe use of Prolog negation on a correct predicate. The sample consists of 8 problems suitable for student exercises. The first program is the standard buggy "quicksort" (Shapiro 1982). Programs 2 and 7 are buggy versions of "substitute" and "wolf, goat and cabbage" from (Sterling and Shapiro 1986). The sample also includes 2 student programs which should compute the ways that a rectangle can be covered with a series of squares of given sizes (programs 3 and 4). Program 5 produces a list of (coordinates of) unary squares contained in a rectangle whose dimensions are given. Program 6 is a buggy "4Queens" program and the eighth program is to solve the famous "Missionaries and Cannibals" problem. The example session for diagnosing the *qsort* program is included in the appendix.

4.1 Discussion

Our experiments with the diagnosing system showed that reduction in queries put to the user can often be achieved by specification of simple properties of the intended model. However, at times it was not obvious whether it is worthwhile to give an assertion in reply to a query. It may for instance be difficult to formulate an assertion and/or a given assertion may not be applicable for future queries. Programs 3, 6, and 7 represent some such cases. The assertions formulated when testing the other sample programs were both easy to make and applicable to more than one query. This is not the case in general.

Our suggestion is that every time a universal or an existential question is asked, the user should consider whether an assertion can be easily formulated to describe a relevant property of the intended model in question. Such an assertion may or may not be applicable to subsequent queries in the same session. But it is a correct specification of some property of the intended model which may be used to reduce the number of queries when the program is further developed.

For a given incorrectness error, a reduction in the number of queries asked using our diagnoser (in comparison with Shapiro's) can only be achieved by introduction of appropriate assertions. However, assertions introduced earlier, e.g. for diagnosis of insufficiency, may result in a decrease in number of queries while detecting incorrectness. The diagnosis of the *qsort* program in our sample included one such case.

Although it may not be easy to answer *incompleteness* questions if the solutions to a goal are many and made up of large terms, answering these

KEY:

(1) error types present in the program: insufficiency/ incorrectness.
(2) atoms subjected to queries by the Shapiro algorithms.
(3) atoms instantiated by the user.
(4) queries asked by our system when no assertions are given.
(5) incompleteness questions asked by our system.
(6) queries by our system when some assertions are given.
(7) The number of assertions formulated.

Program		Shapiro algorithms		our algotihms without assertions		our algorithms with assertions	
	(1)	(2)	(3)	(4)	(5)	(6)	(7)
1.qsort	ins/inc	14	7	16	0	5	4
2.substitute	ins	7	3	6	1	2	2
3.sq1	ins	14	6	10	2	-	-
4.sq2	ins/inc	21	9	25	13	19	2
5.rectangle	ins/inc	13	5	10	0	5	3
6.queens	ins	15	6	7	0	-	-
7.wgc	ins	15	4	9	0	-	-
8.M&C	ins	29	8	18	0	14	1

Table: The number of user interactions on a test sample

questions are considerably easier than giving all correct instances of some goals (Shapiro 1983) (or instantiating whole clauses (Sterling and Shapiro 1986)). In many of our test cases it turned out to be very convenient that the user is not required to provide any goal instances. This was particularly the case in programs 3, 4 , 7 and 8.

Our experience shows that the present formulation of incompleteness questions can lead to user mistakes. To avoid such mistakes it may be more appropriate to query *completeness* of a set of answers instead of *incompleteness* of it.

During diagnosis of insufficiency providing a valid instance of an atom for which the actual program fails may considerably reduce the search space of the algorithm. This can be done in the present prototype implementation by simply starting the diagnoser with the goal instance as an argument. In any case, the decision whether a binding is to be given or not should be left to the user. Our algorithms can be easily extended with that option.

It may happen that a program is both incorrect and insufficient. An example is a program giving a wrong answer and missing a correct one. In such cases it is more convenient to perform incorrectness diagnosis first. The incorrectness diagnoser usually searches a smaller search space, does not ask incompleteness questions and produces more informative answers: an incorrect clause instance refers to a wrong clause while a not completely covered atom refers to a whole procedure. A particular case is when one of the atoms displayed by an incompleteness question is not valid in I_P. Then it is convenient to interrupt the insufficiency diagnosis and start diagnosing incorrectness with such an atom. This usually leads to a faster and more informative result.

5 COMPARISONS

The types of assertions introduced originate from the analysis of the logical nature of answers given by the oracles of Shapiro. They also have their counterparts in the algorithms of Ferrand (1987) and Lloyd (1987) where the oracles are represented by the predicates *valid* and *unsatisfiable* (and to certain extent *impossible* (Ferrand 1987)). But oracles have complete knowledge of the intended model while assertions only approximate it. A given atom may belong to none of the sets specified by *true* and *false* while the validity oracles of Ferrand and Lloyd can always decide its validity. However, the oracles are outside the system, while the assertions constitute a part of the system (which is incrementally developed during the external interactions).

The algorithms of (Sterling and Shapiro 1986), (Ferrand 1987) and (Lloyd 1987) require that the oracle is able to deliver elements of the intended model. If the oracle is the user, this type of interaction may create difficulties and increase the probability of giving wrong answers. One of our objectives has been to free the user from this burden. A similar approach is presented in (Pereira 1986). However, that work seems to rely on procedural semantics of Prolog, while ours has a clean logical foundation and the algorithms are proved correct and complete.

Another difference concerns the results produced by the debugger. Since we do not enforce the user to produce bindings during the debugging process the final result may come out less instantiated than in the other systems. The incorrectness diagnoser returns a clause with the body valid and the head not valid in I_P, while in most of the other approaches the head is unsatisfiable in I_P. The insufficiency diagnoser returns a not completely covered atom while in most of the other approaches it is an uncovered atom (For definitions and comparisons see Drabent et al. 1988).

The last difference to be mentioned concerns input to the algorithms. Usually it is supposed to be a wrong (not valid in I_P) instance of a goal produced by the program (in the case of incorrectness) or a finitely failed goal satisfiable in I_P (in the case of insufficiency). The algorithms used here allow a broader class of inputs. In the first case it is a goal for which a wrong answer is produced. In the second case it is a goal for which some answers are missing.

An earlier work using assertions within logic programming is (Drabent and Maluszynski 1987). Here assertions are used to prescribe a predicates call and success patterns. Preassertions in this sense describe all the predicate calls that are possible: those which succeed and those which fail. The described form of procedure calls is not expressible in terms of declarative semantics and is therefore, in general, not related to the assertions introduced in this paper. Nevertheless, it is possible to make use of such assertions in the debugging process by detecting inadmissible call patterns. We believe that this can be a generalization of Pereira's queries relating to admissibility of a goal (Pereira 1986), (Pereira and Calejo 1988).

6 CONCLUSIONS

We have demonstrated the use of assertions in algorithmic debugging. Assertions provide a formal description of some properties of the intended model, thus "approximating" it. They give a flexible framework for its formal description. On one end of the spectrum the yes/no oracle answers provide rudimentary but easy to produce information about the intended model. On the other end the full formal specification of the intended model can be used, if so desired. The system's knowledge is incrementally built up from users assertions. Assertions can be seen as

generalizations of the simple oracle answers and include them as special cases.

A prototype debugger using assertions has been implemented. Our experiments show that even the use of rather simple assertions may dramatically reduce the number of oracle interactions as compared with the Shapiro debugger. The new insufficiency diagnoser leads to simplified user interaction even in cases where no assertions are given.

The debugging process may start with a non-empty set of assertions. It is the user who decides the extent to which the intended model is described. Modifications of the initial assertions may be preserved from session to session. In this way the debugging process gives as a side effect an interactively developed formal description of some properties of the intended model.

6.1 Future work

An important extension of the method presented in this paper is to include such features of Prolog as the cut, negation, *setof* etc. Our intention is that they should be treated as declaratively as possible. Further experiments with Prolog programs are needed to better understand the debugging process, to fully evaluate the presented approach and to develop pragmatics of declarative debugging with assertions.

For constructs like *assert*, *retract* and input-output that depend very strongly on the execution algorithm, it may be impossible to include them into the declarative debugging framework. It may turn out that only methods based on operational semantics are applicable.

Another subject of future work is to discuss testing of logic programs and correcting of errors. The objective would be a testing-diagnosing-correcting methodology. It should be based on declarative features of existing logic programming languages and may be a complement to methods of systematic construction and verification of programs. Although proving programs correct seems to be a more important target, programs still need debugging and providing sound methods and tools for this is a significant research task.

AKNOWLEDGEMENTS

This report has been partially supported by the National Swedish Board for Technical Development, project number: 87-02926P, and a stipendium by The Royal Swedish Academy of Engineering Sciences (IVA). The first author was supported by Polish Academy of Sciences.

REFERENCES

Dershowitz, N., and Lee, Y., (1987) Deductive Debugging, *Proceedings of the Symposium on Logic Programming* - San Francisco : 298-306.

Drabent, W., and Maluszynski, J., (1987) Inductive Assertion Method for Logic Programs, *Proceedings of the International Conference on Theory and Practice of Software Development* (TAPSOFT), LNCS 250, Springer Verlag : 167-181.

Drabent, W., Nadjm-Tehrani, S., and Maluszynski, J., (1988) Algorithmic Debugging with Assertions, Research Report LiTH-IDA-R-88-04, (An abridged version appeared in: *proceedings for the Workshop on Meta-Programming in Logic Programming*, Bristol: 365-378).

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

Hill, P.M., Lloyd, J.W., (1988) Analysis of Meta-Programs, *proceedings of the workshop on Meta-Programming in Logic Programming*, Bristol: 27-42.

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

Mycroft, A., O'Keefe, R.A., (1984) A Polymorphic Type System for Prolog, *Artificial Intelligence* 23 : 295-307.

Nilsson, J.F., (1983) On the Compilation of a Domain-based Prolog, in: Mason, R.E.A.(ed), *Information Processing 83*, North Holland: 293-298.

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

Pereira, L.M., Calejo, M., (1988) A Framework for Prolog Debugging,in: Kowalski, R.A., Bowen, K.A., (ed), *Procceedings of the Fifth International Conference and Symposium on Logic Programming*, Seatle: 481:495.

Shapiro, E.Y., (1982) Algorithmic Program Debugging, *Proceedings of the 9th annual ACM Symposium on Principles of Programming Languages.*

Shapiro, E.Y., (1983) *Algorithmic Program Debugging*, MIT Press. .b2 **Sterling, L. and Shapiro, E.Y., (1986)** *The Art of Prolog*, MIT Press.

Zobel, J., (1987) Derivation of Polymorphic Types for Prolog Programs, in: Lassez, J.L., (ed), *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne: 817-838.

APPENDIX

The following example session demonstrates diagnosis of 4 errors present in the standard buggy quicksort program (Shapiro 1982). The other 2 errors in the standard buggy qsort are automatically detected by the Quintus Prolog parser. inc/1 and ins/1 are the incorrectness and insufficiency diagnosis procedures respectively. The assertions introduced by the user make use of library procedures integer_list/1, sorted/1, not_sorted/1, and permutation/2 for which the definitions are not given here. The user replies to queries consist of y(yes) n(no) and a(assert an assertion).

The buggy qsort is listed below:

```
%qsort([],[]).                          %error 1
qsort([X|L],L0) :-
        partition(L, X, L1, L2),
        qsort(L1,L3),
        qsort(L2,L4),
        append([X|L3],L4, L0).          %error 4

partition([X|L], Y, L1, [X|L2]) :-
        partition(L,Y,L1,L2).           %error 2
partition([X|L], Y, [X|L1], L2) :-
        Y =< X,                         %error 3
        partition(L,Y,L1,L2).
partition([],_X,[],[]).

append([X|L1], L2, [X|L3]) :-
        append(L1,L2,L3).
append([],X,X).

| ?- qsort([2,3,1,5],L).

no
| ?- ins(qsort([2,3,1,5],L)).
Is qsort([],B) satisfiable? a.
|: posex(qsort(L, 'VAR'(_))):-
        integer_list(L).

This atom is not completely covered:
qsort([],B)
```

%The session continues after amending the program %by inserting the clause qsort([],[]).

```
| ?- qsort([2,3,1,5],L).

L = [2,3,1,5]

yes
| ?- inc(qsort([2,3,1,5],L)).
Is qsort([2,3,1,5],[2,3,1,5]) true? a.
|: negex(qsort(_,L)):-
        integer_list(L),
        not_sorted(L).
Is append([3],[1,5],[3,1,5]) true? y.
Is qsort([1,5],[1,5]) true? a.
|: true(qsort(L1,L2)):-
        integer_list(L1),
        integer_list(L2),
        permutation(L1,L2),
        sorted(L2).
Is partition([1,5],3,[],[1,5]) true? n.
Is partition([5],3,[],[5]) true? y.

This is an incorrect clause:
partition([1,5],3,[],[1,5]) :-
        partition([5],3,[],[5]).

L = [2,3,1,5]
```

%The user includes a test in the first clause of %partition and the session continues:

```
| ?- listing(partition).

partition([A|B],C,D,[A|E]) :-
        A>C,
        partition(B,C,D,E).
partition([A|B],C,[A|D],E) :-
        C=<A,
        partition(B,C,D,E).
partition([],A,[],[]).


yes
| ?- qsort([2,3,1,5],L).

no
| ?- ins(qsort([2,3,1,5],L)).
Is partition([3,1,5],2,B,C) satisfiable? a.
|: posex(partition(L,X,'VAR'(_),'VAR'(_))):-
        integer_list(L),
        integer(X).

This atom is not completely covered:
partition([1,5],2,B,C)
```

%Now the test in the second clause of partition is %reversed to correct the procedure

```
| ?- qsort([2,3,1,5],L).

L = [2,1,3,5] ;

no
| ?- inc(qsort([2,3,1,5],L)).
Is append([2,1],[3,5],[2,1,3,5]) true? y.
Is partition([3,1,5],2,[1],[3,5]) true? y.
This is an incorrect clause:
qsort([2,3,1,5],[2,1,3,5]) :-
        partition([3,1,5],2,[1],[3,5]),
        qsort([1],[1]),
        qsort([3,5],[3,5]),
        append([2,1],[3,5],[2,1,3,5]).
```

```
%The user corrects the call to append in the second
%clause for qsort and the last error is removed:

| ?- qsort([2,3,1,5],L).

L = [1,2,3,5]

%The system has accumulated some properties of the
%intended model of the program in the form of
%assertions.

| ?- listAssertions.

posex(qsort(A,'VAR'(B))) :-
        integer_list(A).

posex(partition(A,B,'VAR'(C),'VAR'(D))) :-
        integer_list(A),
        integer(B).

negex(qsort(A,B)) :-
        integer_list(B),
        not_sorted(B).

true(append([3],[1,5],[3,1,5])).

true(qsort(A,B)) :-
        integer_list(A),
        integer_list(B),
        permutation(A,B),
        sorted(B).

true(partition([5],3,[],[5])).

true(append([2,1],[3,5],[2,1,3,5])).

true(partition([3,1,5],2,[1],[3,5])).

false(partition([1,5],3,[],[1,5])).
```