LiTH-IDA-R-88-04 ISSN-0281-4250

INSTITUTIONEN FÖR DATAVETENSKAP
UNIVERSITETET OCH TEKNISKA HÖGSKOLAN
LINKÖPING

Algorithmic Debugging with Assertions

by

Wlodek Drabent*
Simin Nadjm-Tehrani
Jan Maluszynski

*Institute of Computer Science
Polish Academy of Science
Warszawa, Polen

RESEARCH REPORT
LOGPRO, March 1988

Postadress:
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

Mailing address:
Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

Computer and Information Science Department
Linköping University

PhD theses:

* Available at: University of Microfilms Intl., 300 N. Zeeb Road, Ann Arbor, MI 48106, USA.

(*Linköping Studies in Science and Technology. Dissertations.*)

*No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982.

*No 94 **Hans Lunell:** Code Generator Writing Systems, 1983.

*No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983.

*No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984.

*No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons. 1987.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning. 1987.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems. 1987.

Licentiate of engineering Theses:

(*Linköping Studies in Science and Technology. Theses.*)

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching. 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems. 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of an Architecture based on Asynchronous Processes. 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System. 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems. 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs. 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behaviour of the Greedy Triangulation. 1986.

No 104 **Shamsul I. Chowdhury:** Statistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology. 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-based Editing. 1987

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments. 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge. 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs. 1987

No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs. 1987

No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems. 1987

LiTH-IDA-R-88-04 ISSN-0281-4250

Algorithmic Debugging with Assertions

by

Wlodek Drabent

Simin Nadjm-Tehrani *

Jan Maluszynski

Abstract: Algorithmic debugging, as presented by Shapiro, is an interactive process where the debugging system acquires knowledge about the expected behaviour of the debugged program and uses it to localize errors. This paper suggests a generalization of the language used to communicate with the debugger. In addition to the usual "yes" and "no" answers formal specifications of some properties of the intended model are allowed. The specifications are logic programs. They employ library procedures and are developed interactively in the debugging process. An experimental debugging system incorporating this idea has been implemented. In contrast to some other systems, its insufficiency diagnoser does not require instantiation of unsolved goals by the oracle. A formal proof of correctness and completeness of this algorithm is presented.

This work was supported by STU, the Swedish National Board for Technical Development.

* é–mail: simin@ ida.liu.se

Postadress:
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

Mailing address:
Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

Algorithmic debugging with assertions

Wlodek Drabent
Simin Nadjm-Tehrani
Jan Maluszynski

1.Introduction

This paper deals with diagnosis of logic programs and extends the pioneering work of Shapiro [Sh83] by studying the role of assertions in debugging of logic programs. Algorithmic debugging, as presented by Shapiro, is an interactive process where the debugging system acquires knowledge about the expected behaviour of the debugged program and uses it to localize errors. This paper analyzes the language used to communicate with the debugger and suggests its generalization. The objective is to provide the debugging system with high-level descriptions of program properties rather than with examples of concrete bindings to be computed by the program.

Logical foundations of algorithmic debugging can be found in [Fe87] and [Ll87]. Our basic notions, though slightly different, have been strongly influenced by these papers. (For discussion of the differences see Section 7).

Every (pure) logic program P has a model [AE82]. P is often considered to be the specification of the least Herbrand model Mp. 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 Ip of the Herbrand base. If Ip differs from Mp the program is erroneous.

The program P is said to be:

- incorrect iff Mp - Ip $\neq \phi$, i.e. iff it specifies some element which is not in the intended model.

- insufficient iff Ip - Mp $\neq \phi$, i.e. iff some elements of the intended model are not specified by the program.

In this paper we do not deal with the termination aspect; we concentrate on tracing incorrectness and insufficiency of a logic program.

The objective of debugging is to find a cause of an error in the program. In the case of incorrectness it is a clause which "produces" elements not in Ip. More precisely, it is a clause whose body is valid in Ip and whose head is not valid in Ip. Such a clause will be called *incorrect*. It has a ground instance, such that all atoms of its body are in Ip and its head is not in Ip.

In the case of insufficiency it is a predicate p for which some atom p(t1,...,tn) valid in Ip can not be produced by the clauses defining p. More precisely, there is no ground instance H ←B1,...,Bm (m≥0) of a clause of P such that H is an instance of p(t1,...,tn) and B1,...,Bm are in Ip. Such an atom is called *uncovered*.

The elements of Mp 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 formalised. To solve this problem Shapiro introduces the concept of oracle to compare the results of computations with the intended model. In practice, it is the user who answers the questions concerning the intended model.

Shapiro pointed out that incorporation of "constraints and partial specifications" into the algorithmic debugging scheme may reduce the number of interactions with the user [Sh83 p.79]. This paper formalizes and explores this idea. 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 are developed interactively in the debugging process. The diagnosis system uses its actual 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 systems knowledge about the intended model. This scheme includes as special cases the answers used in Shapiros 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 dramatically as will be illustrated in the paper.

As it is unlikely that the complete specification of the model is conveniently provided by the user, there will be a number of interactions between the system and the user. Another aspect of our debugging methodology is the relative ease with which the user can interact with the system. Our insufficiency diagnoser, in contrast to Shapiros, will not require the user to provide instances of unsolved goals. Instead, the user is expected to recognise the solutions to a goal and to identify a case where some answer is missing. This reduces the likelihood of mistakes made by the user in the process of instantiation and results in a more reliable implementation of the oracle.

The rest of the paper is organized as follows. Section 2 outlines the idea of algorithmic debugging. Particular attention is devoted to the language used to communicate with Shapiro's debugger. We also discuss some limitations of the debugger. In Section 3 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 4 describes the necessary oracle interactions. In section 5 debugging algorithms based on assertions are presented. The section also includes a formal proof of correctness and completeness of our insufficiency diagnosis algorithm. In section 6 some implementation issues are discussed. Conclusions, comparisons with related work and topics for future research are presented in Section 7. Examples of debugging sessions as compared with corresponding sessions based on Shapiro algorithms are given in the appendix.

2. Shapiro's algorithmic debugging of logic programs

This section gives a brief survey of Shapiro's algorithmic debugging [Sh83]. To avoid technical details the discussion is restricted to "pure Prolog" i.e. the programs are assumed to consist of definite clauses, possibly including calls of

standard arithmetic procedures.
The cornerstone of the method is the concept of oracle which interactively provides the debugging system with needed information about the intended model. The information used to trace incorrectness is different from that used for tracing insufficiency. We now outline seperately these cases.

The basic idea of tracing incorrectness is as follows. The input is a ground proof tree whose root is not in the intended model. Such a tree consists of a finite number of ground instances of program clauses pasted together. It must include a ground instance of an incorrect clause (i.e. an istance such that all atoms of the body are in Ip and the head is not in Ip). The objective is to find it. A systematic traversal of the proof tree solves the problem with the help of an oracle. The oracle decides whether the traversed nodes are in the intended model or not, thus effectively specifying the model.

In actual computations of logic programs the proof trees constructed need not be ground. The idea can be extended for non-ground trees in two different ways. The suggestion of [SS86 p.325] is that the oracle should instantiate the visited node, if possible to an instance not included in the intended model, otherwise to any instance. Another possible solution used in this paper is to require that the oracle decides whether the traversed node is valid in the intended model or not. It is a generalization of the original *ground oracle* since validity of a ground atom in Ip means that it is an element of Ip.

Example 1.

As a reference for comparison with our method we quote from [Sh83] the following example of tracing incorrectess in a buggy program.

isort([X|Xs], Ys) ← isort(Xs,Zs), insert(X,Zs, Ys).
isort([],[]).

insert(X,[Y|Ys], [Y,Zs]) ← Y > X, insert(X, Ys,Zs).
insert(X,[Y|Ys], [X,Y|Ys]) ← X ≤ Y.
insert(X,[],[X]).

|? - isort([2,1,3],X).

X = [2,3,1]
yes

The following record of interactions between the debugger and the user leads to identification of an incorrect clause, whose instance is returned as the answer.

|? - fp(isort([2,1,3],[2,3,1]), C).

query: isort([],[])? y.

query: insert(3,[],[3])? y.

query: isort([3],[3])? y.
query: insert(1,[],[1])? y.

query: insert(1,[3],[3,1])? n.

C = (insert(1,[3],[3,1]) ← 3 > 1, insert(1,[],[1]))
yes

End of example.

The insufficiency errors are diagnosed with the help of an *existential* oracle. The oracle is capable of answering the following question types:

- is there a solution to a given goal? (the goal need not be ground). Logically, this is the question whether the goal is satisfiable in the intended model.

- if yes, and the goal includes variables give such a ground instance of it which represents a solution (i.e. produce an element of the intended model which is an instance of the goal). It may happen that several such instances of an atom are required.

The algorithm for tracing insufficiency is invoked with an atom A in Ip for which P fails. It attempts to find a clause H←B of P such that H unifies with A with unifier δ and Bδ is satisfiable in Ip. To check satisfiability the existential oracle is used. If the check is successful, the result is a ground instance Hθ←Bθ of the clause such that Bθ is true in Ip. If there is no such clause then A is an uncovered atom and is returned as the result. Otherwise Bθ includes an atom A' in Ip for which P fails. The algorithm is called recursively with A'.

Example 2.

As a reference for comparison with our method we quote from [Sh83] the following example of tracing insufficiency in a buggy program.

isort([X|Xs], Ys) ← isort(Xs,Zs), insert(X,Zs, Ys).
isort([],[]).

insert(X,[Y|Ys], [Y,Zs]) ← X > Y, insert(X, Ys,Zs).
insert(X,[Y|Ys], [X, Y|Ys]) ← X ≤ Y.

|? - isort([3,2,1],X).

no

Shapiros insufficiency diagnosis procedure *ip* returns an uncovered atom X as follows:

|? - ip(isort([3,2,1],[1,2,3], X).

query: isort([2,1],X)? y.
which X? [1,2].

query: insert(3,[1,2],[1,2,3])? y.
query: isort([1],X)? y.
which X? [1].

query: insert(2,[1],[1,2])? y.

query: isort([],X)? y.
which X? [].

query: insert(1,[],[1])? y.

X = insert(1,[],[1])
yes

End of example.

We conclude our brief survey with the following remarks:

The debugger 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 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. Therefore frequent interactions with the oracle become necessary. As it can be seen in Example 1 the questions often concern recursive calls of the same procedure (in this case *insert*) with the same kind of arguments. However, the communication language does not allow to formulate a more general property sufficient to answer many similar queries without repeated interactions.

It is required that the oracle can produce all solutions of uninstantiated goals. In the case when input and output data are large terms and the oracle is the user this requirement seems to be unrealistic.

3. Assertions

We suggest to extend the communication language of the algorithmic debugger. In addition to the simple YES and NO answers we want to 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.

Let S be a set of atoms in the language L of a given logic program P. The objective is thus to construct a logic program Q with a unary predicate s "specifying" set S. The clauses of Q beginning with the symbol s will be called assertions.

More precisely, there should be a one-to-one correspondence between set S and the set of all ground atoms of the form s(t) which are logical consequences of Q. In other words, each atom A of L should be coded as a ground term A' of the language M of the program Q. We adopt here the following coding scheme:

- if A is a ground atom then the image of A is A ,
- if A is a non ground atom and $X_1,...,X_n$ are variables occurring in A , the image of A is A'= $A[X_i/var(i)]$ $(1 \leq i \leq n)$ where var is a functor not used in the object language. (To obtain uniqueness it may be assumed that $X_1,...,X_n$ are ordered according to their first occurrences in A).

Notice that all the predicate symbols and functors of L become functors of M. Atoms which are not the same up to variable renaming have different images.

The debugging algorithms of section 5 refer to four properties of intended model Ip. The properties are discussed below. They are specified in the above-mentioned sense by four fixed predicate symbols in a program As(Ip). The ways the algorithms use As(Ip) are described in section 4.

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 debugged program [DL87]. Such a solution is completely unrealistic in most cases. There is no reason to believe that the user who has just written a buggy program is able to give a complete formal specification which properly reflects all requirements. It is often suggested e.g. [SS86] that while developing a new version of an existing program the existing version can be used as an oracle. 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. [SS86 p. 323] mentions permutation sort and quicksort [SS86 p.55,56] as an example. However, quicksort contains procedures partition and append that are not specified by the permutation sort 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. This gives rise to the following 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(Ip) and A' is an image of A then for all substitutions θ, $A\theta \in Ip$ provided $A\theta$ is ground.

Example 3.

Consider the intended relation *insert* of Example 1. It includes (as a proper subset) all triples *(x,y,z)* such that *x* is an integer, *y* is a sorted list of integers and *z* is a sorted list whose elements are *x* and all elements of *y*. This property can be formalised as the following assertion:

true(insert(x,y,z)) ←
    is–integer(x),
    is–sorted–integer–list(y),
    is–sorted–integer–list(z),
    permutation([x|y],z).

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

This type of assertions will be used for tracing incorrectness. It is worth noticing that the YES answers of Example 1 can be seen as equivalent to singleton positive assertions. The assertion of Example 3 includes all YES answers of Example 1.

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(Ip) and A' is an image of A then there exists a substitution θ, such that Aθ is ground and A$\theta \notin$ Ip.

Example 4.

The NO answer of Example 1 specifies a single atom which is not valid in the intended model. This can be formalized as the negative assertion

false(insert(1,[3],[3,1])).

A more general assertion characterizes an infinite set of atoms non-valid in the intended model:

false(insert(X,[Y|Z1],[Y|Z2])) ← *X<Y* .

This set includes the queried atom in Example 1.

The intended relation *insert* of Example 1 may be such that the third component of any of its elements is a sorted integer list. In our language this can be formalized as the following assertion:

false(insert(X,Y,Z)) ← *not–sorted–integer–list(Z)*.

The algorithm for tracing incorrectness presented in Section 5 allows for inserting positive assertions (including YES answers) and negative assertions (including NO answers). Interaction with the user takes place only in the case when the actual query cannot be answered by the assertions in the actual database.

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(Ip) and A' is an image of A then there exists a substitution θ such that Aθ is ground and A$\theta \in$ Ip.

Example 5.

The intended *isort* predicate of Example 2 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))) ← is–integer–list(X)

The positive existential assertions generalize YES answers to existential queries.

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(Ip) and A' is an image of A then for all substitutions θ, $A\theta \notin$ Ip provided $A\theta$ is ground.

Example 6.

The intended *isort* predicate of Example 2 has the property that whenever it is called with the first argument being the empty list and the second argument being a non-empty list (possibly with variables) then none of the instances of the call is in the intended model. In the language of assertions this can be formalized as follows:

negex(isort([ ],y)) ← is–nonempty–list(y).

The negative existential assertions generalize NO answers to existential queries. It is worth noticing that various notions of types for logic programs discussed in the literature e.g. [Zo87], [MO84], [Ni83], can be seen as negative existential assertions (if an argument in an atom is of a wrong type then the atom is unsatisfiable).

The algorithm for tracing insufficiency presented in Section 5 allows for inserting positive existential assertions (including YES answers to an existential query) and negative existential assertions (including NO answers). Interaction with the user takes place only in the case when the actual query cannot be answered by the assertions in the actual database.

This is not the case if, for example, both *true(A')* and *false(A')* are logical consequences of As(Ip). That corresponds to A being both valid and not valid in Ip. The responsibility for providing consistent assertions is on the user. In the next section it is described how (partial) consistency checking of As(P) is performed. Notice, that even the basic debugging algorithms are not free of the danger of inconsistent answers. Let A be an atom with variables and B its ground instance. When tracing incorrectness the answer concerning B may be YES, i.e. B is in Ip. Independently, when tracing insufficiency the answer concerning A may be NO, i.e. there is no instance of A in Ip.

4.Oracle interactions

In order to show how the assertions are used, we set out the questions that are posed by the diagnosis algorithms and the way they are answered. Then we describe the actual improved interaction procedure between the algorithms and As(Ip). The ground image of atom A in the coding scheme used by assertions is denoted by A'.

(1) Universal questions:

This type of question is asked by the incorrectness diagnoser:

"Is the atomic formula A valid in the intended model?" (That means whether all its ground instances are members of Ip.)

- If *true(A')* is a logical consequence of As(Ip) then the answer to this question is YES.
- If *false(A')* is a logical consequence of As(Ip) then the answer to this question is NO.
- If neither *true(A')* nor *false(A')* are logical consequences of As(Ip) then the user has to be queried. She may choose at this point to augment As(Ip) by adding a new assertion. Alternatively, she may reply with YES or NO. If an assertion is added, the universal question is retried with augmented version of As(Ip).

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

(2) Existential questions:

"Is A satisfiable in the intended model?" (That means there exists a ground instance of A which is a member of Ip.)

- If *posex(A')* is a logical consequence of As(Ip) then the answer to this question is YES.
- If *negex(A')* is a logical consequence of As(Ip) then the answer to this question is NO.
- If neither *posex(A')* nor *negex(A')* are logical consequences of As(Ip) then the user has to be queried. At this point she may choose to augment As(Ip) by adding a new assertion. Alternatively, she may reply with YES or NO. If an assertion is added the existential question is retried with the augmented version of As(Ip).

(3) Incompleteness questions:

The algorithm also requires to consider whether certain solved goals have produced all the expected answers in the intended model. This is done by asking:

"For the atom A, is there such an instance $A\theta \in Ip$ 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 ←Aθ and P).

A YES or NO answer has to be provided by the user.

The question answering procedure described above can be improved to decrease the number of questions posed to the user. Some questions 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(Ip) but *posex(A')* is not. However in this case the answer to the existential question for A is YES and querying the user is unnecessary.

Let A be an atom and B its instance. The following properties hold:

(1) If A is valid in Ip then it is also satisfiable in Ip.

(2) if A is unsatisfiable in Ip then it is not valid in Ip.

(3,4) If atom A is valid (unsatisfiable) in Ip then B is also valid (unsatisfiable) in Ip.

(5,6) if B is not valid (satisfiable) in Ip then A is not valid (satisfiable) in Ip.

An improved procedure for answering universal questions employs properties (2) and (6). For atom A:

- If *true(A')* is a logical consequence of As(Ip) then the answer to this question is YES.
- If *false(B')* or *negex(B')* is a logical consequence of As(Ip) for some B being an instance of A then the answer to this question is NO.
- Otherwise the user is queried.

An improved procedure for answering existential questions employs properties (1) and (5). For atom A:

- If *posex(B')* or *true(B')* is a logical consequence of As(Ip) for some B being an instance of A then the answer to this question is YES.
- If *negex(A')* is a logical consequence of As(Ip) then the answer to this question is NO.
- Otherwise the user is queried.

The properties (3) and (4) are not used by the answering procedures due to implementation difficulties. However, if the decoded set of atoms defined by *true* (resp. *negex*) is closed under substitution then employing properties (3) and (4) does not change anything. In practice, these two sets are closed under substitution in every reasonable As(Ip).

The answering procedures require checking whether there exists an instance B of a given atom A such that *Assertion(B')* is logical consequence of As(Ip). To do this, program As(Ip) is queried with the goal ←*Assertion(A)* (note: not coded A). This is because coded image of any instance of A is also an instance of A and if a coded image of a term is an instance of A then the term is an instance of A.

If, according to the above-mentioned procedures, both YES and NO is the answer to a question then As(Ip) is inconsistent and the debugging algorithm

is aborted.

To accumulate the knowledge implied by user answers to universal and existential queries, new assertions can be added to As(Ip) by the system. If the user answer to the universal query for A is NO then assertion

false(A') ←

is added.

If the user answer to the universal query for A is YES then any instance of A is valid in Ip. So assertion

true(A) ←

is added. Now for any instance B of A, *true(B')* is logical consequence of As(Ip) (since coded image of any instance of A is also an instance of A and vice versa).

If the user answer to the existential query for A is YES then assertion

posex(A') ←

is added.

If the user answer to the existential query for A is NO then assertion

negex(A) ←

is added (since any instance of A is unsatisfiable in the intended model).

User answers to incompleteness questions can also be recorded. If the answer is NO then unary clause *complete-solutions(A)* is recorded. Then a success of ←*complete-solutions(B')* implies a NO answer to the incompleteness question concerning B since the answer is NO for any instance of A. (It turns out that recording of YES answers is unnecessary.)

5. Diagnosis algorithms

(1) Incorrectness diagnosis

If the SLD-resolution of a goal ←A with program P produces a substitution θ such that Aθ is not valid in Ip, then an incorrect clause instance has to be found. We use the top down version of Shapiro's basic algorithm as suggested in [SS86] with the difference that we do not require the queried goals to be ground. The queries posed by the algorithm are dealt with in the manner described in section 4. The user of this algorithm has therefore the benefit of using a spectrum of assertion levels. At the one end of the spectrum lies the low level assertions characterized by Shapiro's yes and no answers, and at the other end the possibility to use a complete specification. The systems knowledge is incrementally built up from users assertions. 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 Ip and the head is not.

(2) Insufficiency diagnosis

We will say that program P is *insufficient* for ←A if there exists θ such that A$\theta \in$ Ip and no answer more general than θ is a computed answer substitution

for $\leftarrow$A and P. An atom A is *completely covered* by program P if for every θ such that $A\theta \in Ip$, P contains a clause which has an instance where $A\theta$ is the head and all the atoms in its body are in Ip. Note that every not completely covered atom has an instance in Ip.

If a program P is insufficient for a goal $\leftarrow$A then the insufficiency diagnoser is called to identify an atomic formula C not completely covered by the program P. Since our diagnosis algorithm is different from the original Shapiro algorithm in that it does not require instantiation of goals in reply to existential queries, we include a description of the algorithm. In the algorithm, it is assumed that Prolog computation rule is used in the resolution.

Before describing the algorithm we introduce the following definition:

The **search forest** for A consists of a tree for each non unary clause of P whose head is unifiable with A. Let $H \leftarrow B_1, ..., B_n$ $(n > 0)$ be a variant of such a clause.

Then

(B_1, γ), where γ is an mgu of H and A, is the root of the corresponding tree

and

if (B_i, θ) is a node of the tree, and program P gives $\{\sigma_1, ..., \sigma_m\}$ $(m \geq 0)$ as computed answer substitutions to goal $B_i\theta$

then $(B_{i+1}, \theta\sigma_j)$ for $j = 1,...,m$ is a child of this node if $i < n$

and $(\Box, \theta\sigma_j)$ for $j = 1,...,m$ is a child of this node if $i = n$.

Note that (B_i, θ) is a node in the forest iff B_i instantiated to $B_i\theta$ is a selected goal (on the top level) in the computation for A. Note also that $(\Box, ...)$ leaves correspond to successes of $\leftarrow$A. If $(\Box, \theta)$ is a leaf in the forest then goal $\leftarrow$A succeeds with computed answer substitution θ/variables(A) (where variables(A) stands for the set of variables occurring in A and θ/S stands for the restriction of θ to the elements of S). For a given A, the search forest is unique up to variable renaming.

The Algorithm

The input to the algorithm is an atomic formula A for which the program is insufficient and the computation for $\leftarrow$A is finite; the output is a not completely covered atom.

The insufficiency diagnoser asks questions about the nodes of the search forest for A. The types of questions asked have been discussed in section 4. (The order of visiting the nodes is irrelevant to the correctness of the algorithm).

For (B, θ) being a leaf of the forest, $B \neq \Box$, the existential question is asked about $B\theta$. If the answer is YES then the algorithm is recursively called with $B\theta$. (No questions are asked about a success leaf.)

For (B, θ) being an internal node with children (C, $\theta\sigma_1$), ..., (C, $\theta\sigma_m$) (where C is an atomic formula or □), the incompleteness question is asked about the set {B$\theta\sigma_1$, ..., B$\theta\sigma_m$} and the goal Bθ. If the answer is YES then the insufficiency diagnoser is called recursively on Bθ.

If for all nodes of the forest the answers for all the questions are NO, then A is returned as a not completely covered atom and the algorithm terminates. Otherwise, a not completely covered atom is found by the recursive call(s) of the algorithm.

Correctness and completeness of the algorithm

Lemma

Consider program P and a search forest for atom C.

If

for every node in the forest the answer to the question asked by the algorithm is NO and

C is completely covered by P

then

P is sufficient for ←C.

Proof

Let C$\gamma \in$ Ip (without loss of generality it may be assumed that the domain of γ is variables(C)). We show that there exists a computed answer substitution for C that is more general than γ. As C is completely covered by P, there exists

$$A \leftarrow B_1, ..., B_n \qquad (*)$$

which is a variant of a clause of P and there exists a substitution δ such that

$C\gamma = C\delta = A\delta$,

$B_1\delta,...,B_n\delta \in$ Ip.

If n = 0 then an mgu of C and A (restricted to variables(C)) is the required computed answer substitution. Assume n > 0. We show that in the search forest for C, in the tree corresponding to (*) there exists a leaf (□, θ) such that θ is more general that δ. Assume that this does not hold. Note that for the root (B_1, θ_1) substitution θ_1 is more general that δ. Let i be the greatest number for which there exists a node (B_i, θ) in the tree where θ is more general than δ. Then $B_i\delta$ is an instance of $B_i\theta$. Two cases are possible.

1. (B_i, θ) is a leaf of the tree. The answer to the existential question about $B_i\theta$ is YES (as $B_i\delta \in$ Ip). Hence contradiction.

2. (B_i, θ) has sons (C, $\theta\sigma_1$), ..., (C, $\theta\sigma_m$) where C = B_{i+1} or C = □. As the answer to the incompleteness question about $B_i\theta$ is NO, for some j substitution $\theta\sigma_j$ is more general that δ. Contradiction.

Now, the computed answer substitution corresponding to (□, θ) is θ / variables(C). This solution is more general than δ / variables(C) = γ. This concludes the proof.

As the algorithm is (recursively) called for atom C only if P is insufficient for C, from the Lemma it follows that the atom returned by the algorithm is not

completely covered by P. Hence the algorithm is correct.
Notice that each search forest is finite and the recursion depth of the algorithm is finite (otherwise the computation for ←A would be infinite). Thus the algorithm always returns an answer. So the algorithm is also complete in the sense that it returns a correct answer for any atom satisfying input conditions.

6. Implementation

The algorithms described in section 5 have been implemented in Prolog. The implementation employs assertions according to the improved procedures described in section 4. In addition, queries to the user are delayed as much as possible.

For the incorrectness diagnoser, this means that the solved goals in 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 a given search forest. Only if this does not determine a node for which a recursive call of the algorithm should be done, the user is queried, first with the remaining existential questions then with incompleteness questions.

User answers are recorded to avoid repeating of questions. The answers to universal and existential questions and NO answers to incompleteness questions are dealt with as described in section 4. YES answers to incompleteness questions are not recorded because YES-answered incompleteness questions are not repeated by our algorithm. More precisely, let A be an instance of B. If the algorithm asks the incompleteness question for A and the answer is YES then no question is asked about B later on (otherwise the computation for A would have been infinite).

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

Example debugging sessions are given and discussed in the appendix. Basically, they confirmed that assertions can significantly reduce the number of queries posed to the user. Assertions that only roughly approximate the intended model proved to be able to generate answers to a significant deal of questions made by the algorithms.

It also turned out to be very convenient that the user is not required to provide any goal instances. On the other hand, in the insufficiency diagnoser providing a valid instance for which the actual program fails may drastically 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.

To fully evaluate and further develop the ideas presented in this paper, more experience with using the debugger is needed. We plan to make experiments with debugging of student programs.

7. Conclusions and Comparisons

Assertions

The main contribution of this paper is formalization of the concept of assertion for 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, but in most cases its construction is practically unfeasible.

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. The examples of the appendix show that even the use of rather simple assertions may dramatically reduce the number of oracle interactions as compared with Shapiro debugger.

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 [Fe87] and Lloyd [Ll87] where the oracles are represented by the predicates valid and unsatisfiable (and to certain extent impossible [Fe87]). But the oracles have complete knowledge of the intended model while the assertions only approximate it. For example the assertions true and false provide an incomplete information about validity of a given atom in the intended model. The first of them specifies a set of atoms valid in the intended model, the other a set of atoms non-valid in the intended model. A given atom may belong to none of the sets 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). External interactions are necessary in our system only if the actual assertions cannot produce the required answer. In this case the external interaction provides an increment for the existing assertions so that the question can be answered.

It is worth noticing that the concept of assertion is orthogonal to the concept of debugging algorithm: any debugging algorithm based on oracle interactions can also use appropriate assertions.

The debugging process may start with non-empty set of assertions. It is up to the user how much she wants to describe the intended model. 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.

An earlier work using assertions within logic programming is [DM87]. 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 may not be expressible in terms of declarative semantics and are 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 [Pe86].

Oracle questions

The algorithms of Shapiro [Sh83] [SS86], Ferrand [Fe87] and Lloyd [Ll87] 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 or even lead to wrong answers. One of our objectives has been to free the user from this burden.

Our algorithm for incorrectness diagnosis is similar to that suggested in [SS86]. However, it works on non-ground proof trees so that (in contrast to the remark in [SS86] p. 325) the user is not asked to instantiate a queried goal when it contains variables.

A new algorithm for insufficiency diagnosis presented in this paper generates automatically answers for atomic subgoals. Instead of generating bindings the user is (sometimes) asked whether the set of generated answers is complete. A similar approach is presented by Pereira [Pe86]. However that work seems to rely on procedural semantics of Prolog, while ours has a clean logical foundation and our algorithm is proved correct and complete.

Clearly, the bindings provided by the user can speed-up the diagnosis process. However, 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.

Comparisons

There are some differences in basic definitions used in this paper and the papers by Ferrand and Lloyd which give logical foundations for declarative debugging.

We follow Lloyd in that that our intended model is a ground Herbrand model in contrast to the nonground term model of Ferrand. In contrast to Lloyd we require that the intended model of a program without errors is its least Herbrand model. It seems natural to expect that a correct program P when called with any goal $\leftarrow$ A, where A is in the intended model I, will terminate. Our requirement is necessary for that. Thus, our notion of correct program is more restrictive: we want that I is the least fixpoint of Tp while both Ferrand and Lloyd want I to be some fixpoint of Tp.

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. To be more precise consider separately the form of our results in diagnosing incorrectness and insufficiency.

For incorrectness, the result returned is an incorrect instance of a program clause, that is H ← B such that B is valid in I and H is non-valid in I.

This is similar to Ferrand's definition of incorrectness ([Fe87] Definition 4). However, his debugger is only able to return such H ← B that B is valid and H is unsatisfiable. This is due to representing variables of the program by variables of the debugger. The results produced by the debugger of Lloyd have also this property. The approaches are equivalent, since any incorrect (in our terminology) clause has an instance where the head is unsatisfiable and the body valid.

For insufficiency diagnosis the situation is similar. The results produced by our debugger are atoms which are not completely covered while Lloyd's debugger produces uncovered atoms. (An atom A is called uncovered if A is valid in I and none of its instances is in Tp(I)). Comparing the definitions one can see that every not completely covered atom has an instance which is uncovered. This instance is not produced by our system. This is because we do not enforce the user to produce bindings for subgoals during the debugging process. Ferrand's notion of insufficiency is a counterpart of Lloyd's uncovered atom but it is weaker than the latter (A is an insufficiency if A is valid in I and not all its instances are in Tp(I).) However, the answers really produced by the algorithm are similar to those of Lloyd. More precisely, in both cases the result of insufficiency diagnosis is an uncovered atom.

The last difference to be mentioned concerns inputs for insufficiency diagnosis. Usually it is supposed to be a finitely failed goal which is satisfiable in the intended model. However, the finite failure of this goal may be caused by the fact that some subgoal of the computation does not fail but produces insufficient number of answers. In most systems this situation is handled by asking the oracle to provide all intended answers for the subgoal. The answer not produced by the insufficient program will cause its failure and eventual localization of insufficiency. Our system does not require the user to provide correct subgoal instances. This also results in extending the allowed inputs for the diagnoser: the input is a goal whose computation terminates and delivers an incomplete set of computed answers.

Future work

An important extension of the method presented in this paper is to include such features of Prolog as the cut, negation, set-of etc. Our intention is that they should be treated as declaratively as possible.

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.

Experiments with debugging of actual programs are needed to better understand the debugging process, to evaluate the presented approach and to develop pragmatics of declarative debugging with assertions.

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. Such a methodology may constitute a bridge between pure declarative programming (which is still a rather unrealistic concept) and imperative programming based on an operational semantics.

References

[AE82] Apt, K.R. and Van Emden, M.H., Contributions to the Theory of Logic Programming, Journal of the ACM 29(3), July 1982 : 841-863.

[DL87] Dershowitz, N. and Lee, Y., Deductive Debugging, Proceedings of the Symposium on Logic Programming - San Francisco 1987 : 298-306.

[DM87] Drabent, W.,and Maluszynski, J., Inductive Assertion Method for Logic Programs, Proceedings of the International Conference on Theory and Practice of Software Development (TAPSOFT) 1987, LNCS 250, Springer Verlag : 167-181.

[Fe87] Ferrand, G., Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method, Journal of Logic Programming 1987(4): 177-198.

[Ll87] Lloyd, J.W., Declarative Error Diagnosis, New Generation Computing 5(2), 1987: 133-154.

[MO84] Mycroft, A., O'Keefe, R.A., A Polymorphic Type System for Prolog, Artificial Intelligence 23 , 1984 : 295-307.

[Ni83] Nilsson, J.F., On the Compilation of a Domain-based Prolog, in: Mason, R.E.A.(ed), *Information Processing 83*, North Holland 1983 : 293-298.

[Pe86] Pereira, L. M., Rational Debugging in Logic Programming, Proceedings of the 3rd International Conference on Logic Programming, LNCS 225, Springer Verlag , 1986 : 203-210.

[Sh82] Shapiro, E.Y., Algorithmic Program Debugging, Proceedings of the 9th annual ACM Symposium on Principles of Programming Languages, 1982.

[Sh83] Shapiro, E.Y., *Algorithmic Program Debugging*, MIT Press, 1983.

[SS86] Sterling, L. and Shapiro, E.Y., *The art of Prolog*, MIT Press 1986.

[Zo87] Zobel, J., Derivation of Polymorphic Types for Prolog Programs, in: Lassez, J.L.,(ed), *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne 1987 : 817-838.

Appendix: Example Sessions

In the following example sessions we illustrate the qualitative and quantitative improvements in the user queries made by our diagnosis algorithms. inc/1 and ins/1 are the incorrectness and insufficiency diagnosis procedures respectively.

Example 1:

During the following Prolog session 4 of the errors given in the standard buggy quicksort program [Sh82] are detected by our algorithms. The other two errors in the standard buggy qsort are automatically detected by the Quintos prolog parser. The asserions 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([],[]).                                          %error1
qsort([X|L],L0) :-
          partition(L, X, L1, L2),
          qsort(L1,L3),
          qsort(L2,L4),
          append([X|L3],L4, L0).                        %error4

partition([X|L], Y, L1, [X|L2]) :-
          partition(L,Y,L1,L2).                         %error2
partition([X|L], Y, [X|L1], L2) :-
          Y =< X,                                       %error3
          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)
L = _1
```

%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)
L = _64
```

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

```
| ?- listing(partition).

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

yes
| ?- 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]).

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

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

```
| ?- listing(qsort).
```

```
qsort([],[]).
qsort([A|B],C) :-
            partition(B,A,D,E),
            qsort(D,F),
            qsort(E,G),
            append(F,[A|G],C).

yes
| ?- 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])).

no
| ?-
```

Comparison with Shapiro algorithm

Program 19.17 given in [SS86] detects the bugs in the in the same buggy program as follows:

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

no
| ?- missing_solution( qsort([2,3,1,5],L), UA).
Enter a true ground instance of
qsort([2,3,1,5],_119):-
            partition([3,1,5],2,_321,_322),
            qsort(_321,_331),
            qsort(_322,_340),
append([2|_331],_340,_119)
if there is such,or "no" otherwise
|: no.

L = _119,
UA = qsort([2,3,1,5],L)
```

% the call to append in qsort is corrected

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

no
| ?- missing_solution( qsort([2,3,1,5],L), UA).
Enter a true ground instance of
qsort([2,3,1,5],_119):-
            partition([3,1,5],2,_321,_322),
            qsort(_321,_331),
            qsort(_322,_340),
            append(_331,[2|_340],_119)
if there is such,or "no" otherwise
|: qsort([2,3,1,5],[1,2,3,5]):-
            partition([3,1,5],2,[1],[3,5]),
            qsort([1],[1]),
            qsort([3,5],[3,5]),
            append([1],[2,3,5],[1,2,3,5]).
Enter a true ground instance of
partition([3,1,5],2,[1],[3,5]):-
            partition([1,5],2,[1],[5])
if there is such,or "no" otherwise
|: partition([3,1,5],2,[1],[3,5]):-
```

```
          partition([1,5],2,[1],[5]).
Enter a true ground instance of
partition([1,5],2,[1],[5]):-
          2=<1,
          partition([5],2,[],[5])
if there is such,or "no" otherwise
|: no.

L = [1,2,3,5],
UA = partition([1,5],2,[1],[5])
```

% A test is added to the first clause of partition

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

no
| ?- missing_solution( qsort([2,3,1,5],L), UA).
Enter a true ground instance of
partition([3,1,5],2,[1],[3,5]):-
          3>2,
          partition([1,5],2,[1],[5])
if there is such,or "no" otherwise
|: partition([3,1,5],2,[1],[3,5]):-
          3>2,
          partition([1,5],2,[1],[5]).

L = [1,2,3,5],
UA = partition([1,5],2,[1],[5])
```

% The test in the 2nd clause to partition is reversed

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

no
| ?- missing_solution( qsort([2,3,1,5],L), UA).
Enter a true ground instance of
qsort([1],[1]):-
          partition([],1,_1516,_1517),
          qsort(_1516,_1526),
          qsort(_1517,_1535),
          append(_1526,[1|_1535],[1])
if there is such,or "no" otherwise
|: qsort([1],[1]):-
          partition([],1,[],[]),
          qsort([],[]),
```

```
            qsort([],[]),
            append([],[1],[1]).

L = [1,2,3,5],
UA = qsort([],[])
```

% The clause for qsort([],[]) is added

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

L = [1,2,3,5]

yes
| ?-
```

Comments:

Note that every query in the new version of Shapiros algorithm [SS86], is equivalent to asking "Is Atom satisfiable?" for every Atom in the body of the clause in question. The total number of atomic queries answered in this example (not considering the duplicate questions asked by the [SS86] program which we have removed from the log) amounted to 17 in Shapiros case and 5 in our case. Note that in addition to yes/no anwsers we were required to give instances of 8 atoms for Shapiros algorithm. Using our implementation 4 assertions were given in addition to yes/no answers. The use of *true* assertion in this example resulted in giving a complete specification which in many other cases is impractical. We included this assertion in order to demonstrate our implementation. In practice, however, it is more likely that *posex*, and *negex* will be used in easy to formulate assertions.

Example 2:

We show an example session for diagnosing insufficiency in a buggy version of the procedure substitute/4 [SS86]. We show the application of simple *posex* assertions containing a call to the library procedure forall_members(M,List,Condition) which checks that every element of the List satisfies a Condition.

Here is a listing of the buggy version:

```
/* substitute(Old, New,OldTerm,NewTerm) :-
NewTerm is the result of replacing all occurrences of Old in
OldTerm by New.
*/

substitute(Old, _New, Term, Term):-
            atomic(Term), \+ Old = Term.
substitute(Old, New, Term, Term1):-
            \+ atomic(Term),
```

```
            functor(Term, F, N),
            functor(Term1,F,N),
            substitute(N, Old, New, Term, Term1).

substitute(N, Old, New, Term, Term1):-
            N > 0,
            arg(N, Term, Arg),
            substitute(Old, New, Arg, Arg1),
            arg(N, Term1, Arg1),
            N1 is N - 1,
            substitute(N1, Old, New, Term, Term1).
substitute(0, _Old, _New, _Term, _Term1).
```

% Diagnosis session

```
| ?- substitute(x, z, f(y, g(w, h(x)), s(x)), NewTerm).

no
| ?- killAssertions.
killed assertions

yes
| ?- ins(substitute(x, z, f(y, g(w, h(x)), s(x)), NewTerm)).
Is substitute(3,x,z,f(y,g(w,h(x)),s(x)),f(B,C,D)) satisfiable? a.
|:posex(substitute(_N, _Old, _New, OldTerm, NewTerm)):-
            OldTerm =.. [ F| OldArgs],
            NewTerm =.. [ F| NewArgs],
            equal_length( OldArgs, NewArgs),
            forall_members(M, NewArgs, (M = 'VAR'(_))).
|: Is substitute(x,z,s(x),B) satisfiable? a.
|: posex(substitute( _Old, _New, _OldTerm, 'VAR'(_))).
|: This atom is not completely covered:
substitute(x,z,x,B)
NewTerm = _33
```

% The missing clause substitute(Old, New, Old, New) is added to the program

```
| ?- substitute(x, z, f(y, g(w, h(x)), s(x)), NewTerm).

NewTerm = f(y,g(w,h(z)),s(z))

| ?- listing(posex).

posex(substitute(A,B,C,D,E)) :-
            D=..[F|G],
```

```
            E=..[F|H],
            equal_length(G,H),
            forall_members(I,H,I='VAR'(J)).
posex(substitute(A,B,C,'VAR'(D))).

yes | ?-
```

Comparison with Shapiro algorithm

We list below the corresponding session when program 19.17 of [SS86] is used to diagnose the same error:

```
| ?- substitute(x ,z, f(y, g(w, h(x)), s(x)), NewTerm).

no
| ?- missing_solution( substitute(x ,z, f(y, g(w, h(x)), s(x)),
NewTerm), UA).
Enter a true ground instance of
substitute(x,z,f(y,g(w,h(x)),s(x)),f(y,g(w,h(x)),s(x))):-
            atomic(f(y,g(w,h(x)),s(x))),
            \+x=f(y,g(w,h(x)),s(x))
if there is such,or "no" otherwise

|: no.
Enter a true ground instance of
substitute(x,z,f(y,g(w,h(x)),s(x)),_534):-
            \+atomic(f(y,g(w,h(x)),s(x))),
            functor(f(y,g(w,h(x)),s(x)),_790,_791),
            functor(_534,_790,_791),
            substitute(_791,x,z,f(y,g(w,h(x)),s(x)),_534)
if there is such,or "no" otherwise
|: substitute(x,z,f(y,g(w,h(x)),s(x)), f(y, g(w, h(z)), s(z))):-
            \+atomic(f(y,g(w,h(x)),s(x))),
            functor(f(y,g(w,h(x)),s(x)),f,3),
            functor(f(y,g(w,h(z)),s(z)),f,3),
            substitute(3,x,z,f(y,g(w,h(x)),s(x)),f(y,g(w,h(z)),s(z))).

Enter a true ground instance of
substitute(3,x,z,f(y,g(w,h(x)),s(x)),f(y,g(w,h(z)),s(z))):-
            3>0,
            arg(3,f(y,g(w,h(x)),s(x)),_2429),
            substitute(x,z,_2429,_2440),
            arg(3,f(y,g(w,h(z)),s(z)),_2440),
            _2457 is 3-1,
            substitute(_2457,x,z,f(y,g(w,h(x)),s(x)),f(y,g(w,h(z)),s(z)))
```

```
if there is such,or "no" otherwise
|: substitute(3,x,z,f(y,g(w,h(x)),s(x)),f(y,g(w,h(z)),s(z))):-
            3>0,
            arg(3,f(y,g(w,h(x)),s(x)),s(x)),
            substitute(x,z,s(x),s(z)),
            arg(3,f(y,g(w,h(z)),s(z)),s(z)),
            2 is 3-1,
            substitute(2,x,z,f(y,g(w,h(x)),s(x)),f(y,g(w,h(z)),s(z))).
Enter a true ground instance of
substitute(x,z,s(x),s(z)):-
            \+atomic(s(x)),
            functor(s(x),_4184,_4185),
            functor(s(z),_4184,_4185),
            substitute(_4185,x,z,s(x),s(z))
if there is such,or "no" otherwise
|: substitute(x,z,s(x),s(z)):-
            \+atomic(s(x)),
            functor(s(x),s,1),
            functor(s(z),s,1),
            substitute(1,x,z,s(x),s(z)).
Enter a true ground instance of
substitute(1,x,z,s(x),s(z)):-
            1>0,
            arg(1,s(x),_5256),
            substitute(x,z,_5256,_5267),
            arg(1,s(z),_5267),
            _5284 is 1-1,
            substitute(_5284,x,z,s(x),s(z))
if there is such,or "no" otherwise
|: substitute(1,x,z,s(x),s(z)):-
            1>0,
            arg(1,s(x),x),
            substitute(x,z,x,z),
            arg(1,s(z),z),
            0 is 1-1,
            substitute(0,x,z,s(x),s(z)).
Enter a true ground instance of
substitute(x,z,x,z):-
            \+atomic(x),
            functor(x,_6457,_6458),
            functor(z,_6457,_6458),
            substitute(_6458,x,z,x,z)
```

```
if there is such,or "no" otherwise
|: no.

NewTerm = f(y,g(w,h(z)),s(z)),
UA = substitute(x,z,x,z)

| ?-
```

Comments:

This example illustrates a case where tedious work is involved in instantiation of clauses using Shapiros algorithm for insufficiency. The total number of atoms queried amounted to 22 of which 16 had to be instantiated by the user.

This implementation of Shapiros algorithm is worse than the original one [Sh 83] in cases where many built in predicates are involved. The earlier implementation, however, had the drawback that it required the user to give all true instances of an atom in reply to an existential query.

Our algorithm detects the same error by using 2 *posex* assertions. Without assertions the same error is discovered after 4 queries:

```
| ?- ins( substitute(x ,z, f(y, g(w, h(x)), s(x)), NewTerm)).
Is substitute(3,x,z,f(y,g(w,h(x)),s(x)),f(B,C,D)) satisfiable? y.
Is substitute(x,z,s(x),B) satisfiable? y.
Is substitute(1,x,z,s(x),s(B)) satisfiable? y.
Is substitute(x,z,x,B) satisfiable? y.
This atom is not completely covered:
substitute(x,z,x,B)
NewTerm = _32

| ?-
```

However, the same assertions are useful in detection of other errors in the same program. This is illustrated in Example 3.

Example 3:

Consider a version of the same program with an additional error:

```
%substitute(Old, New, Old, New).                    %error1
substitute(Old, _New, Term, Term):-
            atomic(Term), \+ Old = Term.
substitute(Old, New, Term, Term1):-
            \+ atomic(Term),
            functor(Term, F, N),
            functor(Term1,F,N),
            substitute(N, Old, New, Term, Term1).
```

```
substitute(N, Old, New, Term, Term1):-
            N > 0,
            arg(N, Term, Arg),
            substitute(Old, New, Arg, Arg1),
            arg(N, Term1, Arg1),
            N1 is N - 1,
            substitute(N1, Old, New, Term, Term).     %error2...Term,Term1)
substitute(0, _Old, _New, _Term, _Term1).
```

Comments:

Our algorithm first detects error1 as in the previous session. Then the second error is discovered using only two new queries. When Shapiros algorithm is applied, the second error is found after repeating two earlier questions and then asking for a clause instance with 6 atoms in the body. This is illustrated below:

% Diagnosis session

```
| ?- listAssertions.

posex(substitute(A,B,C,D,E)) :-
            D=..[F|G],
            E=..[F|H],
            equal_length(G,H),
            forall_members(I,H,I='VAR'(J)).
posex(substitute(A,B,C,'VAR'(D))).

| ?- ins( substitute(x ,z, f(y, g(w, h(x)), s(x)), NewTerm)).
This atom is not completely covered:
substitute(x,z,x,B)
NewTerm = _32
```

% first error is fixed: substitute(Old, New, Old, New) is added to program

```
| ?- substitute(x ,z, f(y, g(w, h(x)), s(x)), NewTerm).

no
| ?- ins( substitute(x ,z, f(y, g(w, h(x)), s(x)), NewTerm)).
Is        substitute(2,x,z,f(y,g(w,h(x)),s(x)),f(y,g(w,h(x)),s(x)))
satisfiable? n.
Is the following set of answers INCOMPLETE for the goal:
substitute(x,z,s(x),B)
answers:
substitute(x,z,s(x),s(z))
|: n.
```

```
This atom is not completely covered:
substitute(3,x,z,f(y,g(w,h(x)),s(x)),f(B,C,D))
NewTerm = _100

| ?-
```

Comparison with Shapiros algorithm:

```
| ?- missing_solution( substitute(x ,z, f(y, g(w, h(x)), s(x)),
NewTerm),UA).
Enter a true ground instance of
substitute(x,z,f(y,g(w,h(x)),s(x)),f(y,g(w,h(x)),s(x))):-
           atomic(f(y,g(w,h(x)),s(x))),
           \+x=f(y,g(w,h(x)),s(x))
if there is such,or "no" otherwise
|: no. Enter a true ground instance of
substitute(x,z,f(y,g(w,h(x)),s(x)),_100):-
           \+atomic(f(y,g(w,h(x)),s(x))),
           functor(f(y,g(w,h(x)),s(x)),_356,_357),
           functor(_100,_356,_357),
           substitute(_357,x,z,f(y,g(w,h(x)),s(x)),_100)
if there is such,or "no" otherwise
|: substitute(x,z,f(y,g(w,h(x)),s(x)), f(y, g(w, h(z)), s(z))):-
           \+atomic(f(y,g(w,h(x)),s(x))),
           functor(f(y,g(w,h(x)),s(x)),f,3),
           functor(f(y,g(w,h(z)),s(z)),f,3),
           substitute(3,x,z,f(y,g(w,h(x)),s(x)),f(y,g(w,h(z)),s(z))).
Enter a true ground instance of
substitute(3,x,z,f(y,g(w,h(x)),s(x)),f(y,g(w,h(z)),s(z))):-
           3>0,
           arg(3,f(y,g(w,h(x)),s(x)),_1995),
           substitute(x,z,_1995,_2006),
           arg(3,f(y,g(w,h(z)),s(z)),_2006),
           _2023 is 3-1,
substitute(_2023,x,z,f(y,g(w,h(x)),s(x)),f(y,g(w,h(x)),s(x)))
if there is such,or "no" otherwise
|: no.

NewTerm = f(y,g(w,h(z)),s(z)),
UA = substitute(3,x,z,f(y,g(w,h(x)),s(x)),f(y,g(w,h(z)),s(z)))
```

Avdelning, institution, fakultet
Division, department, faculty

Department of Computer and
Information Science
Linköping University

ISBN:

ISSN: ISSN-0281-4250

Rapportnr: Report no: LiTH-IDA-R-88-04

Upplagans storlek: Number of copies:

Datum: Date: March 1988

Projekt: Project:

Titel: Title: Algorithmic Debugging with Assertions

Författare: Author Wlodek Drabent, Simin Nadjm-Tehrani, Jan Maluszynski

Uppdragsgivare: Commissioned by:

Dnr.: Call no:

Rapporttyp: Kind of report:

- ☐ Examensarbete/Final project
- ☐ Delrapport/Progress report
- ☐ Reserapport/Travel report
- ☐ Slutrapport/Final report
- ☒ Övrig rapport/Other kind of report

Rapportspråk: Language:

- ☐ Svenska/Swedish
- ☒ Engelska/English
- ☐ _____

Sammanfattning (högst 150 ord): Abstract (150 words):

Abstract: Algorithmic debugging, as presented by Shapiro, is an interactive process where the debugging system acquires knowledge about the expected behaviour of the debugged program and uses it to localize errors. This paper suggests a generalization of the language used to communicate with the debugger. In addition to the usual "yes" and "no" answers formal specifications of some properties of the intended model are allowed. The specifications are logic programs. They employ library procedures and are developed interactively in the debugging process. An experimental debugging system incorporating this idea has been implemented. In contrast to some other systems, its insufficiency diagnoser does not require instantiation of unsolved goals by the oracle. A formal proof of correctness and completeness of this algorithm is presented.

Nyckelord (högst 8): Keywords (8):

Bibliotekets anteckningar

A Selection of Previous Research Reports.

LiTH-IDA-R-87-21 **Harold W. Lawson, Jr.:** Challenges and Directions in Computers and Education.

LiTH-IDA-R-87-20 **Krzysztof Kuchcinski, Zebo Peng:** Parallelism Extraction from Sequential Programs for VLSI Applications. This paper is to appear in *Microprocessing and Microprogramming, the Euromicro Journal*, 1988.

LiTH-IDA-R-87-19 **Zebo Peng:** Construction of Asynchronous Concurrent Systems from their Behavioral Specifications. Also in *Proc. of the 10th World Computer Congress*, Dublin, Ireland, Sept. 1-5, 1986.

LiTH-IDA-R-87-18 **Henrik Nordin:** Reuse and Maintenance Techniques in Knowledge-Based Systems.

LiTH-IDA-R-87-17 **Tony Larsson:** Specification and Verification of VLSI Systems Actional Behaviour This is a close version of a paper presented at *the 8th international conference on Computer Hardware Description Languages, CHDL,* 87.

LiTH-IDA-R-87-16 **Shamsul I. Chowdhury:** State of the Art in Statistical Expert Systems. Also in *Proc. of the Conference on Expert Systems and their Applications*, Avignon, May 1987.

LiTH-IDA-R-87-15 **Nils Dahlbäck:** Kommunikation med datorer i naturligt språk - vad är det och vem behöver det?

LiTH-IDA-R-87-14 **Peter Fritzson:** Window System Architectures - an Overview. Also presented at *A Tutorial on Software Development Environments,* Linköping University, Nov 24-25 1986.

LiTH-IDA-R-87-13 **Mats Wirén:** A Comparison of Rule-Invocation Strategies in Context-Free Chart Parsing. Also in *Proc. of the Third Conference of the European Chapter of the Association for Computational Linguistics,* Copenhagen, April 1—3, 1987.

LiTH-IDA-R-87-11 **Sture Hägglund, Christer Hansson, Tomas Sokolnicki:** Knowledge-Based Training of Case Management Routines and Emergency Procedures. Also in *Proc. of the 3rd Int. Conf. on Expert Systems*, London, 1987.

LiTH-IDA-R-87-10 **Andrzej Lingas:** On Parallel Complexity of the Subgraph Isomorphism Problem.

LiTH-IDA-R-87-09 **Andrzej Lingas, Marek Karpinski:** Subtree Isomorphism and Bipartite Perfect Matching are Mutually NC Reducible.

LiTH-IDA-R-87-08 **Christos Levcopoulos, Andrzej Lingas, Jörg-R. Sack:** Nearly Optimal Heuristics for Binary Search Trees with Geometric Generalizations. Also in *Proc of ICALP'87*, Karlsruhe, July 1987.

LiTH-IDA-R-87-07 **Peter Haneclou:** A Formal Approach to Reason-maintenance Based on Abstract Domains.

LiTH-IDA-R-87-06 **Johan Hultman:** COPPS - A Software System for Defining and Controlling Actions in a Mechanical System.

LiTH-IDA-R-87-05 **Christer Bäckström:** Logical Modelling of Simplified Geometrical Objects and Mechanical Assembly Processes.

LiTH-IDA-R-87-03 **Mikael R.K. Patel:** A Threaded Interpretive Language Supporting Programming in the Large. Also in *Proc. of The sixth Rochester Forth Conference*, University of Rochester, Rochester, New York, June 11-14, 1986.

LiTH-IDA-R-87-02 **Rober Bilos:** A Token-Based Syntax Sensitive Editor. Also presented at the *Workshop on Programming Environments - Programming Paradigms*, Roskilde, Denmark, October 22-24, 1986.

LiTH-IDA-R-87-01 **Wlodzimierz Drabent:** Do Logic Programs Resemble Programs in Conventional Languages? Also in *Proc of the Fourth IEEE Symposium on Logic Programming*, San Francisco, Aug. 31- Sept. 4, 1987.

LiTH-IDA-R-86-44 **Erik Sandewall et al.:** Department of Computer and Information Science Annual Research Report 1986.

The Department of Computer and Information Science
Linköping University

organizes undergraduate and graduate studies in *Computer Science*, *Telecommunication and Computer Systems*, and *Administrative Data Processing*. Research activities have an emphasis on advanced software technology and computer systems design and are organized in a number of research laboratories:

- *ACTLAB - Laboratory for Complexity of Algorithms*, which is concerned with the design and analysis of efficient sequential and parallel algorithms, and complexity theory, especially in the areas of computational geometry, data structures on bounded domains and graph algorithms.
- *AIELAB - Artificial Intelligence Environments Laboratory,* which conducts research on representation and manipulation of knowledge, problem solving techniques and expert systems with mechanical engineering applications.
- *ASLAB - Application Systems Laboratory,* which studies design of advanced support systems for interactive use of computers, including tools for automated construction of applications software.
- *CADLAB - Laboratory for Computer-Aided Design of Electronics*, which concentrates its research activities around tools for integrated development of hardware and software, graphics-based modelling and simulation techniques.
- *LIBLAB - Laboratory for Library and Information Science,* which studies methods for access to documents and the information contained in the documents, concentrating on catalogs and bibliographic representations, and on the human factors of library use.
- *LOGPRO - Laboratory for Logic Programming*, which concentrates its research activities around foundations of logic programming, relations to other programming paradigms and methodology.
- *NLPLAB - Natural Language Processing Laboratory*, which conducts research related to the development and use of natural language interfaces to computer software.
- *PELAB - Programming Environments Laboratory*, which works with design of tools for software development, specific functions in such tools and theoretical aspects of programs under construction.
- *RKLLAB - Laboratory for Representation of Knowledge in Logic*, which covers issues and techniques such as non-monotonic logic, probabilistic logic, modal logic and truth maintenance algorithms and systems.

Research Reports 1987 and 1988

LiTH-IDA-R-88-04 **Wlodek Drabent, Simin Nadjm-Tehrani, Jan Maluszynski:** Algorithmic Debugging with Assertions.

LiTH-IDA-R-88-03 **Staffan Bonnier, Jan Maluszynski:** Towards a Clean Amalgamation of Logic Programs with External Procedures.

LiTH-IDA-R-88-02 **Ulf Nilsson:** Inferring Restricted AND-Parallelism in Logic Programs using Abstract Interpretation.

LiTH-IDA-R-88-01 **Jacek Leszczylowski, Staffan Bonnier, Jan Maluszynski:** Logic Programming with External Procedures: Introducing S-Unification. A shorter version is to appear in *Information Processing Letters*, North Holland.

LiTH-IDA-R-87-26 **Jonas Löwgren:** Applying a Rapid Prototyping System to Control Panel Dialgues.

LiTH-IDA-R-87-24

LiTH-IDA-R-87-22 ctive System for Tagging
nce of the Association
rg 1-5 June, 1987.