TR-447

Logic Program Diagnosis from Specifications

by
T. Kanamori, T. Kawamura, M. Maeji
& K. Horiuchi (Mitsubishi)

March, 1989

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Logic Program Diagnosis from Specifications

Tadashi KANAMORI Tadashi KAWAMURA
Machi MAEJI Kenji HORIUCHI

Mitsubishi Electric Corporation
Central Research Laboratory
Tsukaguchi-Honmachi 8-1-1
Amagasaki, Hyogo, JAPAN 661

Abstract

This paper presents a framework for locating bugs of Prolog programs from specifications. To be debugged, each predicate is specified by a set of universal formulas either of the form $A \subset A_1 \wedge A_2 \wedge \cdots \wedge A_k$ or of the form $A \wedge A_1 \wedge A_2 \wedge \cdots \wedge A_l \supset A_{l+1} \wedge A_{l+2} \wedge \cdots \wedge A_{l+k}$. The system generates each test case by executing the antecedant part of such a formula, and checks the test case by executing the consequence part under the answer substitution just obtained. If the latter execution cannot succeed without instantiating the variables, it turns out that either the execution of atom A (in the consequence part) under the answer substitution has failed unexpectedly, or the execution of atom A (in the antecedant part) under the answer substitution has succeeded unexpectedly. Then, the system locates the bug using the specifications by checking whether each subcomputation has proceeded as intended while tracing the execution of the atom in a top-down manner. Our approach is a generalization of both the bug location alogorithm in Shapiro's "Algorithmic Debugging" and that in Dershowitz-Lee's "Deductive Debugging." The relation between the generation of test cases and the verification of logic programs is discussed as well.

Keywords: Program Diagnosis, Program Modification, Program Debugging, Prolog.

Contents

1. Introduction

Debugging programs is an integral part of our programming activity. It is often said that programmers usually spend more than 50 percent of the time for debugging, of which the work for locating bugs is most time-consuming, while that for correcting bugs is relatively easy for human programmers once the bugs are found. For locating bugs, there have been developed several tools for various programming languages, e.g., those for displaying a stack backtrace, stopping at specific location or at specific event (breaker), displaying the value of variables by name, executing programs step by step (stepper), tracing the execution of all procedures (tracer), and spying the excution of specified procedures (spy). Within the framework of logic programming, similar tools have been developed as well, e.g., the "tracer" command and the "spy" command of DEC10 Prolog.

Compared with the conventional debugging tools, "Algorithmic Debugger" by Shapiro [15], [16] is slightly different in its nature. While tracing (or backtracing) the execution of logic programs, his debugger asks the programmer whether each procedure (predicate) call just traced has returned a correct result or what answer should have been returned if the program is the intended one. The programmer just need to answer the queries according to his declarative knowledge about each procedure, i.e., he/she does not need to follow the operational behavior of the execution in his/her mind.

In Shapiro's algorithmic debugging, the programmer was considered just a device to answer the queries from the system. Any device which can answer the queries is able to play the role of the programmer, and called an *oracle* in general. The idea to use specifications as an oracle instead of human programmers was hinted by Shapiro [15] pp.77–80, and developed by Dershowitz and Lee [1]. Their "Deductive Debugger" used *executable specifications* (i.e., another possibly inefficient definite clause programs for specification) not only for answering the queries but also for generating test cases, which human programmers have to give in Shapiro's debugger. For example, if "A :- $A_1, A_2, \ldots, A_k$" is an executable specification, and θ is an answer substitution obtained by executing the body $A_1, A_2, \ldots, A_k$, then the execution of $A\theta$ using the program (being debugged) must succeed without instantiating the variables in $A\theta$.

This paper presents a framework for locating bugs of Prolog programs from specifications. To be debugged, each predicate is specified by a set of universal formulas either of the form $A \subset A_1 \wedge A_2 \wedge \cdots \wedge A_k$ or of the form $A \wedge A_1 \wedge A_2 \wedge \cdots \wedge A_l \supset A_{l+1} \wedge A_{l+2} \wedge \cdots \wedge A_{l+k}$. The system generates each test case by executing the antecedant part of such a formula, and checks the test case by executing the consequence part under the answer substitution just obtained. If the latter execution cannot succeed without instantiating the variables, it turns out that either the execution of atom A (in the consequence part) under the answer substitution has failed unexpectedly, or the execution of atom A (in the antecedant part) under the answer substitution has succeeded unexpectedly. Then, the system locates the bug using the specifications by checking whether each subcomputation has proceeded as intended while tracing the execution of the atom in a top-down manner. Our approach is a generalization of both the bug location alogorithm in Shapiro's "Algorithmic Debugging" and that in Dershowitz-Lee's "Deductive Debugging." The relation between the generation of test cases and the verification of logic programs is discussed as well.

The rest of this paper is organized as follows: Section 2 fixes our programming language and specification language. Section 3 shows the outline of our framework for logic program modification consisting of three phases. The first phase, *test case generation phase*, is explained in Section 4, and the second phase, *bug location phase*, in Section 5. (The third phase, *bug correction phase*, is not explained in this paper.) Section 6 explains our implementation, and Section 7 presents an example of logic program modification. Last, Section 8 discusses the relation between the generation of test cases and the verification of logic programs.

2. Preliminaries

2.1 Programming Language

A finite set P of definite clauses is called a *program*. As was mentioned in Section 1, the execution of a goal w.r.t. a program can be traced using the "trace" command of DEC10 Prolog.

Example 2.1.1 The following is a wrong program of *quicksort* with 6 bugs [16].

```
                                                      % qsort([ ],[ ]) is missed
qsort([X|L],M) :- partition(L,X,L1,L2),
                  qsort(L1,M1), qsort(L2,M2),
                  append([X|M1],M2,M).                % append(M1,[X|M2],M)
partition([ ],X,[ ],[ ]).
partition([Y|L],X,[Y|L1],L2) :- Y≥X, partition(L,X,L1,L2).    % Y≤X
partition([Y|L],X,L1,[Y|L2]) :- partition(L,X,L1,L2).          % Y>X
append([ ],M,[ ]).                                             % append([ ],M,M)
append([X|L],M,[X|N]) :- append(L,M,M).                        % append(L,M,N)
```

Example 2.1.2 The following is the execution trace of "$qsort([2,1],X)$" w.r.t. the program above + unit clause "$qsort([\,],[\,])$."

```
| ?- trace, qsort([2,1], X).
Debug mode switched on.
    (1)   0  Call   : qsort([2,1], _40)
      (2)   1  Call   : partition([1], 2, _105, _106)
        (3)   2  Call   : 1≥2
        (3)   2  Fail   : 1≥2
        (4)   2  Call   : partition([ ], 2, _120, _106)
        (4)   2  Exit   : partition([ ], 2, [ ], [ ])
      (2)   1  Exit   : partition([1], 2, [1], [ ])
      (5)   1  Call   : qsort([1], _107)
        (6)   2  Call   : partition([ ], 1, _149, _150)
        (6)   2  Exit   : partition([ ], 1, [ ], [ ])
        (7)   2  Call   : qsort([ ], _151)
        (7)   2  Exit   : qsort([ ], [ ])
        (8)   2  Call   : qsort([ ], _152)
        (8)   2  Exit   : qsort([ ], [ ])
        (9)   2  Call   : append([ ], [1], _107)
        (9)   2  Exit   : append([ ], [1], [ ])
      (5)   1  Exit   : qsort([1], [ ])
      (10)  1  Call   : qsort([ ], _108)
      (10)  1  Exit   : qsort([ ], [ ])
      (11)  1  Call   : append([ ], [2], _40)
      (11)  1  Exit   : append([ ], [2], [ ])
    (1)   0  Exit   : qsort([2,1], [ ])
 X = [ ] ;
    (1)   0  Redo   : qsort([2,1], [ ])
     (11)   1  Redo   : append([ ], [2], [ ])
     (11)   1  Fail    : append([ ], [2], _40)
     (10)   1  Redo   : qsort([ ], [ ])
     (10)   1  Fail    : qsort([ ], _108)
      (5)   1  Redo   : qsort([1], [ ])
        (9)   2  Redo   : append([ ], [1], [ ])
        (9)   2  Fail    : append([ ], [1], _107)
        (8)   2  Redo   : qsort([ ], [ ])
        (8)   2  Fail    : qsort([ ], _152)
        (7)   2  Redo   : qsort([ ], [ ])
        (7)   2  Fail    : qsort([ ], _151)
        (6)   2  Redo   : partition([ ], 1, [ ], [ ])
        (6)   2  Fail    : partition([ ], 1, _149, _150)
      (5)   1  Fail    : qsort([1], _107)
      (2)   1  Redo   : partition([1], 2, [1], [ ])
        (4)   2  Redo   : partition([ ], 2, [ ], [ ])
        (4)   2  Fail    : partition([ ], 2, _120, _106)
      (2)   1  Fail    : partition([1], 2, _105, _106)
    (1)   0  Fail    : qsort([2,1], _40)
 no
```

Remark. In this paper, only terminating Prolog programs are considered. For debugging of nonterminating Prolog programs, see [15], [1].

2.2 Specification Language

(1) Specification Formulas for Unexpected Failure

A formula of the form

$\forall X_1, X_2, \ldots, X_m \ (p(t_1, t_2, \ldots, t_n) \ \subset \ A_1 \wedge A_2 \wedge \cdots \wedge A_k)$

is called a *specification formula for unexpected failure of p* when the predicates of $A_1, A_2, \ldots, A_k$ are predefined primitive predicates. Hereafter, such a formula is represented simply by

$p(t_1, t_2, \ldots, t_n)$:- $A_1, A_2, \ldots, A_k$.

$p(t_1, t_2, \ldots, t_n)$ is called the *head*, while "$A_1, A_2, \ldots, A_k$" is called the *body* of the specification formula.

Example 2.2.1 The following are formulas for unexpected failure:

qsort(L,M) :- permute(L,M), ordered(M),
partition(L,X,L,[ ]) :- ge-all(X,L),
partition(L,X,[ ],L) :- lt-all(X,L),

where "*permute*," "*ordered*," "*ge-all*" and "*lt-all*" are predefined primitive predicates. ("*ge-all*" denotes that the first argument is greater than or equal to all the elements of the second list argument, while "*lt-all*" denotes that the first argument is less than all the elements of the second list argument.)

(2) Specification Formulas for Unexpected Success

A formula of the form

$\forall X_1, X_2, \ldots, X_m \ (p(t_1, t_2, \ldots, t_n) \wedge A_1 \wedge A_2 \wedge \cdots \wedge A_l \ \supset \ A_{l+1} \wedge A_{l+2} \wedge \cdots \wedge A_{l+k})$

is called a *specification formula for unexpected success of p* when the predicates of $A_1, A_2, \ldots,$ A_{l+k} are predefined primitive predicates. Hereafter, such a formula is represented simply by

$p(t_1, t_2, \ldots, t_n)$ -: if $A_1, A_2, \ldots, A_l$ **then** $A_{l+1}, A_{l+2}, \ldots, A_{l+k}$.

$p(t_1, t_2, \ldots, t_n)$ is called the *head*, while "$A_1, A_2, \ldots, A_l$" is called the *if part*, and "$A_{l+1}, A_{l+2}, \ldots, A_{l+k}$" is called the *then part* of the specification formula. In particular, when $l = 0$, it is represented by

$p(t_1, t_2, \ldots, t_n)$ -: $A_1, A_2, \ldots, A_k$.

Example 2.2.2 The following are formulas for unexpected success:

```
qsort(L,M) -: permute(L,M), ordered(M),
partition(L,X,L1,L2) -: sublist(L1,L), sublist(L2,L), ge-all(X,L1), lt-all(X,L2),
partition(L,X,L1,L2) -: if length(L1,N1), length(L2,N2), length(L,N)
                        then add(N1,N2,N),
```

where "*sublist*" and "*length*" are predefined primitive predicates.

(3) Specification of Predicate

Each predicate to be debugged is specified in one of the following three ways.

(i) Executable Specification

spec(p).
  $p(X_1, X_2, \ldots, X_n)$:-: $A_1, A_2, \ldots, A_k$.
endspec.

This is equivalent to writing

spec(p).
  $p(X_1, X_2, \ldots, X_n)$:- $A_1, A_2, \ldots, A_k$.
  $p(X_1, X_2, \ldots, X_n)$ -: $A_1, A_2, \ldots, A_k$.
endspec.

Executable specifications are used when it is not very hard to write concise specifications which completely characterize the predicates.

(ii) Specification by Properties

spec(p).
  F_1.
  F_2.
  $\vdots$
  $F_{\alpha-1}$.
  $p(X_1, X_2, \ldots, X_n)$:- *query-p*$(X_1, X_2, \ldots, X_n)$.
  G_1.
  G_2.
  $\vdots$
  $G_{\beta-1}$.
  $p(X_1, X_2, \ldots, X_n)$ -: *query-p*$(X_1, X_2, \ldots, X_n)$.
endspec.

Each F_i is a specification formula for unexpected failure of p, while each G_j is a specification formula for unexpected success of p. The α-th formula, denoted by F_α, is called an *query formula for unexpected failure*, while $(\alpha + \beta)$-th formula, denoted by G_β, is called an *query formula for unexpected success*, where "*query-p*" is a special predicate such that "*query-*$p(t_1, t_2, \ldots, t_n)$ succeeds by asking the programmer an appropriate answer substitution θ" if and only if "$p(t_1, t_2, \ldots, t_n)$ succeeds with answer substitution θ when the program of p is correctly written." Specifications by properties are used when it is hard to write a concise executable specification but it is relatively easy to write various properties the intended program should have.

(iii) Specification by Queries

```
spec(p).
   p(X1, X2, ..., Xn) :-: query-p(X1, X2, ..., Xn).
endspec.
```

This is equivalent to writing

```
spec(p).
   p(X1, X2, ..., Xn) :- query-p(X1, X2, ..., Xn).
   p(X1, X2, ..., Xn) -: query-p(X1, X2, ..., Xn).
endspec.
```

Specifications by queries are used when the predicate is so trivial that it is tiresome to write even its properties all the way, hence it is much easier to answer whether atoms with individual arguments are true or not.

Example 2.2.3 The following is an example of executable specifications [2].

```
spec(qsort).
   qsort(L,M) :-: permute(L,M), ordered(M).
endspec.
```

The following is an example of specifications by properties.

```
spec(partition).
   partition(L,X,L,[ ]) :- ge-all(X,L).
   partition(L,X,[ ],L) :- lt-all(X,L).
   partition(L,X,L1,L2) :- query-partition(L,X,L1,L2).
   partition(L,X,L1,L2) -: sublist(L1,L), sublist(L2,L), ge-all(X,L1), lt-all(X,L2).
   partition(L,X,L1,L2) -: if length(L1,N1), length(L2,N2), length(L,N)
                           then add(N1,N2,N).
   partition(L,X,L1,L2) -: query-partition(L,X,L1,L2).
endspec.
```

The following is an example of specifications by queries.

```
spec(append).
   append(L,M,N) :-: query-append(L,M,N).
endspec.
```

(4) Specification of Program

A set S consisting of the specifications of predicates in program P is called a *specification* of P. Non-primitive predicates without specifications are assumed to be specified by "specification by queries."

Example 2.2.4 Let S be the set consisting of the specifications of "*qsort*," "*partition*" and "*append*" of Example 2.2.3. Then S is a specification of the program of Example 2.1.1.

3. Outline of A Logic Program Modification System

3.1 Top Level of the Logic Program Modification System

In our program modification process, each predicate is either marked "debugged" or unmarked, and each specification formula is either marked "checked" or unmarked. For a given initial program P_0, the top-level of our logic program modification system is as below:

```
logic-program-modification($P_0$ : program; $S$ : specification);
  $P := P_0$;
  repeat
        select an unmarked predicate in $S$ to be debugged;
        repeat
              select an unmarked specification formula of the predicate;
              modify $P$ using the specification formula;
              mark the specification formula "checked";
        until all the specification formulas of the predicate are marked "checked";
        mark the predicate "debugged";
  until all the predicates are marked "debugged";
  return the modified $P$;
```

Figure 3.1 Top Level of the Logic Program Modification System

The algorithm first selects one of the predicates to be debugged. Next the algorithm selects one of the specification formulas in the specification of the predicate. Then the algorithm modify the program using the specification formula, and continues the modification using other specification formulas until all the specification formulas of the predicate are used. This process continues until all the predicates are debugged.

3.2 Strategies in Selecting Test Predicates and Specification Formulas

In Figure 3.1, how to select the predicates to be debugged and how to select the specification formulas to be checked are left unspecified. In this paper, we have adopted the following strategies:

Strategy of Test Predicate Selection

The predicates to be debugged is interactively specified by the programmer. (For other strategies to automatically select the predicates, see Section 8.)

Strategy of Specification Formula Selection

The specification formulas appearing earlier between **spec** and **endspec** in the specification of a predicate are selected earlier than those appearing later. (Hence, specification formulas for unexpected failure are selected earlier than ones for unexpected success.)

3.3 Three Phases in the Logic Program Modification

The innermost of our logic program modification at line 7 in Figure 3.1
"modify P using the specification formula"
consists of three phases, *test case generation phase*, *bug location phase* and *bug correction phase* as below:

```
modify($P$ : program; $F$ : specification formula);
  repeat
      generate a next test case using $F$;        (Test Case Generation Phase)
      locate a bug in $P$;                              (Bug Location Phase)
      correct the bug in $P$;                         (Bug Correction Phase)
  until either the test cases using $F$ are exhausted up or the programmer says "o.k." ;
```

Figure 3.3 Three Phases in Logic Program Modification

The detail of the "test case generation phase" is to be explained in Section 4, and the "bug location phase" in Section 5. The "bug correction phase" searches the candidates of corrected programs. It is, however, very hard for general bugs, in particular when even the skeletal recursion structures of the programs are wrong. In this paper, we assume for simplicity that this phase is done through interactive editting of the program text by the programmer.

4. Test Case Generation Phase

4.1 The Principle of the Test Case Generation

Let P be a program (containing at least one constant symbol), U_{EH} be the set of all terms composed of constant and function symbols occurring in P and countably infinite number of variables, and B_{EH} be the set of all atoms with predicate symbols occuring in P and arguments contained in U_{EH}. A model of P over domain U_{EH} is called an *extended Herbrand model* when constant and function symbols are interpreted symbolically as they are. An extended Herbrand model can be considered a subset of B_{EH} if an atom is interpreted *true* when the atom is an element of the subset and *false* otherwise. Hereafter, we will use a prefix "B_{EH}-" when we would like to emphasize that an expression is consisting of atoms in B_{EH}.

Let M be the set of all B_{EH}-atoms that succeeds using P without instantiating the variables in it. Then, M is a model of P over domain U_{EH}, and called the *least extended Herbrand model* of P. Let F be a universal formula of the form

$B_1 \land B_2 \land \cdots \land B_l \supset B_{l+1} \land B_{l+2} \land \cdots \land B_{l+k}$.

Suppose that this formula is valid in M, and the execution of

?- $B_1, B_2, \ldots, B_l$

using P succeeds with answer substitution θ. (Different answer substitutions give different test cases.) Then, universal formula

$(B_1 \land B_2 \land \cdots \land B_l \supset B_{l+1} \land B_{l+2} \land \cdots \land B_{l+k})\theta$.

must be valid in M. Because $\forall(B_1 \land B_2 \land \cdots \land B_l)\theta$ is valid in M,

$\forall(B_{l+1} \land B_{l+2} \land \cdots \land B_{l+k})\theta$

must be valid in M. Therefore, the execution of

?- $(B_{l+1}, B_{l+2}, \ldots, B_{l+k})\theta$

must succeeds using P without instantiating the variables in $(B_{l+1}, B_{l+2}, \ldots, B_{l+k})\theta$. This is the principle of our test case generation.

Hence, in the following, we need to execute a goal

?- $(B_{l+1}, B_{l+2}, \ldots, B_{l+k})\theta$

without instantiating the variables in it. In general, we need to execute a goal

?- $A_1, A_2, \ldots, A_n$

while prohibiting the instantiation of some variables $X_1, X_2, \ldots, X_l$ and permitting the instantiation of another variables $Y_1, Y_2, \ldots, Y_k$, which corresponds to prove

$\forall\, X_1, X_2, \ldots, X_l\ \exists\, Y_1, Y_2, \ldots, Y_k\ (A_1 \wedge A_2 \wedge \cdots \wedge A_n)$.

(Such a goal appears during the execution even if the top-level goal is just universally quantified.) Hereafter, we will consider such goals, and assume that we somehow make a distinction between the variables quantified universally at the outermost and another variables quantified existentially at the innermost (e.g., by attaching the tilde mark over the universal variables). When we would like to emphasize that some universally quantified variables might be contained in an atom, we call the atom a *quantified atom*, and when we would like to emphasize that no existentially quantified variables are contained in the atom, we call it a *universally quantified atom*.

4.2 Generating Test Cases for Unexpected Failure

When a specification formula for unexpected failure is selected, unexpected failure of the predicate is checked as below:

```
generate-a-test-case-for-unexpected-failure($P$ : program; $F$ : specification formula);
  let $A$ be the head of $F$;
  let $\theta$ be an answer substitution obtained by executing the body of $F$;
  if the execution of the head of $F\theta$ can succeed
     without instantiating the variables in the head
  then return "$A\theta$ has succeeded as expected"
  else return "$A\theta$ has failed unexpectedly";
```

Figure 4.2 Generating A Test Case for Unexpected Failure

Example 4.2 Suppose that the selected specification formula is

qsort(L,M) :- permute(L,M), ordered(M).

The execution of the body can succeed, for example, with answer substitution

$<L \Leftarrow [\,], M \Leftarrow [\,]>$.

Then, the execution of "*qsort*([ ],[ ])" must succeed using the current program if it is the intended program.

Similarly, if the selected specification formula is

partition(L,X,L,[ ]) :- ge-all(X,L),

the execution of the body can succeed, for example, with answer substitution

$<L \Leftarrow [\,]>$.

Then, the execution of "*partition*([ ], X, [ ], [ ])" must succeed without instantiating X.

If the selected specification formula is

append(L,M,N) :- query-append(L,M,N).

then the execution of the body asks the programmer

WHAT GROUND INSTANCE OF *append*(L, M, N) SHOULD SUCCEED?

If we answer it, say, *append*([0, 1], [2], [0, 1, 2]), then the execution of the atom must succeed.

Remark. The skeleton of the test case generation is easily implemented in Prolog by

```
generate-test-cases((Head :- Body), Test-Result) :-
```

```
        execute(Body),
          replace-variable-with-fresh-constant(Head, Head0), check(Head0, Test-Result).
    generate-test-cases((Head :- Body), test-cases-exhausted).
    check(Head0, has-succeeded-as-expected) :- execute(Head0), !, fail.
    check(Head0, has-failed-unexpectedly).
```

although it might loop forever.

4.3 Generating Test Cases for Unexpected Success

When a specification formula for unexpected success is selected, unexpected success of the predicate is checked as below:

```
generate-a-test-case-for-unexpected-success($P$ : program; $F$ : specification formula);
  let $A$ be the head of $F$;
  let $\theta$ be an answer substitution obtained by executing the head of $F$;
  let $\eta$ be an answer substitution obtained by executing the "if" part of $F\theta$;
  if the execution of the "then" part of $F\theta\eta$ can succeed
    without instantiating the varaibles in the "then" part
  then return "$A\theta\eta$ might succeed"
  else return "$A\theta\eta$ has succeeded unexpectedly";
```

Figure 4.3 Generating A Test Case for Unexpected Success

Example 4.3 Suppose that the selected specification formula is

qsort(L,M) -: permute(L,M), ordered(M).

The execution of the head can succeed, for example, with answer substitution

$<L \Leftarrow [X], M \Leftarrow [X]>$

using the program of Example 2.1.2. If the program were the intended one, the execution of "*permute*$([X],[X])$, *ordered*$([X])$" would have to succeed without instantiating variable X.

Similarly, if the selected specification formula is

partition(L,X,L1,L2) -: sublist(L1,L), sublist(L2,L), ge-all(X,L1), lt-all(X,L2),

the execution of the head can succeed, for example, with answer substitution

$<L \Leftarrow [Y], L1 \Leftarrow [\,], L2 \Leftarrow [Y]>$.

If the program were the intended one, the execution of "*sublist*$([\,],[Y])$, *sublist*$([Y],[Y])$, *ge-all*$(X,[\,])$, *lt-all*$(X,[Y])$" would have to succeed without instantiating variables X and Y.

If the selected specification formula is

append(L,M,N) -: query-append(L,M,N)

test cases are generated and checked in a special way as follows:

- either the head atom is executed, and the programmer is asked whether the execution result is the intended one,
- or the programmer is asked a ground instance of *append*(L, M, N) which should fail, and the ground atom is executed w.r.t. the current program.

In the former case, since the execution of the head can succeed, for example, with answer substitution

$<L \Leftarrow [\,], N \Leftarrow [\,]>$

the programmer is asked

IS SOME INSTANCE OF append([ ],M,[ ]) FALSE?

In the latter case, he/she is asked

WHAT GROUND INSTANCE OF append(L,M,N) SHOULD FAIL?

If he/she answers it, say, *append*$([0],[1],[1,0])$, then the execution of the atom must fail.

Remark. The skeleton of the test case generation is also easily implemented in Prolog by

```
generate-test-cases((Head -: if-then(If,Then)), Test-Result) :-
        execute(Head), execute(If),
        replace-variable-with-fresh-constant(Then, Then0), check(Then0, Test-Result).
generate-test-cases((Head -: if-then(If, Then)), test-cases-exhausted).
check(Then0, might-succeed) :- execute(Then0), !, fail.
check(Then0, has-succeeded-unexpectedly).
```

5. Bug Location Phase

5.1 Top Level of the Bug Location

(1) Success Trace

Suppose that the execution of a quantified atom A has succeeded with answer substitution θ_n after returning answer substitutions $\theta_1, \theta_2, \ldots, \theta_{n-1}$. The trace of the execution since calling A until exiting with $A\theta_n$ is called a *success trace with label* $(A, A\theta_n)$. The quantified atom A is called the *initial label* of the success trace, while the quantified atom $A\theta_n$ is called the *final label* of the success trace.

Let ST be the success trace with label $(A, A\theta_n)$. Suppose that the execution of A with answer substitution θ_n has succeeded using a clause

B :- $B_1, B_2, \ldots, B_k$,

for the top-level atom A. Then there exist substitutions $\sigma, \eta_1, \ldots, \eta_k$ such that θ_n is the restriction of $\sigma\eta_1 \cdots \eta_k$ to the variables in A and

(0) A and B are unifiable by m.g.u. σ,

(1) the execution of $B_1\sigma$ has succeeded with answer substitution η_1 (possibly after succeeding with several other answer substitutions),

(2) the execution of $B_2\sigma\eta_1$ has succeeded with answer substitution η_2 (possibly after succeeding with several other answer substitutions),

$\vdots$

(k) the execution of $B_k\sigma\eta_1 \cdots \eta_{k-1}$ has succeeded with answer substitution η_k (possibly after succeeding with several other answer substitutions).

Then, the success trace with label $(B_i\sigma\eta_1 \cdots \eta_{i-1}, B_i\sigma\eta_1 \cdots \eta_{i-1}\eta_i)$ contained in ST is called an *immediate success subtrace* of ST.

(2) Failure Trace

Suppose that the execution of a quantified atom A has failed after returning answer substitutions $\theta_1, \theta_2, \ldots, \theta_n$. The trace of the execution since calling A until failing is called a *failure trace with label* $(A, [A\theta_1, A\theta_2, \ldots, A\theta_n])$. The success trace with label $(A, A\theta_i)$ in the failure trace is called the *composing success subtrace of* the failure trace.

Let FT be the failure trace with label $(A, [\ldots])$. Suppose that the execution of A in the failure trace has used a clause

B :- $B_1, B_2, \ldots, B_k$

for the top-level atom A. Then, if (an instance of) atom B_i is executed in FT, there exist substitutions $\sigma, \eta_1, \ldots, \eta_{i-1}$ such that

(0) A and B are unifiable by m.g.u. σ,

(1) the execution of $B_1\sigma$ has succeeded with answer substitution η_1 (possibly after succeeding with several other answer substitutions),

(2) the execution of $B_2\sigma\eta_1$ has succeeded with answer substitution η_2 (possibly after succeeding with several other answer substitutions),

$\vdots$

(i) the execution of $B_i\sigma\eta_1 \cdots \eta_{i-1}$ has failed (possibly after succeeding with several other answer substitutions).

Then, the failure trace with label $(B_i\sigma\eta_1 \cdots \eta_{i-1}, [\ldots])$ contained in FT is called the *immediate failure subtrace of FT*.

Note that, when FT is a failure trace with label $(A, [\ldots])$ and B is an instance of A without instantiating the universally quantified variables in A, the failure trace FT' with label $(B, [\ldots])$ is easily obtained from FT

(a) by eliminating some subtraces of FT which never appear due to the instantiation of A to B, and

(b) by instantiating the label of each subtrace according to the instantiation of A to B.

(3) Intended Model and Computed Model

Suppose that the programmer knows whether any given B_{EH}-atom should be true for any assignment for the variables in it. The set of all B_{EH}-atoms that should be true is called the *intended (extended Herbrand) model.*

Let P be the current program the programmer has written. The least extended Herbrand model defined by P is called the *computed (extended Herbrand) model* of the program.

Let "$B_1, B_2, \ldots, B_l$" be an atom sequence and M be an extended Herbrand model. Then, "$B_1, B_2, \ldots, B_l$" is said to be *valid* in M if all B_{EH}-instances of $B_1, B_2, \ldots, B_l$ are true in M, while "$B_1, B_2, \ldots, B_l$" is said to be *invalid* in M if some B_{EH}-instance of $B_1, B_2, \ldots, B_l$ is false in M.

(4) Uncovered Atom and Wrong Clause Instance

When the programmer's intention does not conform to the current program, the intended model does not conform to the computed model.

Let P be a program and M be an intended model. An atom A is called an *uncovered atom* in P w.r.t. M, when there exists some B_{EH}-instance $A\sigma$ such that

(a) $A\sigma$ is true in M, and

(b) for any B_{EH}-instance "$A\sigma$:-$B_1, B_2, \ldots, B_l$" of definite clauses in P, the body $B_1, B_2, \ldots, B_l$ is false in M.

Example 5.1.1 Let P be the program of Example 2.1.1, A be *partition*([0], 1, [0], [ ]), and M be the usual intended model. Then A is an uncovered atom.

Let P be a program and M be an intended model. An instance "H:-$B_1, B_2, \ldots, B_l$" of a definite clause in P is called a *wrong clause instance* in P w.r.t. M when

(a) H is invalid in M, and

(b) $B_1, B_2, \ldots, B_l$ is valid in M.

Example 5.1.2 Let P be the program of Example 2.1.1, C be an instance of the third clause for *partition*

partition([Y],X,[ ],[Y]) :- partition([ ],X,[ ],[ ]),

and M be the usual intended model. Then C is a wrong clause instance.

(5) Locating Bugs Using Traces

"locate-a-bug" below is the bug location algorithm by tracing the execution of the head of the specification formula in a top-down manner. (See [11] for soundness and completeness of the algorithm.) For simplicity, assume that the execution trace T of an atom which has succeeded or failed unexpectedly is completely recorded. (See Section 6 for the actual implementation.) "*locate-a-bug*" works as follows:

(a) If the application of "*locate-a-bug*" to trace T returns an atom "B," it means that B is an uncovered atom found in T.

(b) If the application of "*locate-a-bug*" to trace T returns a clause "C," it means that C is a wrong clause instance found in T.

(c) If the application of "*locate-a-bug*" to trace T returns a message "no bug is found," it means that no bug is found in T.

How to check (i) unexpected failure of the label of a failure trace FT and (ii) unexpected success of the label of a success trace ST using the specification formulas during the top-down tracing is explaind in Section 5.2 and 5.3. Note in advance that, when the unexpected failure check of the label of a failure trace FT returns a quantified atom A', the atom A' is an instance of the label without instantiating the universally quantified variables in it.

```
locate-a-bug($T$ : trace) : bug-message ;
  when $T$ is a failure trace with label $(A, [A_1, A_2, \ldots, A_k])$
    let $T_1, T_2, \ldots, T_k$ be the composing success subtraces of $T$;
    if the application of "locate-a-bug" to some $T_i$ returns a bug
    then return it
    else check unexpected failure of the label of $T$
         if the answer is "No"
         then return "no bug is found"
         else let "Yes($B$)" be the returned answer;
              let $T'$ be the failure trace with label $(B, [\ldots])$ (obtained from $T$);
              let $FT_1, FT_2, \ldots, FT_n$ be the immediate failure subtraces of $T'$;
              if the application of "locate-a-bug" to some $FT_j$ returns a bug
              then return it
              else return the atom "$B$" as a bug
  when $T$ is a success trace with label $(A, B)$
    check unexpected success of the label of $T$
    if the answer is "No"
    then return "no bug is found"
    else let "Yes($C$)" be the returned answer ;
         let $ST_1, ST_2, \ldots, ST_n$ be the immediate success subtraces of $T$;
         if the application of "locate-a-bug" to some $ST_j$ returns a bug
         then return it
         else return the definite clause "$C$" as a bug
```

Figure 5.1 Top Level of the Bug Location Phase

5.2 Checking Unexpected Failure

In Figure 5.1, unexpected failures needed to be checked using the specifications. The check is done as follows:

```
check-unexpected-failure($FT$ : trace);
    let $(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$ be the label of $FT$ ;
    for $i$ from 1 to $\alpha$ do
        if $A$ is unifiable with the head of a specification formula $F_i$, say by m.g.u. $\sigma$,
            without instantiating the universally quantified variables in $A$,
            the excution of the body of $F_i\sigma$ can succeed, say with answer substitution $\eta$,
            without instantiating the universally quantified variables in $A$,
            and $A\sigma\eta$ is not an instance of $A\theta_1, A\theta_2, \ldots, A\theta_k$
        then return "Yes($A\sigma\eta$)" ;
    return "No" ;
```

Figure 5.2 Checking Unexpected Failure

Example 5.2 Suppose that predicate "*partition*" is defined by

partition([ ],X,[ ],[ ]).

partition([Y|L],X,[Y|L1],L2) :- Y≥X, partition(L,X,L1,L2).

partition([Y|L],X,L1,[Y|L2]) :- partition(L,X,L1,L2).

and the failure trace of *partition*([0], 1, [0], [ ]) is given. Because the execution of atom *partition*([0], 1, [0], [ ]) fails without returning any answer, there is no composing success subtraces so that "*locate-a-bug*" immediately checks the label of the failure trace.

Because the label *partition*([0], 1, [0], [ ]) is unifiable with the head of the first specification formula

partition(L,X,L,[ ]) :- ge-all(X,L),

the corresponding body "*ge-all*(1, [0])" is executed. Since it succeeds, "***check-unexpected-failure***" returns

Yes(*partition*([0], 1, [0], [ ])).

Similarly, *partition*([0], 1, [0], [ ]) is unifiable with the head of the third specification formula

partition(L,X,L1,L2) :- query-partition(L,X,L1,L2),

the system asks the programmer

IS partition([0],1,[0],[ ]) TRUE? :

Since the programmer answers "Yes", "*check-unexpected-failure*" also returns

Yes(*partition*([0], 1, [0], [ ])).

(In general, if the atom is not ground and the execution of atom fails after returning answer substitutions $\theta_1, \theta_2, \ldots, \theta_k$, "query-p" in the body asks the programmer whether some other instance of A not subsumed by $A\theta_1, A\theta_2, \ldots, A\theta_k$ is true. When $k = 0$, it simply asks the programmer whether some instance of A is true. In particular, when $k = 0$ and A is a ground atom, it simply asks the programmer whether A is true.)

5.3 Checking Unexpected Success

Similarly, unexpected successes needed to be checked using the specifications in Figure 5.1. The check is done as follows:

```
check-unexpected-success($ST$ : trace);
  let $B$ be the final label of $ST$;
  let $C$ be the definite clause used for $ST$ at its root;
  let $\theta$ be an m.g.u. of $B$ and the head of $C$;
  for $j$ from 1 to $\beta$ do
     if $B$ is unifiable with the head of a specification formula $G_j$, say by m.g.u. $\sigma$,
        without instantiating the variables in $B$,
        the execution of the "if" part of $G_j\sigma$ can succeed, say with answer substitution $\eta$,
        without instantiating the variables in $B$,
        and the execution of the "then" part of $G_j\sigma\eta$ cannot succeed
        without instantiating the variables in the "then" part
     then return "Yes($C\theta$)" ;
  retuen "No" ;
```

Figure 5.3 Checking Unexpected Success

Example 5.3 Suppose that predicate "*partition*" is defined by

partition([ ],X,[ ],[ ]).
partition([Y|L],X,[Y|L1],L2) :- Y≤X, partition(L,X,L1,L2).
partition([Y|L],X,L1,[Y|L2]) :- partition(L,X,L1,L2).

and the success trace of universally quantified atom *partition*$([Y], X, [\ ], [Y])$ is given. Becanse *partition*$([Y], X, [\ ], [Y])$ is an instance of the head of the fourth specification formula

partition(L,X,L1,L2) -: sublist(L1,L), sublist(L2,L), ge-all(X,L1), lt-all(X,L2)

without intantiating X and Y, the corresponding body "sublist([ ],[Y]), sublist([Y],[Y]), ge-all(X,[ ]), lt-all(X,[Y])" is executed. Since it cannot succeed without instantiating variables X and Y, "*check-unexpected-success*" returns

Yes("partition([Y],X,[ ],[Y]) :- partition([ ],X,[ ],[ ])")

Similarly, *partition*$([Y], X, [\ ], [Y])$ is an instance of the head of the sixth specification formula

partition(L,X,L1,L2) -: query-partition(L,X,L1,L2),

the system asks the programmer

IS SOME INSTANCE OF partition([Y],X,[ ],[Y]) FALSE? :

Since the programmer answers "Yes", "*locate-unexpected-success*" also returns

Yes("partition([Y],X,[ ],[Y]) :- partition([ ],X,[ ],[ ])")

(When the atom B is ground, "query-p" in the body simply asks the programmer whether B is false.)

6. Implementation

Three databases are utilized for improving the efficiency at each modification cycle, and five windows are used as interface to display the modification process.

(1) Database of Intended Execution Results (INTENDED-MODEL-BASE)

The first database is "INTENDED-MODEL-BASE" to record the intended execution results which have been confirmed by the specifications (or by the programmer) during the modification cycles so far. (Hence, the size of this database increases as the modification process proceeds.)

INTENDED-MODEL-BASE records the facts either of the form

$should\text{-}fail(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$

or of the form

should-succeed(A).

The first fact says that the execution of quantified atom A should fail after returning solutions that are subsumed by $A\theta_1, A\theta_2, \ldots, A\theta_k$ and that subsume $A\theta_1, A\theta_2, \ldots, A\theta_k$. (Hence, *should-fail*$(A, [\,])$ says that the execution of atom A should fail without returning any solution.) The second fact says that the execution of universally quantified atom A should succeed (without instantiating the variables in it).

A new fact is added to INTENDED-MODEL-BASE in the following cases:

(a) a fact *should-succeed*$(A\theta)$ is added when the test case $A\theta$ is generated in the test case generation phase for unexpected failure,

(b) a fact *should-fail*$(A\theta\eta, [\,])$ is added when the execution of $A\theta\eta$ has succeeded unexpectedly,

(c) a fact *should-fail*$(A\theta, [\,])$ is added when a ground atom $A\theta$ is given from the programmer as an atom which should fail in the test case generation phase for G_β,

(d) a fact *should-fail*$(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$ is added when check of failure trace with label $(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$ ends without detecting unexpected failure in the bug location phase,

(e) a fact *should-succeed*(A) is added when check of success trace labelled with final label A ends without detecting unexpected success in the bug location phase.

This database is used for checking whether the execution results using the new program are consistent with the programmer's intention. Whenever the program is modified to a new program at some modification cycle, the contents in INTENDED-MODEL-BASE is rechecked w.r.t. the new program. (If a fact in INTENDED-MODEL-BASE is consistent with the new program, it is recorded in COMPUTED-MODEL-BASE explained in the next section.) If some fact in INTENDED-MODEL-BASE is not consistent with the new program, it automatically causes a new modification cycle. (Cf. Section 8.)

This database is also used for improving the efficiency of checking unexpected failure and unexpected success in the bug location phase as below: (The algorithm for unexpeceted failure in Figure 6.3 has not yet fully utilized the information, because more computation is necesary to do that.)

```
check-unexpected-failure($FT$ : failure trace);
  let $(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$ be the label of $FT$ ;
  if there is a fact should-fail$(A, [A\zeta_1, A\zeta_2, \ldots, A\zeta_l])$ in INTENDED-MODEL-BASE
     such that $\{A\theta_1, A\theta_2, \ldots, A\theta_k\}$ is identical to $\{A\zeta_1, A\zeta_2, \ldots, A\zeta_l\}$
  then return "No"
  else for $i$ from 1 to $\alpha$ do
          if $A$ is unifiable with the head of $F_i$, say by m.g.u. $\sigma$,
             without instantiating the universally quantified variables in $A$,
             the excution of the body of $F_i\sigma$ can succeed, say with answer substitution $\eta$,
             without instantiating the universally quantified variables in $A$,
             and $A\sigma\eta$ is not an instance of $A\theta_1, A\theta_2, \ldots, A\theta_k$
          then return "Yes($A\sigma\eta$)" ;
       record should-fail$(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$ to INTENDED-MODEL-BASE ;
  record has-failed$(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$ to COMPUTED-MODEL-BASE;
  return "No" ;
```

Figure 6.1 Checking Unexpected Failure Using INTENDED-MODEL-BASE

```
check-unexpected-success($ST$ : success trace);
  let $B$ be the final label of $ST$ ;
  let $C$ be the definite clause used for $ST$ at its root;
  let $\theta$ be an m.g.u. of $A$ and the head of $C$;
  if there is a fact should-succeed($B$) in INTENDED-MODEL-BASE
     such that $A$ is an instance of $B$
  then return"No"
  else for $j$ from 1 to $\beta$ do
          if $B$ is unifiable with the head of a specification formula $G_j$
             without instantiating the variables in $B$, say by m.g.u. $\sigma$,
             the execution of the "if" part of $G_j\sigma$ can succeed
             without instantiating the variables in $B$, say with answer substitution $\eta$,
             and the execution of the "then" part of $G_j\sigma\eta$ cannot succeed
             without instantiating the variables in the "then" part
          then return "Yes($C\theta$)" ;
       record should-succeed($A$) to INTENDED-MODEL-BASE ;
  record has-succeeded($A$) to COMPUTED-MODEL-BASE;
  retuen "No" ;
```

Figure 6.2 Checking Unexpected Success Using INTENDED-MODEL-BASE

(2) Database of Actual Execution Results (COMPUTED-MODEL-BASE)

The second database is "COMPUTED-MODEL-BASE" to record the actual execution results which the current program gives. (Hence, each program at each modification cycle has respective COMPUTED-MODEL-BASE.)

COMPUTED-MODEL-BASE records the facts either of the form

$has\text{-}failed(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$

or of the form

$has\text{-}succeeded(A)$.

The first fact says that the execution of quantified atom A has failed as expected after returning solutions $A\theta_1, A\theta_2, \ldots, A\theta_k$. (Hence, $has\text{-}failed(A, [\,])$ says that the execution of atom A has failed as expected without returning any solution.) The second fact says that the execution of universally quantified atom A has succeeded as expected (without instantiating the variables in it).

A new fact is added to COMPUTED-MODEL-BASE in the following cases:

(a) a fact $has\text{-}succeeded(A\theta\eta)$ is added when the execution of $A\theta\eta$ has succeeded as expected in the test case generation phase,

(b) a fact $has\text{-}failed(A\theta, [\,])$ is added when the execution of $A\theta$ has failed as expected in the test case generation phase for B_β,

(c) a fact $has\text{-}failed(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$ is added when check of failure trace with label $(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$ ends without detecting unexpected failure in the bug location phase, or

(d) a fact $has\text{-}succeeded(A)$ is added when check of success trace with final label A ends without detecting unexpected success in the bug location phase.

Note that, if *has-failed*$(A, [\ldots])$ is in COMPUTED-MODEL-BASE, *should-fail*$(A, [\ldots])$ is also in INTENDED-MODEL-BASE, and if *has-succeeded*(A) is in COMPUTED-MODEL-BASE, *should-succeed*(A) is also in INTENDED-MODEL-BASE.

The facts of the form "*has-failed*$(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$" in this database are used as below for efficient execution of quantified atom B when its unexpected success is checked,i.e.,

(a) when B is an instance of the head of a specification formula in the test case generation phase for unexpected success, or

(b) when a fact "*should-fail*$(B, [B\zeta_1, B\zeta_2, \ldots, B\zeta_l])$" in INTENDED-MODEL-BASE is rechecked w.r.t. a new program.

```
execute1($B$ : atom);
  if there exists has-failed($A, [A\theta_1, A\theta_2, \ldots, A\theta_k]$) in COMPUTED-MODEL-BASE
    such that $B$ is an instance of $A$ without instantiating the $\forall$-quantified variables in $A$
  then unify $B$ with some $A\theta_i$ and return true
  else execute $B$ one step using the current program;
       apply execute1 to the resulting sequence of atoms one by one
```

Figure 6.3 Executing A "*should-fail*" Atom Using COMPUTED-MODEL-BASE

Similarly, the facts of the form "*has-succeeded*(A)" in this database are used as below for efficient execution of quantified atom B when its unexpected failure is checked, i.e.,

(a) when B is an instance of the head of a specification formula in the test case generation phase for unexpected failure, or

(b) when a fact "*should-succeed*(B)" in INTENDED-MODEL-BASE is rechecked w.r.t. a new program.

```
execute2($B$ : atom);
  if there exists has-succeeded($A$) in COMPUTED-MODEL-BASE
    such that $B$ is an instance of $A$
  then return $B$
  else execute $B$ one step using the current program;
       apply execute2 to the resulting sequence of atoms one by one
```

Figure 6.4 Executing A "*should-succeed*" Atom Using COMPUTED-MODEL-BASE

(3) Database of Execution Traces (TRACE-BASE)

The third database is "TRACE-BASE" to record the information about the execution trace of a quantified atom being debugged. In the explanation of the bug location phase in Section 5, we have assumed for simplicity that all the execution trace is completely recorded to be utilized in the later top-down re-tracing. We, however, only need the relations between the label of the subtraces and the clauses used so that our implementation records much less information than expected from the explanation in Section 5.

This database is created when (an intance of) the head of a specification formula is executed in the test case generation phase or a quantified atom in INTENDED-MODEL-BASE is executed for rechecking w.r.t. a new program. This database is discarded as soon as the execution result is known to be as expected or the bug location phase ends.

Moreover, although we have shown in Section 5 an algorithm which traces the execution of all predicates, it is easy to modify the algorithm to "spy" the execution of a specific predicate first so that the calls with the same predicate are checked continually. This modified algorithm, called *zooming algorithm* in [11], makes the space temporarily necessary for storing trace records much smaller.

(4) Interface Using Five Subwindows

The following five subwindows are used to make it easy to see the modification process:

(i) "SESSION Window" (the middle in the figure below),
(ii) "SPECIFICATION Window" (the top-left in the figure below),
(iii) "PROGRAM Window" (the top-right in the figure below),
(iv) "INTENDED-MODEL-BASE Window" (the bottom-left in the figure below), and
(v) "COMPUTED-MODEL-BASE Window" (the bottom-right in the figure below).

The "SPECIFICATION Window" and the "PROGRAM Window" are the subwindows that provide the capability of displaying and editting text, while the other three are the subwindows that only provide the capability of displaying, scrolling and reading-in text. See Section 7 for how those five subwindows are used.

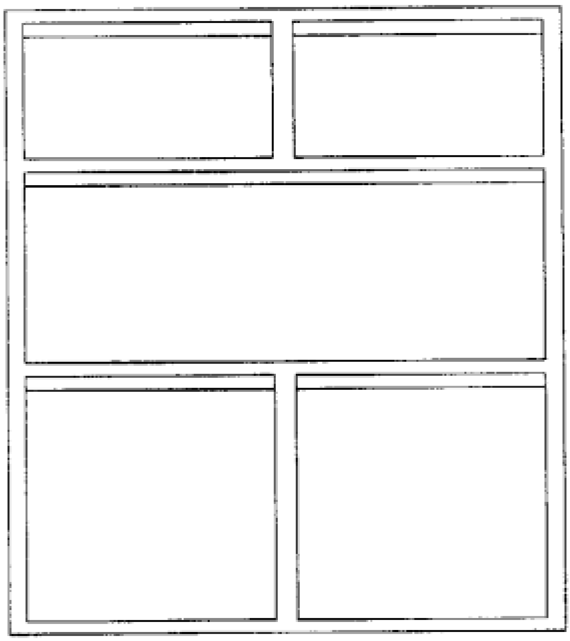

Figure 6.5 Interface Using Five Subwindows

7. An Example of Program Modification

Let us examine how the logic program modification proceeds. Suppose that we have the program of Example 2.1.1 as an intial program, and the specification of Example 2.2.3.

(1) Selecting "*partition*"

First of all, the system asks the programmer which predicate to debug as below:

```
MODIFICATION SESSION [0]:
WHICH PREDICATE TO DEBUG [1]? :
```

Figure 7.1 SESSION Window at Step 0

"[*i*]" to the right of "MODIFICATION SESSION" denotes that we have finished the *i*-th session. "[*j*]" to the right of "WHICH PREDICATE TO DEBUG" denotes that we are starting the *j*-th session. Following the prompt ":", we may type in any predicate name we would like to debug. Suppose that the programmer has typed in "*partition*" and the carriage return. The system will first show the information about "*partition*" in other four windows as below:

```
partition SPECIFICATION [0]:
partition(L,X,L,[ ]) :- ge-all(X,L).
partition(L,X,[ ],L) :- lt-all(X,L).
partition(L,X,L1,L2) :- @.
partition(L,X,L1,L2) -: sublist(L1,L), sublist(L2,L), ge-all(X,L1), lt-all(X,L2).
partition(L,X,L1,L2) -: if length(L1,N1), length(L2,N2), length(L,N) then add(N1,N2,N).
partition(L,X,L1,L2) -: @.
```

```
partition PROGRAM [0]:
partition([ ],X,[ ],[ ]).
partition([Y|L],X,[Y|L1],L2) :- Y≥X, partition(L,X,L1,L2).
partition([Y|L],X,L1,[Y|L2]) :- partition(L,X,L1,L2).
```

Figure 7.2 SPECIFICATION and PROGRAM Windows of "partition" at Step 0

```
partition INTENDED-MODEL-BASE [0]:
```

```
partition COMPUTED-MODEL-BASE [0]:
```

Figure 7.3 MODEL-BASE Windows of "partition" at Step 0

The SPECIFICATION window shows the current specification, while the PROGRAM window the current program. (@ is an abbreviation of "*query-partition*$(L, X, L1, L2)$.") Because no modification has been ever done, the INTENDED-MODEL-BASE window and the COMPUTED-MODEL-BASE window show nothing.

The system starts the modification process by selecting a specification formula of "*partition*." Althogh it is not relevant how the predicates in the bodies of the specification formulas are defined, they are probably predefined as below. (Note that several test cases are given for primitive predicates "≥" and "<.")

```
ge-all(X,[ ]).
ge-all(X,[Y|L]) :- X≥Y, ge-all(X,L).
lt-all(X,[ ]).
lt-all(X,[Y|L]) :- X<Y, lt-all(X,L).
0≥0.
1≥0.
2≥0.
1≥1.
2≥1.
2≥2.
0<1.
0<2.
1<2.
sublist([ ],L).
sublist([X|M],[X|L]) :- sublist(M,L).
sublist(M,[X|L]) :- sublist(M,L).
length([ ],0).
length([X|L],A+1) :- length(L,A).
add(0,B,B).
add(A+1,B,C+1) :- add(A,B,C).
```

The system first selects the specification formula
partition(L,X,L,[ ]) :- ge-all(X,L),
since it is at the top of the specification formulas of "*partition.*" The first test case is generated by executing the body "ge-all(X,L)." The SESSION window responds as below:

```
MODIFICATION SESSION [1]:
WHICH PREDICATE TO DEBUG [1]? : partition

GENERATING A TEST CASE ...
  FROM partition(L,X,L,[ ]) :- ge-all(X,L)
  partition([ ],X,[ ],[ ]) SHOULD SUCCEED WITHOUT INSTANTIATING X.
  partition([ ],X,[ ],[ ]) HAS SUCCEEDED AS EXPECTED.

CONTINUE THIS TEST BRANCH [2]? (y/n/nn/nnn) :
```

Figure 7.4 SESSION Window at Step 1

The generated test case is added to INTENDED-MODEL-BASE, while the checked test case is added to COMPUTED-MODEL-BASE. A fact "*should-succeed*(A)" in INTENDED-MODEL-BASE is represented by
"A" SHOULD SUCCEED WITHOUT INSTANTIATING ... ,
where "..." are the variables in A, while a fact "*has-succeeded*(A)" in COMPUTED-MODEL-BASE is represented by
"A" HAS SUCCEEDED AS EXPECTED.
In particular, when A is a ground atom, "WITHOUT INSTANTIATING ..." is omitted.

```
partition INTENDED-MODEL-BASE [1]:
partition([ ],X,[ ],[ ]) SHOULD SUCCEED WITHOUT INSTANTIATING X.
```

```
partition COMPUTED-MODEL-BASE [1]:
partition([ ],X,[ ],[ ]) HAS SUCCEEDED AS EXPECTED.
```

Figure 7.5 MODEL-BASE Windows of "partition" at Step 1

After finishing the first cycle, the system asks the programmer whether the test case generation along the same branch should be continued. If the programmer types in ("y" and) the carriage return, the modification process continues by redoing the body of the specification formula as below:

```
MODIFICATION SESSION [2]:
   :
CONTINUE THIS TEST BRANCH [2]? (y/n/nn/nnn) : y

GENERATING A TEST CASE ...
  FROM partition(L,X,L,[ ]) :- ge-all(X,L)
  partition([0],0,[0],[ ]) SHOULD SUCCEED.
  partition([0],0,[0],[ ]) HAS SUCCEEDED AS EXPECTED.

CONTINUE THIS TEST BRANCH [3]? (y/n/nn/nnn) :
```

Figure 7.6 SESSION Window at Step 2

Again, the generated test case and the checked test case are added to INTENDED-MODEL-BASE and COMPUTED-MODEL-BASE. This test branch generates a similar test case as below:

```
MODIFICATION SESSION [3]:
   :
CONTINUE THIS TEST BRANCH [3]? (y/n/nn/nnn) : y

GENERATING A TEST CASE ...
  FROM partition(L,X,L,[ ]) :- ge-all(X,L)
  partition([0,0],0,[0,0],[ ]) SHOULD SUCCEED.
  partition([0,0],0,[0,0],[ ]) HAS SUCCEEDED AS EXPECTED.

CONTINUE THIS TEST BRANCH [4]? (y/n/nn/nnn) :
```

Figure 7.7 SESSION Window at Step 3

This test branch seems to be just extending the length of list $[0, 0, \ldots, 0]$, hence have dived into a loop for generating trivial test cases. To prune off this test branch, we will type in "n" and the carriage return.

```
MODIFICATION SESSION [4]:
   :
CONTINUE THIS TEST BRANCH [4]? (y/n/nn/nnn) : n

GENERATING A TEST CASE ...
  FROM partition(L,X,L,[ ]) :- ge-all(X,L)
  partition([0],1,[0],[ ]) SHOULD SUCCEED.
  partition([0],1,[0],[ ]) HAS FAILED UNEXPECTEDLY !!!

LOCATING A BUG ...
  partition([0],1,[0],[ ]) IS AN UNCOVERED ATOM.

CORRECTING THE BUG ...
   :
```

Figure 7.8 SESSION Window at Step 4

In the test case generation phase, the next answer substitution $< X \Leftarrow 1, L \Leftarrow [0] >$ is generated by letting the body "ge-all(X,L)" fail. Since the execution of the instance of the head "*partition*([0], 1, [0], [ ])" fails, an unexpected failure has been detected.

In the bug location phase, the failure trace of "*partition*([0], 1, [0], [ ])" is checked first to return an atom "*partition*([0], 1, [0], [ ])," since the failure trace has no composing success subtrace. Then, "*locate-a-bug* is applied to the only one immediate failure subtrace to return "no bug is found," because the predicate "$\geq$" of the label $(0 \geq 1, [\,])$ of the failure subtrace is primitive so that we can assume that "$\geq$" is always correct. After all, the system will find an uncovered atom "*partition*([0], 1, [0], [ ])."

In the bug correction phase, the second clause of "*partition*" is corrected. ("$\geq$" in the body is in the reverse direction.) According to this correction, the PROGRAM window of "*partition*" is updated to the new program. The contents of INTENDED-MODEL-BASE is rechecked w.r.t. this new program, and the results are shown in the COMPUTED-MODEL-BASE window. (Notice "RECHECKING ..." between the brackets.)

```
partition PROGRAM [4]:
partition([ ],X,[ ],[ ]).
partition([Y|L],X,[Y|L1],L2) :- Y≤X, partition(L,X,L1,L2).
partition([Y|L],X,L1,[Y|L2]) :- partition(L,X,L1,L2).
```

Figure 7.9 PROGRAM Window of "partition" at Step 4

```
partitition INTENDED-MODEL-BASE [4]:
partition([ ],X,[ ],[ ]) SHOULD SUCCEED WITHOUT INSTANTIATING X.
partition([0],0,[0],[ ]) SHOULD SUCCEED.
partition([0,0],0,[0,0],[ ]) SHOULD SUCCEED.
partition([0],1,[0],[ ]) SHOULD SUCCEED.
```

```
partition COMPUTED-MODEL-BASE [RECHECKING ...]:
partition([ ],X,[ ],[ ]) HAS SUCCEEDED AS EXPECTED.
partition([0],0,[0],[ ]) HAS SUCCEEDED AS EXPECTED.
partition([0,0],0,[0,0],[ ]) HAS SUCCEEDED AS EXPECTED.
partition([0],1,[0],[ ]) HAS SUCCEEDED AS EXPECTED.
```

Figure 7.10 MODEL-BASE Windows of "partition" at Step 4

The modification process proceeds in the same way as far as we type in ("y" and) the carriage return. To quit from the test case generation using the current specification formula, we type in "nn" and the carriage return.

```
MODIFICATION SESSION [5]:
   :
CONTINUE THIS TEST BRANCH [5]? (y/n/nn/nnn) : y

GENERATING A TEST CASE ...
   FROM partition(L,X,L,[ ]) :- ge-all(X,L)
   partition([0,0],1,[0,0],[ ]) SHOULD SUCCEED.
   partition([0,0],1,[0,0],[ ]) HAS SUCCEEDED AS EXPECTED.

CONTINUE THIS TEST BRANCH [6]? (y/n/nn/nnn) :
```

Figure 7.11 SESSION Window at Step 5

```
MODIFICATION SESSION [6]:
   :
CONTINUE THIS TEST BRANCH [6]? (y/n/nn/nnn) : nn

GENERATING A TEST CASE ...
   FROM partition(L,X,L,[ ]) :- lt-all(X,L)
   partition([1],0,[ ],[1]) SHOULD SUCCEED.
   partition([1],0,[ ],[1]) HAS SUCCEEDED AS EXPECTED.

CONTINUE THIS TEST BRANCH [7]? (y/n/nn/nnn) :
```

Figure 7.12 SESSION Window at Step 6

We can similarly traverse the computation tree of "*lt-all*(X, L)" as below:

```
MODIFICATION SESSION [7]:
   :
CONTINUE THIS TEST BRANCH [7]? (y/n/nn/nnn) : y

GENERATING A TEST CASE ...
  FROM partition(L,X,L,[ ]) :- lt-all(X,L)
  partition([1,1],0,[ ],[1,1]) SHOULD SUCCEED.
  partition([1,1],0,[ ],[1,1]) HAS SUCCEEDED AS EXPECTED.

CONTINUE THIS TEST BRANCH [8]? (y/n/nn/nnn) :
```

Figure 7.13 SESSION Window at Step 7

```
MODIFICATION SESSION [8]:
   :
CONTINUE THIS TEST BRANCH [8]? (y/n/nn/nnn) : n

GENERATING A TEST CASE ...
  FROM partition(L,X,L,[ ]) :- lt-all(X,L)
  partition([2],0,[ ],[2]) SHOULD SUCCEED.
  partition([2],0,[ ],[2]) HAS SUCCEEDED AS EXPECTED.

CONTINUE THIS TEST BRANCH [9]? (y/n/nn/nnn) :
```

Figure 7.14 SESSION Window at Step 8

If we type in "nn" here, the third specification formula is used as below:

```
MODIFICATION SESSION [9]:
   :
CONTINUE THIS TEST BRANCH [9]? (y/n/nn/nnn) : nn

GENERATING A TEST CASE ...
  FROM partition(L,X,L1,L2) :- @
  WHAT GROUND INSTANCE OF partition(L,X,L1,L2) SHOULD SUCCEED? :
```

Figure 7.15 SESSION Window at Step 9

Because the body of the specification formula is @, the system asks the programmer to type in a test case. Say that "*partition*$([2,0], 1, [0], [2])$" is typed in, the SESSION Window responds as below:

```
MODIFICATION SESSION [9]:
    :
CONTINUE THIS TEST BRANCH [9]? (y/n/nn/nnn) : nn

GENERATING A TEST CASE ...
   FROM partition(L,X,L1,L2) :- @
   WHAT GROUND INSTANCE OF partition(L,X,L1,L2) SHOULD SUCCEED? :
      partition([2,0],1,[0],[2])
   partition([2,0],1,[0],[2]) HAS SUCCEEDED AS EXPECTED.

CONTINUE THIS TEST BRANCH [10]? (y/n/nn/nnn) :
```

Figure 7.16 SESSION Window at Step 9 (Continued)

We can test as many ground atoms as we desire using this query formula. To proceed to the next specification formulas for unexpected success, we will type in "nn" again. Notice the new messages below:

```
MODIFICATION SESSION [10]:
    :
CONTINUE THIS TEST BRANCH [10]? (y/n/nn/nnn) : nn

GENERATING A TEST CASE ...
   FROM partition(L,X,L1,L2) -: sublist(L1,L), sublist(L2,L), ge-all(X,L1), lt-all(X,L2)
   partition([ ],X,[ ],[ ]) HAS SUCCEEDED WITHOUT INSTANTIATING X.
   partition([ ],X,[ ],[ ]) MIGHT SUCCEED.

CONTINUE THIS TEST BRANCH [11]? (y/n/nn/nnn) :
```

Figure 7.17 SESSION Window at Step 10

```
MODIFICATION SESSION [11]:
    :
CONTINUE THIS TEST BRANCH [11]? (y/n/nn/nnn) : y

GENERATING A TEST CASE ...
   FROM partition(L,X,L1,L2) -: sublist(L1,L), sublist(L2,L), ge-all(X,L1), lt-all(X,L2)
   partition([0],0,[0],[ ]) HAS SUCCEEDED.
   partition([0],0,[0],[ ]) MIGHT SUCCEED.

CONTINUE THIS TEST BRANCH [12]? (y/n/nn/nnn) : y
```

Figure 7.18 SESSION Window at Step 11

After several modification sessions, we shall encounter a wrong case. In the test case generation phase, atom "*partition*$([Y], X, [\,], [Y])$" succeeds using the current program without instantiating X and Y, while the execution of the body of the specification formula says that it may not succeed if X and Y are not instantiated. In the bug location phase, the success trace of "*partition*$([Y], X, [\,], [Y])$" is checked to immedeately find a wrong clause instance, since "*partition*$([\,], X, [\,], [\,])$" is known to be true from the contents of INTENDED-MODEL-BASE. In the bug correction phase, the third clause is corrected. (The comparison "$Y > X$" is missed in the body.)

```
MODIFICATION SESSION [14]:
   ⋮
CONTINUE THIS TEST BRANCH [14]? (y/n/nn/nnn) : n

GENERATING A TEST CASE ...
  FROM partition(L,X,L1,L2) -: sublist(L1,L), sublist(L2,L), ge-all(X,L1), lt-all(X,L2)
  partition([Y],X,[ ],[Y]) HAS SUCCEEDED WITHOUT INSTANTIATING X,Y.
  partition([Y],X,[ ],[Y]) SHOULD FAIL IF X,Y ARE NOT INSTANTIATED !!!

LOCATING A BUG ...
  "partition([Y|L],X,L1,[Y|L2]) :- partition(L,X,L1,L2)" HAS A WRONG INSTANCE
       "partition([Y],X,[ ],[Y]) :- partition([ ],X,[ ],[ ])"

CORRECTING THE BUG ...
   ⋮
```

Figure 7.19 SESSION Window at Step 14

The PROGRAM window is updated accordingly. INTENDED-MODEL-BASE is re-checked, and the results are added to COMPUTED-MODEL-BASE. A fact "*should-fail*$(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$" in INTENDED-MODEL-BASE is represented by

"A" SHOULD FAIL AFTER $A\theta_1, A\theta_2, \ldots, A\theta_k$ IF ... ARE NOT INSTANTIATED.

where "..." are the universally quantified variables in A, while a fact "*has-failed*$(A, [A\theta_1, A\theta_2, \ldots, A\theta_k])$" in COMPUTED-MODEL-BASE is represented by

"A" HAS FAILED AS EXPECTED.

In particular, when $k = 0$, "AFTER $A\theta_1, A\theta_2, \ldots, A\theta_k$" is omitted, and when there is no universally quantified variables in A, "IF ... ARE NOT INSTANTIATED" is ommitted.

```
partition PROGRAM [14]:
partition([ ],X,[ ],[ ]).
partition([Y|L],X,[Y|L1],L2) :- Y≤X, partition(L,X,L1,L2).
partition([Y|L],X,L1,[Y|L2]) :- Y>X, partition(L,X,L1,L2).
```

Figure 7.20 PROGRAM Windows of "partition" at Step 14

```
partitition INTENDED-MODEL-BASE [14]:
partition([ ],X,[ ],[ ]) SHOULD SUCCEED WITHOUT INSTANTIATING X.
partition([0],0,[0],[ ]) SHOULD SUCCEED.
partition([0,0],0,[0,0],[ ]) SHOULD SUCCEED.
partition([0],1,[0],[ ]) SHOULD SUCCEED.
    ⋮
partition([2,0],1,[0],[2]) SHOULD SUCCEED.
    ⋮
partition([Y],X,[ ],[Y]) SHOULD FAIL IF X,Y ARE NOT INSTANTIATED.
```

```
partition COMPUTED-MODEL-BASE [RECHECKING ...]:
partition([ ],X,[ ],[ ]) HAS SUCCEEDED AS EXPECTED.
partition([0],0,[0],[ ]) HAS SUCCEEDED AS EXPECTED.
partition([0,0],0,[0,0],[ ]) HAS SUCCEEDED AS EXPECTED.
partition([0],1,[0],[ ]) HAS SUCCEEDED AS EXPECTED.
    ⋮
partition([2,0],1,[0],[2]) HAS SUCCEEDED AS EXPECTED.
    ⋮
partition([Y],X,[ ],[Y]) HAS FAILED AS EXPECTED.
```

Figure 7.21 MODEL-BASE Windows of "partition" at Step 14

The modification proceeds in the same way. When the sixth specification formula is selected, the system first execute the head, and ask the programmer whether the execution result is true or not only if the system cannot answer it for itself by consulting INTENDED-MODEL-BASE.

```
MODIFICATION SESSION [16]:
    ⋮
CONTINUE THIS TEST BRANCH [16]? (y/n/nn/nnn) : nn

GENERATING A TEST CASE ...
  FROM partition(L,X,L1,L2) -: @
  partition([ ],X,[ ],[ ]) HAS SUCCEEDED WITHOUT INSTANTIATING X.
  partition([ ],X,[ ],[ ]) SHOULD SUCCEED WITHOUT INSTANTIATING X.

CONTINUE THIS TEST BRANCH [17]? (y/n/nn/nnn) :
```

Figure 7.22 SESSION Window at Step 16

If we type in "nn" here, the system asks the programmer using the same sixth specification formuls as below:

```
MODIFICATION SESSION [17]:
    :
CONTINUE THIS TEST BRANCH [17]? (y/n/nn/nnn) : nn

GENERATING A TEST CASE ...
  FROM partition(L,X,L1,L2) -: @
  WHAT GROUND INSTANCE OF partition(L,X,L1,L2) SHOULD FAIL? :
    partition([2,0],1,[2],[0])
  partition([2,0],1,[0],[2]) HAS FAILED AS EXPECTED.

CONTINUE THIS TEST BRANCH [18]? (y/n/nn/nnn) :
```

Figure 7.23 SESSION Window at Step 17

(2) Selecting "*qsort*"

If we type in "nn" after using the last specification formula of "*partition*," the system again asks the programmer which predicate to debug. Suppose that the programmer has typed in "*qsort*." The system will now show the information about "*qsort*."

```
qsort PROGRAM [17]:
qsort([X|L],M) :- partition(L,X,L1,L2), qsort(L1,M1), qsort(L2,M2), append([X|M1],M2,M).
```

```
qsort SPECIFICATION [17]:
qsort(L,M) :- permute(L,M), ordered(M).
qsort(L,M) -: permute(L,M), ordered(M).
```

Figure 7.24 SPECIFICATION and PROGRAM Windows of "qsort" at Step 17

```
qsort INTENDED-MODEL-BASE [17]:

```

```
qsort COMPUTED-MODEL-BASE [17]:

```

Figure 7.25 MODEL-BASE Windows of "qsort" at Step 17

The system starts the modification process by selecting the specification formula
qsort(L,M) :- permute(L,M), ordered(M).
Again, it is not relevant how "*permute*" and "*ordered*" are defined, but they are probably predefined as below:

```
permute([ ],[ ]).
permute([X|L],M) :- permute(L,N), insert-randomly(X,N,M).
insert-randomly(X,N,[X|N]).
insert-randomly(X,[Y|N],[Y|M]) :- insert-randomly(X,N,M).
ordered([ ]).
ordered([X]).
ordered([X,Y|L]) :- X≤Y, ordered([Y|L]).
0≤0.
0≤1.
0≤2.
1≤1.
1≤2.
2≤2.
```

The SESSION window responds as below:

```
MODIFICATION SESSION [18]:
   ⋮
WHICH PREDICATE TO DEBUG [18]? : qsort

GENERATING A TEST CASE ...
  FROM qsort(L,M) :- permute(L,M), ordered(M)
  qsort([ ],[ ]) SHOULD SUCCEED.
  qsort([ ],[ ]) HAS FAILED UNEXPECTEDLY !!!

LOCATING A BUG ...
  qsort([ ],[ ]) IS AN UNCOVERED ATOM.

CORRECTING THE BUG ...
   ⋮
```

Figure 7.26 SESSION Window at Step 18

In the test case generation phase, the first answer substitution $< L \Leftarrow [\,], M \Leftarrow [\,] >$ is generated by executing the body "$permute(L, M), ordered(M)$." Since the execution of "$qsort([\,],[\,])$" does not succeed, an unexpected failure has been detected.

In the bug location phase, the unexpected failure of "$qsort([\,],[\,])$" is checked immediately to find an uncovered atom "$qsort([\,],[\,])$," since the failure trace of "$qsort([\,],[\,])$" has neither composing success subtraces nor immediate failure traces.

In the bug correction phase, a unit clause "$qsort([\,],[\,])$" is added. According to this correction, the windows change as below:

```
qsort PROGRAM [18]:
qsort([ ],[ ]).
qsort([X|L],M) :- partition(L,X,L1,L2), qsort(L1,M1), qsort(L2,M2), append([X|M1],M2,M).
```

Figure 7.27 PROGRAM Windows of "qsort" at Step 18

```
qsort INTENDED-MODEL-BASE [18]:
qsort([ ],[ ]) SHOULD SUCCEED.
```

```
qsort COMPUTED-MODEL-BASE [RECHECKING ...]:
qsort([ ],[ ]) HAS SUCCEEDED AS EXPECTED.
```

Figure 7.28 MODEL-BASE Windows of "qsort" at Step 18

In the second cycle of the test of "*qsort*," the next answer substitution $< L \Leftarrow [X], M \Leftarrow [X] >$ is generated by redoing the body "$permute(L, M), ordered(M)$." Because the execution of "$qsort([X], [X])$" succeeds, this case ends without the bug location phase and the bug correction phase. (Note that two bugs of "*append*" have cancelled each other so that the execution of "$qsort([X], [X])$" has superficially succeeded.)

```
MODIFICATION SESSION [19]:
   :
CONTINUE THIS TEST BRANCH [19]? (y/n/nn/nnn) : y

GENERATING A TEST CASE ...
  FROM qsort(L,M) :- permute(L,M), ordered(M).
  qsort([X],[X]) SHOULD SUCCEED WITHOUT INSTANTIATING X.
  qsort([X],[X]) HAS SUCCEEDED AS EXPECTED.

CONTINUE THIS TEST BRANCH [20]? (y/n/nn/nnn) :
```

Figure 7.29 SESSION Window at Step 19

The results of this cycle are added to the INTENDED-MODEL-BASE window and the COMPUTED-MODEL-BASE window.

```
qsort INTENDED-MODEL-BASE [19]:
qsort([ ],[ ]) SHOULD SUCCEED.
qsort([X],[X]) SHOULD SUCCEED WITHOUT INSTANTIATING X.
```

```
qsort COMPUTED-MODEL-BASE [19]:
qsort([ ],[ ]) HAS SUCCEEDED AS EXPECTED.
qsort([X],[X]) HAS SUCCEEDED AS EXPECTED.
```

Figure 7.30 MODEL-BASE Windows of "qsort" at Step 19

The modification cycle continues in the same way. The next answer substitution $< L \Leftarrow [0,0], M \Leftarrow [0,0] >$ is generated by redoing "$permute(L, M), ordered(M)$" to detect an unexpected failure.

```
MODIFICATION SESSION [20]:
   :
CONTINUE THIS TEST BRANCH [20]? (y/n/nn/nnn) : y

GENERATING A TEST CASE ...
  FROM qsort(L,M) :- permute(L,M), ordered(M).
  qsort([0,0],[0,0]) SHOULD SUCCEED.
  qsort([0,0],[0,0]) HAS FAILED UNEXPECTEDLY !!!

LOCATING A BUG ...
  IS append([0,0],[ ],[0,0]) TRUE? : y
  IS append([0],[ ],[ ]) TRUE? : n
  append([0,0],[ ],[0,0]) IS AN UNCOVERED ATOM.

CORRECTING THE BUG ...
   :

CONTINUE THIS TEST BRANCH [21]? (y/n/nn/nnn) :
```

Figure 7.31 SESSION Window at Step 20

Note that the specification formula "*append*(L, M, N) :- @" has been used in the bug location phase so that we have needed to answer Yes/No to two queries from the system. In the bug correction phase, the second clause of "*append*" is corrected. (The third argument in the recursive call is a wrong term.)

```
append PROGRAM [20]:
append([ ],M,[ ]).
append([X|L],M,[X|N]) :- append(L,M,N).
```

Figure 7.32 PROGRAM Window of "append" at Step 20

The contents of INTENDED-MODEL-BASE are rechecked whether they are consistent with the new program. (Because "*partition*" has no "caller-calee" relation with the modified predicate "*append*," just those of "*append*" and "*qsort*" are rechecked.)

```
append INTENDED-MODEL-BASE [20]:
append([0,0],[ ],[0,0]) SHOULD SUCCEED.
append([0],[ ],[ ]) SHOULD FAIL.
```

```
append COMPUTED-MODEL-BASE [RECHECKING ...]:
append([0,0],[ ],[0,0]) HAS SUCCEEDED AS EXPECTED.
append([0],[ ],[ ]) HAS FAILED AS EXPECTED.
```

Figure 7.33 MODEL-BASE Windows of "append" at STEP 20

```
qsort INTENDED-MODEL-BASE [20]:
qsort([ ],[ ]) SHOULD SUCCEED.
qsort([X],[X]) SHOULD SUCCEED WITHOUT INSTANTIATING X.
qsort([0,0],[0,0]) SHOULD SUCCEED.
```

```
qsort COMPUTED-MODEL-BASE [RECHECKING ...]:
qsort([ ],[ ]) HAS SUCCEEDED AS EXPECTED.
qsort([X],[X]) HAS SUCCEEDED AS EXPECTED.
qsort([0,0],[0,0]) HAS SUCCEEDED AS EXPECTED.
```

Figure 7.34 MODEL-BASE Windows of "qsort" at STEP 20

The new program is consistent with INTENDED-MODEL-BASE so that the SESSION window shows the prompt to continue the next modification cycle. A new test case for "*qsort*" is generated in the next test case generation phase to find unexpected failure, and a bug of "*append*" is discovered again in the bug location phase as below:

```
MODIFICATION SESSION [21]:
    ⋮
CONTINUE THIS TEST BRANCH [21]? (y/n/nn/nnn) : y

GENERATING A TEST CASE ...
  FROM qsort(L,M) :- permute(L,M), ordered(M).
  qsort([0,1],[0,1]) SHOULD SUCCEED.
  qsort([0,1],[0,1]) HAS FAILED UNEXPECTEDLY !!!

LOCATING A BUG ...
  IS append([0],[1],[0,1]) TRUE? : y
  IS append([ ],[1],[1]) TRUE? : y
  append([ ],[1],[1]) IS AN UNCOVERED ATOM.

CORRECTING THE BUG ...
    ⋮
```

Figure 7.35 SESSION Window at Step 21

The first clause of "*append*" is corrected. (The third argument of the head is a wrong term.) The contents of INTENDED-MODEL-BASE is rechecked again to show the results in the COMPUTED-MODEL-BASE window.

```
append PROGRAM [21]:
append([ ],M,M).
append([X|L],M,[X|N]) :- append(L,M,N).
```

Figure 7.36 PROGRAM Window of "append"at Step 21

```
append COMPUTED-MODEL-BASE [RECHECKING ...]:
append([0,0],[ ],[0,0]) HAS SUCCEEDED AS EXPECTED.
append([0],[ ],[ ]) HAS FAILED AS EXPECTED.
append([0],[1],[0,1]) HAS SUCCEEDED AS EXPECTED.
append([ ],[1],[1]) HAS SUCCEEDED AS EXPECTED.
```

```
qsort COMPUTED-MODEL-BASE [RECHECKING ...]:
qsort([ ],[ ]) HAS SUCCEEDED AS EXPECTED.
qsort([X],[X]) HAS SUCCEEDED AS EXPECTED.
qsort([0,0],[0,0]) HAS SUCCEEDED AS EXPECTED.
qsort([0,1],[0,1]) HAS SUCCEEDED AS EXPECTED.
```

Figure 7.37 COMPUTED-MODE-BASE Windows at Step 21

After several modification sessions for predicate "*qsort*," we shall encounter a wrong test case as below:

```
MODIFICATION SESSION [27]:
   :
CONTINUE THIS TEST BRANCH [27]? (y/n/nn/nnn) : n

GENERATING A TEST CASE ...
  FROM qsort(L,M) :- permute(L,M), ordered(M).
  qsort([1,0],[0,1]) SHOULD SUCCEED.
  qsort([1,0],[0,1]) HAS FAILED UNEXPECTEDLY !!!

LOCATING A BUG ...
  IS append([1,0],[ ],[0,1]) TRUE? : n
  qsort([1,0],[0,1]) IS AN UNCOVERED ATOM.

CORRECTING THE BUG ...
   :
```

Figure 7.38 SESSION Window at Step 27

This time, a bug of "*qsort*" has been discovered so that the second clause of "qsort" is corrected. (The first argument of "*append*" in the body is a wrong term. The element used for partition must be at the head of the second argument.)

```
qsort PROGRAM [27]:
qsort([ ],[ ]).
qsort([X|L],M) :- partition(L,X,L1,L2), qsort(L1,M1), qsort(L2,M2), append(M1,[X|M2],M).
```

Figure 7.39 PROGRAM Window of "qsort" at Step 27

```
qsort COMPUTED-MODEL-BASE [RECHECKING ...]:
qsort([ ],[ ]) HAS SUCCEEDED AS EXPECTED.
qsort([X],[X]) HAS SUCCEEDED AS EXPECTED.
qsort([0,0],[0,0]) HAS SUCCEEDED AS EXPECTED.
qsort([0,1],[0,1]) HAS SUCCEEDED AS EXPECTED.
    ⋮
qsort([1,0],[0,1]) HAS SUCCEEDED AS EXPECTED.
```

Figure 7.40 COMPUTED-MODEL-BASE Window of "qsort" at Step 27

The modification of "*qsort*" proceeds in the same way without encountering a wrong test case.

(3) Selecting "*append*"

After typing several "nn"s to the prompts (or "nnn" to immediately quit the debugging of the predicate), we can skip to the next predicate.

```
MODIFICATION SESSION [31]:
    ⋮
CONTINUE THIS TEST BRANCH [31]? (y/n/nn/nnn) : nnn

WHICH PREDICATE TO DEBUG [31]? : append

GENERATING A TEST CASE ...
  FROM append(L,M,N) :- @
  WHAT GROUND INSTANCE OF append(L,M,N) SHOULD SUCCEED? :
```

Figure 7.41 SESSION Window at STEP 31

The other four windows show the information about "*append.*" Because we have checked several atoms with predicate "*append*" in the modification sessions for "*qsort*," the INTENDED-MODEL-BASE window and the COMPUTED-MODE-BASE window are not empty.

```
append SPECIFICATION [31]:
append(L,M,N) :- @.
append(L,M,N) -: @.
```

```
append PROGRAM [31]:
append([ ],M,M).
append([X|L],M,[X|N]) :- append(L,M,N).
```

Figure 7.42 SPECIFICATION and PROGRAM Windows of "append" at STEP 31

```
append INTENDED-MODEL-BASE [31]:
append([0,0],[ ],[0,0]) SHOULD SUCCEED.
append([0],[ ],[ ]) SHOULD FAIL.
append([0],[1],[0,1]) SHOULD SUCCEED.
append([ ],[1],[1]) SHOULD SUCCEED.
```

```
append COMPUTED-MODEL-BASE [31]:
append([0,0],[ ],[0,0]) HAS SUCCEEDED AS EXPECTED.
append([0],[ ],[ ]) HAS FAILED AS EXPECTED.
append([0],[1],[0,1]) HAS SUCCEEDED AS EXPECTED.
append([ ],[1],[1]) HAS SUCCEEDED AS EXPECTED.
```

Figure 7.43 MODEL-BASE Windows of "append" at STEP 31

Because "*append*" is specified by queries, the programmer need to type in ground instances of "*append*(L, M, N)" which should succeed or fail, or confirm the execution results of "*append*(L, M, N)." The modification of "*append*" proceeds without encountering a wrong case.

Remark. Although we have prepared several test cases for primitive predicates "$\geq$," "$<$" and "$\leq$," our diagnosis algorithm works as well even if those predicates are defined by

$X \geq 0.$
$X+1 \geq Y+1$:- $X \geq Y.$
$0 < Y+1.$
$X+1 < Y+1$:- $X < Y.$
$0 \leq Y.$
$X+1 \leq Y+1$:- $X \leq Y.$

provided that we do not mind typing in more "nn"s to the prompts. (The more we prepare for generating desirable test cases in advance, the more we can dispense with tiresome interactions for skipping undesirable test cases.)

8. Discussion

(1) Strategies of Test Predicate Selection

In Section 3.2, we have adopted the interactive test predicate selection. To automate the selection, we need to introduce some strategies taking all the three phases into consideration, in particular, the *side effects* by modifying the program at the "bug correction phase." The following is one of the strategies which takes advantages of executable specifications:

(a) All the predicates that are called from the selected predicate (either directly or indirectly) are either already debugged (i.e., marked "debugged") or with executable specifications except for the cases when it is impossible to select any predicate while keeping the restriction above due to the mutual recursions.

(b) The predicate at the upper level in the "caller-callee" relation is prefered to that at the lower level except for the cases when it is impossible to compare the levels due to the mutual recursions.

Then, our logic program modification proceeds in principle as follows:

(a) The predicates that do not have executable specifications are debugged earlier than those that have executable specifications.
(b) When the predicates do not have executable specifications, they are debugged in the bottom-up manner in the "caller-callee" relation, i.e., the predicates at the lower level in the "caller-callee" relation are selected earlier than those at the upper level.
(c) When the predicates have executable specifications, they are debugged in the top-down manner in the "caller-callee" relation, i.e., the predicates at the upper level in the "caller-callee" relation are selected earlier than those at the lower level.

This strategy is aiming at utilizing executable specifications instead of the corresponding correct programs even if the correct programs are not yet completed, so that the execution of atom A is done as follows:
(a) if the predicate of the atom is specified by an executable specification, it is not yet marked "debugged," and it is not the predicate just being debugged, then atom A is executed using the executable specification,
(b) otherwise, atom A is executed using the program.

This strategy is also aiming at avoiding the superficial coincidences of the execution reults by observing the hierarchical "caller-callee" relation. If more than one bugs in the lower level predicates accidentally gives the intended execution results for the upper level predicates, rechecking of INTENDED-MODEL-BASE at the later modification cycle causes inconsistency with the new program, which automatically starts a new modification cycle as below. Frequent occurrence of such inconsistency is likely to confuse the debugging process.

```
MODIFICATION SESSION [32]:
   ⋮
CONTINUED AUTOMATICALLY [32]

GENERATING A TEST CASE...
  A CASE INCONSISTENT WITH INTENDED-MODEL-BASE
  ... SHOULD ...
  ... HAS NOT ...

LOCATING A BUG ...
   ⋮

CORRECTING THE BUG ...
   ⋮
```

Figure 8 Session Caused by Rechecking of INTENDED-MODEL-BASE

(2) The Relation between the Test Case Generation and Our Verification

As was explained in Section 4.1, the antecedant part of a specification formula is executed first in the test case generation phase, and then the consequence part under the answer substitution just obtained is executed without instantiating the variables in it. These two kinds of execution are special cases of the first order inference rules developed for proving properties of Prolog programs [5],[7]. Let G be a formula of the form ($n \geq 0$)

$$A_1 \wedge A_2 \wedge \cdots \wedge A_k \supset \exists Y_1, Y_2, \ldots, Y_n(A_{k+1} \wedge A_{k+2} \wedge \cdots \wedge A_{k+l})$$

where the variables other than $Y_1, Y_2, \ldots, Y_n$ are universally qunatified implicitly at the outermost. Then, "Negation as Failute Inference (NFI)" rule and "Definite Clause Inference (DCI)" rule are as follows for this class of formulas. (NFI and DCI are defined for a wider class of formulas. See [5], [7] for the details.)

"Negation as Failure" Inference (NFI)

Let A be an atom in the antecedant part of G. Then, for every definite clause "B :- $B_1, B_2, \ldots, B_m$" in P, whose head B is unifiable with A, say by an m.g.u. τ, we generate a new goal obtained from G by applying τ after replacing A with $B_1 \wedge B_2 \wedge \cdots \wedge B_m$. ($B_1 \wedge B_2 \wedge \cdots \wedge B_m$ is *true* when $m = 0$.) All new variables introduced are treated as new universally quantified variables.

Definite Clause Inference (DCI)

Let A be an atom in the consequence part of G, and "B :- $B_1, B_2, \ldots, B_m$" be a definite clause in P whose head B is unifiable with A without instantiating universally quantified variables, say by an m.g.u. σ. Then, we generate a new goal obtained from G by applying σ after replacing A with $B_1 \wedge B_2 \wedge \cdots \wedge B_m$. ($B_1 \wedge B_2 \wedge \cdots \wedge B_m$ is *true* when $m = 0$.) All new variables introduced are treated as new existentially quantified variables.

Obviosly, the former execution in the test case generation corresponds to the consecutive application of the NFI rule, while the latter execution corresponds to that of the DCI rule. NFI and DCI are, however, just two inference rules of our verification system, and not enough to prove logical properties of logic programs. To *prove* that a specification formula is valid in the least Herbrand model of a given program, another two inference rules, called *simplification* and *computational induction*, need to be used extensively. (Roughly speaking, the simplification rule is for cancelling the antecedant and the consequent when they are identical [5], [7], while the computational induction rule is for utilizing the properties that holds for subcomputation [6].) To *generate* a test case from a specification formula that makes the antecedant part true and *test* whether the generated goal (the consequence part under the answer substitution) is valid in the least Herbrand model, however, NFI and DCI are enough. If the number of the test cases to be generated are finite, there is no difference in the power between the test case generation using just NFI and DCI and the verification using additional inference rules. If the number of the test cases is infinite, however, we cannot check all the test cases. This point is the crucial difference between the two approaches. See [5],[6],[7],[8],[9] for the details.

(3) Relations to Other Works

It was Shapiro that first invented a new debugging style of logic programs [15], [16]. Taking advantages of the declarative character of Prolog (and the automatic database capability for recording the previous answers), he very impressively demonstrated the power of his "Algorithmic Debugger". (Although the style of his debugger seems drastically novel at first glance, it is just automatically providing the points at which we need to check the results so that it is not *completely* discontinuous with the debugging which the programmers usually do by following the execution traces. Our more naive formalization using execution traces in

21

Section 5 is based on [11].) The Shapiro's algorithmic debugging has been extended and refined in several directions, e.g., by Plaisted [14], Pereira [12], Lloyd [10], Maeji and Kanamori [11], Ferrand [4], Pereira and Calejo [13], among others. Although Shapiro suggested the basic idea for mechanizing oracles [15] pp.77–80, he focused his attention on the case when the programmers play the role of the oracle, which corresponds to our "specification by queries" in Section 2.2 (3).

The early proposal to utilize specifications for mechanizing oracles (and even for generating test cases) was done by Edman and Tärnlund [3]. Although it is very sketchy, a succinct explanation of the framework is found in [3] p.554, Section 5. Their approach is different from ours in the following respects:

(a) They utilized *full specifications* rather than *partial specifications.* (Section 1–4 of their paper [3] are cencerned with how to strengthen partial specifications to full specifications.) We have, in general, utilized *partial specifications.*

(b) Their full specifications are, however, not necesarily *executable specifications in the usual sense*, but more general first order formulas, although their extended system (called *programming calculus*) can (probably less efficiently) *execute* them in a more general sense. Our specifications are the class of formulas for which the usual Prolog execution can be effectively utilized.

Dershowitz and Lee not only developed Shapiro's idea for mechanizing oracles but also proposed to utilize specifications for generating test cases [1]. Their approach is different from ours in the following respects:

(a) Their approach used executable specifications of the form

$$p(X_1, X_2, \ldots, X_n) \text{ :- } A_1, A_2, \ldots, A_k.$$

which corresponds to the first half of our "specification by executable specifications" in Section 2.2 (3). Such executable specifications are *full specifications* in the sense that they fully characterize the predicates. Our approach has permitted *partial specifications* as well as the reverse implication of such specifications. Their approach does not utilize the reverse direction *in their test case generation*.

(b) Following the original Shapiro's bug location algorithm, their approach traces the [illegible]x-ecution in a bottom-up manner for unexpected success. Their debugging progra [illegible]n Prolog was very elegantly derived from a Prolog meta-interpreter. Our approac [illegible] o-cates bugs in a top-down manner for both unexpected failure and unexpected success. Although our algorithm does not directly correspond to the behavior of the Prolog meta-interpreter, the top-down tracing very naturally simulates the debugging human programmers usually do.

(c) They made an attempt to utilize specifications in the bug correction phase. Compared with the information the answers of the programmer (to individual queries) give, specifications are more structured and more informative. When some specifications are given, the problem of bug correction is in close relation with the program synthesis/transformation with an approximate structure of the desired program given, or the problem of incorporating the information contained in (possibly partial) specifications into the (possibly incomplete) final program. We expect that their approach is more promising than the inductive inference approach *with naive enumeration* only, although the discussion employing enumeration is useful for investigating the theoretical limits of inductive inference.

Recently, Drabent, Nadjm-Tehrani and Maluszynski [2] showed an approach to utilize partial specifications, called *assertions* in their paper, for (partly) mechanizing oracles independently of us. They emphasized that full specification or executable specifications are sometimes unrealistic so that partial specifications to approximate the intended models are of great practical importance. (See [8],[9] for similar discussion in verification of logic programs.) Moreover, they adopted a top-down bug location algorithm similar to us [11] which requires only Yes/No answers but no instantiation of goals. Their approach is different from ours in the following respects:

(a) The class of formulas for specifications are different between theirs and ours. They used four types of specifications, *positive assertions, negative assertions, positive existential assertions* and *negative existential assertions*, which directly correspond to four types of answers in their debugging system, while we have used two types of implicative formulas. Some of their specifications are included in ours, while some are derived from our specifications. (To immediately include their positive existential assertions, we need to permit existential qunatifiers in the head in our specification formulas for unexpected failure, and slightly modify the conditions in generating test cases and in checking failure traces.)

(b) They had used their specifications only for answering queries, not for generating test cases. We have used them for both generating test cases and answering queries.

9. Conclusions

We have presented a framework for locating bugs of logic programs from specifications. This method is an element of our logic program modification system Argus/M developed from 1987 April to 1989 March.

Acknowledgements

This research was done as a part of the Fifth Generation Computer Systems project of Japan. We would like to thank Dr. K. Fuchi (Director of ICOT) for the opportunity of doing this research, and Dr. K. Furukawa (Deputy Director of ICOT), Dr. R. Hasegawa (Chief of ICOT 1st Laboratory) and Dr. H. Ito (Chief of ICOT 3rd Laboratory) for their advice and encouragement.

References

[1] Dershowitz, N. and Y-J Lee, "Deductive Debugging," Proc. of 1987 Symposium on Logic Programming, pp.298–306, Salt Lake City, August 1987.

[2] Drabent, W., S.Nadjm-Tehrani and J.Maluszynski, "The Use of Assertions in Algorithmic Debugging," Proc. of the International Conference on Fifth Generation Computer Systems 1988, Tokyo, November 1988.

[3] Edman, A. and S.A.Tärnlund, "Mechanization of An Oracle in A Debugging System," Proc. of 8th International Joint Conference on Artificial Intelligence, pp.553–555, 1983.

[4] Ferrand, G., "Error Diagnosis in Logic Programming," J. of Logic Programming, Vol.4, pp.177–198, 1987.

[5] Kanamori, T. and H.Seki, "Verification of Prolog Programs Using An Extension of Execution," Proc. of 3rd International Conference on Logic Programming, pp. 475–489, London, July 1986. Also a preliminary version appeared as ICOT Technical Report TR-096, ICOT, Tokyo, December 1984.

[6] Kanamori, T. and H.Fujita, "Formulation of Induction Formulas in Verification of Prolog Programs," Proc. of 8th International Conference on Automated Deduction, pp. 281–299, Oxford, July 1986. Also a preliminary version appeared as ICOT Technical Report TR-094, ICOT, Tokyo, December 1984.

[7] Kanamori, T., "Soundness and Completeness of Extended Execution for Proving Properties of Prolog Programs," Proc. of 1st France-Japan Artificial Intelligence and Computer Science Symposium, pp. 219–238, Tokyo, October 1986. Also in *Programming in Future Generation Computers* (K.Fuchi and M.Nivat Eds.), pp. 259–281, North-Holland, 1988. Also a preliminary version appeared as ICOT Technical Report TR-175, ICOT, Tokyo, May 1986.

[8] Kanamori, T., H.Fujita, H.Seki, K.Horiuchi and M.Maeji, "Argus/V : A System for Verification of Prolog Programs," Proc. of Fall Joint Computer Conference 86, pp. 994–999, Dallas, October 1986. Also a preliminary version appeared as ICOT Technical Report TR-176, ICOT, Tokyo, May 1986.

[9] Kanamori, T., "Verification of Logic Programs," to appear in *Introduction to Fifth Generation Computers* (K.Fuchi Ed.), The SRI Tokyo Series on Advanced Technology, Prentice-Hall, 1989.

[10] Lloyd, J.W., "Declarative Program Diagnosis," Technical Report 86/3, Department of Computer Science, University of Melbourne, 1986. Also New Generartion Computing, Vol.5, No.2, pp.133-154, 1987.

[11] Maeji, M. and T.Kanamori, "Top-down Zooming Diagnosis of Logic Programs," Presented at RIMS Symposium on Mathematical Methods in Software Science and Engineering '87, Kyoto, September 1987. Also RIMS Research Report 655, pp. 147–166, Research Institute for Mathematical Sciences, Kyoto University, April 1988. Also ICOT Technical Report TR-290, ICOT, Tokyo, August 1987.

[12] Pereira, L.M., "Rational Debugging in Logic Programming," Proc. of 3rd International Conference on Logic Programming, pp. 203–210, 1986.

[13] Pereira, L.M. and M.Calejo, "A Framework for Prolog Debugging," Proc. of 5th International Conference and Symposium on Logic Programming," pp.481–495, Seatle, 1988.

[14] Plaisted, D., "An Efficient Bug Location Algorithm," Proc. of 2nd International Logic Programming Conference, pp. 151–157, 1984.

[15] Shapiro, E.Y., "Algorithmic Program Debugging," An ACM Distinguished Dissertation 1982, The MIT Press, 1983. Also Research Report 237, Yale University, Department of Computer Science, 1982.

[16] Shapiro, E.Y., "Algorithmic Program Diagnosis," Conf. Rec. of the 9th ACM Symposium on Principles of Programming Languages, pp. 299–308, 1984.