J. Symbolic Computation (1993) **15**, 745–773

Logical Debugging†

NACHUM DERSHOWITZ‡ AND YUH-JENG LEE§

‡*Department of Computer Science, University of Illinois at Urbana-Champaign, U.S.A.*
§*Computer Science Department, Naval Postgraduate School, U.S.A.*

(Received 15 March 1993)

A methodology for reasoning about logic programs and their specifications is applied to program debugging as well as program synthesis. Specifications in the form of an executable model of the desired program are used to generate test cases, locate bugs, and guide bug correction. Both deductive and inductive inference techniques are employed for bug correction and synthesis. The behavior of the automated debugger is demonstrated through several examples.

1. Introduction

Logic programs have relatively simple syntax and semantics. In addition, logic programming offers an attractive feature rarely met in traditional programming languages, namely, the ability to use the same language—that of logic—for both specification and computation. We took advantage of this capability in the design of an implemented system for finding and correcting errors in logic programs. This debugging process comprises three steps: testing, bug location, and a mixture of heuristic and deductive bug correction. The specification is used to aid all three tasks: (1) when supplied as an executable model, it is used to generate test cases for discovery of bugs and determine that an error is present; (2) it helps isolate the offending program statements when test data cause the program to fail; and (3) in conjunction with a theorem prover, it guides deductive inference of corrections. Our system also attempts to verify termination of the program by checking for reduction with respect to a user-supplied ordering.

Program synthesis, that is, deriving a program from its specifications, can be thought of as debugging an empty program. We use an executable model, a predefined search space for programs, and a deductive mechanism. One needs to supply the model and ordering, as well as a skeleton of the desired recursive structure for the program; the system tries to do the rest.

We begin with a summary of the basic ideas of logic programming and with a presentation of the standard meta-interpreter for Prolog on which our debugger is based. Then, Section 3 discusses the use we make of specifications as a prototype of the correct

† This is a substantially revised version of a paper, "Deductive debugging", presented at the *Fourth IEEE Symposium on Logic Programming* (San Francisco, CA, U.S.A., Aug. 1987).

0747-7171/93/5-6745+29 $08.00/0
© 1993 Academic Press Limited

program. Section 4 gives a methodology for locating program errors and Section 5 introduces heuristics for correcting them. The integrated automated debugger, called the *Constructive Interpreter*, is presented in Section 6. The synthesis process is illustrated in Section 7. Finally, Section 8 relates this work to other research in automated debugging and synthesis.

This paper is based largely on the second author's Ph.D. dissertation (Lee, 1988).

2. Logic Programming

Broadly defined, a "logic programming language" is a language based on a formal logic, with deduction in that logic serving as operational semantics. Pure Lisp (McCarthy, 1960), for example, is a logic programming language based on the lambda-calculus. Languages based on equational logic, such as EQLOG (Goguen and Meseguer, 1986) and RITE (Josephson and Dershowitz, 1989), also fall in this category.

More specifically, "logic programming" usually refers to languages loosely based on the Horn-clause subset of first-order predicate calculus, with variations of the resolution principle used as efficient schemes for computation; see (Kowalski, 1974; Kowalski and van Emden, 1976). This procedural interpretation of Horn clauses accelerated progress in the development, implementation and acceptance of logic programming. Prolog (Clocksin and Mellish, 1987), the most popular such language, is nowadays a viable alternative to Lisp for symbolic processing and in artificial intelligence research.

2.1. PROLOG

A *definite Horn clause* is a logical formula written in (Edinburgh) Prolog in the form

$$A \text{ :- } B_1,\ B_2,\ \ldots\ ,\ B_k$$

meaning that, for all values assigned to the variables appearing in clause, the atomic formula A, called the *head*, is true whenever all the atomic formulæ B_i in the *body* are. When the body is empty ($k = 0$), the formula is called a *unit clause*, and asserts that A is true unconditionally. A *query* $B_1,\ B_2,\ \ldots\ ,\ B_k$ asks if there are values for the variables for which the truth of all the B_i is provable from the set of clauses constituting the *program*. The operational semantics is based on (what is called LUSH, or) SLD-resolution (Apt and van Emden, 1982; Lloyd, 1984). During the computation process, the heads of clauses are examined for one which unifies with the current goal. The atomic formulæ in the body, instantiated with the unifier of the head and goal, become new subgoals. The solution is the composite substitution for variables of the goal which allows all the subgoals to succeed.

Prolog's execution follows a sequential simulation of this nondeterministic computation, using a depth-first search strategy, and incorporating a backtracking mechanism. Prolog tries to unify the current goal with heads of clauses sequentially, in the order they occur in the program text. When unification succeeds with a unit clause, the solution to that subgoal is the most general unifying substitution. For other clauses, Prolog attempts to solve the subgoals in the body, from left to right. Whenever Prolog fails to find a clause whose head can be unified with the current goal, it backtracks to the most recently executed goal, undoes any substitutions made by the unification, and tries to resatisfy that goal using another clause. If no solution can be found to the initial goal,

```
interpret( (Goal1, Goal2) )  :-  interpret( Goal1 ), interpret( Goal2 )
interpret( Goal )            :-  system( Goal ), call( Goal )
interpret( Goal )            :-  clause( Goal, Subgoals ), interpret( Subgoals )
```

Figure 1. A Prolog meta-interpreter

the entire computation fails. Backtracking can also be used to obtain additional solutions to the given goal. (See Section 3.2 for an example.)

This computation process can also be described as the traversal of a computation tree. A *computation tree* of a program is a rooted, ordered tree. Each node in the tree has the form $p(x, y)$, where p is a procedure (predicate) name, and x and y represent input and output vectors over some domain. Though standard Prolog does not require the declaration of input and output variables (and one variable can serve both purposes), we make this distinction—which is usually not a limitation—to allow for generation of test cases (see Section 3.2).† For each clause

$$p(x, y) \text{ :- } p_1(x_1, y_1),\ p_2(x_2, y_2),\ \ldots\ ,\ p_k(x_k, y_k)$$

involved in a computation, the corresponding part of the tree includes the internal node $p(x, y)$ and its children $p_1(x_1, y_1)$, $p_2(x_2, y_2)$, ... , and $p_k(x_k, y_k)$. The meaning of this tree is as follows: procedure p, on input x, calls p_1 with x_1; if this call returns y_1, then p calls p_2 with x_2 (which may include outputs y_1 of the previous subgoal); and so on until p calls p_k with x_k; if this call returns y_k, then p returns y as its output. If a node $p(x, y)$ has no children, then procedure p has a computation that returns y on input x without performing any procedure calls.

Programs, for our purposes, are presumed to obey the operational semantics of "pure" Prolog, that is, there is a predefined sequential order for both clauses in a program (from top to bottom) and subgoals in a clause (from left to right). The "extra-logical" cut "predicate" is allowed to affect efficiency and termination, but not correctness.

2.2. Meta-Programming

As defined by Fuchi and Furukawa (1986), *meta-programming* can be characterized as: (1) handling programs as data; (2) handling data as programs and evaluating them; and (3) handling the result of a computation as data. Some programming languages make it particularly easy to build meta-programming systems that manipulate and execute other programs written within the *same* language.

This meta-programming capability is of great help when implementing a system to reason about programs. It provides a basis for building a powerful programming environment. Prolog is especially attractive in this respect, since one can easily write a meta-interpreter to execute pure Prolog programs in just a few lines, as shown in Figure 1. The first clause solves a conjunctive goal (*Goal1*, *Goal2*) by recursively solving its two conjuncts. The second clause checks if *Goal* involves a built-in predicate (procedure ***system*** is defined to recognize if its argument is a built-in predicate), and, if it is, executes the goal directly by passing it on to the Prolog system (via the built-in predicate

† In other words, we deal with "well-moded" logic programs; that is, those in which input and output variables are distinct.

call). The third clause uses a built-in predicate ***clause*** both to find a clause the head of which can be unified with *Goal* and to reduce *Goal* to the list of subgoals in the body of that clause. The interpreter then solves these subgoals recursively. As will be seen, our debugging system for pure Prolog programs is based on the scheme of this interpreter.

3. Executable Models

In software development, a specification may be regarded as an abstraction of a concrete problem, as the starting point for the development of a concrete solution, and as the criterion for judging the correctness of the final product. Specifications are procedural abstractions, mappings from sets of input values to sets of output values (Liskov and Berzins, 1986). A specification is a model which can be considered as a precise and independent description of the expected program behavior, a description of *what* is desired, rather than *how* it is to be achieved or implemented. When formulated in a language with operational semantics, a specification can also serve as a prototype program—a partial model of the functionality of the target system—the behavior of which may be scrutinized to determine if it is in fact what is desired.

Logic-based languages serve well for specification (Clark, 1981; Kowalski, 1985), since they have simple syntax, well-defined declarative semantics, and well-understood deductive mechanisms. Simplicity of syntax is highly desirable in that it makes a specification easier to write and understand. Having a well-defined declarative semantics facilitates the construction of high-level specifications, since a specification language is to describe intended behavior without prescribing a particular algorithm. The deductive mechanism provides the operational ability to validate that the specifier's intentions have been fulfilled.

3.1. Specifications in Prolog

First-order predicate calculus has long been used as a specification language. The typical approach to program verification (Floyd, 1967; Hoare, 1969; Katz and Manna, 1976) expresses specifications in first-order logic, and relates them to conventional programs by defining the semantics of programs in a logic of programs. Since Horn clauses are a sufficiently powerful subset of first-order logic, Prolog itself can often be used for specifications with the advantageous extra feature of executability: models written in Prolog can be directly executed by the Prolog interpreter or compiled and then executed. It may be argued that executable models are no different from programs (cf., (Kowalski, 1985)). For our purpose, execution efficiency is the main criterion for distinguishing programs from specifications, expressed in Horn-clause logic. ¿From this point of view, models are intended to emphasize clarity and simplicity, but not efficiency. In implementing software, on the other hand, efficiency becomes a primary consideration.

We will (somewhat optimistically; cf. (Gerhart and Yelowitz, 1976)) presume that specifications faithfully reflect the intended requirements of a program. For an executable model to produce the desired effect, it is sometimes necessary to use impure features of Prolog, that is, its non-logical control structures. More expressive languages, such as EQLOG (Goguen and Meseguer, 1986) and RITE (Josephson and Dershowitz, 1989), may be even more suitable for specifications.

Another requirement for program verification is a well-founded ordering of input arguments for recursive procedures, to be used to test for termination.† The ordering specifies in what sense the arguments should decrease for each recursive call. This is used for detecting looping.

3.2. Generation of Test Cases

The information contained in specifications regarding the expected output behavior is indispensable for checking the correctness of the results of a particular run of a program, while test cases can help reveal instances of incorrect output. Executable models can be used not only to check the output of the specified program for a given input, but also to generate useful test cases, with the proviso that axioms for built-in predicates that deal with equality, arithmetic, and the like are supplied. Test data for logic programs can be symbolic, that is, they can contain variables which may be instantiated in the process of checking the program.

To generate test cases for a given procedure $p(x, y)$, we first query its model with this goal to obtain a sample input s and expected output t. We then use the input value alone, that is, $p(s, y)$ to query the program to be debugged. If execution fails, goes into a loop, or returns an incorrect output value, then this test case has shown that there is at least one bug in the program. In other words, a test case consisting of a correct input/output pair can be used to discover bugs should they cause the program to fail to compute the correct answer. If one of the predicates in the model of a program is defined in the form of a "generator", then we can generate alternate test cases by utilizing Prolog's built-in backtracking facility.

Example 3.1. *Generating test cases from specifications.*

Figure 2 gives specifications for a sorting algorithm. They state that the list "*Out_List*" is a correct result of procedure *sort* if it is in (non-decreasing) order and is a permutation of "*In_List*". Given that *perm* is defined in a way that generates all possible permutations of a list (Figure 3), *ordered* accepts a list of numbers in strictly ascending order (see Figure 4),‡ and the "primitive" (not built-in) predicate *lt* defines the basic "*less than*" relation (Figure 5),§ then by executing *spec*(*sort*(*In_List*, *Out_List*)) with uninstantiated variables, *In_List* and *Out_List*, we can generate a sequence of input/output pairs. The first value generated for *In_List* is an empty list (that is, [ ]), then a one-element list (that is, $[X]$), then two-element lists with all possible permutations (that is, $[0, 1]$ and $[1, 0]$), then three-element lists with all possible permutations, etc. For each of these test inputs, the variable *Out_List* is bound to the specified result, and can be used to verify the correctness of the program.

† A *well-founded ordering* is a irreflexive transitive binary relation on domain elements that allows no infinite descending sequences.

‡ We are taking advantage of the fact that it suffices to check correctness of comparison-based sorting methods on lists of natural numbers without repetitions (Knuth, 1973).

§ We have to use *lt* and not Prolog's built-in operator "<" for it to also work on symbolic values. Also note that the keyword "*is*" in Figure 5 is Prolog's assignment operator for arithmetic operations.

$spec(sort(In\_List, Out\_List))$	:-	$ordered(Out\_List)$, $perm(In\_List, Out\_List)$

Figure 2. Specifications for sorting

$perm([\,],[\,])$		
$perm([X\|Xs], Ys)$	:-	$del(X, Ys, Zs)$, $perm(Xs, Zs)$
$del(X,[X\|Xs],Xs)$		
$del(X,[Y\|Xs],[Y\|Ys])$	:-	$del(X,Xs,Ys)$

Figure 3. Executable model of *perm*

$ordered([\,])$		
$ordered([X])$		
$ordered([X1,X2\|Xs])$	:-	$lt(X1,X2)$, $ordered([X2\|Xs])$

Figure 4. Executable model of *ordered*

$lt(X,Y)$	:-	$is\_number(X)$, $is\_number(Y)$, $X < Y$
$is\_number(0)$		
$is\_number(X)$	:-	$is\_number(Y)$, $X\ is\ Y+1$

Figure 5. Primitive predicate *lt*

3.3. Validation of Computation Results

We assume that the properties of each procedure in the program have been described in the program's specifications, which detail the relationships between program variables. In other words, they define all legal input/output pairs for each procedure. To check for termination, we also need a well-founded ordering under which successive input values to recursive procedures are intended to form a descending sequence. Any unspecified procedures are presumed correct and terminating.

Suppose S specifies program P and A contains domain facts, including any inductive (in the mathematical sense) theorems needed to verify the program. We say that P is *partially correct* with respect to S if $A, S \vdash P$, that is, if each clause of the program can be proved from the domain facts and specification by first-order reasoning. We *test* for partial correctness by checking that $A, S \vdash R$ for various atomic formulæ R that follow from the program, that is, for which $A, P \vdash R$. If there is a computation result R that can not be so deduced, then P is *incorrect* with respect to S.

On the other hand, if $A, P \vdash S$, then P is *complete* with respect to S. This is tested, analogously, by showing that program P (and facts A) derives test facts R that follow

from the specification S. If there is an R specified by S that can not result from executing the program, then that fact is "uncovered" and P is *incomplete.*

If during a computation, P generates an infinite sequence of procedure calls, then P is *nonterminating.* Otherwise, it *terminates.*

We test *partial correctness* and *completeness* by checking a program's computation results against its specifications. *Termination* is tested for by routines that compare the inputs with respect to a specified well-founded ordering whenever a procedure is invoked.

4. Automated Bug Location

When a Prolog program does not compute correct results, it may be that the program contains incorrect clauses, is incomplete in defining certain relationships between program variables, or gets into an infinite procedure invocation sequence. In this section, we discuss how each of these three types of errors can be detected and located automatically, based on the meta-programming capability of Prolog and our use of an executable model. The method for detecting errors and incompleteness is the same as in (Shapiro, 1983), except for our use of a model, instead of repeatedly querying the user.

4.1. Locating Incorrect Clauses

Consider the computation $p(x', y')$ of procedure p with input x' and output y'. Suppose y' is incorrect with respect to the specifications of p. We trace the computation and check the result of each procedure call (by executing its model) as soon as it is completed. Suppose

$$p(u, v) \quad :- \quad r_1, ..., r_n$$

is the (temporally) first clause to return an incorrect output, say v' on input u', and that

$$p(u', v') \quad :- \quad r'_1, ..., r'_n$$

is the offending instance. This instance of procedure p is incorrect, since all the procedure calls $r'_1, ..., r'_n$ must have completed earlier and returned correct results. Thus, the clause contradicts the specifications (with u', v' as witness).

The method can be summarized as in Figure 6. Conjunctive goals are broken down. To compute a noncomposite goal, we first find a clause whose head can be unified with *Goal* and recursively solve the subgoals in the clause. If all subgoals return correct results, the system checks if *Goal* is satisfied, by running the model on the instantiated *Goal*. If the result is consistent with the specifications of *Goal*, then the clause is okay, according to the first clause of *diagnose*. If the computed *Goal* is inconsistent with its specification, the next clause returns an instance of the incorrect clause. On the other hand, if the debugger identifies an error in the subgoals, it returns an error message to the top level, using the last clause.

Example 4.1. *Locating an incorrect clause.*

Consider the (buggy) insertion sort program in Figure 7, adopted from (Shapiro, 1983), with specifications for each of its procedures shown in Figure 8. The specification for *isort* is the same as that for *sort* in Section 3.2. For *insert*, the specification means that

```
execute( (Goal1, Goal2), Message ) :-
    execute( Goal1, Message1 ),
    if Message1 = ok( Goal1 )
        then execute( Goal2, Message )
        else Message = Message1
execute( Goal, Message ) :-
    clause( Goal, Subgoal ),
    execute( Subgoal, SubMessage ),
    diagnose( Goal, Subgoal, SubMessage, Message )
diagnose( Goal, Subgoal, ok( Subgoal ), Message ) :-
    if spec( Goal )
        then Message = ok( Goal )
        else Message = incorrect( clause(Goal, Subgoal) )
diagnose( Goal, Subgoal, SubMessage, SubMessage )
```

Figure 6. Algorithm for locating incorrect clauses

```
isort([X|Xs], Ys)      :-  isort(Xs, Zs), insert(X, Zs, Ys)
isort([ ], [ ])
insert(X, [Y|Ys], [Y|Zs])  :-  Y > X, insert(X, Ys, Zs)
insert(X, [Y|Ys], [X, Y|Ys])  :-  X <= Y
insert(X, [ ], [X])
```

Figure 7. An incorrect insertion sort

```
spec(isort(X, Y))      :-  ordered(Y), perm(X, Y)
spec(insert(X, Y, Z))  :-  if ordered(Y)
                             then  ordered(Z), perm([X|Y], Z)
                             else  true
```

Figure 8. Specifications for insertion sort

$insert(X, Y, Z)$ is correct if Z is in order and is a permutation of the list consisting of the element X and list Y, provided that Y is in order in the first place.

Running *isort* on input [2,1,3], we find that the instance

$$insert(1, [3], [3, 1]) \quad :- \quad 3 > 1, \; insert(1, [\,], [1])$$

of the first clause of *insert* is incorrect. (The user need not supply the input list [2,1,3], since it will be generated by running the model of *isort*, as shown in Section 3.2.) The error is due to the arithmetic test. Since the positions of the two arguments are exchanged, it forces a smaller element to be inserted after a larger element. The result is an unsorted list which fails the specification check. The error message will be passed to the bug fixing routine discussed in Section 5.

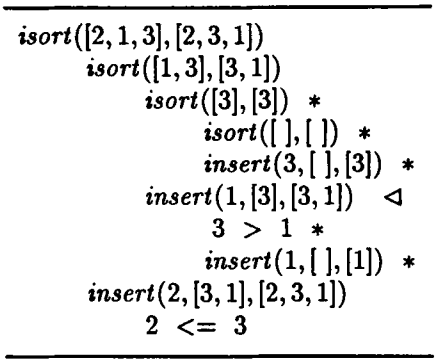

Figure 9. Computation tree for *isort*([2, 1, 3], [2, 3, 1])

The computation tree in Figure 9 shows how the diagnostic process works on *isort* with input [2, 1, 3]. It traverses the computation tree in post-order and checks each procedure for its correctness. Each of the nodes marked with an asterisk has been verified by the interpreter as correct with respect to its specifications, while the node pointed by "◁" is the first node to contain results inconsistent with its specifications. Therefore, the interpreter returns this node along with its two children as a counterexample.

4.2. Locating Incomplete Procedures

Suppose x' is a legal input, and the query $p(x', y)$ is satisfiable according to the specifications, but the program fails on procedure call $p(x', y)$. Then the program must contain at least one incomplete procedure. This incompleteness corresponds to a computation tree which is finite but contains a node which represents an unsuccessful branch. There are two possibilities: if p with input x' invokes no other procedures, then p is incomplete; if, on the other hand, p calls other procedures, then p or one of the procedures invoked after p must be incomplete. Accordingly, we trace the execution of p. If a satisfiable call to a procedure q fails, while all procedures called by q return an answer whenever the call is satisfiable, then it is q that is deemed incomplete. The above method is summarized in Figure 10. The interpreter for locating an incomplete procedure first tries to build a computation tree by executing the goal and recursively executing subgoals. The procedure *satisfiable* computes whether the goal supplied as its argument has an answer. When a satisfiable call *Goal* fails to find a clause that can complete the computation, one can be sure that *Goal* is not covered by the program.

Example 4.2. *Locating an incomplete procedure.*

Suppose we are given the incomplete program of Figure 11. With the same specifications as in Figure 8, we try *isort* on [3,2,1], and detect that the goal

$$insert(1, [\,], [1])$$

is not covered. We now have an instance of the uncovered goal and the debugger detects that the incomplete procedure is *insert*, which does not have a clause to cover the base case (when inserting an element to an empty list).

```
execute( Goal, Message ) :-
    clause( Goal, Subgoals ),
    satisfiable( Subgoals ),
    execute( Subgoals, Message )
execute( Goal, uncovered( Goal ) ) :-
    satisfiable( Goal )
satisfiable( (Goal1, Goal2) ) :-
    satisfiable( Goal1 ),
    satisfiable( Goal2 )
satisfiable( Goal ) :-
    spec( Goal )
```

Figure 10. Algorithm for locating incomplete procedures

$$
\begin{array}{rcl}
isort([X|Xs], Ys) & :- & isort(Xs, Zs),\ insert(X, Zs, Ys) \\
isort([\,], [\,]) & & \\
insert(X, [Y|Ys], [Y|Zs]) & :- & X > Y,\ insert(X, Ys, Zs) \\
insert(X, [Y|Ys], [X, Y|Ys]) & :- & X <= Y
\end{array}
$$

Figure 11. An incomplete insertion sort

$$
\begin{array}{l}
isort([3,2,1], Answer) \\
\quad isort([2,1], X_1) \\
\quad\quad isort([1], X_2) \\
\quad\quad\quad isort([\,], [\,]) \\
\quad\quad\quad insert(1, [\,], X_2)\ \lhd
\end{array}
$$

Figure 12. Computation tree for $isort([3,2,1], Answer)$

The incomplete computation tree of *isort* on [3,2,1] is in Figure 12. The first goal in the computation tree that can not be unified with any clause in the program is $insert(1, [\,], X_2)$. The computation stops at this point because of the failure of this node.

4.3. Locating a Diverging Procedure

If a program is partially correct, but nonterminating, then during computation, some procedure p must be invoked infinitely often. (However, there may be calls to other procedures in between calls to p.) In that case, the sequence of input values to p can not decrease in any well-founded ordering.

In the computation tree, a diverging computation is reflected by an infinite branch. We check for nontermination by tracing the procedures and checking that each call (except the first one) is smaller than the previous one with respect to the given well-founded ordering. The algorithm for this task is straightforward and is shown in Figure 13. Note

```
execute( Goal, looping(Goal) ) :-
    not decreasing( Goal )
execute( Goal, Message ) :-
    clause( Goal, Subgoals ),
    execute( Subgoals, Message )
decreasing( Goal ) :-
    retract_goal( PreviousGoal ),
    wfo( Goal, PreviousGoal ),
    assert_goal( Goal )
```

Figure 13. Algorithm for locating diverging procedures

$$
\begin{array}{rcl}
isort([X|Xs], Ys) & :- & isort(Xs, Zs),\ insert(X, Zs, Ys) \\
isort([\,], [\,]) & & \\
insert(X, [Y|Ys], [Y|Zs]) & :- & insert(X, Ys, \mathrm{Ws}),\ insert(Y, Ws, Zs) \\
insert(X, [Y|Ys], [X, Y|Ys]) & :- & X <= Y \\
insert(X, [\,], [X]) & &
\end{array}
$$

Figure 14. A looping insertion sort

that in order to incorporate this function, we need to modify procedure *execute* so that it "remembers" the previous call to every procedure invocation. (In our Prolog implementation this is achieved by building an explicit run-time stack in the meta-interpreter via the use of "assertions".)

EXAMPLE 4.3. *Locating a diverging procedure.*

The program in Figure 14 contains a loop. Its designated well-founded ordering is given in Figure 15. The predicate *wfo* specifies the ordering by looking at pairs of input values. For both *isort* and *insert*, the number of elements in the input list should decrease with each recursive call. Note that in our implementation every procedure call is recorded using Prolog's *assert* and is available for global checking; therefore, the well-founded ordering can be verified even for mutually recursive predicates.

Running *isort* on [2,1,3], we find that the goal $insert(3, [1], X)$ in the clause

$$insert(1, [3], X) \ :- \ insert(1, [\,], [1]), insert(3, [1], X)$$

is looping. As can be seen from the infinite computation tree (Figure 16) for this goal,

$$
\begin{array}{rcl}
wfo(isort(X, Y), isort(U, V)) & :- & shorter(X, U) \\
wfo(insert(X, Y, Z), insert(U, V, W)) & :- & shorter(Y, V) \\
shorter([X, Xs], [\,]) & & \\
shorter([X, Xs], [Y, Ys]) & :- & shorter(Xs, Ys)
\end{array}
$$

Figure 15. Well-founded ordering for recursive procedures

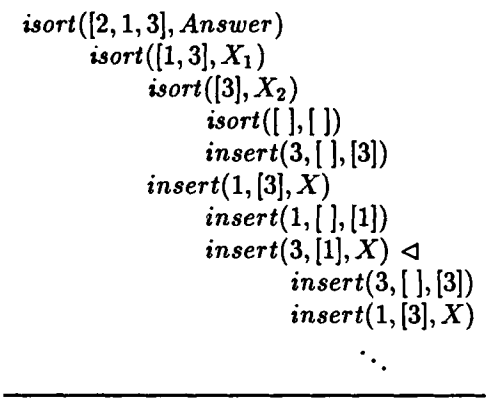

Figure 16. An infinite computation tree

the second argument of the goal $insert(3,[1],X)$ has the same length as the second argument of the head $insert(1,[3],X)$ of the invoked clause. This clearly violates the relationship defined in $wfo(insert)$ which requires that the length of lists get shorter with each recursive call.

4.4. The Automatic Bug Locator

Based on the above analysis of possible errors, we can construct a meta-interpreter which executes programs, diagnoses errors according to the specifications of programs, and locates and reports bugs once they are identified. This meta-interpreter is summarized in Figure 17. Procedure *execute*(*Goal*, *Message*) serves two functions: goal reduction and bug location. The first clause deals with conjunctive goals. If the first conjunct executes correctly, the remaining conjuncts will be tried in order; otherwise, it just returns the error found to the top level. The second clause executes built-in primitives directly. The next three clauses detect bugs of nontermination, incorrect clauses, and uncovered goals, respectively, using subprocedures *diagnose* (Figure 6), *satisfiable* (Figure 10), and *decreasing* (Figure 13). It first checks if the input variables violate the well-founded ordering given for the procedure that covers the goal. If such is the case, we have an instance of a looping goal. If the input can not cause an infinite sequence of procedure calls, the interpreter will proceed to check if the program can actually complete the computation on the given input. It first finds a clause whose head can be unified with *Goal* and then recursively executes (and debugs) the subgoals in the body of that clause. If a bug is found in the body of a clause, it will be returned to the top level for correction. If all the subgoals complete successfully, then all the output variables in *Goal* will be instantiated. The interpreter then checks if the output value is correct with respect to the specifications of *Goal*. If not, then we have found an incorrect clause. On the other hand, if there is no clause in the program that covers the goal for the input data (that is, a subgoal fails for every clause with unifying head), then, since *Goal* is satisfiable according to the specifications, the program must be incomplete and we have discovered an instance of an uncovered goal.

```
execute( (Goal1, Goal2), Message ) :-
    execute( Goal1, Message1 ),
    if Message1 = ok( Goal1 )
        then execute( Goal2, Message )
        else Message = Message1
execute( Goal, ok(Goal) ) :-
    system( Goal ),
    call( Goal )
execute( Goal, looping(Goal) ) :-
    not decreasing( Goal )
execute( Goal, Message ) :-
    not system( Goal ),
    clause( Goal, Subgoals ),
    satisfiable( Subgoals )
    execute( Subgoals, SubMessage ),
    diagnose( Goal, Subgoal, SubMessage, Message )
execute( Goal, uncovered(Goal) ) :-
    satisfiable( Goal )
```

Figure 17. Algorithm for bug location

5. Bug Correction

Although Myers (1979) has claimed that bug correction is a much easier task for a human than bug location, correcting a bug after it is identified appears to be the more difficult when performed by a machine. This is because bug location only requires tracing the execution of procedures and checking the results of computation against a (user-supplied) specification. Bug correction, on the other hand, requires reasoning with knowledge both of the domain and of the intended algorithm, as well as the semantics of the programming language.

5.1. Fixing an Incorrect Clause

A clause

$$p(x,y) \quad :- \quad r_1,...,r_n$$

is incorrect if it has an instance, say,

$$p(x',y') \quad :- \quad r'_1,...,r'_n$$

such that all the r'_i's are true (that is, their specifications hold), but $p(x',y')$ is false. (Here x' denotes the test input value(s) to p and y' is the output after the call $p(x',y)$ returns.) To fix this incorrect clause, we first rerun the specification of p to get a correct output, say y'', for the given input x'. How the program behaves with the *solved* goal $p(x',y'')$ will help guide the debugger.

If $p(x',y'')$ is covered by another clause in the program (that is, there exists at least one clause in the procedure that computes this goal correctly), then the incorrect clause should not have completed and returned a wrong result. Instead, the clause should pre-

sumably have failed for this input. We can, therefore, attempt to include extra conditions that prevent computation for the improper input x'. To add subgoals to the clause, we try to construct a proof that the subgoals (on the right) imply the head (for all instances of the variables). If the proof fails because of some missing conditions, we can add them as subgoals to the clause (see Section 5.2). Alternatively, we can use the offending clause as a starting point for an inductive synthesis of a correct clause. In the worst case, we can always add the subgoal ***fail*** (***false***) to the clause. Although this might be too strong a fix and might result in some other goals becoming uncovered, adding ***fail*** as a subgoal does make the clause (vacuously) partially correct. Later, we will see how to deal with uncovered goals.

If the *solved* goal $p(x', y'')$ is only covered by the incorrect clause, then we proceed to add conditions that preclude computation of the wrong answer y', with input x', as above. A sufficient condition (namely, that $x = x'$ implies $y = y''$) can be deduced from the variable bindings obtained when unifying $p(x', y'')$ with the clause head $p(x, y)$ and may be added to the clause as subgoals. Or, an inductive approach may be taken, guessing patches that make the program work for more and more examples.

If the *solved* goal $p(x', y'')$ is not covered by any clause, then the fix proceeds in different directions, depending on whether $p(x', y'')$ can be unified with the head of the incorrect clause. If the head does unify, but some of the subgoals fail for y'', then we presume that the incorrect clause should cover the goal $p(x', y)$ and compute y'' instead of y'. In this case, we can combine fixes for the uncovered goal, $p(x', y'')$, and the incorrect clause that computes the erroneous solution $p(x', y')$. We check, for $p(x', y'')$ (that is, under the current input and *correct* output), which of the subgoals in the clause fail with the output constrained to be y''. After identifying any such incorrect subgoals, we try to fix them by either applying a heuristic rule or an inductive method (see Section 5.3). We rearrange, replace, delete, or add new variables within subgoals until the original incorrect clause computes $p(x', y'')$—and subsequent tests—correctly.

The last possibility is that $p(x', y'')$ can not be unified with the head of the incorrect clause, nor is it covered by other clauses in the program. In this case, we assume that the incorrect clause we have identified should cover this goal. Accordingly, the only way to correct the bug is to first fix (that is, weaken) the clause head so that it is unifiable with $p(x', y'')$. The methods described above can then be used to fix any incorrect subgoals.

We summarize the strategies for correcting a clause in the following heuristic rules:

(a) If the solved goal is covered by a clause in the program, then deduce missing subgoals and add them to the incorrect clause to preclude the wrong answer.

(b) If the solved goal can be unified with the head of the incorrect clause and is not covered by any clause in the program, then fix the subgoals that fail for the correct answer and continue debugging the clause.

(c) If the solved goal can not be unified with the head of the incorrect clause and is not covered by any clause in the program, then fix the clause head and continue debugging the clause.

EXAMPLE 5.1. *Fixing an incorrect clause.*

We demonstrate this process on the insertion sort program in Figure 18. The debugger detects an incorrect clause in procedure *insert* when trying to solve the goal

```
isort([X|Xs], Ys)      :-  isort(Xs, Zs), insert(X, Zs, Ys)
isort([ ], [ ])
insert(X, [Y|Ys], [Y|Zs])  :-  insert(X, Ys, Zs)
insert(X, [Y|Ys], [X, Y|Ys])  :-  X <= Y
insert(X, [ ], [X])
```

Figure 18. An incorrect insertion sort

isort([0, 1], *Answer*). After some analysis, it determined that the clause

$$insert(X, [Y|Zs], [Y|Vs]) \quad :- \quad insert(X, Z, Vs)$$

is false for $X = 0$, $Y = 1$, $Z = [\,]$, $Vs = [0]$ (the debugger occasionally renames variables). Furthermore, it need not be covering the subgoal $insert(0, [1], Z)$, since the solved subgoal $insert(0, [1], [0, 1])$ is in fact covered by another clause,

$$insert(X, [Y|Zs], [X, Y|Zs]) \quad :- \quad X <= Y$$

in the program. The debugger then tries to deduce a missing subgoal by constructing a proof. It attempts to prove that $insert(X, Z, V)$ implies $insert(X, [Y|Zs], [Y|Vs])$, and concludes that, by adding the subgoal "$Y < X$" to the body of the clause, the implication will hold. Therefore, it removes the incorrect clause and asserts the newly synthesized clause as part of the program. This proof process requires the theorem prover described next.

5.2. Deducing Missing Subgoals

According to the declarative semantics of Prolog, the body (subgoals) of a clause should imply the truth of the head. On the other hand, trying to prove the implication for an incorrect clause must result in failure. The basic idea for correcting a clause in this event is to identify sufficient conditions that would have allowed the proof to go through. This approach is inspired by the work of (Smith, 1982) in which a deductive theorem prover was used to derive a sufficient precondition such that a goal can be shown to logically follow from the conjunction of the precondition and a hypothesis. In other words, the condition provides additional hypotheses with which a goal can be proved. We modified this method and constructed an, albeit incomplete, theorem prover for Horn clauses, with no guarantee of finding a satisfactory correction. The prover is presented with a goal of deriving the truth of a program clause from the given specifications (and domain information). Definitions of procedures, as given in the specifications, are assumed to satisfy the "closed world assumption". (We need definitions for the correct procedures, as well as the incorrect ones, to be able to deduce corrections.)

The deductive proof proceeds by reducing both sides of the clause to simpler forms, by replacing a subgoal with its definitions or with something that implies it, and each hypothesis with its definition or something that it implies, until the implication is found to hold, or sufficient *primitive* conditions (that is, predicates usable in the program) for it to hold are isolated.

The prover employs the following rules for simplifying implications, which are typical of natural deduction proofs. In the rules we use G (possibly with a subscript) to represent

a goal, H (possibly with a subscript) for a hypothesis, $\wedge$, $\vee$, and $\neg$ for logical "and", "or", and "not", "$H \rightarrow G$" for "if H then G", and "$A \Leftarrow B$" for "to prove A, it is sufficient to prove B".

Rule 1. $H \rightarrow (G_1 \wedge G_2) \Leftarrow (H \rightarrow G_1) \wedge (H \rightarrow G_2)$

Rule 2. $H \rightarrow (G_1 \vee G_2) \Leftarrow (H \rightarrow G_1) \vee (H \rightarrow G_2)$

Rule 3. $(H_1 \vee H_2) \rightarrow G \Leftarrow (H_1 \rightarrow G) \wedge (H_2 \rightarrow G)$

Rule 4. $H \rightarrow (G_1 \rightarrow G_2) \Leftarrow (H \wedge G_1) \rightarrow G_2$

Rule 5. $(H_1 \rightarrow H_2) \rightarrow G \Leftarrow (\neg H_1 \rightarrow G) \wedge (H_2 \rightarrow G)$

Rule 6. $\neg H \rightarrow \neg G \Leftarrow G \rightarrow H$

Rule 7. $\neg H_1 \wedge H_2 \rightarrow \neg G \Leftarrow G \wedge H_2 \rightarrow H_1$

We replace a goal with its definition, as given in the goal's specification:

Rule 8. $H \rightarrow G \Leftarrow H \rightarrow G'$, *if* $G = G'$.

A logical simplifier (cf. (Waldinger and Levitt, 1974)) is invoked after each reduction step and performs tasks such as removing nested conjunctions, duplicate goals, and tautologies, by applying propositional rules governing $\wedge$, $\vee$, $\neg$, and $\rightarrow$. Also, domain facts can be used to replace a goal with something equivalent.

We use transitivity of implication:

Rule 9. $H \rightarrow G \Leftarrow H \rightarrow G'$, *if* $G' \rightarrow G$.

In particular, if the head of a correct program clause matches the goal, we can replace the goal with the subgoals obtained from that clause. Also, when a specific domain fact is known, it can be used to strengthen a goal.

Similarly, domain facts may be used to weaken hypotheses:

Rule 10. $H \rightarrow G \Leftarrow H' \rightarrow G$, *if* $H \rightarrow H'$.

An effort was made in our implementation to build in some domain knowledge about lists and inequalities that can be employed by the above rules.

The proof process terminates when one of the following conditions is met:

(a) the original goal is reduced to ***true***, in which case the clause is proved correct;
(b) the original set of hypotheses (that is, the subgoals in the body of the clause) is reduced to ***false***, meaning that there are conflicting subgoals in the clause, and that the clause is vacuously correct;
(c) the goal is reduced to a subset of the hypotheses, in which case the implication is also established; and

(d) the original goal is reduced to primitives and hypotheses, in which case those goals not appearing as hypotheses are added as subgoals to the original clause. In this case, we have identified those missing subgoals which will make the clause correct.

5.3. Fixing Incorrect Subgoals

Once we identify an incorrect subgoal, we can try to correct it using heuristic rules, in addition to the deductive methods outlined in the previous sections. The method for correcting subgoals is a modification of the refinement method in (Shapiro, 1983).

We have incorporated heuristics meant to correct an incorrect subgoal expeditiously when certain patterns are encountered. For example, one of the rules is to swap the variables if there are only two variables in the subgoal. Other rules include moving a simple variable to a different position, replacing simple variables with more complicated terms, deleting seemingly redundant variables, and adding free variables that have appeared elsewhere in the same clause. The purpose of our heuristic rules is to attempt to fix some common, easily corrected errors.

If these heuristics can not correct the errors in a subgoal, a general strategy is employed. Refinement operators are applied to terms in the subgoals until the test case succeeds. For example, we can try to unify two free variables, or unify a compound term with variables appearing elsewhere in the same clause. Of course, all heuristic fixes will be tested immediately after the changes are made; and if the fixes can not correct the errors, the suggested changes will be undone.

5.4. Fixing an Incomplete Program

To remedy the problem of an uncovered goal, we first check if the goal can be unified with the head of a clause. If indeed such a clause exists, then we presume that it should cover this goal. Since the original clause might be useful for other goals, instead of modifying the clause directly, we make local changes to a copy. We locate the subgoal that causes this clause to fail and either try to fix it inductively (by repeatedly rearranging, replacing, deleting, or adding variables within the subgoal, until enough test cases work), or eliminate the offending subgoal entirely and use deductive means to correct it, if necessary.

When there is no clause whose head unifies with the uncovered goal, we use the specifications to synthesize a new clause. This can be done by using the uninstantiated goal as the clause head and the specifications as the clause body, simplifying the resulting clause as much as possible, or by using the specifications to guide a search of a tree of modifications to the erroneous clause. We can also fix a clause head so that it can be unified with the uncovered goal, and then debug the subgoals in the clause.

The above strategies for dealing with uncovered goals can be summarized as follows:

(a) If the uncovered goal can be unified with the head of a clause, then duplicate the clause, and locate and fix its unsatisfiable subgoals.

(b) If the uncovered goal can not be unified with the head of a clause, then use the specifications for that goal to synthesize a new clause.

```
interpret( Goal ) :-
    spec( Goal ),
    freeze_input_variables( Goal, Goal' ),
    execute( Goal', Message ),
    if Message ≠ ok(Goal')
        then fix_bug( Message )
```

Figure 19. The Constructive Interpreter

5.5. Fixing a Looping Procedure

When the input to a procedure call violates the well-founded ordering given for that procedure, a likely cause is that the input argument of the call is too general. For example, it may contain an irrelevant variable that does not appear in either the clause head or other subgoals of the same clause. Other possibilities are that some variables are missing or that the order of arguments is wrong. In any of these cases, what we have is a clause that contains a looping call caused by incorrect arguments. We try to fix the offending subgoal, using the same heuristic method as for fixing incorrect subgoals. Alternatively, we can weaken the subgoal and employ deductive techniques to ensure that the well-founded condition is met.

It is also possible that a subgoal that would preclude the looping case is missing (and that the goal is covered by another clause). This can be treated in the same way as an incorrect clause, by adding subgoals.

6. The Constructive Interpreter

Based on the analyses in previous sections, we can integrate the functions of test case generation, bug discovery, bug location, and bug correction into an automated debugging environment. The realization of this framework is the *Constructive Interpreter*. The top-level structure of this interpreter is described in Figure 19. Upon receiving a goal, the interpreter first examines the input variables. If the input is symbolic (partially uninstantiated), then by executing the model of the procedure, the interpreter will generate test cases. If the input variables are instantiated, then running the model on the given input checks if the input values are satisfiable. Once the legality of the input is established or a legal test input generated, the interpreter proceeds to execute the program on the input. Note that the interpreter will "freeze" the variables at this point, treating them as constants so that they will not be changed by the Prolog system. If execution completes successfully, the interpreter returns correct output values. In the case of symbolic input, the user can continue to generate alternate test cases and execute the program on different inputs. If the execution ever fails, that is, if the program contains an incorrect, incomplete, or nonterminating procedure, then the interpreter returns a diagnostic message with the location. Bug-fixing routines will then be invoked to correct the bug that has been identified and located.

Procedure *execute* does goal reduction and bug location, and has been discussed in Section 4.4. Procedure *fix_bug*(*Message*) implements the bug correction heuristics discussed in Section 5.

```
qsort([X|L], L0)              :-  part(L, X, L1, L2), qsort(L1, L3),
                                  qsort(L2, L4), append([X|L3], L4, L0)
part([X|L], Y, L1, [X|L2])    :-  part(L, Y, L1, L2)
part([X|L], Y, [X|L1], L2)    :-  X <= Y, part(L, Y, L1, L2)
part([ ], X, [X], [ ])
append([X|L1], L2, [X|L3])    :-  append(L1, L2, L3)
append([ ], L, L)
```

Figure 20. A buggy Quicksort program

```
spec(qsort(X, Y))             :-  ordered(Y), perm(X, Y)
spec(part(L, E, X, Y))        :-  rm_list([E|X], L, Y),
                                  gt_all(E, X), lt_all(E, Y)
spec(append(X, Y, Z))         :-  length(X, N), front(N, Z, X),
                                  rm_list(X, Z, Y)
wfo(qsort(X, Y), qsort(U, V)) :-  shorter(X, U)
wfo(part(X, A, B, C), part(Y, D, E, F)) :-  shorter(X, Y)
wfo(append(X, A, B), append(Y, C, D))   :-  shorter(X, Y)
```

Figure 21. Specifications for Quicksort

This interpreter is constructive in the sense that it assumes an active role during the debugging process and actually tries to complete the construction of the program being debugged, all with very little user involvement. It is based on the meta interpreter introduced in Figure 1 and consists of the three major components: test case generator, bug locator, and bug corrector. The test case generator executes a model to either generate test input or verify the satisfiability of user-supplied input. The bug locator also carries out the computation. It has a run-time stack that records all the procedure invocations. This information and the specified well-founded ordering are used to check against looping. The execution is simulated to perform depth-first search and backtracking upon failure. A message stack is maintained during execution, and an error message is recorded whenever an error occurs. The bug corrector contains three main procedures, dealing with the three different kinds of errors. In addition to performing error analysis and suggesting fixes, they all have access to the deductive theorem prover and heuristic subgoal refiner.

In the remainder of this section, we illustrate the integrated functions, including test case generation, bug location, and correction, of the *Constructive Interpreter*. Our experimental implementation is able to generate test cases that reveal errors and locate bugs for all the sorting examples in (Shapiro, 1983).

EXAMPLE 6.1. *Debugging a Quicksort program.*

We reproduce a session with the *Constructive Interpreter*, debugging the Quicksort program in Figure 20, with the specifications in Figure 21. The specifications say that

qsort(X, Y) holds if Y is an ordered permutation of X, that *part*(L, E, X, Y) holds if Y is the list obtained by removing elements of X from L (in other words, L is a permutation of X and Y combined) and E is greater than or equal to all the elements in X and smaller then or equal to all the elements in Y, and that *append*(X, Y, Z) is true if Z is the concatenation of lists X and Y. The predicate *wfo* specifies the well-founded ordering for sequences of input values. For procedures *qsort*, *part*, and *append*, the number of elements in the input list should decrease with each recursive call. Predicates *rm_list*, *gt_all*, and *lt_all* can be defined as ordinary Prolog procedures, as were *perm*, *ordered*, and *shorter*. The third argument of *rm_list* is a list like the second argument, but without the elements of the first. (These procedures can be regarded as standard building blocks for specification, available in the debugger's library, since they all apply across a wide gamut of specific programs. For example, *lt_all* would play a role in many sorting and searching programs and *rm_list* in various list manipulation programs.)

Given a query *qsort*(U, V) the *Constructive Interpreter* proceeds as follows:

(1) Since qsort(U,V) is symbolic, the debugger first generates a test case *qsort*$([\,], X)$ and tries to satisfy it. It discovers that *qsort*$([\,], X)$ should have a solution *qsort*$([\,],[\,])$ according to the specification of *qsort*, but can not get it from the given program. The debugger, therefore, reports a bug and then tries to fix it.

(2) Since no clause head in the original program unifies with *qsort*$([\,],[\,])$, the debugger uses the specification for *qsort* and synthesizes the clause

$$qsort([\,],[\,]) \;:-\; ordered([\,]), perm([\,],[\,])$$

to cover that goal. Since the body of this clause can be reduced to ***true***, the debugger adds a unit clause to the program (by asserting it to the database):

$$qsort([\,],[\,]) \;:-\; \textbf{\textit{true}}$$

The goal *qsort*$([\,],[\,])$ is now satisfiable.

(3) Since we initially supplied a symbolic input, we try another test case (the user answers "yes" to the system's query whether to continue). The debugger now generates a one element list as test input: *qsort*$([x], X)$. (Note that the generated input, $[x]$, contains a frozen variable x.) This time, it finds an incorrect clause in procedure *part*, because partitioning an empty list should result in two empty sublists, so the result of *part*$([\,], x, X, Y)$ should be *part*$([\,], x, [\,], [\,])$ instead of *part*$([\,], x, [x], [\,])$.

(4) After further analysis, the debugger concludes that the head of the clause

$$part([\,], X, [X], [\,]) \;:-\; \textbf{\textit{true}}$$

is incorrect. Since the debugger can not fix the head, it retracts the clause.

(5) After synthesizing the unit clause *part*$([\,], x, [\,], [\,])$, the debugger re-executes all the test goals generated so far to make sure the changes do not destroy anything. (Note that there is no way a correctly synthesized clause can cause a problem; retracting an incorrect clause, however, could conceivably cause some goals to become uncovered.) Since every goal generated so far can be satisfied, the debugger prompts the user whether to continue.

(6) The next test case generated is *qsort*$([0, 1], X)$. Unlike the previous two test cases, the goal *qsort*$([0, 1], X)$ is solved directly by the clauses currently in the program.

(7) The next test goal, *qsort*$([1, 0], X)$, results in the location of an incorrect clause in procedure *part*. A trace of the procedures shows that the correct solution to

```
msort([ ],[ ])
msort(X,Z)                  :-  length(X,L), L1 is L // 2,
                                break(X,L1,X1,X2),
                                msort(X1,Z1), msort(X2,Z2),
                                merge(Z1,Z2,Z)
break(X,0,[ ],X)
break([A|Xs],L,[A|Ys],Z)    :-  L1 is L - 1, break(X,L1,Y,Z)
merge([ ],X,X)
merge(X,[ ],X)
merge([A|Xs],[B|Ys],[A|Zs]) :-  A <= B, merge(X,[B|Ys],Z)
merge([A|Xs],[B|Ys],[B|Zs]) :-  A > B, merge([A|Xs],Y,Z)
```

Figure 22. A buggy merge-sort program

part([0], 1, *X*, *Y*), namely *part*([0], 1, [0], [ ]), can be obtained from the other clause of *part*. Thus, this incorrect clause should have failed, but did not because of a missing subgoal. The debugger is able to deduce this missing subgoal:

$$part([X|Y], Z, U, [X|W]) \text{ :- } Z <= X, part(Y, Z, U, W)$$

(8) After correcting for the missing subgoal (by retracting an incorrect clause and asserting a correct one), the debugger re-executes all the test goals again. This time the debugger catches another bug. Further diagnosis narrows down the bug's location to the subgoal *append*([*X*|*Y*], *Z*, *U*) in the clause

$$qsort([X|W], U) \text{ :- } part(W, X, U1, V1), qsort(U1, Y), qsort(V1, Z), append([X|Y], Z, U)$$

(9) A local fix gives instead:

$$qsort([X|Y], Z) \text{ :- } part(Y, X, W, X1), qsort(W, Z1), qsort(X1, V1), append(Z1, [X|V1], Z)$$

(10) Up to this point, all the bugs in the original program have been detected and corrected. If we now continue to debug the program, the debugger will keep on generating arbitrarily long lists as test input without reporting an error. (The uncovered cases of *part* are of no consequence.) We would be led to believe, by induction, that the program is correct with respect to its specifications, though no formal proof has been obtained.

EXAMPLE 6.2. *Debugging a merge-sort program.*

We now demonstrate how the debugger deals with a looping error in the merge-sort program in Figure 22 ("//" is Prolog's symbol for integer division). Figure 23 contains the specifications.

Like Quicksort, merge-sort is another example of solving a problem by divide and conquer. The program accepts a list, breaks it into roughly equivalent halves, recursively sorts the sublists, then merges the sorted halves. Note that the predicates used in the above specifications are the same ones as were used for Quicksort.

$$
\begin{array}{rcl}
spec(msort(X,Y)) & :- & ordered(Y),\ perm(X,Y) \\
spec(break(X,N,Y,Z)) & :- & append(Y,Z,X),\ length(Y,N) \\
spec(merge(X,Y,Z)) & :- & rm\_list(X,Z,Y),\ ordered(Z) \\
wfo(msort(X,Y),\ msort(U,V)) & :- & shorter(X,U) \\
wfo(break(X,A,B,C),\ break(Y,D,E,F)) & :- & shorter(X,Y) \\
wfo(merge(X,Y,A),\ merge(U,V,B)) & :- & shorter(X,U) \\
wfo(merge(X,Y,A),\ merge(U,V,B)) & :- & shorter(Y,V)
\end{array}
$$

Figure 23. Specifications for merge-sort

(1) The program has no problem solving the empty list: the goal $msort([\,],X)$ leads to the solution $msort([\,],[\,])$.
(2) However, the debugger quickly gets into trouble with the one-element list: $msort([x],X)$. The goal clause instance

$$
\begin{array}{rcl}
msort([x],X) & :- & length([x],1),0\ is\ 1\ //\ 2,break([x],0,[\,],[x]), \\
 & & msort([\,],[\,]),msort([x],Y),merge([\,],Y,X)
\end{array}
$$

is found to be looping. The debugger found that the procedure call $msort([x],Y)$ to the second clause of *msort* violated the well-founded ordering for that recursive procedure.
(3) The debugger adds subgoals to ensure a decrease in the clause:

$$
\begin{array}{rcl}
msort(X,Y) & :- & length(X,U),V\ is\ U\ //\ 2,break(X,V,Z1,U1), \\
 & & msort(Z1,W1),msort(U1,Y2),merge(W1,Y2,Y)
\end{array}
$$

and continues debugging.
(4) Now it finds the goal $msort([x],[x])$ uncovered.
(5) This suggests the clause

$$msort([X],[X])\ :-\ true$$

(6) Since the error was due to a missing case, the debugger restores the original clause.
(7) It turns out that the looping bug was due to the behavior of procedure *break*. A one-element list is always broken into an empty list and the original one-element list. This list is never reduced in the recursive call, and, therefore, needs to be treated as a special case. Adding a unit clause for one-element lists resolves the problem completely.

Since the knowledge necessary for the discovery, location, and correction of bugs is either built into the debugger or furnished as program specifications, user intervention during a debugging session is reduced to a minimum. The user simply needs to supply top-level goals, and type "yes" when asked whether to continue with additional test cases.

7. Program Synthesis

Shapiro's (1983) Model Inference System can generate Prolog programs from a sequence of examples. The system is also supplied with the skeleton of the desired program, indicating which procedures call which. We show how an executable model can be incorporated into such a system, replacing the oracle, usually played by the user, thereby automating the whole synthesis process.

EXAMPLE 7.1. *Program synthesis using an executable model.*

This example shows the synthesis of an insertion sort program (for distinct elements). The specifications of *isort* in Figure 8 is given to the system, along with the declaration that *isort* calls procedures *isort* and *insert* (with two and three arguments, respectively) and *insert* calls itself, recursively, and "<". Note that this declaration is also required for Shapiro's system to work, and forms the basis for the construction of the search tree. The process, starting with an empty *isort* program, is summarized below:

(1) Test goal generated: $isort([\,], X)$
Error: missing solution $isort([\,],[\,])$
Diagnosis: $isort([\,],[\,])$ is uncovered
Action: Add clause $isort(X, X)$:- ***true***
Note: The system quickly finds a clause to cover the first goal.
This is considered the most general simple covering clause.
(2) Test goal generated: $isort([x], Y)$
Error: None
(3) Test goal generated: $isort([0,1], X)$
Error: None
(4) Test goal generated: $isort([1,0], X)$
   (a) Error: wrong solution $isort([1,0],[1,0])$
   Diagnosis: $isort([1,0],[1,0])$:- ***true*** is false
   Action: Remove clause $isort(X, X)$:- ***true***
   Note: An incorrect clause is identified and deleted (the program is now empty again). The system then checks to make sure that all the goals are still covered.
   (b) Error: missing solution $isort([\,],[\,])$
   Diagnosis: $isort([\,],[\,])$ is uncovered
   Action: add clause $isort(X,[\,])$:- ***true***
   Note: This is the next best covering clause.
   (c) Error: missing solution $isort([x],[x])$
   Diagnosis: $isort([x],[x])$ is uncovered
   Action: add clause $isort([X|Y], Z)$:- $insert(X, Y, Z)$
   (d) Error: missing solution $isort([x],[x])$
   Diagnosis: $insert(x,[\,],[x])$ is uncovered
   Action: add clause $insert(X, Y, [X|Y])$:- ***true***
   (e) Error: missing solution $isort([1,0],[0,1])$
   Diagnosis: $insert(1,[0],[0,1])$ is uncovered
   Action: add clause $insert(X,[Y|Z],[Y,X|Z])$:- ***true***
   (f) Error: wrong solution $isort([1,0],[1,0])$
   Diagnosis: $insert(1,[0],[1,0])$:- ***true*** is false
   Action: remove clause $insert(X, Y, [X|Y])$:- ***true***
   Note: This clause is incorrect, since according to the specification of *insert*, inserting the element 1 into the list [0] should result in the output [0, 1] instead of [1, 0]. It is being replaced.
   (g) Error: missing solution $isort([x],[x])$
   Diagnosis: $insert(x,[\,],[x])$ is uncovered
   Action: add clause $insert(X, Y, [X])$:- ***true***
   Note: This clause does not solve the problem.

(h) Error: missing solution $isort([0,1],[0,1])$
Diagnosis: $insert(0,[1],[0,1])$ is uncovered
Action: add clause $insert(X,[Y|Z],[X,Y|Z]) :- X < Y$
Note: The program is now correct with respect to all the facts known to the system. (A *fact* is a ground term with a value of ***true*** or ***false***; it may be supplied by the user, or generated by the system when solving goals. For example, $isort([1,0],[0,1]) =$ ***true*** is a fact, as is $isort([1,0],[1,0]) =$ ***false***. A correct program should succeed for true goals, and fail on known false ones.) The system proceeds to check if the program satisfies the goals generated so far.

(i) Error: wrong solution $isort([x],[\,])$
Diagnosis: $isort([x],[\,]) :-$ ***true*** is false
Action: remove clause $isort(X,[\,]) :-$ ***true***
Note: Removing a clause usually causes problems. The system has to recheck all the facts and goals.

(j) Error: missing solution $isort([\,],[\,])$
Diagnosis: $isort([\,],[\,])$ is uncovered
Action: add clause $isort([\,],[\,]) :-$ ***true***

(k) Error: wrong solution $isort([0,1],[1,0])$
Diagnosis: $insert(0,[1],[1,0]) :-$ ***true*** is false
Action: remove clause $insert(X,[Y|Z],[Y,X|Z]) :-$ ***true***
Note: Search continues ...

(l) Error: missing solution $isort([1,0],[0,1])$
Diagnosis: $insert(1,[0],[0,1])$ is uncovered
Action: add clause $insert(X,[Y|Z],[Y,X|Z]) :- Y < X$
Note: Found the right clause, but a base clause is still incorrect.

(m) Error: wrong solution $isort([0,1],[0])$
Diagnosis: $insert(0,[1],[0]) :-$ ***true*** is false
Action: remove clause $insert(X,Y,[X]) :-$ ***true***

(n) Error: missing solution $isort([x],[x])$
Diagnosis: $insert(x,[\,],[x])$ is uncovered
Action: add clause $insert(X,[\,],[X]) :-$ ***true***
Note: Up to this point, the synthesized program solves all the generated goals, $isort([\,],[\,])$, $isort([x],[x])$, $isort([0,1],[0,1])$, and $isort([1,0],[0,1])$, successfully.

No user involvement was needed in these steps.

(5) Test goal generated: $isort([0,1,2],X)$
Error: None

(6) Test goal generated: $isort([0,2,1],X)$

(a) Error: wrong solution $isort([0,2,1],[0,2,1])$
Diagnosis: $isort([0,2,1],[0,2,1]) :- insert(0,[2,1],[0,2,1])$ is false
Action: remove clause $isort([X|Y],Z) :- insert(X,Y,Z)$

(b) Error: missing solution $isort([x],[x])$
Diagnosis: $isort([x],[x])$ is uncovered
Action: add clause $isort([X|Y],Z) :- isort(Y,V), insert(X,V,Z)$

(7) Test goal generated: $isort([1,0,2],X)$
Error: None

$$
\begin{array}{rcl}
isort([\,],[\,]) & & \\
isort([X|Ys],Zs) & :- & isort(Y,Ys),\ insert(X,Ys,Zs) \\
insert(X,[Y|Zs],[X,Y|Zs]) & :- & X < Y \\
insert(X,[\,],[X]) & & \\
insert(X,[Y|Zs],[Y|Vs]) & :- & insert(X,Zs,Vs),\ Y < X
\end{array}
$$

Figure 24. Synthesized insertion sort

(8) Test goal generated: $isort([1,2,0],X)$
Error: None
(9) Test goal generated: $isort([2,0,1],X)$
(a) Error: wrong solution $isort([2,0,1],[0,2,1])$
Diagnosis: $insert(2,[0,1],[0,2,1])$:- $0 < 2$ is false
Action: remove clause $insert(X,[Y|Z],[Y,X|Z])$:- $Y < X$
(b) Error: missing solution $isort([1,0],[0,1])$
Diagnosis: $insert(1,[0],[0,1])$ is uncovered
Action: add clause $insert(X,[Y|Z],[Y|V])$:- $insert(X,Z,V), Y < X$
Note: Finally, a clause for the recursive case of *insert* is found.
(10) Test goal generated: $isort([2,1,0],X)$
Error: None

The last permutation of the three-element list now executes correctly on the synthesized program shown in Figure 24.

8. Conclusion

Traditional testing methods are mainly concerned with designing test cases that might show a program to be incorrect. Since knowing that a program is incorrect is not the same as knowing the cause, the methods do not deal directly with the problem of isolating and correcting the bugs disclosed by testing. Our methodology, on the other hand, is intended to combine the functions of testing and debugging, for logic programs, in one uniform framework.

We have used logic, in the form of Horn-clause programs, for both specification and computation. We have shown that user-supplied executable models, besides defining the intended behavior of a program, may also be used to generate test cases for bug discovery. We employ a Prolog interpreter to locate bugs, by finding discrepancies with the specified results. We have also incorporated heuristics to analyze bugs and suggest fixes, and used techniques from deductive theorem proving and inductive synthesis to mechanize the bug correction process, with the aid of the specifications.

Our *Constructive Interpreter* contains three major components: test case generator, bug locator, and bug corrector. It performs much as an active human expert does during a typical debugging session. Given a program and its specifications, it can

(i) execute a goal as an ordinary interpreter would,
(ii) generate test cases systematically from symbolic input data,
(iii) verify the results of a computation,

(iv) trace the execution of the program,
(v) locate a bug when a goal does not compute correctly, and
(vi) attempt to repair the bug once located.

When supplied with a program and its executable model, the test case generator can generate test data systematically by executing models. (Were we to use a breadth-first mechanism to generate test cases, we could generate a complete [perhaps infinite] set of test cases for that program.) The *Constructive Interpreter* then executes the program on the test data. Should the execution fail to return an answer that agrees with the specifications, the bug locator will automatically locate a bug causing the failure. The bug corrector then analyzes the nature of the bug and utilizes correction heuristics which guide the use of the specifications and which attempt to repair the bug. This bug-fixing process sometimes involves the use of a deductive theorem prover which tries to construct a proof and deduce sufficient conditions to amend the program, and of a program generator which tries inductively to synthesize the missing part of the program.

One of the earliest works on heuristic debugging and synthesis is (Sussman, 1975). There, the search space for alternative programs consists primarily of permutations of the program statements, and new programs are formed from generalizations of working ones. Invariant assertions are used in (Katz and Manna, 1975; Katz and Manna, 1976; Dershowitz, 1983; Dershowitz, 1985) to diagnose and correct errors in Algol-like programs, by modifying the programs until the required invariants can be shown to hold. Working with logic programs, we use specifications for each procedure (including auxiliary ones), rather than invariants. Unlike methods such as (Katz and Manna, 1975), a correct clause is never "debugged" by our system; only a clause found faulty by testing is subjected to formal verification. Other work on declarative debugging includes (Ferrand, 1985; Pereira, 1986; Lloyd, 1987; Drabent *et al.*, 1988; Pereira and Calejo, 1988; Brna *et al.*, 1992; Naish, 1992; Nilsson and Fritzson, 1992).

A system designed for synthesizing Prolog programs inductively is described in (Shapiro, 1991). Querying an oracle (the user) to verify the results of procedure calls, the system diagnoses errors by isolating an erroneous procedure, and suggesting a correction that agrees with the desired program. A similar diagnostic approach was applied to Pascal programs in (Renner, 1991); a more efficient procedure for diagnosing incorrect clauses was suggested in (Plaisted, 1984); an improved refinement operator can be found in (Huntbach, 1986); a method for generating test sets from algebraic specifications is presented in (Bouge *et al.*, 1986); and an efficient method for the induction of logic programs is described in (Muggleton and Feng, 1990). Our system is similar in structure and employs similar heuristics, but replaces the oracle with an executable model. In addition to heuristics, we use deductive means as tools for correcting errors. We have also incorporated checks for termination.

Another approach to automatic debugging uses a fairly complete description of the algorithm to specify the intended behavior of the program to be debugged. It can either be a "model program" (Ruth, 1976; Adam and Laurent, 1980; Murray, 1986) or a "program description" (Johnson and Soloway, 1985). Debuggers of this kind have to rely on heuristics to match algorithms and programs. A mismatch usually signals the existence and location of a bug, and the stored information can then be used to correct the bug. A Prolog debugger combining program sources and traces of program executions in the debugging process is described in (Ducassé, 1992). A summary of knowledge-based program debugging systems can be found in (Seviora, 1987). In place of a description of

how the desired program should work, we make do with a specification of what goals the procedures are expected to solve. Once a bug is found, deductive and inductive corrective measures are employed in an attempt to bring the program in line with the given specifications.

In this work, we have shown how two different inference mechanisms, deduction and induction, can complement each other. Logical deduction is a powerful technique in that the results of deductive inference are guaranteed correct (consistent with the axioms). In the context of logic programming, deduction can be used to execute, verify, transform, and derive programs. Induction is used to test programs and incrementally modify them if they are incorrect or incomplete, by using deductive and heuristic error-correcting methods.

The deductive approach to synthesis of logic programs starts with a goal representing the desired logic procedure and proceeds by repeatedly applying inference rules, until the original goal reduces to a set of primitive formula (cf. (Clark, 1981; Hogger, 1981)). Other approaches using synthesis rules or transformation rules for program synthesis can be found, for example, in (Manna and Waldinger, 1980; Dershowitz, 1985); approaches to inductive program synthesis are surveyed in (Biermann, 1976; Smith, 1980); a comprehensive review of synthesis methodologies (including synthesis from formal specifications, from examples, through natural language dialogue, and from using a mechanized assistant) is described in (Biermann, 1992).

In conclusion, we have demonstrated that, in the realm of logic programming, the tedious tasks of program synthesis and debugging are at least somewhat amenable to automation.

Acknowledgements

The first author's research was supported in part by the U. S. National Science Foundation under Grants CCR-90-07195 and CCR-90-24271 and by a Meyerhoff Visiting Professorship at the Weizmann Institute of Science. The second author's research has been supported in part by the U. S. Naval Postgraduate School and by the U. S. Army Artificial Intelligence Center. We would also like to thank the referees for their helpful comments on an earlier draft of this paper.

References

Adam, A., Laurent, J.-P. (1980). LAURA, a system to debug student programs. *Artificial Intelligence*, **15**, 75–122.

Apt, K. R., van Emden, M. H. (1982). Contributions to the theory of logic programming. *J. ACM*, **29**, 841–862.

Biermann, A. W. (1976). Approaches to automatic programming. In Rubinoff, M., Yovits, M. C., eds., *Advances in Computers*, volume 15, pages 1–63. New York: Academic Press.

Biermann, A. W., (1992). Automatic programming. In Shapiro, S. C., ed., *Encyclopedia of Artificial Intelligence*, pages 59–83. New York: John Wiley & Sons, Inc., second edition.

Bouge, L., Choquet, N., Fribourg, L., Gaudel, M.-C. (1986). Test sets generating from algebraic specifications using Logic Programming. *J. of Syst. and Softw.*, **6**(4), 343–360.

Brna, P., Bundy, A., Pain, H. (1992). A framework for the principled debugging of Prolog programs: How to debug non-terminating programs. In Brough, D. R., ed., *Logic Programming: New Frontiers*, chapter 2, pages 22–55. Oxford: Intellect.

Clark, K. L. (1981). The synthesis and verification of logic programs. Research Report DOC 81/36, Department of Computing, Imperial College, London, England.

Clocksin, W. F., Mellish, C. S. (1987). *Programming in Prolog*. New York: Springer-Verlag, third edition.

Dershowitz, N. (1983). *The Evolution of Programs*. Boston, MA: Birkhäuser.

Dershowitz, N. (1985). Synthetic programming. *Artificial Intelligence*, **25**, 323–373.

Drabent, W., Nadjm-Tehrani, S., Maluszynski, J. (1988). Algorithmic debugging with assertions. In *Proc. of Workshop on Meta-Programming in Logic Programming*, Bristol. META 88.

Ducassé, M. (1992). Opium — An advanced debugging system. In Comyn, G., Fuchs, N. E., Ratcliffe, M. J., eds., *Proc. of the Second International Logic Programming Summer School*, pages 303–312, Zurich, Switzerland. Available as Vol. 636 of *Lecture Notes in Computer Science*, Springer-Verlag.

Ferrand, G. (1985). Error diagnosis in logic programming, an adaptation of E. Y. Shapiro's method. Rapport 375, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France.

Floyd, R. W. (1967). Assigning meanings to programs. In *Proc. of Symposia in Applied Mathematics, XIX: Mathematical Aspects of Computer Science*, pages 19–32, Providence, RI. American Mathematical Society.

Fuchi, K., Furukawa, K. (1986). The role of logic programming in the fifth generation computer project. In *Third International Conference on Logic Programming*, pages 1–24, London, United Kingdom.

Gerhart, S. L., Yelowitz, L. (1976). Observations of fallibility in applications of modern programming methodologies. *IEEE Trans. on Softw. Eng.*, **SE-2**(3), 195–207.

Goguen, J. A., Meseguer, J. (1986). EQLOG: Equality, types, and generic modules for logic programming. In DeGroot, D., Lindstrom, G., eds., *Logic Programming: Relations, Functions, and Equations*. Englewood Cliffs, NJ: Prentice Hall.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, **12**(10), 576–583.

Hogger, C. J., (1981). Derivation of logic programs. *J. ACM*, **28**(2), 372–392.

Huntbach, M. M. (1986). An improved version of Shapiro's Model Inference System. In Shapiro, E., ed., *Proc. of the Third International Conference on Logic Programming*, pages 180–187, London, UK. Available as Vol. 225, Lecture Notes in Computer Science, Springer-Verlag.

Johnson, W. L., Soloway, E. (1985). PROUST: Knowledge-based program understanding. *IEEE Trans. on Softw. Eng.*, **SE-11**(3), 267–275.

Josephson, N. A., Dershowitz, N. (1989). An implementation of narrowing. *J. of Logic Programming*, **6**(1&2), 57–77.

Katz, S. M., Manna, Z. (1975). Towards automatic debugging of programs. In *Proc. of the International Conference on Reliable Software*, pages 143–155, Los Angeles, CA.

Katz, S., Manna, Z. (1976). Logical analysis of programs. *Commun. ACM*, **19**(4), 188–206.

Knuth, D. E. (1973). *The Art of Computer Programming: Searching and Sorting*, volume 3. Reading, MA: Addison-Wesley Publishing Company.

Kowalski, R. A., van Emden, M. H. (1976). The semantics of predicate logic as a programming language. *J. ACM*, **23**, 733–742.

Kowalski, R. A. (1974). Predicate logic as programming language. In *Proc. of the IFIP Congress*, pages 569–574, Amsterdam, The Netherlands.

Kowalski, R. (1985). The relation between logic programming and logic specification. In Hoare, C. A. R., Shepherdson, eds., *Mathematical Logic and Programming Languages*. Englewood Cliffs, NJ: Prentice/Hall International.

Lee, Y. (1988). *Debugging Logic Programs using Executable Specifications*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois.

Liskov, B. H., Berzins, V. (1986). An appraisal of program specifications. In Gehani, N., McGettrick, A., eds., *Software Specification Techniques*, pages 3–23. Reading, Mass.: Addison-Wesley.

Lloyd, J. W. (1984). *Foundations of Logic Programming*. New York: Springer-Verlag.

Lloyd, J. W. (1987). Declarative error diagnosis. *New Generation Computing*, **5**, 133–154.

Manna, Z., Waldinger, R. J. (1980). A deductive approach to program synthesis. *ACM Trans. on Prog. Lang. Syst.*, **2**(1), 90–121.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM*, **3**, 184–195.

Muggleton, S., Feng, C. (1990). Efficient induction of logic programs. Research Memorandum TIRM-90-044, Turing Institute, Glasgow, Scotland.

Murray, W. R. (1986). *Automatic Program Debugging for Intelligent Tutoring Systems*. PhD thesis, The University of Texas at Austin, Austin, Texas.

Myers, G. J. (1979). *The Art of Software Testing*. New York: Wiley.

Naish, L. (1992). Declarative debugging of lazy functional programs. Report 92/6, Department of Computer Science, University of Melbourne, Australia.

Nilsson, H., Fritzson, P. (1992). Algorithmic debugging for lazy functional languages. In Bruynooghe, M., Wirsing, M., eds., *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 385–399, Leuven, Belgium. Available as Vol. 631 of *Lecture Notes in Computer Science*, Springer-Verlag.

Pereira, L., Calejo, M. (1988). A framework for Prolog debugging. In *Proc. of Fifth International Conference and Symposium on Logic Programming*, Seattle.

Pereira, L. M. (1986). Rational debugging in logic programming. In *Proc. of the Third International Conference on Logic Programming*, pages 203–210, London, United Kingdom. Available as Vol. 225, Lecture Notes in Computer Science, Springer-Verlag.

Plaisted, D. A. (1984). An efficient bug location algorithm. In *Proc. of the Second International Logic Programming Conference*, pages 151–157, Uppsala, Sweden.

Renner, S. A. (1991). *Logical Error Diagnosis.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois.

Ruth, G. (1976). Intelligent program analysis. *Artificial Intelligence*, **7**, 65–85.

Seviora, R. E. (1987). Knowledge-based program debugging systems. *IEEE Software*, **4**, 20–32.

Shapiro, E. Y. (1983). *Algorithmic Program Debugging.* Cambridge, MA: MIT Press.

Shapiro, E. Y. (1991). Inductive inference of theories from facts. In Lassez, J.-L., Plotkin, G., eds., *Computational Logic: Essays in Honor of Alan Robinson*, pages 199–254. Cambridge, MA: MIT Press.

Smith, D. R. (1980). A survey of synthesis of Lisp programs from examples. In *International Workshop on Program Construction*, pages 307–324, Bonas, France.

Smith, D. R. (1982). Derived preconditions and their use in program synthesis. In *Proc. of the Sixth Conference on Automated Deduction*, pages 172–193, New York, NY.

Sussman, G. J. (1975). *A Computer Model of Skill Acquisition.* New York: American Elsevier.

Waldinger, R. J., Levitt, K. N. (1974). Reasoning about programs. *Artificial Intelligence*, **5**(3), 235–316.