Logical Debugging

YUH-JENG LEE
Computer Science Department
Naval Postgraduate School
Code CS/Le
Monterey, CA 93943

NACHUM DERSHOWITZ[1]
Department of Computer Science
University of Illinois
1304 W Springfield Ave.
Urbana, IL 61801

Logic programming offers a distinctive feature that is rarely met by other traditional programming languages: namely, one can use logic for both specification and computation. We present a methodology for reasoning about logic programs and their specifications. This methodology can be applied to program debugging as well as program synthesis. We focus on the use of executable specifications to generate test cases for bug discovery, locate bugs when test data cause a program to fail, and guide deductive and inductive bug correction. The behavior of the automated debugger is demonstrated through several examples.

1 Introduction

Logic programs have relatively simple syntax and well-understood semantics. In addition, logic programming offers an attractive feature rarely met in traditional programming languages, namely, the ability to use logic for both *specification* and *computation*. We present a methodology for reasoning about logic programs and their specifications. Debugging a given program involves three steps: bug discovery, bug location, and bug correction. We focus on the use of executable specifications to generate test cases for bug discovery, locate bugs when test data cause a program to fail, and guide deductive and inductive bug correction.

The typical process of debugging—designing a test case, detecting an error in the program, locating the error, and fixing it—can also be applied in program synthesis. For debugging, we use the *given program* as a starting point to search for a correct one. For program synthesis, we start the search with an *empty program*. With the application of executable specifications, an inductive search space for programs, and a deductive mechanism for synthesis, our system allows one to specify a program and give the skeleton of the recursive structure, and the system tries to do the rest.

[1]Research supported in part by the U. S. National Science Foundation under Grants CCR-90-07195 and CCR-90-24271.

Section 2 reviews some of the related work. Section 3 examines the basic ideas of logic programming and a Prolog meta interpreter on which our debugger is based. Section 4 discusses the use of executable specifications. Section 5 analyzes the algorithm for locating program errors. Section 6 introduces the heuristics for correcting errors. Section 7 shows a mechanism for automatic program synthesis, based on the results from sections 5 and 6. Section 8 presents the integrated automated debugger. Section 9 contains concluding remarks.

2 Related Work

In Shapiro (1983) the Model Inference System (MIS) was designed for synthesizing Prolog programs inductively. By querying an oracle (usually the user) to verify the results of procedure calls, the system can diagnose an error by isolating an erroneous procedure, and suggests a correction to produce the desired program. A similar diagnostic approach was applied to Pascal programs in Renner (1991). A more efficient algorithm to diagnose an incorrect clause was suggested in Plaisted (1984). An improved refinement operator can be found in Huntbach (1986). Other work on declarative debugging can be found in Pereira (1986) and Lloyd (1987).

In Katz & Manna (1975), Katz & Manna (1976), and Dershowitz (1983) it has been shown that it is possible to use invariant assertions to diagnose and correct program errors, by modifying programs so the necessary invariants can be obtained.

Another approach to automatic debugging uses a fairly intensive, complete description of the algorithm to specify the intended behavior of the program to be debugged. It can either be a *model program* (such as in Ruth (1976), Adam & Laurent (1980), and Murray (1986)) or a *program description* (such as in Johnson & Soloway (1985)). Debuggers of this kind have to rely on heuristics to match between 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 summary of knowledge based program debugging systems can be found in Seviora (1987).

The deductive synthesis of logic programs starts with a goal representing the desirable logic procedure and proceeds by applying repeatedly inference rules, until the original goal becomes a set of atomic formula (cf. Hogger (1981) and Clark (1981)). Other approaches using synthesis rules or transformation rules for program synthesis can be found, for example, in Manna & Waldinger (1980) and Dershowitz (1985).

3 Logic Programming

Broadly defined, a logic programming language is a language that is based on a formal logic system, with operational semantics defined by deduction in that system. Lisp (or pre-

cisely, pure Lisp), for example, is a logic programming language based on the λ–calculus. Languages based on equational logic, such as EQLOG by Goguen & Meseguer (1986) and rewrite systems by Dershowitz & Jouannaud (1990), also fall into this category.

A more common definition of logic programming refers to the use of first-order predicate logic, or a subset of it, as a programming language, with emphasis on using predicates and deduction to describe computation. Based on the resolution principle (Robinson (1965)) and its successive improvements, efficient schemes for processing predicate logic by computers have been developed. The principal idea is to represent programs with the (definite) Horn clause subset of the first-order predicate logic (Kowalski (1974) and Kowalski & van Emden (1976)). This breakthrough set the basis for procedural interpretation to Horn clause logic and accelerated progress in the development of logic programming languages. Prolog (Clocksin & Mellish (1987)), the prototypical logic programming language, is nowadays a viable alternative to Lisp in symbolic processing and Artificial Intelligence research.

3.1 PROLOG

The operational semantics of Prolog is based on SLD-resolution (Apt & van Emden (1982) and Lloyd (1984)). That is, Prolog's execution follows a sequential simulation of the nondeterministic computation, using a depth-first search strategy with a backtracking mechanism incorporated. In the computation process, Prolog will try all unifiable clauses sequentially, in the order they occur in the program text, and subgoals are solved from left to right. When it 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 with a different solution. If none can be found, the entire computation fails.

The computation process can also be described as the traversal of a computation tree. A computation tree $\boldsymbol{T}$ of a program $\boldsymbol{P}$ is a rooted, ordered tree. Each node in the tree has the form $\boldsymbol{p}(\boldsymbol{x},\boldsymbol{y})$, where $\boldsymbol{p}$ is a procedure (predicate) name, and $\boldsymbol{x}$ and $\boldsymbol{y}$ represent input and output vectors over some domain. For the clause

$$\boldsymbol{p}(\boldsymbol{x},\boldsymbol{y}) \quad :- \quad \boldsymbol{p}_1(\boldsymbol{x}_1,\boldsymbol{y}_1),\ \boldsymbol{p}_2(\boldsymbol{x}_2,\boldsymbol{y}_2),\ \ldots\ ,\ \boldsymbol{p}_k(\boldsymbol{x}_k,\boldsymbol{y}_k)$$

involved in a computation, the corresponding part in $\boldsymbol{T}$ includes the internal node $\boldsymbol{p}(\boldsymbol{x},\boldsymbol{y})$ and its sons $\boldsymbol{p}_1(\boldsymbol{x}_1,\boldsymbol{y}_1)$, $\boldsymbol{p}_2(\boldsymbol{x}_2,\boldsymbol{y}_2)$, $\ldots$, and $\boldsymbol{p}_k(\boldsymbol{x}_k,\boldsymbol{y}_k)$. The meaning of this tree is as follows: procedure $\boldsymbol{p}$, on input $\boldsymbol{x}$, calls $\boldsymbol{p}_1$ on $\boldsymbol{x}_1$, and if this call returns $\boldsymbol{y}_1$, then $\boldsymbol{p}$ calls $\boldsymbol{p}_2$ on $\boldsymbol{x}_2$, and if this call returns $\boldsymbol{y}_2$ then ..., then $\boldsymbol{p}$ calls $\boldsymbol{p}_k$ on $\boldsymbol{x}_k$, and if this call returns $\boldsymbol{y}_k$, then $\boldsymbol{p}$ returns $\boldsymbol{y}$ as its output. If a node $\boldsymbol{p}(\boldsymbol{x},\boldsymbol{y})$ has no sons, then the procedure $\boldsymbol{p}$ has a legal computation that returns $\boldsymbol{y}$ on input $\boldsymbol{x}$ without performing any procedure calls.

Programs, for our purposes, are presumed to obey their Horn clause declarative semantics, i.e., those extra-logical features, such as cuts, clause order, and subgoal order, may affect efficiency and termination, but not correctness.

3.2 Meta Programming

One important feature of a programming language that has simple, well-defined semantics (such as pure Prolog or pure Lisp) is that it can easily be used to build a system that manipulates and executes other programs written in that language. As pointed out in Fuchi & Furukawa (1986), meta programming can be characterized as programming that: (1) handles programs as data; (2) handles data as programs and evaluates them; and (3) handles a result (success or fail) of computation as data.

This meta programming capability is essential when implementing a system to reason about programs. It provides a basis for building a powerful programming environment. Prolog is especially attractive in this aspect, since one can easily write a meta interpreter to execute pure Prolog programs in just three lines, as shown in Figure 1. The first clause

```
interpret( (G1,G2) )  :-  interpret(G1),
                          interpret(G2)
interpret(Goal)       :-  system(Goal),
                          Goal
interpret(Goal)       :-  clause(Goal,SubGoals),
                          interpret(SubGoals)
```

Figure 1: A Prolog meta interpreter

solves a conjunctive goal by recursively solving its components. The second clause checks if (a noncomposite) *Goal* is a system (built-in) predicate (***system*** itself is a built-in predicate that succeeds if *Goal* is a call to a built-in procedure) and, if it is, executes the goal directly. The third clause uses a built-in predicate ***clause*** both to find a clause whose head 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.

4 Executable Specifications

In software development, a specification may be regarded as an abstraction of a concrete problem at hand, as the starting point for the subsequent program development, and as the criterion for judging the correctness of a final software product. It is 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.

As long as the specification is formulated in a language which has operational semantics, the specification becomes a prototype (a partially complete functional model of the target system), and the behavior of which may be scrutinized to determine if it is in fact the behavior of the desired software product. A logic-based language would serve

this purpose well (cf. Clark (1981) and Kowalski (1985)), since it is a formal language and has simple syntax, well-defined declarative semantics, and a well-understood deductive mechanism. Simple in syntax makes a specification easier to understand. Having a well-defined declarative semantics facilitates the construction of high-level specifications, since a specification language is to describe intended behavior (*what*) without prescribing a particular algorithm (*how*). The deductive mechanism provides operational validation of the specifier's intentions.

4.1 Specifications in Prolog

First-order predicate calculus has long been used as a specification language. The typical approach to program verification (e.g., Hoare (1969) and Katz & Manna (1976)) expresses specifications in first-order logic, and relates them to conventional programs by defining the semantics of programs in a "programming" logic. As mentioned earlier, in logic programming, one can use a single language for both specification and computation.

Since Horn clauses are a powerful subset of first-order logic, Prolog can often be used for specifications with the advantageous extra feature of executability: a program's specifications can be written in Prolog itself and can be executed by the Prolog interpreter or compiler directly.

For our debugging purpose, the specifications of a program describe the relationships between program variables by giving input/output constraints. They define the functionalities of the program without imposing a restriction on how these functionalities are to be achieved. The specifications can be viewed as procedural abstractions (cf. Liskov & Berzins (1986)). A procedural abstraction performs a mapping from a set of input values to a set of output values.

It may then be argued that specifications are no different from programs. Indeed, in logic programming, as Kowalski (1985) has contended, *execution efficiency* is the main criterion for distinguishing programs from complete specifications. Specifications emphasize clarity and simplicity but not efficiency, while in the implementation of programs, efficiency is the main consideration. In other words, specifications written in Prolog can be considered to be *nonalgorithmic*, *executable*, and perhaps *inefficient* programs.

Another aspect of specifications is that they provide information on the well-founded ordering of input arguments for recursive procedures. A well-founded ordering $\succ$ is a binary relation on elements of a nonempty set $\boldsymbol{S}$ such that the relation is transitive, asymmetric, and irreflexive, and such that $\boldsymbol{S}$ has no infinite descending sequences. The ordering specifies, for a particular recursive call, which arguments should be decreasing. This is used for detecting looping.

In this research, we presume that specifications faithfully reflect the intended requirements of a program (cf. Gerhart & Yelowitz (1976)). To obtain the desired effect, it is sometimes necessary to use impure features, i.e., non-logical control structures, of Prolog.

More expressive languages, e.g., EQLOG (Goguen & Meseguer (1986)), HOPE with unification (Darlington *et al.* 1986), and RITE (Josephson & Dershowitz (1986)) may be even more suitable for specifications.

4.2 Generation of Test Cases

Executable specifications of a program not only compute the desired output, but also generate useful test cases for that program, provided that axioms for primitive predicates are supplied. The information contained in specifications regarding the expected output behavior is indispensable for checking the correctness of the results of program execution, while test cases help reveal instances of incorrect output.

To generate test cases for a given goal, we first run the specifications of that goal to obtain a pair consisting of an input along with its expected output. We then use only the input value to run the goal on the program to be debugged. If the execution fails, goes into a loop, or returns an incorrect output value, then this test case has shown us 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 specifications 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. If we use a breadth-first mechanism to generate test cases, we can generate a complete (perhaps infinite) set of test cases for that program.

EXAMPLE 1. *Generating test cases from specifications*

Suppose we have the following specification for a sorting procedure:

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

which says that feeding a list "In_List" to the procedure *sort*, the list "Out_List" is a correct result if it is in 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 2), *ordered* accepts a

$$\begin{array}{rcl} perm([\,],[\,]) & & \\ perm([X|Xs],Ys) & :- & del(X,Ys,Zs), \\ & & perm(Xs,Zs) \\ del(X,[X|Xs],Xs) & & \\ del(X,[Y|Xs],[Y|Ys]) & :- & del(X,Xs,Ys) \end{array}$$

Figure 2: Procedure *perm*

list in ascending order (Figure 3), and the primitive predicate *lt* defines the basic *less than* relation (Figure 4), then by executing $spec(sort(In\_List, Out\_List))$ with uninstantiated

```
ordered([ ])
ordered([X])
ordered([X1, X2|X])   :-   lt(X1, X2),
                           ordered([X2|X])
```

Figure 3: Procedure *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 4: Primitive predicate *lt*

variables we can generate a sequence of input/output pairs. The first value generated for *In_List* is an empty list (i.e., [ ]), then a one-element list (i.e., $[X]$), then two-element lists with all possible permutations (i.e., $[0, 1]$ and $[1, 0]$), then three-element lists with all possible permutations, etc. The variable *Out_List* contains the expected result for each given input, and can be used to verify the correctness of a program (see section 4.3).

4.3 Validation of Computation Results

When a program is to be debugged, we assume that the properties of each procedure in the program can be described by the program's specifications. These nonalgorithmic specifications detail the relationships between program variables as well as the well-founded ordering under which successive input values to recursive procedures form a descending sequence. In other words, they define all legal input/output pairs for each procedure. Unspecified procedures are presumed correct.

Suppose we have a relation $\boldsymbol{R}$ that is defined by specifications $\boldsymbol{S}$ and is to be computed by program $\boldsymbol{P}$. If every instance of $\boldsymbol{R}$ computed by $\boldsymbol{P}$ can also be deduced from $\boldsymbol{S}$, then $\boldsymbol{P}$ is *partially correct* with respect to $\boldsymbol{S}$, i.e.,

$$\boldsymbol{if}\ \ \boldsymbol{P} \vdash \boldsymbol{R}\ \ \boldsymbol{then}\ \ \boldsymbol{S} \vdash \boldsymbol{R},$$

where $X \vdash Y$ denotes that conclusion Y can be derived or proved from assumption X. This actually means that the program $\boldsymbol{P}$ is consistent with the specification $\boldsymbol{S}$, or

$$\boldsymbol{S} \vdash \boldsymbol{P}.$$

If there is a computation result of $\boldsymbol{P}$ that cannot be deduced from $\boldsymbol{S}$, then $\boldsymbol{P}$ is *incorrect* with respect to $\boldsymbol{S}$.

On the other hand, if every instance of $\boldsymbol{R}$ defined by $\boldsymbol{S}$ can be obtained by executing $\boldsymbol{P}$, then $\boldsymbol{P}$ is *complete* with respect to $\boldsymbol{S}$, i.e.,

$$\boldsymbol{if}\ \ \boldsymbol{S} \vdash \boldsymbol{R}\ \ \boldsymbol{then}\ \ \boldsymbol{P} \vdash \boldsymbol{R}.$$

This means that the program $\boldsymbol{P}$ derives every instance of $\boldsymbol{R}$ that is defined by the specification $\boldsymbol{S}$, or

$$\boldsymbol{P} \vdash \boldsymbol{S}.$$

If there is an instance of $\boldsymbol{R}$ that is defined by $\boldsymbol{S}$ but cannot be the result of executing $\boldsymbol{P}$, then that instance is "uncovered" and $\boldsymbol{P}$ is *incomplete*.

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

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

5 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 has an infinite procedure invocation sequence. We now discuss how each of these three types of errors can be detected and located automatically, based on the meta programming capability of Prolog and executable specifications.

5.1 Locating Incorrect Clause

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

$$\boldsymbol{q}(\boldsymbol{u'}, \boldsymbol{v'}) \ :- \ \boldsymbol{r}_1\boldsymbol{'}, ..., \boldsymbol{r}_n\boldsymbol{'}$$

is the first application (instance) of a clause to return an incorrect output $\boldsymbol{v'}$ on input $\boldsymbol{u'}$, then the applied clause

$$\boldsymbol{q}(\boldsymbol{u}, \boldsymbol{v}) \ :- \ \boldsymbol{r}_1, ..., \boldsymbol{r}_n$$

of procedure $\boldsymbol{q}$ is incorrect. This is explained by the fact that, if $\boldsymbol{q}(\boldsymbol{u'}, \boldsymbol{v'})$ is the first call returning an incorrect output, all the procedure calls $\boldsymbol{r}_1\boldsymbol{'}, ..., \boldsymbol{r}_n\boldsymbol{'}$ must have completed earlier and returned correct results. Thus, the implication

$$\boldsymbol{q}(\boldsymbol{u}, \boldsymbol{v}) \ :- \ \boldsymbol{r}_1, ..., \boldsymbol{r}_n$$

is false (for the instance $\boldsymbol{u'}$, $\boldsymbol{v'}$) with respect to the specifications.

The algorithm can be summarized as the pseudo-Prolog code in Figure 5. To compute

```
execute( Goal, Message ) :-
    clause( Goal, Subgoal )
    execute( Subgoal, Message1 )
    diagnose( Goal, Subgoal, Message, Message1 )
diagnose( Goal, Subgoal, ok( Goal ), ok( Subgoal ) ) :-
    spec( Goal )
diagnose( Goal, Subgoal, incorrect( Goal :- Subgoal ), ok( Subgoal ) ) :-
    not spec( Goal )
diagnose( Goal, Subgoal, Message1, Message1 )
```

Figure 5: An algorithm for locating an incorrect clause

a goal, we first find a clause whose head can be unified with *Goal* and recursively solve the subgoals in the clause. If we can identified an error in the subgoals, we return the error message to the top level, using the third clause of *diagnose*. On the other hand, if all the subgoals return correct results, then we check if *Goal* is satisfied, by running specifications on the instantiated *Goal*. If the result is consistent with the specifications of *Goal*, then the clause is correct. The first clause of *diagnose* shows this result. If the computed *Goal* is not consistent with its specification, the second clause of *diagnose* will return an instance of the incorrect clause.

EXAMPLE 2. *Locating an incorrect clause.*

Consider the insertion sort program in Figure 6, adapted from Shapiro (1983), with specifications for each of its procedures shown in Figure 7. The specification for *isort* is the

```
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 6: An incorrect insertion sort

same as that for *sort* in section 4.2 (actually this definition can be used for any sorting routines). For *insert*, the specification means that $insert(X, Y, Z)$ 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. Note that we use the *if-then-else* symbol "$\rightarrow$" of Prolog in the specification of *insert* to have a precise definition. Without using "$\rightarrow$", the complete specification for *insert* would require the following two clauses:

$$\begin{array}{rcl} spec(isort(X,Y)) & :- & ordered(Y), \\ & & perm(X,Y) \\ spec(insert(X,Y,Z)) & :- & ordered(Y) \rightarrow \\ & & \quad ordered(Z), \\ & & \quad perm([X|Y],Z) \end{array}$$

Figure 7: Specifications for the insertion sort program

$$spec(insert(X,Y,Z)) \ :- \ ordered(Y),\ ordered(Z),\ perm([X|Y],Z)$$

and

$$spec(insert(X,Y,Z)) \ :- \ not \ \ ordered(Y).$$

We now run *isort* on input [2,1,3] (the user actually need not supply the input list [2,1,3], since it can be generated by running the specifications of *isort*, as shown in section 4.2. Here is the result (for the examples used in this paper, user input is shown in bold face and system generated output is shown in typewriter type font):

```
| ?- execute(isort([2,1,3],Answer), Message).
Error detected.   Debugging ...

The clause
    insert(1,[3],[3,1]):-3>1,insert(1,[],[1])
is false!

Answer = X
Message = [wrong_clause,insert(1,[3],[3,1]),(3>1,insert(1,[],[1]))]

yes
```

We found a false instance of the first clause of *insert*. The error was 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 that fails on the specification check. Note that the variable $Message$ is actually passed, in our debugging system, to the bug fixing routine which is discussed in section 8.

The computation tree in Figure 8 shows how the diagnostic procedure works on *isort* with input $[2,1,3]$. It traverses the computation tree in post-order and checks each procedure of its correctness. With reference to the tree, during the diagnostic process 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 "$\longleftarrow$" is the first node that contains results inconsistent with its specifications. Therefore, the interpreter returns this node along with its two sons (equivalent to an instantiated clause) as a counterexample.

5.2 Locating Incomplete Procedures

If $\boldsymbol{P}$ finitely fails (cf. Lloyd (1984)) on a procedure call $\boldsymbol{p(x', y)}$ with legal input $\boldsymbol{x'}$ and uninstantiated output $\boldsymbol{y}$ (i.e., the specification of $\boldsymbol{p(x', y)}$ is satisfiable), then $\boldsymbol{P}$

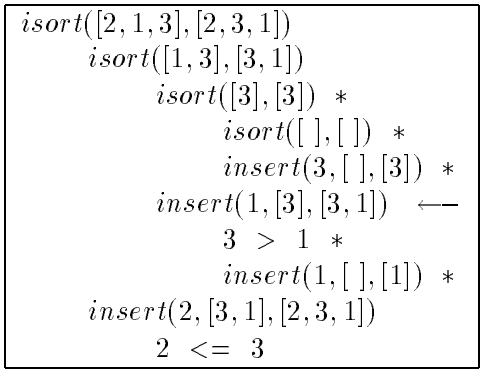

Figure 8: The computation tree for $isort([2,1,3],[2,3,1])$

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 $\boldsymbol{p}$ with input x' invokes no other procedures, then $\boldsymbol{p}$ is incomplete; if, on the other hand, $\boldsymbol{p}$ calls other procedures, then $\boldsymbol{p}$ or one of the procedures invoked after $\boldsymbol{p}$ must be incomplete. Accordingly, we trace the execution of $\boldsymbol{p}$. If a satisfiable call to a procedure $\boldsymbol{q}$ fails, while all procedures called by $\boldsymbol{q}$ return an answer whenever the call is satisfiable, then it is $\boldsymbol{q}$ that is deemed incomplete.

We summarize the above algorithm in Figure 9. In other words, the interpreter for

```
execute( Goal, Message ) :-
    clause( Goal, Subgoals )
    satisfiable( Subgoals )
    execute( Subgoals, Message )
execute( Goal, uncovered( Goal ) ) :-
    satisfiable( Goals )
```

Figure 9: An algorithm for locating incomplete procedures

locating an incomplete procedure can be built in a way that it first tries to establish a computation tree from the execution of the goal and recursively executes the new subgoals. When a satisfiable call *Goal* fails to find a clause that can complete the computation, one can be sure that *Goal* is not covered.

EXAMPLE 3. *Locating an incomplete procedure.*

Suppose we have an incomplete program as in Figure 10. With the same specifications in Figure 7, we try *isort* on [3,2,1]:

```
| ?- execute(isort([3,2,1],Answer), Message).
```

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

Figure 10: An incomplete insertion sort

```
Error detected.   Debugging ...

The goal
    insert(1,[],[1])
is not covered!

Answer = X
Message = [uncovered,insert(1,[],[1])]

yes
```

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).

The incomplete computation tree of *isort* on [3,2,1] is in Figure 11. In the computation

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

Figure 11: The computation tree for $isort([3,2,1], Answer)$

tree, $insert(1, [\,], X_2)$ is the first goal that cannot be unified with any clause in the program. The computation stops at this point because of the failure of this node.

5.3 Locating a Diverging Procedure

If $\boldsymbol{P}$ is partially correct, but nonterminating, then during the computation, some procedure $\boldsymbol{p}$ must be invoked repeatedly (however, there may be calls to other procedures in between the calls to $\boldsymbol{p}$), with the sequence of input values to $\boldsymbol{p}$ not decreasing in the specified well-founded ordering $\succ$ for $\boldsymbol{p}$. In the computation tree, a diverging computation corresponds to the infinite growth on one branch of the tree. This nonterminating computation can be

detected by tracing $\boldsymbol{P}$ and checking that each call is smaller with respect to $\succ$ than the previous one.

EXAMPLE 4. *Locating a diverging procedure.*

The program in Figure 12 contains a loop. Its well-founded ordering specifications are in Figure 13. The predicate wfo specifies the well-founded ordering for sequences of input

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

Figure 12: A looping insertion sort

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

Figure 13: Well-founded ordering for recursive procedures

values. For both *isort* and *insert*, the number of elements in the input list should decrease with each recursive call. As with the case for predicates *perm* and *ordered*, *shorter* can be defined in Prolog in a straightforward, declarative manner.

Running *isort* on [2,1,3], we have

```
| ?- execute(isort([2,1,3],Answer), Message).

Error detected.   Debugging ...

The goal  "insert(3,[1],X)"  in the clause
    insert(1,[3],X):-insert(1,[],[1]),insert(3,[1],X)
is looping!

Answer = Y
Message = [looping,insert(1,[3],X), (insert(1,[],[1]), insert(3,[1],X)),
                  insert(3,[1],X)]

yes
```

As can be seen from the infinite computation tree (Figure 14) for this goal, the second argument (i.e., [1]) of the goal $insert(3,[1],X)$ has the same length as the second argument

(i.e., [3]) of the head (i.e., $insert(1, [3], X)$) of the invoked clause. This clearly violates the relationship defined in $wfo(insert)$ which says that the length of lists should get shorter with each recursive call.

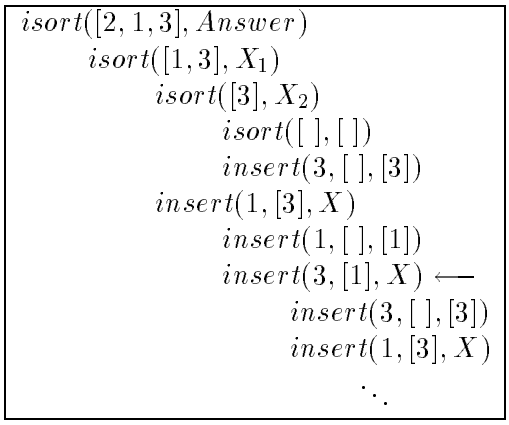

Figure 14: An infinite computation tree

5.4 A Meta Interpreter for Automatic Bug Location

Given the above analysis, 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 15. The 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. It first checks if the input variables violate the well-founded ordering defined in the specification of the procedure that covers the goal. If such is the case, we have an instance of a looping goal. If the input cannot 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 (i.e., no unifying clause or a subgoal fails in every unifying clause), then, since *Goal* is satisfiable according to the specifications, the program must be incomplete and we have an instance of an uncovered goal.

```
execute( (Goal1, Goal2), Message ) :-
    execute( Goal1, Msg_Goal1 ),
    if Msg_Goal1 = ok( Goal1 )
        then execute( Goal2, Message )
        else Message = Msg_Goal1
execute( Goal, ok(Goal) ) :-
    system( Goal ), Goal
execute( Goal, looping(Goal) ) :-
    not_decreasing( Goal )
execute( Goal, Message ) :-
    clause( Goal, Subgoals ),
    execute( Subgoals, Msg_Subgoal ),
    if Msg_Subgoal = ok( Subgoals )
        then if spec( Goal )
                then Message = ok( Goal )
                else Message = incorrect( (Goal :- Subgoals) )
        else Message = Msg_Subgoal
execute( Goal, uncovered(Goal) )
```

Figure 15: An automatic meta interpreter for bug location

6 Heuristic Bug Correction

Just as knowing that a program is incorrect does not mean that one knows where the bug is, knowing the location of a bug does not imply that one knows how to correct it. Although Myers (1979) has claimed that that bug correction is a much easier task than bug location, we believe that correcting a bug after it is identified is generally a more difficult task than locating the bug, especially when it is to be performed by a machine. This is because bug location only requires tracing the execution of procedures and checking the results of computation. Bug correction, on the other hand, requires reasoning with knowledge of the domain and intended algorithm, the semantics of the programming language and the input/output specifications.

In the automation process, it is intricate to formalize the complex knowledge involved in bug correction and represent it in a form that can be utilized by the debugger. Some automatic debugging system (e.g., Murray (1986)) uses the stored information in their system's knowledge base for bug correction by matching (maybe partially) and replacing the buggy program with the established code fragments. In our case, we have only the knowledge contained in the specifications of the individual procedures and the operational semantics of pure Prolog. In addition, we have devised some heuristics—based on a classification of Prolog bugs—that suggest a possible cause for the error. Deductive or inductive corrective measures or both are then employed in an attempt to bring the program in line with the given specifications.

6.1 Fixing an Incorrect Clause

A clause

$$p(x, y) \ :- \ p_1, ..., p_n$$

is incorrect if there is an instance of that clause, say,

$$p(x', y') \ :- \ p_1', ..., p_n'$$

such that all the p_i''s are true (i.e., 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 goal $p(x', y'')$ will help guide the debugger.

If the *solved* goal $p(x', y'')$ is covered by another clause in the program (i.e., 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 presumably 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 right hand side of the clause implies the left hand side. If the proof fails because of some missing conditions, we can add them as subgoals to the clause (detail below). Alternatively, we can use the offending clause as a starting point for an inductive synthesis of a correct clause (see below). In the worst case, we can always add the subgoal **fail** 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) correct. We will discuss below how to deal with any 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 (viz. if $x = x'$ then $y = y''$) 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.

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 $p(x', y'')$ (i.e., 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. We rearrange, replace, delete, or add new variables within subgoals until the original incorrect clause computes $p(x', y'')$ correctly. The induction method that we use to correct incorrect subgoals is a modification of the refinement method in Shapiro (1983).

The last possibility is that $\boldsymbol{p(x', y'')}$ cannot 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 (i.e., weaken) the clause head so that it is unifiable with $\boldsymbol{p(x', y'')}$. The methods described above can then be used to fix any incorrect subgoals.

We summarize the strategies for correcting an incorrect clause as the following heuristic rules:

- **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.
- **If** the solved goal can be unified with the head of the incorrect clause **but** is not covered by any clause in the program, **then** fix the subgoals that fail for the correct answer and continue debugging the clause.
- **If** the solved goal cannot 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. *Fixing an incorrect clause.*

We demonstrate this process with insertion sort program in Figure 16.

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

Figure 16: An incorrect insertion sort

We now test the program on input list [0,1]:

```
| ?- debug(isort([0,1],Answer)).

Error detected.   Debugging ...

The clause
    insert(0,[1],[1,0]):-insert(0,[],[0])
is false!

The goal "insert(0,[1],[0,1])" is covered
There are missing subgoals in the clause:
     insert(X,[Y|Z],[Y|V]):-insert(X,Z,V)
Retract erroneous clause:
```

```
     insert(X,[Y|Z],[Y|V]):-insert(X,Z,V)

Generating missing subgoals ...
Assert clause:
    insert(X,[Y|Z],[Y|V]):-Y<X,insert(X,Z,V)
```

The debugger detected an incorrect clause in procedure *insert* when trying to solve the goal $isort([0,1], Answer)$. After some analysis, it determined that the clause

$$insert(X,[Y|Z],[Y|V]) \ :- \ insert(X,Z,V)$$

is false for $X = 0,\ Y = 1,\ Z = [\,],\ V = [0]$ (note that 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,

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

in the program. The debugger then tried to deduce a missing subgoal by constructing a proof. It tried to prove that $insert(X,Z,V)$ implies $insert(X,[Y|Z],[Y|V])$, and concluded that, by adding $Y < X$ to the right-hand side of the clause, the implication will hold. Therefore, the debugger removed the incorrect clause and asserted the synthesized clause to the program. This proof process requires the theorem prover for Horn clauses described in section 6.4.

6.2 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 on a copy. We locate the subgoal that causes this clause to fail and either try to fix it inductively (by rearranging, replacing, deleting, or adding variable within the subgoal) 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 an inductive method, using the specifications to guide the search. 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:

- **If** the uncovered goal can be unified with the head of a clause, **then** duplicate the clause, and locate and fix its unsatisfiable subgoals.
- **If** the uncovered goal cannot be unified with the head of a clause, **then** use the specifications for that goal to synthesize a new clause.

6.3 Fixing a Looping Procedure

When the input to a procedure call violates the well-founded ordering defined 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 inductive method as for fixing incorrect subgoals. Alternatively, we can weaken it 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.

6.4 Deducing Missing Subgoals

According to the deductive semantics of Prolog, the right hand side (the body) of a clause should imply the left hand side (the head). Therefore, in proving the correctness of a correct clause, the implication should be found to hold. On the other hand, trying to prove the implication for an incorrect clause must result in failure. The basic idea is to try to prove the head of the clause, given the subgoals in the body as hypotheses, and in the process identify and derive those sufficient conditions that will allow a proof to go through. (Unlike some method such as that in Katz & Manna (1975), a correct clause would never be "debugged"; only a clause found faulty by testing is subjected to formal verification.)

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 precondition provides any additional conditions under which a goal can be proved from a hypothesis. We adopted and modified this method and constructed a theorem prover for Horn clauses.

The deductive proof proceeds by reducing both sides of the clause to simpler forms, by replacing each goal (or subgoal) with its definitions or with something that implies it, and each hypothesis with its definition or something that it implies, until a termination condition is met.

It employs the following rules which, for the most part, are modifications of typical rules for deductive proof (cf. Loveland (1978)). 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 "$lhs \Rightarrow rhs$"

for "given lhs (left hand side), it is sufficient to prove rhs (right hand side)".

Rule 1. $H \rightarrow G_1 \wedge G_2 \quad \Rightarrow \quad (H \rightarrow G_1) \wedge (H \rightarrow G_2)$
Reduction of a conjunctive goal: To prove a conjunctive goal, prove each conjunct separately.

Rule 2. $H \rightarrow G_1 \vee G_2 \quad \Rightarrow \quad (H \rightarrow G_1) \vee (H \rightarrow G_2)$
Reduction of a disjunctive goal: To prove a disjunctive goal, prove one of the disjuncts.

Rule 3. $(H_1 \vee H_2) \rightarrow G \quad \Rightarrow \quad (H_1 \rightarrow G) \wedge (H_2 \rightarrow G)$
Reduction of a disjunctive hypothesis: To prove a goal with disjunctive hypotheses, one can prove that the goal can be proved from each disjunct.

Rule 4. $H \rightarrow (G_1 \rightarrow G_2) \quad \Rightarrow \quad (H \wedge G_1) \rightarrow G_2$
Reduction of an implicative goal: To prove a goal which is an implication itself, include the precondition of the implication as part of the hypothesis and prove the postcondition of the implication.

Rule 5. $(H_1 \rightarrow H_2) \rightarrow G \quad \Rightarrow \quad (\neg H_1 \rightarrow G) \wedge (H_2 \rightarrow G)$
Reduction of an implicative hypothesis: To prove a goal with an implicative hypothesis, first prove the goal with the negation of the precondition of the implication, then prove the goal with the postcondition of the implication.

Rule 6. $\neg H \rightarrow \neg G \quad \Rightarrow \quad G \rightarrow H$
Contraposition: If both the hypothesis and the goal are in negation form, then one can drop both negations and reverse the hypothesis and the goal for the proof.

Rule 7. $\neg H_1 \wedge H_2 \rightarrow \neg G \quad \Rightarrow \quad G \wedge H_2 \rightarrow H_1$
Generalized contraposition: If the goal and one part of the hypothesis are in negation form, then the proof can be established if one can show that the negation part of the hypothesis can be derived from the negation of the goal and the non-negation part of the hypothesis combined.

In addition to these proof rules, there are three ways of reducing a goal or subgoal. First, we can replace the goal with its definition as described in the goal's specification. This is substitution of equivalent terms:

$$H \rightarrow G \quad \Rightarrow \quad H \rightarrow G', \quad \textit{if} \quad G = G'.$$

It is obvious that, if one substitutes the goal with equal terms, the proof condition will remain the same. Second, if there is a correct program clause whose head matches the goal, we can replace the goal with the subgoals in that clause. Note that this is just like the goal reduction in normal Prolog computation. It can also be regarded as the application of implicative terms:

$$H \rightarrow G \quad \Rightarrow \quad H \rightarrow G', \quad \textit{if} \quad G \rightarrow G'.$$

Third, if a specific domain fact is known, it can be used to weaken a goal or replace it with something equivalent (e.g., replacing a list with one of its permutations when the order does not affect the truth value of the predicate). This is an effort to build into the debugger a knowledge handling capability such that it can have some common sense when reasoning about programs. Similar methods also apply to hypothesis reduction.

The proof process terminates when one of the following conditions is met: (1) the original goal is reduced to **true**, in which case the clause is proved correct; (2) the original set of hypothesis is reduced to **false**, meaning that there are conflicting subgoals in the clause, and that the clause is vacuously correct; (3) the goal is reduced to a subset of the hypotheses, in which case the implication is also established; and (4) 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. If the proof ends in condition (4), then we have identified those missing subgoals that will make the clause correct.

A logical simplifier (cf. Waldinger & Levitt (1974)) is built to aid goal reduction. It is invoked after each reduction step and performs tasks such as removing nested conjunctions, duplicate goals, and tautologies (i.e., the goal **true**). It also simplifies the goal structures according to the logical rules governing **and**, **or**, **not**, and **implication**. For example, if a conjunctive goal contains the subgoal **false**, then the whole goal will be reduced to **false**.

6.5 Fixing Incorrect Subgoals

Once we identify an incorrect subgoal, we can correct it using either a heuristic rule or an inductive method, besides using the deductive methods outlined in the previous sections.

We have developed heuristics that are meant to correct an incorrect subgoal quickly when a certain pattern of subgoals is 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 this kind of heuristic rules is to attempt to fix some commonly made, yet easily corrected, errors.

When our heuristic rules cannot correct the errors in a subgoal, a general inductive strategy will be employed with the hope of fixing the bugs. This is done by applying some refinement operations on terms within the subgoals. For example, we can try to unify two free variables, or unify a compound term with variables appearing elsewhere in the same clause.

It should be noted that all heuristic fixes will be tested immediately after the changes are made; and if the fixes cannot correct the errors, all the changes will be undone.

7 Automated Program Synthesis

A major use of software specifications is to provide a very high level descriptive tool so one can build a large system in top-down fashion. If the specification truly embodies what one needs, then one should be able to provide that abstract specification as input to an automatic programming system and be able to receive, as a result, a low level program that can be executed on the target machine more efficiently. Ideally, this practice would salvage much of the grievance in current software development processes. Given the current state of technology, however, such an automatic programming system is still remote for general software production.

Nontheless, by restricting the problem domain of such a system, it is possible to apply such technology and build systems for practical applications. The system described in (Barstow *et al.* 1982) deals with a class of numerical software for scientific processing and has allowed the client scientists both greater flexibility in their ability to specify program behavior and much more rapid program development to establish the validity of that behavior.

The Model Inference System (Shapiro (1983)) can generate Prolog programs from examples. We now show how executable specifications can be incorporated into the this system. The major benefit of using specifications is to replace the oracle, usually played by the user, and, therefore, automate the synthesis process. We first modify the querying process of the diagnosis routine in a way that whenever the user is to be queried to confirm or supply a computation result, the system instead executes the specifications for an answer. We also need to add procedures for goal generation and goal rechecking.

EXAMPLE 6. *Program synthesis using executable specifications*

This example shows the synthesis of an insertion sort program. With all the modifications discussed above, the synthesis proceeds with very little user involvement. In fact, the user only needs to type in the initial request to start the system, and answer "yes" or "no" when the system prompts for instruction on whether to continue with the generation of new goals. This example starts with an empty *isort* program and the specifications of *isort* in Figure 7 is given to the system. The process 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.

2. Test goal generated: $isort([x], Y)$
   Error: None

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

4. Test goal generated: $isort([1,0],X)$

   - 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.
   - Error: missing solution $isort([\,],[\,])$
     Diagnosis: $isort([\,],[\,])$ is uncovered
     Action: add clause $isort(X,[\,]) : -true$
   - Error: missing solution $isort([x],[x])$
     Diagnosis: $isort([x],[x])$ is uncovered
     Action: add clause $isort([X|Y],Z) : -insert(X,Y,Z)$
   - Error: missing solution $isort([x],[x])$
     Diagnosis: $insert(x,[\,],[x])$ is uncovered
     Action: add clause $insert(X,Y,[X|Y]) : -true$
   - Error: missing solution $isort([1,0],[0,1])$
     Diagnosis: $insert(1,[0],[0,1])$ is uncovered
     Action: add clause $insert(X,[Y|Z],[Y,X|Z]) : -true$
   - 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.
   - 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.
   - 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 known *facts* to the system. (A *fact* is defined as 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, so is $< isort([1,0],[1,0]),\ false >$. A correct program should succeed on a *true fact*, and fail on a *false fact*.) The system proceeds to check if the program satisfies the goals generated so far.

- Error: wrong solution $isort([x],[\,])$
  Diagnosis: $isort([x],[\,]) : -trueisfalse$
  Action: remove clause $isort(X,[\,]) : -true$
  Note: Removing a clause usually causes problems. The system has to recheck all the facts and goals.
- Error: missing solution $isort([\,],[\,])$
  Diagnosis: $isort([\,],[\,])$ is uncovered
  Action: add clause $isort([\,],[\,]) : -true$
- 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 ...
- 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.
- Error: wrong solution $isort([0,1],[0])$
  Diagnosis: $insert(0,[1],[0]) : -true$ is false
  Action: remove clause $insert(X,Y,[X]) : -true$
- 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.

  Also, all the operations within this step are done with NO user involvement.

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

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

   - 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 $isort([X|Y],Z) : -insert(X,Y,Z)$
   - 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

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

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

   - 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 $insert(X,[Y|Z],[Y,X|Z]) : -Y < X$
   - 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 which is shown in Figure 17.

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

Figure 17: A synthesized program of insertion sort

8 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 structure of this interpreter is described in Figure 18 in pseudo-Prolog code. Upon receiving a goal, the interpreter first examines the input variables. If the input is symbolic, then by executing the specifications of the procedure, the interpreter will generate test cases. If the input variables are instantiated, then running the specifications 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 skolemized input. (Skolemization forces the program to find one symbolic output for all inputs with the same given structure.) If execution completes successfully, the interpreter returns

```
interpret( Goal(Input,Output) ) :-
        spec( Goal(Input,Output) ),
        skolemize( Input,Skolem ),
        execute( Goal(Skolem,Output), Message ),
        fix_bug( Message )
```

Figure 18: The Constructive Interpreter

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 ever the execution fails, i.e., if the program contains an incorrect, incomplete, or nonterminating procedure, then the interpreter will locate a bug and return a diagnostic message. Bug-fixing routines will then be invoked to correct the bug that have been identified and located.

The procedure *execute* does goal reduction and bug location, and has been discussed in section 5.4. The procedure $fix\_bug(Message)$ implements the bug correction heuristics discussed in sections 6.1 through 6.3.

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 specifications 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 three different kind of errors respectively. In addition to performing error analysis and suggesting fixes, they all have access to the deductive theorem prover and inductive 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 7. *Debugging a Quicksort program.*

We show an annotated script of the *Constructive Interpreter* debugging the Quicksort program in Figure 19, with the specifications in Figure 20. The specifications say that $qsort(X,Y)$ holds if Y is sorted and Y is a permutation of X, that $part(L,E,X,Y)$

```
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 19: A buggy Quicksort program

```
spec(qsort(X, Y))               :- ordered(Y),
                                   perm(X, Y)
spec(part(L, E, X, Y))          :- rm_list(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 20: Specifications for the Quicksort program

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 all the elements in X and smaller then all the elements in Y, and that $append(X, Y, Z)$ is true if Z is the combination of lists X and Y, in their original order. The predicate wfo specifies the well-founded ordering for sequences of input values. For all procedures *qsort*, *part*, and *append* the number of elements in the input list should decrease with each recursive call. As is also the case for the insertion sort program, the predicates *perm*, *ordered*, *rm_list*, *gt_all*, *lt_all*, and *shorter* can be defined as usual Prolog procedures. (These procedures should be regarded as standard building blocks for specification, available in the debugger's library, since they all apply across a whole gamut of specific programs. For example, *lt_all* would play a role in virtually all sorting and most searching programs and *rm_list* in practically all programs with destructive list manipulation.)

We now show how the *Constructive Interpreter* analyzes the above insertion sort program. The top level command is *apd* (for *a*utomated *p*rogram *d*ebugger); it prompts with an asterisk. User input is shown in boldface.

Invoking the debugger, we proceed as follows:

```
| ?- apd.
* qsort(U,V).

Solving goal: qsort([],X) ...

Error detected.   Debugging ...

The goal
    qsort([],[])
is not covered!
```

Since **qsort(U,V)** is symbolic, the debugger first generated a test case $qsort([\,], X)$ and tried to satisfy it. It discovered that $qsort([\,], X)$ should have a solution $qsort([\,],[\,])$ according to the specification of *qsort*, but cannot get it from the program we supplied. The debugger therefore reported a bug and tried to fix it.

```
Synthesizing a clause to cover qsort([],[]) ...
Assert clause:
    qsort([],[]) :- true

Listing of qsort(X,Y):
    qsort([],[]) :- true.
    qsort([X|Y],Z) :- part(Y,X,W,X1), qsort(W,Z1), qsort(X1,V1), append([X|Z1],V1,Z).
```

Since no clause head in the original program unified with $qsort([\,],[\,])$, the debugger used the specification for *qsort* and synthesized the clause

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

to cover that goal. Since the body of this clause can be reduced to *true*, the debugger added a unit clause to the program (by asserting it to the database). The goal $qsort([\,],[\,])$ is now satisfiable. Since we initially supplied a symbolic input, we now try for another test case:

```
* Try another test case?  y.

Solving goal: qsort([x],X) ...

Error detected.   Debugging ...

The clause
    part([],x,[x],[]) :- true
is false!
```

The debugger now generated a one element list as test input: $qsort([x], X)$. (Note that the input generated, $[x]$, contains a skolem constant x.) This time, it found an incorrect clause in the procedure *part*, because partitioning an empty list should result in two empty sublist, so the result of $parti([\,], x, X, Y)$ should be $part([\,], x, [\,], [\,])$ instead of $part([\,], x, [x], [\,])$. After further analysis, the debugger concludes that:

```
The head of the clause
    part([],X,[X],[]) :- true
is incorrect.   Fixing ...

Cannot fix clause head!
Retract clause:
    part([],X,[X],[]) :- true
Synthesizing a clause to cover part([],x,[],[]) ...
Assert clause:
    part([],X,[],[]) :- true

Listing of part(X,Y,Z,U):
    part([],X,[],[]) :- true.
    part([X|Y],Z,U,[X|W]) :- part(Y,Z,U,W).
    part([X|Y],Z,[X|V],W) :- X <= Z, part(Y,Z,V,W).
```

Since the unit clause in the procedure *part* was incorrect, and the debugger could not fix the head, it retracted the clause. After synthesizing a clause that covers $part([\,], x, [\,], [\,])$, the debugger reexecuted 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.)

```
Checking previous goal qsort([],X) ...
Found solution: qsort([],[])

Checking previous goal qsort([x],X) ...
Found solution: qsort([x],[x])
```

Since every goal generated so far can be satisfied, the debugger prompts the user:

```
* Try another test case?  y.

Solving goal: qsort([0,1],X) ...
Found solution: qsort([0,1],[0,1])
```

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.

```
* Try another test case?  y.

Solving goal: qsort([1,0],X) ...

Error detected.   Debugging ...

The clause
    part([0],1,[],[0]) :- part([],1,[],[])
is false!
```

The next test goal $qsort([1,0], X)$ resulted in the location of an incorrect clause in the procedure *part*. A trace of the procedures shows that the correct solution to $part([0], 1, X, Y)$, *viz.* $part([0], 1, [0], [\,])$, can be obtained from the other clause of *part*. Thus, this incorrect clause should have failed, but did not because of a missing subgoal. Our debugger is able to deduce this missing subgoal:

```
There are missing subgoals in the clause:
    part([X|Y],Z,U,[X|W]) :- part(Y,Z,U,W)
Retract erroneous clause:
    part([X|Y],Z,U,[X|W]) :- part(Y,Z,U,W)

Generating missing subgoals ...

Assert clause:
    part([X|Y],Z,U,[X|W]) :- Z <= X, part(Y,Z,U,W)
Listing of part(X,Y,Z,U):
    part([X|Y],Z,U,[X|W]) :- Z <= X, part(Y,Z,U,W).
    part([],X,[],[]) :- true.
    part([X|Y],Z,[X|V],W) :- X <= Z, part(Y,Z,V,W).
```

After correcting for the missing subgoal (by retracting an incorrect clause and asserting a correct one), the debugger reexecuted all the test goals again.

```
Checking previous goal qsort([],X) ...
Found solution: qsort([],[])

Checking previous goal qsort([x],X) ...
Found solution: qsort([x],[x])

Checking previous goal qsort([0,1],X) ...
Found solution: qsort([0,1],[0,1])

Checking previous goal qsort([1,0],X) ...

Error detected.   Debugging ...

The clause
    qsort([1,0],[1,0]) :- part([0],1,[0],[]), qsort([0],[0]),
        qsort([],[]), append([1,0],[],[1,0])
is false!
```

As shown above, the debugger caught another bug when trying to resatisfy the current test goal. Further diagnosis narrows down the bug's location:

```
The clause
    qsort([X|Y],Z) :- part(Y,X,W,X1), qsort(W,Z1), qsort(X1,V1), append([X|Z1],V1,Z)
contains incorrect subgoals.   Fixing ...

Subgoal "append([X|Y],Z,U)" in clause
    qsort([X|W],U) :- part(W,X,U1,V1), qsort(U1,Y), qsort(V1,Z), append([X|Y],Z,U)
is incorrect

Trying a local fix ...

Retract clause:
    qsort([X|Y],Z) :- part(Y,X,W,X1), qsort(W,Z1), qsort(X1,V1), append([X|Z1],V1,Z)

Assert clause:
    qsort([X|Y],Z) :- part(Y,X,W,X1), qsort(W,Z1), qsort(X1,V1), append(Z1,[X|V1],Z)


Listing of qsort(X,Y):
```

```
qsort([X|Y],Z) :- part(Y,X,W,X1), qsort(W,Z1), qsort(X1,V1), append(Z1,[X|V1],Z).
qsort([],[]) :- true.
```

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. We would be led to believe, in this case, that the program is correct with respect to its specifications. (Formal verification of its correctness would require greater theorem proving capabilities.)

EXAMPLE 8. *Debugging a merge sort program.*

We now demonstrate how the debugger deals with a looping error in the merge sort program in Figure 21. Figure 22 is the specifications.

$$
\begin{array}{rcl}
msort([\,],[\,]) & & \\
msort(X,Z) & :- & length(X,L), \\
 & & L1 \ \ is \ \ L//2, \\
 & & break(X,L1,X1,X2), \\
 & & msort(X1,Z1), \\
 & & msort(X2,Z2), \\
 & & merge(Z1,Z2,Z) \\
break(X,0,[\,],X) & & \\
break([A|X],L,[A|Y],Z) & :- & L1 \ \ is \ \ L-1, \\
 & & break(X,L1,Y,Z) \\
merge([\,],X,X) & & \\
merge(X,[\,],X) & & \\
merge([A|X],[B|Y],[A|Z]) & :- & A \ \leq \ B, \\
 & & merge(X,[B|Y],Z) \\
merge([A|X],[B|Y],[B|Z]) & :- & A \ > \ B, \\
 & & merge([A|X],Y,Z)
\end{array}
$$

Figure 21: A buggy merge sort program

Just as 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 merge sorts the sublists, then merges the sorted halves. Note that the predicates used in the above specifications are the same ones used in the specifications for Quicksort.

The following is a debugging script:

```
| ?- apd.
* msort(U,V).

Solving goal: msort([],X) ...
Found solution: msort([],[])
```

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

Figure 22: Specifications for the merge sort program

The program has no problem solving the empty list. However, it gets into trouble quickly with the one-element list.

```
* Try another test case?  y.

Solving goal: msort([x],X) ...

Error detected.   Debugging ...

The goal  "msort([x],Y)"  in the clause
    msort([x],X) :- length([x],1), 0 is 1//2, break([x],0,[],[x]),
        msort([],[]), msort([x],Y), merge([],Y,X)
is looping!
```

The debugger found that the procedure call $msort([x],Y)$ to the second clause of $msort$ violated the well-founded ordering defined for that recursive procedure.

```
Adding subgoals for well-founded ordering in clause:
    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)

Listing of msort(X,Y):
    msort([],[]):-true.
    msort(X,Y) :- shorter(U1,X), length(X,U), V is U//2,
        break(X,V,Z1,U1), msort(Z1,W1), msort(U1,Y2), merge(W1,Y2,Y)

Continue debugging ...
```

Not being able to fix the offending subgoal directly, the debugger added the well-founded condition as a subgoal to the clause and continues with the analysis.

```
Error detected.   Debugging ...

The goal
    msort([x],[x])
is not covered!
```

```
Synthesizing a clause to cover msort([x],[x]) ...
Assert clause:
    msort([X],[X]) :- true

Listing of msort(X,Y):
    msort([X],[X]) :- true.
    msort([],[]) :- true.
    msort(X,Y) :- shorter(U1,X), length(X,U), V is U//2,
        break(X,V,Z1,U1), msort(Z1,W1), msort(U1,Y2), merge(W1,Y2,Y)
```

After enforcing the well-founded ordering on recursive calls, the debugger detected that the program is incomplete, since the goal $msort([x],[x])$ is not covered. Therefore, it invoked the synthesis routine and found a clause for the goal.

```
Restoring original clause ...
Listing of msort(X,Y):
    msort([X],[X]) :- true.
    msort([],[]) :- true.
    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)
```

Since the error was due to a missing case, the debugger restored the original, correct clause.

It turned out that the looping bug was due to the behavior of procedure *break*. A one-element list to the procedure is always broken into sublists of zero- and one-element. This one-element list is never reduced in the recursive call, and, therefore, need to be treated as a special case. Adding a unit clause for it resolve the problem completely.

Note that, in running the debugger, the user only needs to supply top level goals (in our examples, **qsort(U,V)** and **msort(U,V)**), and types in a *yes* answer for the debugger to continue debugging with alternative test cases. 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 minimal level.

9 CONCLUSION

In this research, we have explored a distinctive feature of logic programming: using logic for both specification and computation. We have shown that user-supplied program specifications can be utilized in many different ways.

We use executable input/output specifications to define the intended behavior of a program and to generate test cases for bug discovery. We employ the execution mechanism of a Prolog machine to locate bugs, using specifications to validate computation results. We also have heuristics to analyze bugs and suggest fixes, and use techniques in deductive

theorem proving and inductive synthesis to mechanize the bug correction process, also with the help of specifications.

With the target language being pure Prolog, we have formulated a computer model to encode the knowledge necessary for automating the debugging process. It includes a classification scheme of program bugs, heuristics that analyze and repair program errors, operational semantics of the language, intended behavior of a program, and deductive and inductive inference strategies to reason with programs and their specifications.

The realization of our methodology is the *Constructive Interpreter*. It contains three major components: test case generator, bug locator, and bug corrector. When supplied with a program and its executable specifications, the test case generator can generate test data systematically by executing specifications. 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 that is 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 might involve the use of (1) a deductive theorem prover which will try to construct a proof and deduce sufficient conditions to amend the program, and (2) an inductive program generator which will synthesize the missing part of the program.

The *Constructive Interpreter* performs much as a active human expert does during a typical debugging session. When given a program and its specifications, it can (1) execute a goal as a regular interpreter does, (2) generate test cases systematically when symbolic input data are supplied, (3) verify the results of a computation, (4) trace the execution of the program, and (5) locate and fix a bug when a goal does not compute correctly.

The traditional testing approach is only concerned with designing test cases that might show a program to be incorrect; it does not deal directly with the problem of locating and correcting bugs. Since knowing that a program is incorrect does not imply knowing the cause, research in testing provides, at most, methods to disclose the existence of bugs in a program. Our methodology, on the other hand, is intended to combine the functions of testing and debugging, for logic programs, under one uniform framework.

Deduction and Induction are two different inference mechanisms. Although they seem to be opposite of each other, we have shown that they can complement each other. Logical deduction is a powerful technique in the sense that the result from deductive inference is guaranteed correct (or consistent with the axioms). In the context of logic programming, deduction can be used to execute, derive, transform, or verify programs. We have applied this procedure to check the inconsistency between a program and its specifications. Inductive inference is employed to generate programs whenever incompleteness is identified.

In conclusion, we have demonstrated that, in the realm of logic programming, the tedious problem of program debugging and synthesis is perhaps amenable to automation.

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. of the Association for Computing Machinery*, 29:841–862.

Barstow, D., Duffey, R., Smoliar, S., Vestal, S. (1982). An automatic programming system to support an experimental science. In *Sixth International Conference on Software Engineering*, pages 360–366.

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.* Springer-Verlag, New York, third edition.

Darlington, J., Field, A. J., Pull, H. (1986). The unification of functional and logic languages. In DeGroot, D., Lindstrom, G., editors, *Logic Programming: Functions, Relations, and Equations*, pages 37–70. Prentice-Hall, Englewood Cliffs, NJ.

Dershowitz, N., Jouannaud, J.-P. (1990). Rewrite systems. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North-Holland, Amsterdam.

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

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

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 Transactions on Software Engineering*, SE-2(3):195–207.

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

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

Hogger, C. J. (1981). Derivation of logic programs. *J. of the Association for Computing Machinery*, 28(2):372–392.

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

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

Josephson, N. A., Dershowitz, N. (1986). An implementation of narrowing: The RITE way. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 187–197, Salt Lake City, UT.

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

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

Kowalski, R. A., van Emden, M. H. (1976). The semantics of predicate logic as a programming language. *J. of the Association for Computing Machinery*, 23:733–742.

Kowalski, R. A. (1974). Predicate logic as programming language. In *Proceedings 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, editors, *Mathematical Logic and Programming Languages*. Prentice/Hall International, Englewood Cliffs, NJ.

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

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

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

Loveland, D. W. (1978). *Automated Theorem Proving: A Logical Basis*. North-Holland, New York.

Manna, Z., Waldinger, R. J. (1980). A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121.

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*. Wiley, New York.

Pereira, L. M. (1986). Rational debugging in logic programming. In *Proceedings 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 *Proceedings of the Second International Logic Programming Conference*, pages 151–157, Uppsala, Sweden.

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

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. of the Association for Computing Machinery*, 12(1):23–41.

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

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

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

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

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