Control-Flow Based Testing of Prolog Programs

Gang Luo* , Gregor v. Bochmann*, Behcet Sarikaya** and Michel Boyer*

* Departement d'IRO, Universite de Montreal,
C.P. 6128,Succ.A, Montreal, P.Q., H3C 3J7, Canada
email:luo@iro.umontreal.ca

** Dept. of Computer and Information Sciences,
the Bilkent University, Ankara, Turkey, 06533
email:sarikaya@trbilun.bitnet.

Abstract

We present in this paper test selection criteria for Prolog programs which are based on control flow. The control flow in Prolog programs is not obvious because of the declarative nature of Prolog. We present two types of control flow graphs to represent the hidden control flow of Prolog programs explicitly. A fault model is developed for Prolog programs for guidance on test selection. Test selection criteria are given in terms of the coverage on these control flow graphs. Under the given fault model, the effectiveness of these criteria is analyzed in terms of fault detection capability of the test cases produced with these criteria.

1. Introduction

A lot of research has been reported about program testing for conventional procedure-oriented programming languages [8, 9, 10, 13, 16, 17, 22], but little has been said about program testing of logic programming languages, such as Prolog. Several Prolog-related issues in the area of software quality assurance have been studied, such as Prolog program debugging [20, 19, 12], recursive program termination checking [21, 18], and the detection of data type anomaly [1]. In particular, the issue of generating test data from Prolog-like specification has been investigated in [2, 3, 6, 7, 5]. Those articles, however, address only the case where logic programs are used as specifications, that is, specification-based testing, but they do not address testing logic programs as implementation. The issue of testing a logic program as an implementation has received very little attention. Because of the wide use of Prolog, this issue seems important.

There exist a few differences between specification-based and implementation-based test selections. With specification-based test selection, one can generate both a set of test data and the expected results from a specification. With implementation-based test selection, one can only generate a set of test data from an implementation, but cannot obtain the expected results from the implementation. In this case, one usually assumes that there exist oracles in the human mind, and people are responsible for checking the test results against the oracles. In this paper, we mainly concentrate upon implementation-based test selection.

Prolog programs are different from conventional procedure-oriented programs in the following aspects. (1) The basic data structures in Prolog are recursive lists, which are less frequently used within conventional programming languages. (2) The unifications of subgoals in Prolog can proceed in two directions: (2a) control flow is two ways (continuing after success and backtracking after failure), (2b) data flow is two ways also (passing an input value and returning an output value for the same variable); whereas there is only one direction in the control flow of conventional programs (see Figure 1 for the comparison on the control flows). (3) The designs of predicates in Prolog are heavily recursion-oriented. (4) There are no type declarations in Prolog. It is these differences that require more specific studies for Prolog program testing.

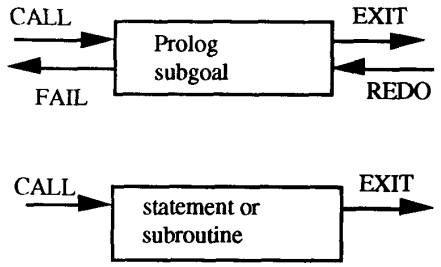


Figure 1: The distinction between a subgoal and a statement or subroutine

Some test selection criteria for Prolog programs were proposed in [14] based on the control flow and functional aspects. However, little has been analyzed to convince readers of the effectiveness of the approach, and in fact the quality of the approach has not been assured. Most Prolog specification-based testing methods [2, 3, 6, 5]

0-8186-2975-4/92 $03.00 © 1992 IEEE

mainly concentrate on generating test cases automatically, by taking advantage of the declarative nature of Prolog.

We first present in Section 2 a fault model for Prolog programs. A fault model consists of a set of fault types; a fault type is a set of faults; and a fault is a textual problem with programs. The fault model serves as a guide to developing test selection criteria. The fault model is defined with respect to the syntactic structure of Prolog Programs.

Guided by the given fault model, we give several test selection criteria in terms of control-flow coverage. Although control-flow-based testing is not a new idea in program testing, control flows in Prolog programs are hidden because of the declarative nature of Prolog. We first introduce in Section 3 two graphs which we call *P-flowgraph* and *reduced global P-flowgraph* to represent the hidden control flow structure in Prolog programs. We then give some test selection criteria based on the two graphs. The soundness of the above test selection criteria is analyzed under the fault model .

Furthermore, we investigate in the Section 4 an instrumentation method and tool which facilitates the generation of test data. We conclude in Section 5 by discussing possible future work.

2. Fault model for prolog programs

We give in this section a fault model for Prolog programs, which serves as a guide to developing test selection criteria, and a quality measure of determining when one test (or testing strategy) is better than another. [16] presented a software quality measure in terms of the detection of prescribed faults, which serves as the basis of his fault-based testing strategy. Testing is fault-based when it seeks to demonstrate that prescribed faults are not in a program [16]. Furthermore, a "fault" is a textual problem with the code, resulting from a mental mistake by a programmer or designer, and the mental mistake is defined as an "error" [11]. We use a fault model to classify the prescribed faults into a set of fault types. Each fault type represents a subset of prescribed faults. The fault types are defined with respect to the syntactic structure of programs.

We assume in this paper that the computation orders in Prolog are fixed. Computations are conducted from left to right within a rule, from the first to the last rule within a predicate. For implementation-based testing, we assume that no written formal specification is available, and that a specification in the human mind serves as an oracle. Furthermore, we only consider the faults which cannot be detected easily by an ordinary compiler. Hence, we consider the following fault types in the implementations:

(1) missing or extra cut,
(2) missing or extra rule,
(3) missing or extra predicate in a rule,
(4) wrong order of called predicates in a rule,
(5) wrong order of rules in a predicate,
(6) missing or extra pair of "[" and "]" for a list,
(7) wrong intermediate variable name in a rule,
(8) wrong replacement between a variable and a value.

Each of the above fault types represents a set of faults. For example, let R1 and R2 be two rules in a predicate. Therefore, "missing a cut in R1" and "missing a cut in R2" are two faults in the above fault type (1). The ordinary Prolog compiler may leave the faults in the above fault types undetected. Most of the fault types which are not in the above list can be modeled as multiple faults from the above lists.

Of these fault types, (6), (7) and (8) are much more difficult to detect than the first five, as discussed in Section 3. Since there exist no methods to ensure the absence of fault types (6), (7) and (8), the best we can do is to find as many faults in these fault types as possible.

3. Test selection based on prolog control-flow

Control-flow oriented testing is not a new idea in the area of software testing; the statement coverage, branch coverage and path coverage for conventional program testing [17] belong to this category. On the other hand, the control flows in Prolog programs are not so obvious as in conventional programs because the control flows in Prolog are hidden. These hidden control flows result from the declarative nature of Prolog which avoids a lot of procedure-oriented programming details to facilitate programming. For the purpose of control-flow-based testing, therefore, a means is needed to present the control flow explicitly . The automaton which was proposed in [5] to control recursion is one kind of abstract description for Prolog control flow, but it does not present the control flows of Prolog program in enough detail; in particular, it fails to present backtracking in Prolog.

We propose in this section two kind of graphs to represent Prolog's hidden control flow explicitly. One graph is the so-called *P-flowgraph* (Prolog control flow graph) which is defined by Algorithm 1 in Section 3.1 and represents the control flow in a given predicate at the top level. The other is the so-called *reduced global P-flowgraph* which is defined by Algorithm 3 in Section 3.2 and represents a portion of the global control flow in a set of predicates which may be called by a given predicate.

Guided by the fault model, we present several test selection criteria in terms of coverage of the P-flowgraph and the reduced global P-flowgraph. However, many other kinds of coverages can be defined based on the two graphs. We therefore have to answer the question of what kinds of coverages is good and why the proposed coverages are good. Answering these two questions, we

analyze the goodness of the given test selection criteria on the basis of how many faults can be ensured to be absent under the fault model.

```
/*****************************************************************
 This is a simplified definition of regular expressions:

 <exp> ::= <term> | <term> + <exp>
 <term> ::= <fac> | <fac> <term>
 <fac> ::= <name> | <name>* | (<exp>) | (<exp>)*
 <name> ::= a | b | c | d

 *****************************************************************/
/* The predicate exp is the main predicate in this program, and  is used to do the syntax analysis of regular expression. */

/* exp-bff, -bbf, -bfb exp succeeds only when (1) the first parameter is <exp>, (2) the second parameter is some left part of
the first parameter and also a <exp>, (3) the concatenation of the second parameter and the third parameter is equal to the
first parameter. For the case of exp-bff, exp finds the longest <exp> from the first parameter and put it into the second
parameter. */
 exp(EXP,TERM,REST):- term(EXP,TERM,REST).
 exp(EXP,EXP1,REST):- term(EXP,TERM,[+ | REST1]), exp(REST1,EXP2,REST), append1(TERM, [+ | EXP2], EXP1).

/* term-bff, -bbf, -bfb term succeeds only when (1) the first parameter is <term>, (2) the second parameter is some left part
of the first parameter and also a <term>, (3) the concatenation of the second parameter and the third parameter is equal to
the first parameter. For the case of term-bff, term finds the longest <term> from the first parameter and put it into the
second parameter. */
 term(EXP,FAC,REST):- fac(EXP,FAC,REST).
 term(EXP,TERM,REST):- fac(EXP,FAC,REST1), term(REST1,TERM1,REST), append1(FAC,TERM1,TERM).

/* fac-bff, -bbf, -bfb fac succeeds only when (1) the first parameter is <fac>, (2) the second parameter is some left part of
the first parameter and also a <fac>, (3) the concatenation of the second parameter and the third parameter is equal to the
first parameter.  For the case of fac-bff, fac finds the longest <fac> from the first parameter and put it into the second
parameter. */
 fac([X | EXP],[X],EXP):-name(X).
 fac([X | [* | EXP]],[X | [*]],EXP):-name(X).
 fac(["(" | EXP], FAC, REST):- exp(EXP,EXP1, [")" | REST]), append1(["(" | EXP1], [")"], FAC).
 fac(["(" | EXP], FAC, REST):- exp(EXP,EXP1, [")" | [* | REST]]), append1(["(" | EXP1], [")" | [ *] ], FAC).

/* append1-bbf, -bbb append1 is used to replace the built-in predicate append */
 append1([],L,L).
 append1([X | L1],L2,[X | L3]):-append1(L1,L2,L3).

/* name-b,-f Database facts*/
 name(a).
 name(b).
 name(c).
 name(d).
```

Figure 2: An example of Prolog program

3.1.Test selection based on P-flowgraph

3.1.1. Construction of P-flowgraph

The control flow in Prolog programs is significantly more complex than control flow in the programs of traditional programming languages. The semantics of Prolog implies the following additional features: (1) Several types of backtracking, caused by the failure of subgoal matching and by the requirement for multi-answers (maybe all answers sometimes) for one single goal. (2) Next answer searching for a subgoal to which control is transferred after backtracking. (3) Enforced

control flow change by the predicate "cut". These features must be considered from the testing point of view.

We therefore use the so-called P-flowgraph to provide a means to describe these features explicitly. We will present in the following an algorithm to construct the P-flowgraph for a predicate, and explain the algorithm through the two Prolog programs in Figures 2 and 4. The algorithm takes a Prolog program as input and produces the corresponding P-flowgraph. The Prolog program in Figure 2 is used to explain Algorithm 1. The Prolog program in Figure 4 is used to explain the problems related to the existence of deterministic predicates and "cuts".

The Prolog program in Figure 2 only serves as an example which is meaningful and able to describe the mutually recursive definition of predicates. The reader does not need to fully understand the meaning of the Prolog program, as long as he/she understands the relationship among the mutually recursively defined predicates and the subgoal-solving order.

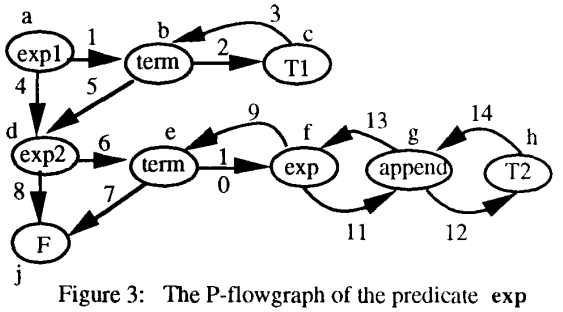


Figure 3: The P-flowgraph of the predicate **exp**

ALGORITHM 1: Constructing the P-flowgraph for a given predicate "p" .

Input: Prolog program containing p

Output: P-flowgraph of p

Step 1: Create nodes for P-flowgraph:

(1.1) For the head of each rule, create a node and label the node with <predicate name><rule number>.

(1.2) For every called predicate in each rule, create a node and label it with the predicate name.

(1.3) For every rule, create a node to indicate the successes of the unifications of all subgoals, and label it with T<rule number>. We call this type of nodes T-nodes.

(1.4) Create a node to represent the failure of the predicate execution and label it with F. We call this node an F-node.

Step 2: Identify whether each predicate, which is used in the definition of the predicate to be tested, is deterministic or nondeterministic.

Step 3: Create a directed edge (i.e. a branch) for each possible control transfer between two above-created nodes as follows:

(3.1) For each control transfer from the head of every rule to the first called predicate of the rule, create a branch to link the corresponding nodes.

(3.2) For the control transfer from each called predicate of every rule to the right side predicate next to it, create a branch to link the corresponding nodes.

(3.3) For successful unifications of all subgoals of every rule, create a branch from the node of the rightmost predicate of the rule, to the T-node of the rule.

(3.4) For each nondeterministic predicate node of every rule and for every T-node, create a backtracking branch from the node to:

(3.4a) the node of the nondeterministic predicate next to it if there exists such a predicate and there is no "cut" between them.

(3.4b) the node of the head of the next rule or F-node if no nodes has been found in (3.4a).

(3.5) For the direct control transfer from the head of each rule to the head of the next rule, create a branch to link the corresponding nodes if the head of the first rule can fail, or create a branch from its corresponding node to the F-node if the rule is the last rule of a predicate.

[*End of algorithm 1*]

Taking the Prolog program of Figure 2 and the corresponding P-graph of Figure 3 as examples, we explain the above algorithm as follows:

Step 1: During (1.1), for the heads of two rules of the predicate exp, create the nodes a and d with labels exp1 and exp2 respectively in the P-flowgraph. During (1.2), for the second rule of exp in Figure 2, resulted nodes are e, f and g in the P-flowgraph. During (1.3), for the predicate exp, the resulting T-nodes are nodes c and h with labels T1 and T2 respectively. During (1.4), the created F-node for exp is node j with label F.

Step 2: Every predicate in the example of Figure 2 is nondeterministic, but the predicate "write" of Figure 4a is deterministic.

Step 3: During (3.1), we create branches 1 and 6 in the P-flowgraph. The resulting branch represents the fact that control is transferred from the head of the rule to the first called predicate of a rule, after the successful unification of the head. During (3.2), we create branches 10 and 11 in the P-flowgraph. During (3.3), we create branches 2 and 12 in the P-flowgraph. During (3.4a), we have branches 3,9,13 and 14 in Figure 3. In the case of the existence of deterministic predicates, for the example in Figure 4a, we create branch 7 in Figure 4b. The backtracking from T-node represents the fact that another answer is required after a successful answer is produced. During (3.4b), we create branches 5 and 7 in Figure 3. For the deterministic predicates, in the example of Figure 4a,

we create branches 6 and 8 in Figure 4b. During (3.5), we have branches 4 and 8 in Figure 3. The resulting branches represent the direct transfer from the head of a rule to the head of the next rule when the unification of the head of the former rule fails.

Comments on Algorithm 1:

(1) T-nodes and F-node created in Step 1 are used to represent the successes and failure of the predicate, and they are needed for revealing the control flow in Prolog although they do not have textual correspondences in the Prolog program.

(2) Generally a predicate of Prolog is nondeterministic, and the corresponding subgoal has two entries(CALL and REDO) and two exits(EXIT and FAIL), as shown in Figure 1. But some of the predicates are deterministic, and each of them only has one entry(CALL) and one exit(EXIT) like subroutines of conventional programs, such as some predicates for printing and so on. Therefore, we need to identify for each predicate in the definition of the tested predicate whether it is deterministic, or not, in order to present the control flow precisely.

(3) The branches resulting from Step 3 reveal the implied control transfers of Prolog.

(4) This algorithm is polynomial with respect to the number of nodes in the P-flowgraph and it terminates after a finite number of steps.

(5) The node in the P-flowgraph which corresponds to the head of the first rule, is the root of the P-flowgraph. A directed path from the root to a T-node represents the execution trace with an answer *yes* ; and a directed path from the root to the F-node represents the execution trace with an answer *fail* . For the example shown in Figure 2, executing exp([a, +, b,e],Term,[ ]), the path "1, 5, 6, 10, 9, 7" in the P-flowgraph shown Figure 3 is traversed, which corresponds the following execution trace: the success of the head of the first rule (branch 1), the failure of the *term* in the first rule (branch 5), the success of the head of the second rule (branch 6), the success of the *term* in the second rule (branch 10), the failure of the *exp* in the second rule (branch 9), the failure of the *term* in the second rule (branch 7).

3.1.2. Test selection criteria

We give in the following two test selection criteria for a Prolog predicate based on the P-flowgraph, and analyze the corresponding fault coverage.

CRITERION 3.1 (Branch coverage of the P-flowgraph): For a given predicate, generate a set of test data such that every branch of the P-flowgraph of the predicate will be traversed by running these test data.

Criterion 3.1 is similar to the branch coverage of conventional programs. According to the criterion all branches of the P-flowgraph should be traversed. Therefore, in the tested predicate, the heads of all rules and all called predicates in the rule bodies should be exercised, and all possible transfers between above heads and called predicates should be exercised too.

In order to analyze the fault coverage clearly, we make the following convention. For a given implementation and the corresponding specification (an oracle in the human mind for implementation-based testing), a P-flowgraph of the wrong implementation can be considered to be obtained by deleting and adding some edges in the P-flowgraph of the specification. Based on such a convention, therefore, we can talk about the question of whether a path in an implementation is a path in the corresponding specification, and vice versa. For analyzing fault coverage easily, we assume furthermore that if the same input causes two different paths in a specification and its implementation, the two results will be different. This assumption will be called *distinct path computation assumption* .

THEOREM 1: If the distinct path computation assumption holds, Criterion 3.1 can ensure the absence of the following fault types:

(1) missing or extra cut,
(2) missing or extra rule,
(3) missing or extra predicate in a rule,
(4) wrong order of called predicates in a rule,
(5) wrong order of rules in a predicate.

Proof: Any fault of the five fault types causes the P-flowgraphs of the implementation and of the corresponding specification to have different edges. In the case of implementation-based testing where test data are developed from the P-flowgraph of an implementation, there exists at least one path, say path A, among the paths resulting from Criterion 3.1, which does not exit in the P-flowgraph of the corresponding specification. The test data for this path A of implementation causes a different path from path A in the specification, therefore it will cause different results between the implementation and the corresponding specification under the distinct path computation assumption, i.e. a fault is found.
[*End of proof*]

This criteria is fault-based. Similar to the branch coverage of conventional program, this criterion is still weak due to the following two problems. First, if the distinct path computation assumption does not hold, the criterion cannot ensure the absence of the above fault types. Second, it cannot ensure the absence of all the faults of the fault types (6), (7) and (8) of the fault model even if the distinct path computation assumption does hold.

Theoretically, no method can solve these problems since they cover the problem of checking whether two Turing machines are equivalent. However, it is possible to go further than Criterion 3.1 and solve part of the problems using the following approach.

Many faults in Prolog programs result in the following situation : Wrong results can be produced only if a particular pair of successive edges of the P-flowgraph are exercised. A similar situation can be found in conventional procedure-oriented program testing, where a method called branch-to-branch pair or 1-switch coverage [4] is presented to deal with this situation. Inspired by this method, we give the following criterion to deal with a similar situation in Prolog programs.

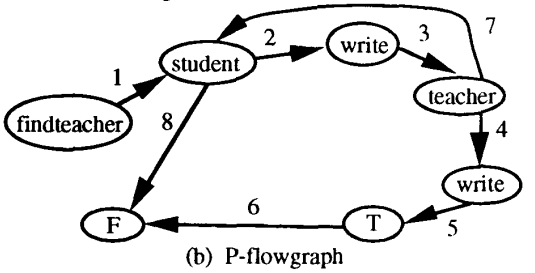
CRITERION 3.2 (Branch-to-branch pair coverage of the P-flowgraph): For a given predicate, generate a set of test data such that every branch-to-branch pair of its P-flowgraph will be traversed by running these test data.

Meeting Criterion 3.2 implies meeting Criteria 3.1. Criterion 3.2 therefore has at least the fault detection power of Criterion 3.1. It furthermore can detect the faults which will cause some wrong control transfers, in particular, the wrong branch-to-branch transfers in the P-flowgraph .

We explain in the following the form of test data for Prolog programs before we give an example to explain Criterion 3.2. Test data for Prolog programs consist of three parts: (1) the values of input variables (usually lists), (2) database facts, and (3) the order of the answer of interest among the alternative answers given by the Prolog program. The execution outcome of a Prolog predicate depends on the values of input variables, the content of the database, and the order numbers of answers. In particular, the order numbers of the answers are needed as a part of test data because of the following reason: By solving a single goal during the execution of the Prolog program, many alternative answers may be produced depending on the user's requests, and the answers are ordered according to their occurrence in time. During testing, in order to traverse a specific path in a control flow graph, maybe only the answer of a particular order number is interesting to us; we therefore require to specify the order numbers of the answers of interest as part of the test data.

```
findteacher :- student(X), write("there exist students !"),
                    teacher(Y), !, write("teacher is ", Y).
```

(a) Program



(b) P-flowgraph

Figure 4: The P-flowgraph of the predicate **findteaher**

Using the example shown in Figure 4a, we now show a fault which cannot be detected by Criterion 3.1, but can be detected by Criterion 3.2. The predicate "findteacher" shown in Figure 4a is supposed to print out only one "there exist students" as one or more "students" are in the database; it then finds a "teacher" for "students" and prints out "teacher is <name>" if there exist teachers in the database; and it prints nothing if there does not exist any teacher . According to the P-flowgraph of the predicate findteacher shown in Figure 4b, the different branch-to-branch pairs are the following:

1-2, 1-8, 2-3, 3-4, 3-7, 4-5, 5-6, 7-2, 7-8

According to Criterion 3.1 the following test data can be adopted:

Test-data 1:
Input variable values: empty;
Database facts: student(R. Roy).
teacher(L. Clarke).
Answers: 1st and 2nd.

Test-data 2:
Input variable values: empty;
Database facts: empty;
Answers: 1st.

Test-data 3:
Input variables values: empty;
Database facts: student(R. Roy).
Answers: 1st and 2nd.

Although Test-data 1, 2 and 3 cover all branches of the P-flowgraph, they do not cover all the branch-to-branch pairs, leaving the pair 7-2 uncovered. Test-data 4 given below is used to cover the pair 7-2.

Test-data 4:
Input variables values: empty;
Database facts: student(R. Roy).
student(G. Cobbert).
Answers: 1st.

By running Test-data 1, the following is printed

"there exist students !"
"teacher is L. Clarke"

With the first answer being true and the second answer false, we find no fault by executing test data 1. We also cannot find any fault by executing Test-data 2 and 3. But by running Test-data 4, the output is:

"there exist students !"
"there exist students !"

This is contrary to our specification that the predicate will print out only one "there exist students" if there exist more than one "students" in the database. A fault is therefore found. The correct version of predicate findteacher is the following, and the fault in the incorrect version is "missing a cut".

```
findteacher :- student(X), write("there exist students !"), !,
               teacher(X), !, write("teacher is ", X).
```

3.2. Test selection based on a reduced global P-flowgraph

We give in this section a test selection criterion to detect the faults which can be exhibited only by exercising the recursive part of Prolog programs. A P-flowgraph only presents the top-level control flow of a tested predicate which does not reflect the recursive nature. Solving such a problem, we require to construct a kind of control flow graph to represent global control flow in order to capture the recursive nature. Test selection criteria will be presented in terms of coverage of the global control flow graph.

Ideally, a global control flow graph of a given predicate can be constructed in the following fashion: Let a modified P-flowgraph be the graph resulting from a P-flowgraph by removing all T-nodes and the F-node; for example, the modified P-flowgraph of Figure 6 corresponds to the P-flowgraph of Figure 3. Starting from the P-flowgraph of the predicate, replace a node corresponding to a predicate by the modified P-flowgraph of the predicate (for instance, by replacing the node b in the P-flowgraph shown in Figure 3 by the modified P-flowgraph of the predicate term, we obtain the graph in Figure 7), and do such replacements repeatedly until no further progress can be made.

Unfortunately, the graph thus resulted is usually infinite or very large so that we cannot use it directly to select test data. To compromise between the completeness and complexity for test selection, we use a so-called *reduced global P-flowgraph* to present a part of global control flow in the above ideal control flow graph. To determine the reduced global P-flowgraph, we require the *Calling-graph* (calling relation graph) to describe the relations among the recursively defined predicates.

ALGORITHM 2: Constructing the Calling-graph for a given predicate "p".
Input: Prolog program containing p
Output: Calling-graph for p
Step 1: Create a graph G of one node with label p. For every predicate which can be called directly or indirectly by the predicate p, create in the graph G a node labeled with the corresponding predicate name.
Step 2: For every ordered pair of nodes in G where the predicate of the first node may call the predicate of the second node directly, create in G a directed edge (or branch) from the first node to the second node.
[*End of algorithm 2*].

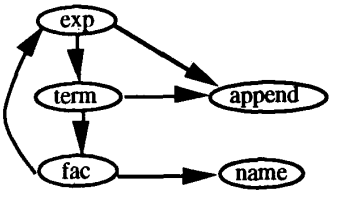


Figure 5: The Calling-graph of the predicate **exp**

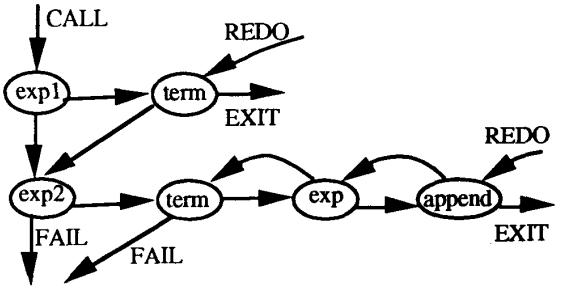


Figure 6: The modified P-flowgraph of the predicate **exp**

Figure 5 shows the Calling-graph of the predicate exp in Figure 2. It is easy to identify a group of mutually recursively defined predicates by means of the Calling-graph, where the group of predicates corresponds to a group of nodes in a strongly connected component of the Calling-graph. For example, from the Calling-graph shown in Figure 5, we know that the predicates exp, term and fac are a group of mutually recursively defined predicates.

However, although the Calling-graph represents the relationship among those mutually recursively defined predicates, it hardly presents any detailed information about the global control flow. But, it gives an idea of how to construct a reduced global control flow graph. By replacing the nodes of all strongly connected components of a Calling-graph with the corresponding P-flowgraphs or modified P-flowgraphs, a reduced global P-flowgraph can be obtained by using the following algorithm. The resulting reduced control flow graph is a compromise between the completeness of test coverage, the complexity of test generation and the size of the test set.

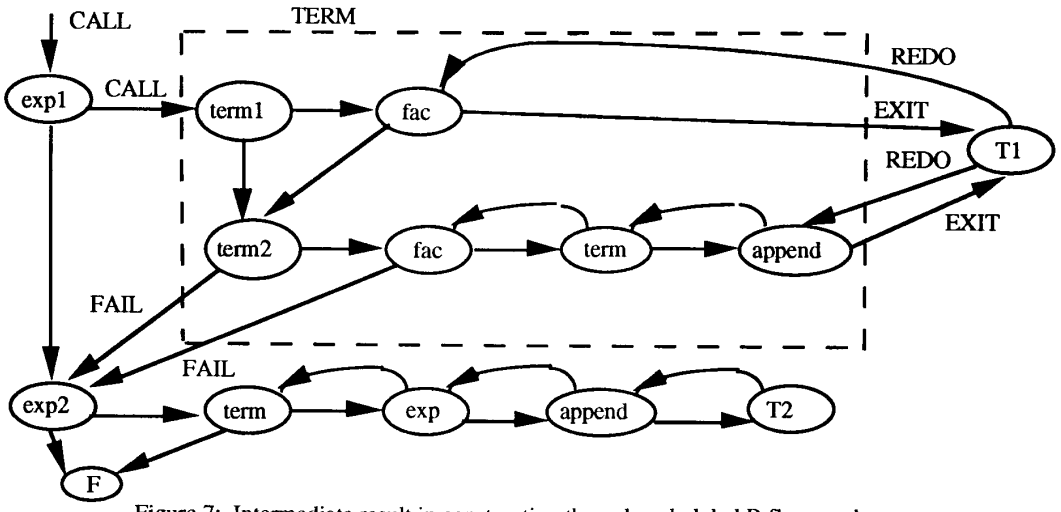


Figure 7: Intermediate result in constructing the reduced global P-flowgraph

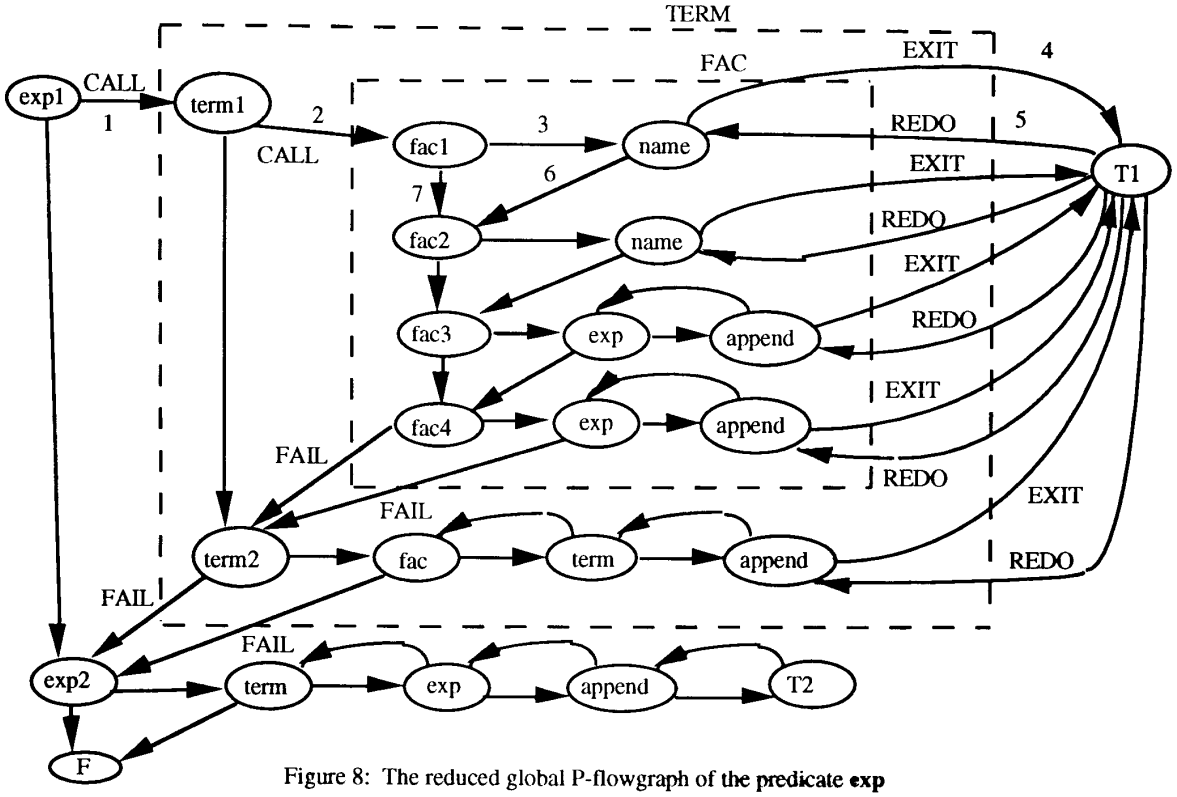


Figure 8: The reduced global P-flowgraph of the predicate **exp**

ALGORITHM 3: Constructing a reduced global P-flowgraph for a given predicate "p".
Input: (1) Prolog program containing p, (2) Calling-graph for p
Output: Reduced global P-flowgraph

Data structure: a stack
Step 1: Put the predicate p into the stack.
Step 2: If the stack is empty , then stop with G being the reduced global P-flowgraph. Otherwise:

(1) Pop out a predicate, say q. Find all successive internal nodes of q in the Calling-graph which have never been pushed into the stack, and push them into the stack.

(2) If q is the given predicate p, let the P-flowgraph of p be graph G. Otherwise, if the P-flowgraph or modified P-flowgraph of q has not yet been used in Step 2 to replace some node in G, find a node with label q in G and replace it by its modified P-flowgraph. Goto Step 2.

[*End of algorithm 3*].

The algorithm is explained in the following by the Prolog program of Figure 2. Suppose that predicate exp in the program of Figure 2 is the given predicate p in Algorithm 3. We first obtain the Calling-graph of exp shown in Figure 5. During Step 1, exp is pushed into the stack. After Step 2 is executed the first time, the P-flowgraph of exp is created; the content of the stack is [term] since exp has in the Calling-graph only one successive internal node "term" and since node "append" is a leaf node. After Step 2 is executed the second time, we obtain the graph shown in Figure 7 with the content of the stack being [fac]. After Step 2 is executed the third time, we finally obtain the graph shown in Figure 8 which is the reduced global P-flowgraph of the predicate exp.

Based on the reduced global P-flowgraphs, we propose the following test selection criterion.

CRITERION 3.3 (Branch coverage of the reduced global P-flowgraph): For a given predicate, generate a set of test data such that every branch of the reduced global P-flowgraph will be traversed by running these test data.

The intent of the criterion is to find the faults which will cause wrong control transfer related to recursive definitions and the integration of predicates. The only way to exhibit this wrong control transfer is to exercise the paths of the global P-flowgraph. Since it is impossible to exercise all different paths of the global P-flowgraph, it is adequate and necessary to generate test data on the basis of the coverage of the reduced control flow graph. Usually we first generate test data according to Criteria 3.1 and 3.2 and find extra test data if the test data thus produced does not satisfy Criterion 3.3.

4. Test data generation tool

Given test selection criteria, one still needs to generate test data according to the criteria efficiently. In order to facilitate test generation, we propose in the following a program instrumentation tool which inserts special predicates (probes) into a given Prolog program. These probes make up a Test Generation Tool (TGT) which generates test data semi-automatically.

4.1. Monitoring execution traces by instrumentation

In order to obtain the execution trace information for evaluating the coverage of the P-flowgraph and the reduced global P-flowgraph, the probes which are also Prolog predicates should be inserted into the given Prolog program to be tested. We study in this section which points in a given Prolog program the probes should be inserted by an instrumentation tool.

In order to record traces, probes are inserted into the four following places: (1) the entry of a rule, (2) the exit of a rule, which is also a successful exit of a corresponding predicate, (3) the entry of a predicate, and (4) the failure exit of a predicate. The probes are used to record the information about the execution traces. For example, according to the above description, the predicate exp shown in Figure 2 would be instrumented as follows:

```
exp(EXP1, EXP2, EXP3):- probe(exp_in), fail.
exp(EXP,TERM,REST):-probe(exp1_in),
    term(EXP,TERM,REST), probe(exp1_T).
exp(EXP,EXP1,REST):-probe(exp2_in),
    term(EXP,TERM,[+ | REST1]),
        exp(REST1,EXP2,REST),
    append(TERM,[+ | EXP2],EXP1), probe(exp2_T).
exp(EXP1, EXP2, EXP3):- probe(exp_F), fail.

probe(St):- "record St in some place".
```

where the predicate "fail" always fails when it is called. For the Prolog program of Figure 2, the completely instrumented program is given in [15].

When we run this instrumented program with the goal exp([a],X,[]), TGT will record the following trace: exp_in, exp1_in, term_in, term1_in, fac_in, fac1_in, fac1_T, term1_T, exp1_T, and return with X = [a] successfully. From this trace, the TGT can find that this trace covers the path "1, 2" of the P-flowgraph shown in Figure 3 and the path "1,2,3,4" of the reduced global P-flowgraph shown in Figure 8. In this example, probe(exp1_T) and probe(exp2_T) are used to monitor the traversal of the T-nodes in the P-flowgraph and the reduced global P-flowgraph; and probe(exp_F) is used to monitor the traversal of the F-node.

If each predicate of a given Prolog program is instrumented in the above four places, the trace information obtained by the probes during the executions is enough to decide what parts of the P-flowgraph or the reduced global P-flowgraph have been traversed.

4.2. Test selection with the help of TGT

We explain in the following the test selection with help of TGT briefly. More details have been presented in [15]. By using TGT, we can generate test data in the following manner:

(1) The probes are inserted into the given Prolog program by the instrumentation tool automatically; the TGT consists of the probes.
(2) Select some test data arbitrarily and run these test data on the instrumented program.
(3) The TGT records the resulting execution traces, reports what part of the P-flowgraph (or the reduced global P-flowgraph) has not yet been exercised. The report provides a guidance for test selection according to the adopted test selection criteria.
(4) According to the report from TGT, find extra test which may increase the coverage on the P-flowgraph (or the reduced global P-flowgraph) intuitively. Repeat this process until no more coverage has been achieved, or the full coverage has been achieved.
(5) In the case where the full coverage has been achieved in (4), no more test cases are needed. Otherwise, find extra test to achieve the full coverage by manual calculation.

Using TGT, we can generate test case semi-automatically. In contrast to the case of procedure-oriented program testing, a single Prolog test case produces a much longer execution path because of backtracking and recursions. Thus, it is much more difficult to manually derive the resulting paths and the coverage of the control-flow-graph for a test. Using TGT, however, we can save a lot of manpower by an automatic evaluation of execution coverage provided by TGT.

5. Conclusion

We have presented several test selection criteria for Prolog programs, which are based on the control flow of Prolog programs. Future work could be done with respect to the data flow of Prolog programs. The test generation tool presented here provides a means to generate test cases semi-automatically.

Acknowledgments: The authors would like to thank many people, in particular Prof. Roland Groz who carefully read our paper and gave us many valuable comments, and Mr. Cheng Wu and Mr. Kaiyuan Huang for many useful suggestions and comments. This work was supported by the IDACOM-NSERC-CWARC Industrial Research Chair on Communication Protocols at the University of Montreal (Canada).

References:

[1] Pierre De Boeck and Baudouin Le Charlier, "Static Type Analysis of Prolog Procedures for Ensuring Correctness", International Workshop PLILP'90, Lecture Notes in Computer Science 456, Springer-Verlag, pp.223-237.

[2] L.Bouge, N.Choquet, L.Fribourg, M.C.Gaudel, "Application of Prolog to Test Sets Generation from Algebraic Specification", TAP Soft Conference on Theory and Practice of Software Development, Berlin, March, 1985, LNCS 186 pp.261-275.

[3] N. Choquet, "Test Data Generation Using a PROLOG with Constraints", Workshop on Software Testing, Verification and Analysis", Banff, Canada, July 1986, pp132--141.

[4] T.S.Chow, "Testing Software Design Modeled by Finite-State Machines, IEEE Transactions on Software Eng., Vol. SE-4, No.3, 1978.

[5] Richard Denney, "Test-Case Generation from Prolog-Based Specifications", IEEE Software, March 1991, pp.49-57.

[6] Michael M.Gorlick, Carl F.Kesselman, Daniel A.Marotta & D. Stott Parker, "MOCKINGBIRD: A Logical Methodology for Testing", J. Logic Programming, No.8, 1990, pp95--119.

[7] Daniel Hoffman, and Paul Strooper, "Automated Module Testing in Prolog", IEEE Transactions on Software Engineering, Vol. SE-17, No.9, 1991, pp.934-943.

[8] William E.Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-2, No.3, 1976, pp.208-215.

[9] William E.Howden, "Functional Program Testing", IEEE Transactions on Software Engineering, Vol. SE-6, No.2, March 1980, pp.162-169.

[10] William E.Howden, Functional Program Testing & Analysis, McGraw-Hill, Inc., New York, 1987.

[11] IEEE Standard Glossary of Software Engineering Terminology, IEEE Standard 729-1983, 1983.

[12] Mariam Kamkar, Nahid Shahmehri and Peter Fritzson, "Bug Localization by Algorithmic Debugging and Program Slicing, International Workshop PLILP'90, Lecture Notes in Computer Science 456, Springer-Verlag, pp.223-237.

[13] B. Korel, "Automated Software Test Data Generation", IEEE Transactions on Software Engineering, Vol.16, No.8, August 1990, pp.870-879.

[14] Gang Luo, Junliang Chen and Xun Yuan, "Prolog Logic Program Testing", Journal of China Institute of Computer, November, 1991, pp.838-844.

[15] Gang Luo, Gregor v. Bochmann, Behcet Sarikaya and Michel Boyer, "Control-flow Based testing of Prolog Programs", Department Report #825, Department of Computer Science, University of Montreal, 1992.

[16] Larry J.Morell, "A Theory of Fault-Based Testing", IEEE Transactions on Software Engineering, Vol.16, No.8, August 1990, pp.844-857.

[17] G.J.Myers, The Art of Software Testing, John Wiley & Sons, Inc. New York, 1979.

[18] Lutz Plumer, Termination Proofs for Logic Programs, Lecture Notes in Artificial Intelligence, Vol. 446 1990, Springer-Verlag.

[19] L.M.Pereira, "Rational Debugging in Logic Programming", Third International Conference on Logic Programming, Ed. E.Y.Shapiro, Lecture Notes on Computer Science 225, 1986, pp.203-210.

[20] E.Y.Shapiro: Algorithmic Program Debugging. MIT press. 1983.

[21] J.D.Ullman & A.V.Gelder, "Efficient Tests for Top-Down Termination of Logical Rules", Journal of ACM, Vol.35, No.2,1988, pp.345-373.

[22] Elaine J.Weyuker and Thomas J.Ostrand, "Theories of Program Testing and Application of revealing subdomains", IEEE Transactions on Software Engineering, Vol. SE-6, No.3, May 1980, pp.236-246.