Generation of test cases for simple Prolog programs*

PEKKA KILPELÄINEN, HEIKKI MANNILA

University of Helsinki, Department of Computer Science
Teollisuuskatu 23, SF-00510 Helsinki

Abstract

We describe a general method for producing complete sets of test data for Prolog programs. The method is based on the classical competent programmer hypothesis from the theory of testing, which states that the program written by the programmer differs only slightly from the correct one. The nearness is expressed by postulating a class of possible errors, and by assuming that the written program contains only errors from this class. Under this assumption the test cases produced by the method are enough to ensure the correctness of the program. The method is based on a result showing that it is sufficient to consider programs from which the written one differs by a single error. Test cases are produced by forming a path condition consisting of equations and universally quantified inequations, and solving the condition. The method is particularly easy to implement for the class of iterative programs; for general programs it can be used as a component of an interactive tool.

1. Introduction

One of the attractive properties of the Prolog programming language is that testing is quite easy. Each predicate can be tested as soon as it has been written. One does not have to write separate test programs, as in many conventional programming languages and environments.

Although testing Prolog programs is easy, the problem of choosing sets of test data is as difficult as in conventional languages. How do we know that the inputs we use for testing really test the program adequately?

There is a fairly large amount of research on testing of programs and systems written in conventional programming languages (see, e.g., the books by Beizer

* This work is supported by the Academy of Finland.

** Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

[Be84] and Myers [My79]). Also the theory of testing has been developed [BA82, GG75, DMLS78, MR89, Ho86]. This paper shows how these ideas can be applied to the generation of test cases for Prolog programs.

We describe a method for producing sets of test cases which are complete in a certain exact sense. For example, for the program

```
member(A, [A|_]).
member(A, [_|X]) :- member(A, X).
```

our method generates the test queries

```
?-member(a, [b]).
?-member(a, [a]).
?-member(a, [b, b]).
?-member(a, [b, a]).
```

These queries can be seen to test the member predicate in a quite natural way.

Our method is based on the competent programmer hypothesis [DMLS78, GG75], which assumes that the program P written by the programmer is fairly close to the intended program Q. We formalize this along the lines of [DMLS78] and [Br80] by assuming there is a class M of possible modifications, and that P and Q differ only by one or more modifications from M. Intuitively the modifications are inverses of possible programming errors. In Prolog the class M can contain modifications like exchanged variables, missing or extraneous functors or missing arguments. The competent programmer hypothesis is used by generating test cases which show the difference between the written program and any other nonequivalent program which differs from it only by modifications in class M. Hence the test cases show the difference of the written and the intended program (if there is any). The equivalence criterion used is based on the box model trace of the predicates. Hence top level tracing must be used in running the test cases to observe all their properties.

The above test cases for member were produced by considering errors in variable names. If the class of modifications contains also missing functors, then the test case set would include the additional query

?-member(a, a).

The choice of the class of modifications M is fairly important for the usefulness of the method: the larger M is, the more likely it is that the formalization of the competent programmer hypothesis holds. On the other hand, a large set M tends to produce more test cases. Fortunately, our method is not very dependent on the properties of M, so that variety of choices can be used.

Our method faces the difficulties inherent in every test data generation method. If the predicate p to be tested calls some other predicate, say q, we have to be able to generate inputs satisfying q. No system can automate this for all possible predicates q, as determining whether q ever can succeed is an unsolvable problem. Therefore our system is particularly well suited for simple programs, e.g., the *iterative* programs defined in [SS86]. Our work is fairly close in spirit to the mutation testing approach [DMLS78] and especially to the work of Brooks [Br80] on generating test cases for Lisp programs.

Work on generating test data is in a sense complementary to the interesting work done on debugging Prolog programs by Shapiro and others (see [Sh82, Pe86]). Algorithmic debugging aims at methods for finding the cause of an erroneous test output; test data generation tries to help in the process of finding the inputs showing the presence of an error. Our work can also be seen to be complementary to the work done on program synthesis [Sh82, MCM83, MCM86], which tries to move from illustrative examples to programs. We move in the opposite direction.

The rest of this paper is organized as follows. Section 2 describes the general framework by defining the concept of complete test data. It also describes a naive method for producing test data and points out its deficiencies. Section 3 contains the theoretical result showing that instead of all programs which can be obtained from the original program P by one or more modifications it suffices to consider those programs which differ from P by one modification only. This is crucial in obtaining reasonably sized sets of test cases.

Section 4 discusses how we find a test case which illustrates the difference of program P and a program obtained from P by applying one modification. We show that the existence of such an input can be characterized by giving a formula containing equations and universally quantified inequations [LMM86]. In Section 5 we discuss how inputs satisfying these formulas are found for simple programs. Section 6 is a short conclusion. For reasons of space some straightforward technical definitions have been omitted.

2. Framework

We consider the testing of a predicate p which has been defined by giving a program P for p, consisting of a list of clauses for p and the definitions of other predicates. We want to generate queries of the form

$$?\text{-}p(d_1, d_2, \ldots, d_k).$$

where k is the arity of p and $d_1, \ldots, d_k$ are terms, such that these queries test predicate p completely in some sense. We call $d = d_1, \ldots, d_k$ an *input* for the program P to achieve compatibility with the usual terminology in the theory of testing. To discuss testing one has to specify the properties of program one is interested in. The basic choice is to consider input-output relations, i.e., the function/relation computed by the program. However, this gives little information about the program. Therefore in the theory of testing it is usual to consider traces of program execution, which give more information about the computations (see, e.g., [Br80] for a discussion of the usefulness of traces).

Given a program P, we define the *top-level trace* of P on input d, denoted by $P(d)$. This is, intuitively, the box model trace of the query

$$?\text{-}p(d).$$

limited to the definition of p and with the unification attempts explicitly represented. For example, let P denote the following definition of append

```
append([], X, X).
append([A|X], Y, [A|Z]):- append(X, Y, Z).
```

Then the top-level trace of P on ([1, 2, 3], [4, 5], U) is

```
CALL append([1, 2, 3], [4, 5], U)
UNIFY with append([], X, X) FAILED
UNIFY with append([A|X], Y, [A|Z]):- ...SUCCEEDED
    CALL append([2, 3], [4, 5], Z)
    EXIT append([2, 3], [4, 5], [2, 3, 4, 5])
EXIT append([1, 2, 3], [4, 5], [1, 2, 3, 4, 5]).
```

Note that the recursive call is not traced; only the CALL and EXIT of it are represented. We omit the straightforward formal definition of the top-level trace.

Two programs P and Q for predicate p are *equivalent*, if for all inputs d the traces of P and Q are equal, i.e., $P(d)=Q(d)$ for all d. If P and Q are equivalent, we write $P \equiv Q$.

Example 1. Let P be the program

```
p(X) :- sort(X, Y),process(Y).
sort(X, Y) :- ...
```

with sort implemented by merge sort, and Q the same program, but sort implemented by quicksort. Then P and Q are equivalent, since the traces $P(d)$ and $Q(d)$ do not include any details of the sort computation. □

The above definition of equivalence is rather strict. Two equivalent programs not only have to compute the same relation for predicate p, but they have to compute it in the same way at the level of p's definition.

Equivalence of programs is undecidable, as, e.g., the halting problem can be reduced to it. Therefore one cannot expect too much from a test method which tries to generate instances separating the given program from all nonequivalent programs in a given class. If the class is wide enough, even recognizing the (non)equivalent programs cannot be done.

As mentioned in the introduction, our work is based on the competent programmer hypothesis [DMLS78]. That is, we assume that the program written by the programmer is reasonably close to the one he/she meant to write.

Following Brooks [Br80] and DeMillo, Lipton, and Sayward [DMLS78], we use this assumption by postulating a set of possible modifications M.

Example 2. We use mainly list processing programs in our examples. For such programs, a suitable class of modifications consists of the following.

- replace an occurence of variable x by variable y, for all variables x and y
- replace term $[t|t']$ by term t or t'
- replace variable x by term $[t|x]$ where t is a variable or an atom

We omit the formal definition of this class. □

We assume, that the programmer has made only errors which are inverses of some modifications in the class M. That is, the intended program differs from the written one only by one or more modifications from the class M. We assume the errors occur only in the clauses of predicate p, i.e., the testing concentrates on this one predicate.

The *neighbourhood* $M^*(P)$ of P consists of programs Q such that P can be transformed to Q by application of zero or more modifications from M. We denote the result of applying a single modification m to a program P by $P.m$. The neighbourhood $M^*(P)$ of P is defined as follows.

$$M^0(P) = \{P\}$$

$$M^i(P) = \{Q.m | Q \in M^{i-1}(P), m \in M\}, i > 0$$

$$M^*(P) = \bigcup_{i \geq 0} M^i(P)$$

Formalized in this framework the competent programmer hypothesis states that the intended program P' belongs to $M^*(P)$.

Let D be a set of inputs for program P. D is an (M-)*complete* test data set for P, if for all programs $Q \in M^*(P)$, such that Q is not equivalent with P, there is an input $d \in D$ such that $P(d) \neq Q(d)$. If the program works correctly on such a set D, then by the competent programmer hypothesis the program is the intended one.

Example 3. The five queries for member given in the introduction form a complete set of test data for member under the errors of Example 2. □

Let C be a set of programs and P a program, and D a set of test cases for P. We say that D separates P from C, if for each $Q \in C$, $Q \not\equiv P$. there is an input $d \in D$ such that $P(d) \neq Q(d)$. Thus a complete set of test cases for P separates P from $M^*(P)$.

How does one generate complete sets of test data? A naive approach would be to generate for each $Q \in M^*(P)$, $Q \not\equiv P$, an input d_Q such that $Q(d_Q) \neq P(d_Q)$, and to collect these inputs into the test data set. However, $M^*(P)$ may be infinite and usually is very large. The naive method takes time proportional to the size of $M^*(P)$ and probably produces test sets having about as many elements as $M^*(P)$ has, which is unacceptable. In the next section we show how this problem can be avoided by considering only a small subset of $M^*(P)$.

The second problem is: given programs P and Q, how do we decide whether they are quivalent or not, and if they are not, how do we generate a test revealing the nonequivalence (i.e., an input d for which $P(d) \neq Q(d)$)? These questions are considered in Sections 4 and 5.

We close this section by giving another example of a complete test data set.

Example 4. Let P be the familiar append program:

```
append([], X, X).
append([A|X], Y, [A|Z]) :- append(X, Y, Z).
```

Let M be the class of modifications consisting of variable name changes and errors in list functors. Then the following queries are a complete test data set for P.

```
?-append([], a, a).
?-append([a], [], [a]).
?-append([a], a, [a, a]).
?-append(a, a, a).
```

□

3. Local neighbourhoods

Let a class M of modifications and a program P be given. We want to generate a test case set D separating P from each program in $M^*(P)$. Given a program $Q \in M^*(P)$, we know that Q can be obtained from P by applying the modifications from M. That is

$$Q = P.m_1.m_2.\dots.m_n$$

for some modifications $m_1, \dots, m_n \in M$.

Example 5. Let the member program be as in the introduction; denote it by P. Let Q be the program

member(A).
member(A, [A|X]):- member(X, X).

Then $Q = P.\ m_1.\ m_2.\ m_3$, where m_1 drops the second argument of the fact and m_2 and m_3 change variable names in the second clause. Thus $Q \in M^*(P)$. □

Our goal is to avoid considering the entire neighbourhood $M^*(P)$. For this, we define the *local* (*M*-)*neighbourhood* $M(P)$ of program P by

$$M(P) = M^1(P) = \{P.m | m \in M\}$$

That is, $M(P)$ consist of those programs obtained from P by applying one modification from M; thus $M(P) \subset M^*(P)$.

Example 6. The program Q in Example 5 belongs to $M^*(P)$, but not to $M(P)$. □

The class $M(P)$ is far smaller than the entire neighbourhood $M^*(P)$. If there are k subgoals in the program and for each subgoal there are on the average b modifications that can change it, then $M(P)$ contains about kb elements, whereas $M^*(P)$ has about b^k elements.

In order to be able to consider only the local M-neighbourhood of P, we need to restrict the possible sets of modifications. A set of modifications M is *term-closed* for P if all programs $Q \in M^*(P)$ differing from P only with respect to a single term belong to $M(P)$. In other words, we require that all sequences of modifications to one term can be expressed as one modification. We also assume, that modifications are *shape preserving* in the sence of Brooks [Br80]. A modification is shape preserving, if it does not change the ordering and number of heads and goals in the program. Thus a shape preserving modification may change argument lists, but it preserves the overall structure of the program. These assumptions are realistic, if we think modifications as inverses of typing errors, for example.

The next theorem shows that under our assumptions a set of test cases separating the program from its local environment is enough to completely test the program.

Theorem 1. Let P be a program for predicate p, M a term-closed set of shape preserving modifications to the clauses of predicate p, and D a set of test cases that separates P from $M(P)$. Then D is a complete set of test cases for P and M, i.e., D separates program P from $M^*(P)$.

Proof. Let Q be a program in $M^*(P)$, $Q \not\equiv P$. We need to show that there is a test case d in D for which the traces $Q(d)$ and $P(d)$ differ from each other. Let d' be a test case, for which $Q(d') \neq P(d')$. Consider the first differing lines in the traces $Q(d')$ and $P(d')$; let them be the ith ones. We have the following lemma.

Lemma 1. The first differing lines in $Q(d')$ and $P(d')$ are of one of the following forms.

1. UNIFY with h' and UNIFY with h, where h' is a clause head in program Q, h a clause head in P, and $h \neq h'$,
2. CALL $r(t_1', \ldots, t_n')\theta$ and CALL $r(t_1, \ldots, t_n)\theta$, where $r(t_1', \ldots, t_n')$ is a subgoal of program Q, $r(t_1, \ldots, t_n)$ a subgoal of P, and $t_i'\theta \neq t_i\theta$ for some $i \in 1, \ldots, n$, or
3. the difference in the traces follows immediately calls to a predicate r (i.e., one trace contains an EXIT or a FAIL line, and the other contains adifferent line or no line at all). Note that because only the definition of predicate p may be modified, the call to r must be a direct or indirect call to p. □

The proof of the theorem uses induction on the number of the recursive calls of p that must be executed before the first differing lines in the traces $Q(d')$ and $P(d')$ are encountered.

For the base case, only the first two cases of the lemma are possible. In the first case the ith line of $Q(d')$ is

$$\text{UNIFY with } h' :- \ldots \tag{1}$$

and the corresponding line in trace $P(d')$ is

$$\text{UNIFY with } h :- \ldots \tag{2}$$

Let m be a modification that changes the head h. Then $P.m \in M(P)$, and $P \cdot m \not\equiv P$, since the ith lines in the traces $P.\ m(d')$ and $P(d')$ differ. Therefore there is an input d in D such that $P(d) \neq P.\ m(d)$. Let the first lines where the traces $P(d)$ and $P.\ m(d)$ differ from each other be the jth ones. Now, if the traces $Q(d)$ and $P(d)$ differ before line j, $Q(d) \neq P(d)$. Otherwise the jth lines are (1) and (2), i.e., they are different, and again $Q(d) \neq P(d)$.

In the second case the ith line in trace $Q(d')$ is

$$\text{CALL } r(t_1', \ldots, t_n')\,\theta$$

and the corresponding line in $P(d')$ is

$$\text{CALL } r(t_1, \ldots, t_n)\,\theta$$

and there is a $k \in 1, \ldots, n$ such that $t_k'\theta \neq t_k\theta$. By the term-closedness of M there is a modification $m \in M$ for which $t_k \cdot m = t_k'$. Now the ith line in the trace $P.\ m(d')$ is

$$\text{CALL } r(t_1, \ldots, t_k.m, \ldots, t_n)\,\theta$$

and the corresponding line in $P(d')$ is

$$\text{CALL } r(t_1, \ldots, t_k, \ldots, t_n)\,\theta$$

Because the traces differ, $P.\,m \not\equiv P$. Since $P.\,m \in M(P)$, there is an input $d \in D$ such that $P(d) \neq P.\,m(d)$. Assume that these traces are the same up to the jth line. Let the jth line of $P(d)$ be

$$\text{CALL } r(t_1, \ldots, t_k, \ldots, t_n)\,\gamma$$

and the corresponding line in $P.\,m(d)$

$$\text{CALL } r(t_1, \ldots, t_k.m, \ldots, t_n)\,\gamma$$

with a substitution γ, for which $t_k\gamma \neq t_k.\,m\gamma$. If the traces $Q(d)$ and $P(d)$ differ before the jth line, then $Q(d) \neq P(d)$. Otherwise the jth line of $Q(d)$ is

$$\text{CALL } r(t_1', \ldots, t_k.m, \ldots, t_n'),$$

which is different from the corresponding line in $P(d)$, since $t_k\gamma \neq t_k.\,m\gamma$.

The induction assumption is that for all d'' such that $Q(d'')$ and $P(d'')$ differ and the number of recursive calls to p before producing the first different lines in $Q(d'')$ and $P(d'')$ is smaller than the number of recursive calls preceding the first difference in $Q(d')$ and $P(d')$, there exists a $d \in D$ such that $Q(d)$ and $P(d)$ differ.

Assume that there are recursive calls of p in the execution preceding the first differing lines in top-level traces $Q(d')$ and $P(d')$. Now all the three cases of the lemma are possible. The first two cases are as in the base case. The third case is that the first line where the traces $Q(d')$ and $P(d')$ differ is immediately after a line

$$\text{CALL } r(u_1, \ldots, u_m),$$

Because the programs Q and P can differ at the definitions of predicate p only, the execution of the call $r(u_1, \ldots, u_m)$ must contain a recursive call $p(t_1, \ldots, t_n)\theta$, and the result of this call differs in Q and P. Let $d'' = t_1\theta, \ldots, t_1\theta$. Now $Q(d'') \neq P(d'')$, and the number of recursive calls of p before the first difference of $Q(d'')$ and $P(d'')$ is smaller than the corresponding number before the first difference between $Q(d')$ and $P(d')$. By the induction assumption there exists a $d \in D$ such that $Q(d) \neq$ $\neq P(d)$. □

Now we have reduced the problem of generating complete sets of tes cases for P to the problem of separating P from the class $M(P) = \{P.\,m \mid m \in M\}$. Our test data generation method can thus be formulated as follows:

```
D:={};
for each modification m∈M do
    if P≢P.m then
        generate a d such that P(d)≠P.m(d);
        D:=D∪{d};
```

The method outlined above has connections to the classical testing method known as path testing [Be84]. This method requires that every path in the program is traversed by execution of some test case. We have the following simple result.

Theorem 2. Let a class of modifications M and a program P be given. If for each subgoal in the clauses of predicate p there is a modification m altering that subgoal so that $P.\,m \not\equiv P$, then the execution of all the queries in a complete set of test data for P traverses every subgoal of the predicate p. □

4. Path conditions and separation

Suppose we are given a program P and a modification m. How do we test whether P and $P.m$ are equivalent and if they are not, generate an input d such that $P(d)\neq P.m(d)$? Such a d (if one exists) must cause the execution of P to proceed to the subgoal c altered by the modification m; additionally, d must be such that it causes different trace lines to be output by subgoals c and $c.m$.

Suppose p is defined by the clauses

$$p(t_1) \text{:-} q_{11}(t_{11}), \ldots, q_{1a_1}(t_{1a_1}).$$
$$\vdots$$
$$p(t_k) \text{:-} q_{k1}(t_{k1}), \ldots, q_{ka_k}(t_{ka_k}).$$
$$\vdots$$
$$p(t_u) \text{:-} q_{u1}(t_{u1}), \ldots, q_{ua_u}(t_{ua_u}).$$

Here $t_1, \ldots, t_u, t_{11}, \ldots, t_{ua_u}$ are parameter lists. Let m alter the goal $q_{kh}(t_{kh})$ in clause k $(1\leqq k\leqq u,\ 1\leqq h\leqq a_k)$. Then an input d such that $P(d)\neq P.m(d)$ must satisfy at least the following conditions.

1. d unifies with t_k; let θ be the unifying substitution,
2. the subgoal $(q_{k1}(t_{k1}), \ldots, q_{kh-1}(t_{kh-1}))\theta$ succeeds
3. the goals $q_{kh}(t_{kh}\theta)$ and $(q_{kh}(t_{kh}).m)\theta$ produce different trace lines.

Here (1) states that execution of the altered clause must be able to start; (2), that this execution proceeds to subgoal q_{kh}; and (3), that the resulting traces are different.

These conditions are not enough, however. We must know that unification of clause k in p's definition is attempted. There are two ways to guarantee this: we can require that no previous clause is applicable, or that no previous clause succeeds (or meets a cut). A third possibility arises when the clauses $1, \ldots, k-1$ contain no cuts. Then clause k can be reached by backtracking.

The first alternative of the condition (4) is formally expressed as

(a) t_i and d do not unify for each $i=1, \ldots, k-1$,

and the second

(b) for each $i=1, \ldots, k-1$, either t_i and d do not unify, or, if they do (with substitution θ_i), the execution of the resulting subgoal $(q_{i1}(t_{i1}), \ldots, q_{ia_i}(t_{ia_i}))\theta_i$ does not succeed.[1]

The conjunction of (1), (2), (3), and (4a) is called the *(strong) path condition* for modification m. The conjunction of (1), (2), (3), and (4b) is the *regular path condition* for m, and the conjunction of (1), (2) and (3) the *weak path condition* for m.

Example 7. Let P be the append-program of Example 4 and let $P'=P.m$ be the program where the second clause is

append$([A|X], Y, [A|Z])$:- append(X, Y, Y).

The strong path condition for the modification m is that input d unifies with $([A|X], Y, [A|Z])$, d does not unify with $([], X, X)$, and the tracing of append

[1] For simplicity we do not discuss cuts here.

(X, Y, Y) and append(X, Y, Z) is different on d. A d satisfying these conditions is, e.g., $([1], [1], [1])$. □

To generate a d such that $P(d) \neq P.m(d)$, we form the path condition for m and try to generate an input satisfying it. For this, we need a formal way of describing path conditions. This is easy to do using the concepts of unification and universally quantified inequalities [LMM86, LM87]. An example should explain how this is done.

Example 8. The path condition in the previous example can be formalized as follows.

$$U = ([A|X], Y, [A|Z]) \wedge \forall X' : U \neq ([\,], X', X') \wedge (X \neq X \vee Y \neq Y \vee Y \neq Z).$$

Here U stands for the input d. The last conjunct comes from the requirement that the traces of append(X, Y, Z) and append(X, Y, Y) differ. □

The reason for introducing regular path conditions is that sometimes the strong path condition is unsatisfiable. For example, let p be defined as follows:

$$\text{p}(A', B') :\text{-} A' < B', q(A', B').$$
$$\text{p}(A, B) :\text{-} r(A, B).$$

The path condition for subgoal $r(A, B)$ altered to $r(A, A)$ is

$$(\forall A', B' : U \neq (A', B')) \wedge U = (A, B) \wedge \ldots$$

which clearly cannot be satisfied. An input reaching the end of the second clause necessarily unifies with the first clause, but fails in its body. The regular path condition contains the subformula

$$U = (A', B') \wedge (\neg(A' < B') \vee \neg q(A', B')),$$

which can be satisfied by letting A' and B' be integers and $A' \geqq B'$.

The technique we use is first to try the strong path condition. If that cannot be satisfied, we move towards the regular path condition by allowing some previous unifications to succeed but requiring that some subgoal in that clause fails. In our example this would mean including the subformula

$$U = (A', B') \wedge \neg(A' < B')$$

in the path condition. In this fashion the path condition is weakened, until it can be satisfied.

Weak path conditions arise in situations where we want to check backtracking behaviour of a program. We omit the discussion on this; in the sequel we concentrate on strong path conditions.

5. Generating inputs satisfying the path condition

Given a path condition, how do we generate an input satisfying it? Here we have to restrict our class of Prolog programs. A path condition can contain conditions of the form $q(t)$, where q is an arbitrary predicate. Generating inputs satisfying q is an unsolvable problem, as, e.g., the halting problem is reduced to it.

There are two ways out of this problem. We could try to generate inputs satisfying the subgoals just by running the subgoals. This alternative seems to be feasible in practice, but it is hardly amenable to an exact analysis.

The second way is to restrict ourselves to programs where the subgoals are easily analyzable. One such class (but by no means the only) is the class of *iterative programs*, defined in [SS86]. An iterative program for predicate p consists of clauses, where the last subgoal in each clause can be a recursive call and all other goals are calls of system predicates. For example the programs for append and member are iterative.

Given a conjunction of equations and universally quantified inequations, we collect first the equations and solve them by using (Robinson's) unification algorithm. This gives us a structure, possibly with free variables, representing the most general solution of the equations.

We then process the universally quantified inequations one by one. If for the inequation $\forall X$: $U \neq t(X)$ the right hand side matches the structure formed so far, we develop the structure by adding functors or atoms so that the inequation holds.

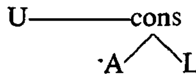
Example 9. Let p be defined by the clauses

$$\begin{array}{l} \mathrm{p}([\,]) \text{ :-} \ldots \\ \mathrm{p}([A', B'|L']) \text{ :-} \ldots \\ \mathrm{p}([A|L]) \text{ :-} B \text{ is } A+1, \mathrm{p}(L). \end{array}$$

and let m be the modification changing the last L in the third clause to a new variable, say X. The path condition for m is

$$U \neq [\,] \wedge (\forall A', B', L' : U \neq [A', B'|L']) \wedge B = A+1 \wedge U = [A|L] \wedge L \neq X^2$$

We start by considering the equations in this conjunction: $U=[A|L]$, $B=A+1$. We form a structure representing the value of U:



Next we check the inequations for U. The right hand side of the equation $U \neq [\,]$ does not match the above partially instantiated value of U, so we proceed to the next inequation, $\forall A', B', L'$: $U \neq [A', B'|L']$. The right hand side of this matches the above structure, so we have to ensure that $\forall B', L'$: $L \neq [B'|L']$. This can be done by instantiating L to the empty list.

The method outlined above is fairly easy to implement by brute force: we use a Prolog query, which first generates the structure from the equations and then tries possible alternatives for the free variables until a suitable case is found. A more refined method would also be quite simple to implement.

[2] Note that this inequation is not universally quantified; it comes from the requirement that the altered subgoal behave differently from the original one.

6. Concluding remarks

We have described a general method for producing complete sets of test data for Prolog programs. The method is based on the competent programmer assumption and on a theoretical result showing how one can concentrate on the local neighbourhood $M(P)$. The test cases were produced by forming path conditions for each modification and by solving them.

Several open problems remain. One is the exact class of programs for which the method can be made fully automatic. For iterative programs this can be done, but they probably do not form the largest such class. Another such class might be the programs without function symbols (Datalog).

Another problem is dealing with general programs. For those interaction with the user is necessary for succesful generation of test cases. How should the interaction be organized?

7. Acknowledgements

We wish to thank Tiina Häkkä for implementing a prototype of the system.

References

[BA82] BUDD, T. & ANGLUIN, D., Two Notions of Correctness and Their Relation to Testing. Acta Informatica 18 1982, pp. 31—45.

[Be84] BEIZER, B., Software System Testing and Quality Assurance. Van Nostrand Reinhold. USA 1984.

[Br80] BROOKS, M., Determining Correctness by Testing. Report No. STAN—CS—80—804, Computer Science Department, Stanford University. May 1980.

[DMLS78] DEMILLO, R. & LIPTON, R. & SAYWARD, F., Hints on Test Data Selection: Help for the Practicing Programmer. Computer Vol. 11 No. 4 April 1978, pp. 34—41.

[GG75] GOODENOUGH, J. & GERHART, S., Toward a Theory of Test Data Selection. IEEE Transactions on Software Engineering Vol. SE—1 No. 2 (1975), pp. 156—173.

[Ho86] HOWDEN, W., A Functional Approach to Program Testing and Analysis. IEEE Transactions on Software Engineering Vol. SE—12 No. 10 (1986), pp. 997—1005.

[LM87] LASSEZ, J-L. & MARRIOT, K., Explicit Representation of Terms Defined by Counter Examples. Journal of Automated Reasoning, 3 (1987), pp. 301—317.

[LMM86] LASSEZ, J-L. & MAHER, M. & MARRIOT, K., Unification Revisited. RC 12394. IBM-T. J. Watson Research Center Yorktown Heights, NY. November 1986.

[MCM83] MICHALSKI, R. & CARBONELL, J. & MITCHELL, T. (eds.), Machine Learning: an Artificial Intelligence Approach, Morgan Kaufmann, 1983.

[MCM86] MICHALSKI, R. & CARBONELL, J. & MITCHELL, T. (eds.), Machine Learning: an Artificial Intelligence Approach, Vol. II, Tioga 1986.

[MR89] MANNILA, H. & RÄIHÄ, K.-J., Automatic Generation of Test Data for Relational Queries. Journal of Computer System Sciences 38, 2 (April 1989), 240—258.

[My79] MYERS, G., The Art of Software Testing. Wiley. New York 1979.

[Pe86] PEREIRA, L., Rational Debugging of Prolog programs. Proceedings of the Third International Conference on Logic Programming, London 1986, pp. 203—210.

[Sh82] SHAPIRO, E., Algorithmic Program Debugging. Ph. D. Thesis, Yale University, 1982.

[SS86] STERLING, L. & SHAPIRO, E., The Art of Prolog. The MIT Press. Massachusetts 1986.