This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: Belli, F., Jack, O., Declarative Paradigm of Test Coverage, Software Testing, Verification and Reliability, Vol. 8 (1), 15-47

Declarative Paradigm of Test Coverage

FEVZI BELLI AND OLIVER JACK*

Universität-GH Paderborn, FB Elektrotechnik, FG Angewandte Datentechnik, D-33095 Paderborn, Germany

SUMMARY

Two facts about declarative programming prevent the application of conventional testing methods. First, the classical test coverage measures such as statement, branch or path coverage, cannot be used, since in declarative programs no control flow notion exists. Second, there is no widely accepted language available for formal specification, since predicate logic, which is the most common formalism for declarative programming, is already a very high-level abstract language. This paper presents a new approach exending previous work by the authors on test input generation for declarative programs. For this purpose, the existing program instrumentation notion is extended and a new logic coverage measure is introduced. The approach is mathematically formalized and the goal of achieving 100% program logic coverage controls the automatic test input generation. The method is depicted by means of logic programming; the results are, however, generally applicable. Finally, the concepts introduced have been used practically within a test environment. **© 1998 John Wiley & Sons, Ltd.**

KEY WORDS software testing; implementation based testing; logic programming; coverage measures; test input generation

1. INTRODUCTION

Programming paradigms developed in the past decades lead from a machine-oriented view to a problem-oriented one. The *imperative* paradigm requires *instructing* a machine on the way to solve the problem. The *functional* paradigm means describing a problem through a set of functions so that the programmer can concentrate on the problem instead of instructing the machine. The *logic* paradigm enables one to specify the problem (Kowalski, 1979). A logic program consists of facts and properties about a problem, but it is the task of the system to find the solution to the problem. Programming languages relying on the logic paradigm are also called *declarative*. In short, declarative programming (Padawitz, 1992) expresses *what* must be achieved and lets the system determine how, whereas imperative programming expresses *how* something is achieved and lets the reader determine what.

In software engineering two important issues are addressed by logic programming:

(1) formal specifications, including test specifications can be given by logic programs (Bougé *et al.*, 1986; Denney, 1991);

*Correspondence to: O. Jack, FB 14 Electrotechnik, FG Angewandte Datentechnik, Universität Gesamthochschule, Paderborn, D-33095 Paderborn, Germany

(2) rapid prototyping for requirements validation can be handled efficiently by logic programs (Leibrandt and Schnupp, 1984; Bobrow, 1985; Hekmatpour and Ince, 1988).

Logic programming, because of its very high abstraction level, is widely seen as a tool for theorem provers. Concerning software engineering, this puts logic programming in the position for proving correctness of imperative programs and for generating tests from logic specifications of imperative programs (cf. Denney, 1991). On the other hand, in the logic programming community there are hardly any well known activities for testing logic programs (Ducassé and Noyé, 1994).

Apart from the previous work of the present authors (Belli and Jack, 1995) there are two approaches to testing logic programs. Test data selection criteria for Prolog programs have been presented by Luo *et al.* (1992). The hidden control flow of a Prolog program is modelled and extracted and then a fault model is used for selecting test data which covers the control flow graph. This method is applicable only to logic programming systems with a certain computational model of transforming declarative programs into procedural programs.

Identifying classes of logic programs with decidable semantics is used for specification based testing by Ruggieri (1996). The work describes a sound treatment of testing for a subset of logic programs (of those programs with decidable semantics) but it lacks generality.

Structural software testing still relies on the imperative paradigm. Coverage measures are based on control flow and data flow. Structural testing of logic programs cannot be performed using conventional testing methods. Since the notion of control makes no sense for logic programs, none of the control flow-based methods is applicable. Data flow is expressed in a logic program completely differently from an imperative program; hence, conventional data flow-based methods are similarly inapplicable.

Structural testing requires an abstract model of the program under test, on which a coverage measure is defined. For an imperative program, such an abstract model could be its control flow graph and the measure could be coverage of all branches in the graph. To emphasize the differences, the following logic program is used:

```
connected(X, X, [X]).
connected(X, Z, [X|Tail]) :-
  arc(X, Y),
  connected(Y, Z, Tail).
```

The program above checks whether two nodes in a graph represented by *facts* arc(x, y) are connected or not and computes a connecting path. This program has no procedural elements. It can be read as follows.

(1) Any node X is connected to itself by path [X].
(2) Two nodes X and Z are connected by path [X|Tail] if
- there is an arc from X to some node Y and
- Y is connected to Z by path Tail.

The problems with structural testing of a logic program are now obvious: *What* is an

appropriate *abstract model* of a logic program for building a coverage measure and *how* is this *coverage measure* defined?

The problem cannot be solved solely by considering a computational model of logic programming, and transforming the logic program into a procedural one, and then extracting its control flow for a conventional coverage test. Remember, as mentioned at the beginning of this section, that in logic programming the system has to determine how a connecting path is computed. One system may perform a breadth first search by first inspecting all nodes Y connected to Z and then inspecting to see if there is an arc from X to Y; another system may perform depth first search by first inspecting all nodes Y having an arc from X and being connected to Z. The resulting control flow graphs are then quite different.

This paper bridges the gap between the use of logic programming as a *verification and testing tool* and the testing of a logic program *itself*. The view taken here is of logic programming as a programming language for the *engineer* rather than as an artificial intelligence tool for the *scientist*.

A declarative coverage notion for logic programs (Belli and Jack, 1995) is reviewed and extended and a method for the generation of covering test input sets is presented. The generation method uses declarative instrumentation of logic programs based on the present authors' previous work (Belli and Jack, 1993). This instrumentation scheme is based on types and modes introduced for run-time checking of logic programs (Dart and Zobel, 1992) and modified in order to control test input generation.

The paper is organized as follows. Section 2 gives a brief description of logic programming and lists the basic definitions used throughout the paper. Sections 3 and 4 review the declarative coverage notion and explain the formal instrumentation scheme for logic programs. Section 5 describes the novel test input generation method. Section 6 contains the description of a test environment which incorporates the test input generation method. In Section 7, some practical results are discussed, and Section 8 contains concluding remarks.

2. PRELIMINARIES

In this section, the notions and vocabulary used throughout the paper are introduced. Consider the example program from the previous section:

```
connected(X, X, [X]).
connected(X, Z, [X|Tail]) :-
  arc(X, Y),
  connected(Y, Z, Tail).
```

Let a graph be given by the *facts*

```
arc(a, b).
arc(b, c).
arc(b, d).
arc(d, a).
```

To check whether node a is connected to node d the *goal*

```
?- connected(a, d, Path).
```

is given to run the program. The logic programming system proceeds as follows to reach the goal. It searches the *program clauses* for a matching *head*. The *literal* connected(a, d, Path) of the goal is compared to the literals connected(X, X, [X]) of the first program clause and connected(X,Z,[X|Tail]) of the second program clause. If there are *substitutions* of *variables* through *terms* for these literals such that the literals can be made identical, then the literals are *unified* and a *resolvent* of the two clauses is generated.

For comparison, first the *predicate symbols* connected must be identical, then the *arguments* of the literal are compared. The arguments of the goal clause are a, d and Path; the arguments of the first program clause are X, X and [X]. The X and Path are *variables*; variables are denoted by strings beginning with an upper case letter. The arguments a and d are *constants*; constants are denoted by strings beginning with a lower case letter. The argument [X] is a *structure*. An argument of any of the above types is called a *term*. Comparison shows that the goal literal and the head of the first program clause do not match, because the program clause forces the first two arguments to be identical, while the goal has these arguments different.

The goal matches the head of the second program clause (such a clause is called a *matching clause*), and the *instance*

```
connected(a, d, [a|Tail]) :-
  arc(a, Y),
  connected(Y, d, Tail).
```

of this program clause is generated by applying the unifying substitution

{X/a, Z/d, Path/[a|Tail]}

to it (see Table I).

The head of this instance of the second program clause is discarded and the body is processed as the goal

```
?- arc(a, Y), connected(Y, d, Tail).
```

This new goal is called the *resolvent* of the original goal and the second program clause.

Matching first finds the program clause arc(a, b) and generates the instance

```
?- arc(a, b), connected(b, d, Tail).
```

Table I. Unification of two literals

Literal	Substitution	Instance
`connected(a, d, Path)`	`Path/[a|Tail]`	`connected(a, d, [a|Tail])`
`connected(X, Z, [X|Tail])`	`X/a, Z/d`	`connected(a, d, [a|Tail])`

Processing continues with the resolvent

```
?- connected(b, d, Tail).
```

which, as above, matches the head of the second program clause. The instance is

```
connected(b, d, [b|Tail]) :-
  arc(b, Y),
  connected(Y, d, Tail).
```

and the actual resolvent is

```
?- arc(b, Y), connected(Y, d, Tail).
```

Now, the program clause arc(b, d) matches and processing continues with the resolvent

```
?- connected(d, d, Tail).
```

This goal matches the first program clause via the unifying substitution {Tail/[d], X/d}, and as the resolvent, the *empty clause* is achieved. This terminates processing and the variable substitutions are traced back via the resolvents to the original goal (now indexed by the depth of the resolution, goal starts with index 0 and matching clause with index 1):

$$\{X_3/d,\ Tail_2/[d]\}$$
$$\circ\{X_2/b,\ Z_2/d,\ Tail_1/[b|Tail_2]\}$$
$$\circ\{X_1/a,\ Z_1/d,\ Path_0/[a|Tail_1]\}$$
$$= \{X_1/a,\ Z_1/d,\ Path_0/[a|[b|[d]]]\}$$

The term [a|[b|[d]]] is written [a, b, d]. This is the list notation known from the programming language Lisp. Restricting to the variables occurring in the original goal, the result of the program run is

```
Path = [a, b, d]
```

Looking closer shows that the last resolvent also matches the second program clause, resulting in the resolvent

```
?- arc(d, Y), connected(Y, d, Tail).
```

which could be further processed by matching the program clause arc(d, a). Then the resolvent is

```
?- connected(a, d, Tail).
```

which looks much like the initial goal. Thus, an alternative result is

```
Path = [a, b, d, a, b, d]
```

Table II. Types of clause

Clauses	
A.	fact: A=head
A:- $B_1, \ldots, B_n$.	rule: A=head, $B_1, \ldots, B_n$=body
?- $B_1, \ldots, B_n$.	goal: $B_1, \ldots, B_n$=body

Summarizing and putting it in a formal way, logic programming is based on *first-order predicate logic* and expresses knowledge about a problem in the form of *clauses*, which are divided into three types: *facts, rules* and *goals* as depicted in Table II. The components A, B_i of the clauses are called *literals* or *atomic formulae*. The literal to the right of the symbol ':-' is the *head* of a clause and the symbols $B_1, \ldots, B_n$ to the right of :- or ?- constitute the *body* of a clause. To avoid confusion, a clause H :- B may be enclosed in parentheses (H :- B). The body of a clause is a conjunction of the literals $B_1, \ldots, B_n$.

A literal is either an *atom* a or a structure $p(t_1, \ldots, t_m)$, where a and p are *predicate symbols* and $t_1, \ldots, t_m$ are *terms*, which are divided into the three types shown in Table III. The set of terms is denoted by $\mathcal{L}(\mathcal{F}, \mathcal{V})$, where $\mathcal{F}$ is the set of *functors* and $\mathcal{V}$ is the set of *variables*. The set of literals is denoted by $\mathcal{L}(\mathcal{R}, \mathcal{F}, \mathcal{V})$, where $\mathcal{R}$ is the set of predicate symbols. The variable-free terms are denoted by $\mathcal{L}(\mathcal{F})$ and the variable-free literals are denoted by $\mathcal{L}(\mathcal{R}, \mathcal{F})$. The predicate symbols have associated with them their number n of arguments, i.e. their *arity*. Since one symbol f can be used with different arities, function symbols and predicate symbols are denoted in the form of *functors f/n*. The set of clauses in a program which have a head literal with functor *p/n* is called the *definition of the predicate p/n*, e.g. the two clauses with functor connected/3 are the definition of the predicate connected/3 and the four clauses with functor arc/2 are the definition of the predicate arc/2.

A *substitution* is a finite mapping σ: $\mathcal{V} \rightarrow \mathcal{L}(\mathcal{F}, \mathcal{V})$ from the set of variables to the set of terms, which is identified with a finite set $\{X_1/t_1, \ldots, X_m/t_m\}$ of ordered pairs X_i/t_i, where X_i are distinct variables and t_i are terms and each t_i is distinct from X_i, $i = 1, \ldots, m$. An ordered pair X/t is called a *binding* of X to t. Substitutions are denoted post-fix, $t\sigma := \sigma(t)$. The result of the application $t\sigma$ of σ to a term t is the term t' obtained by simultaneously replacing each occurrence of a variable X where $X/s \in \sigma$ by s. The term t' is called the *instance* of t by σ. An instance is *ground* if there are no occurrences of variables in it. The set of all ground instances of a term t is denoted by $\mathcal{G}(t)$. The above instance definition extends to sets of terms, clauses and sets of clauses in a canonical way.

The instance relation is denoted by '$\leq$', i.e. $t \leq s$ if t is an instance of s. By '$\leq$', the set of terms and the set of clauses can be factored into equivalence classes. Two

Table III. Types of term

Terms $\mathcal{L}(\mathcal{F}, \mathcal{V})$	
X	variable
c	constant
$f(t_1, \ldots, t_k)$	structure: $t_1, \ldots, t_k$: terms

terms t_1, t_2 or clauses C_1, C_2 are *variants* if $t_1 \leq t_2$ and $t_2 \leq t_1$, respectively $C_1 \leq C_2$ and $C_2 \leq C_1$. The variant relation obviously is an equivalence relation; it is denoted by '$\simeq$'. If $t_1 \leq t_2$ and not $t_1 \simeq t_2$, i.e. t_1 is a *real* instance of t_2, this is denoted by $t_1 < t_2$. A set M of terms or clauses is *unifiable* if there exists a substitution σ such that $M\sigma$ is a singleton. A *most general unifier* (mgu), which exists for any unifiable set M, is a substitution μ such that for any other unifier σ for M, there exists a substitution θ with $\sigma = \theta \circ \mu$. Compositions $\theta \circ \sigma$ of substitutions are denoted $\sigma\theta := \theta \circ \sigma$. The set of clause instances induced by an initial goal, i.e. the result of unifying the initial goal with the head of a matching clause and applying the mgu to this clause, is called the set of *top level instances*.

For the semantics, consider the *Herbrand universe* $\mathcal{H}_{\mathcal{L}} := \mathcal{L}(\mathcal{F})$ and *Herbrand base* $\mathcal{B}_{\mathcal{L}} := \mathcal{L}(\mathcal{R}, \mathcal{F})$ of a program. It is assumed that the first-order language $\mathcal{L}(\mathcal{R}, \mathcal{F}, \mathcal{V})$ for a program P is given by the symbols occurring in the program clauses of P. Hence, $\mathcal{H}_P$ stands for the Herbrand universe and $\mathcal{B}_P$ for the Herbrand base of a program. The semantics of a program P is the set $\mathcal{M}^+(P)$ of ground atomic formulae which are logical consequences of P. $\mathcal{M}^+(P)$ is called the *minimal Herbrand model* of P.

The minimal Herbrand model of a program is computationally handled by the *resolution calculus* (Lloyd, 1987) which *computes an answer* to a goal and a program by subsequently unifying goals and heads of program clauses (using mgu) until the empty clause is achieved. The process of computing an answer to a goal is called *SLD-resolution*, which stands for Selective Linear resolution for Definite clauses (Lloyd, 1987). The set of atomic formulae, which can be computed for a program P by SLD-resolution is the *success set* of P; it is denoted by $S^+(P)$. Since non-ground formulae can be answers to a goal, $S^+(P) \neq \mathcal{M}^+(P)$, but $\mathcal{M}^+(P) \subseteq S^+(P)$ and $\mathcal{G}(S^+(P)) = \mathcal{M}^+(P)$.

3. TEST COVERAGE OF LOGIC PROGRAMS

Covering a program by a set of test inputs means to consider some abstract model of the program and select test inputs which address certain properties of that abstract model. Such an abstract model of a program could be for example its control flow graph and the property could be the set of branches in the graph. Then the program is covered if a test input set is selected which, when run on the program, directs control flow to each of the control flow branches. Control-flow-based as well as data-flow-based test coverage has been studied and used widely (Duran and Ntafos, 1984; Rapps and Weyuker, 1985; Prather and Myers Jr, 1987). However, such coverage notions are not applicable to logic programming, since a logic program does not express control flow and it expresses data flow only implicitly.

In logic programming, unification is used to extract specific information out of a program which asserts general information about a problem. In particular, the program clauses are instantiated via goals, leading to specific instances of these clauses. Repeating this process, concrete information is extracted from a program. A test coverage notion for logic programs must consider this computational model. The coverage notion is based on the abstract model of the program instances induced by goals and resolution. For this, a relation on the set of program instances will be defined. The concept behind it is *anti-unification*. Anti-unification can be used to generalize information contained in a set of formulae. This generalization means finding, for a set M of formulae, a formula that expresses what the formulae in M have in common. This kind of generalization has been

used in theorem proving (Boyer and Moore, 1979) and inductive inference (Angluin and Smith, 1983).

Because anti-unification is not broadly known, a brief overview is given in the following paragraphs. The full theory can be found for example in the work of Lassez *et al.* (1988). For technical reasons, the view is restricted to terms, but all the results can be applied to Horn clauses as well. First, the set of terms is extended by a least element $\perp$. Now, the instance relation is a partial ordering on the set of terms. Most general unification of a (unifiable) set M of terms corresponds to the greatest lower bound of M under $\leq$, i.e. if μ is an mgu for M then $M\mu$ is the greatest lower bound of M. In the case when M is not unifiable, $\perp$ is the greatest lower bound of M.

Definition 1

Let M be a non-empty set of terms. A term t is an *anti-instance* of M if for all $s \in M$, $t \geq s$. A term t is a *least common anti-instance* of M, denoted $t = \text{lca}(M)$, if

(1) t is an anti-instance of M;
(2) for each anti-instance t' of M, $t' \geq t$.

From the definition of lca it follows that all lca's of a set M of terms are instances of each other. Thus an lca is unique modulo renaming.

The concept of the least common anti-instance is inspired by the unification concept of logic programming: unification leads to the greatest lower bound of terms under the instance ordering, lca are least upper bounds (see Figure 1). For this reason, the computation of an lca is called anti-unification.

An important result is the following theorem, the proof of which can be found in the work of Lassez *et al.* (1988).

Theorem 1: Anti-unification

Let M be a non-empty set of terms. Then M has an lca which is unique modulo variants.

Anti-unification provides a mathematical formalization of the concept of generalization. The Anti-unification Theorem states that the set of terms, extended by $\perp$, together with the instance relation, forms a complete lattice. For each non-empty set of terms, there

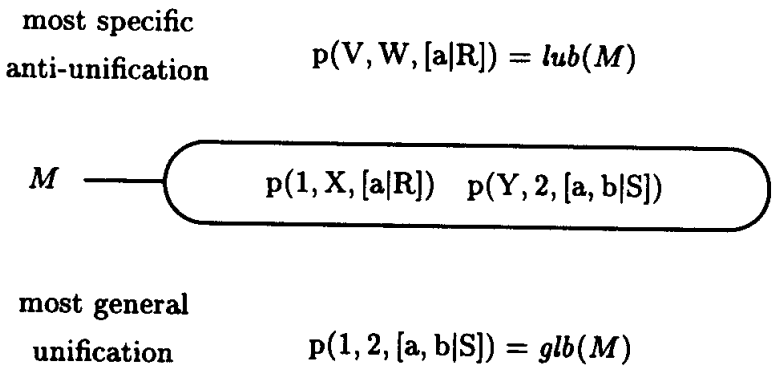

Figure 1. Unification and anti-unification.

exist a least upper bound (by the anti-unification Theorem), and a greatest lower bound (since either the set is unifiable and the mgu is the greatest lower bound or the greatest lower bound is $\perp$). There are efficient algorithms for the computation of the lca of a finite set of terms (see the work of Huet, 1976).

The concept of generalization reviewed above will be used in the following to define the test coverage for logic programs. For a logic program and a goal, unification is used to specialize information contained in the program. The goal is unified with some matching program clause, and an instance of that program clause (the corresponding top level instance) is obtained. Further resolution steps will then compute an answer to the goal. Since goals are the only way to run a logic program, 'covering' the top level instances means to cover the program. Hence, the abstract model of a logic program to be considered for test coverage is the set of top level instances induced by the set of test inputs (Belli and Jack, 1995). The property of interest for this set of instances is its most specific generalization, which is given by the lca. It expresses what all the top level instances have in common, which in fact can be interpreted as how much of the (general) information given in the program is accessed by the test inputs. Here 'how much' is subject to the term ordering. Figure 2 depicts the scheme of obtaining an instance of a program by a set of test inputs. The program is then *covered* by the test inputs if the corresponding program instance is equivalent to the program. In short,

(1) specializing the program by test inputs, then generalizing this specialization gives the test coverage of the program;
(2) if, from the specialization given by a set of test inputs, the whole program can be reconstructed via generalization, then the program is covered.

This is formally expressed as follows.

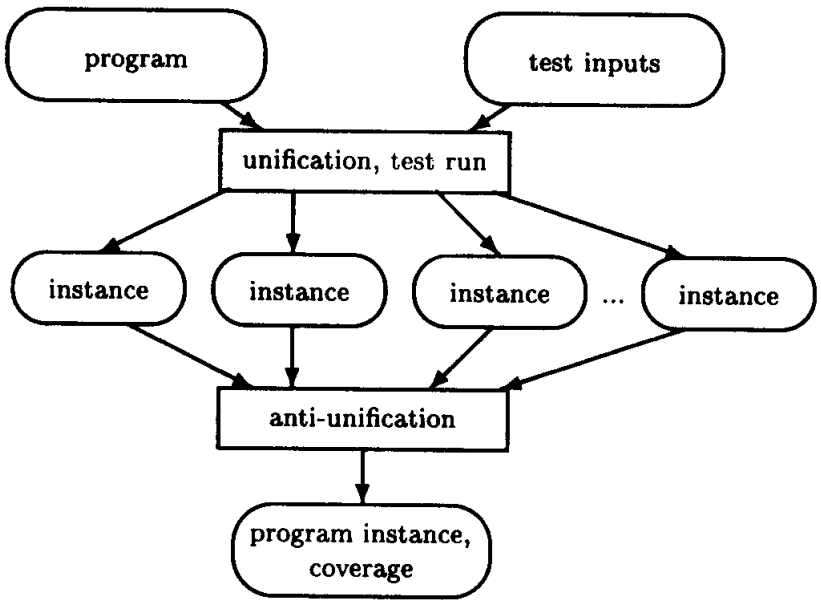

Figure 2. Test coverage for logic programs.

Definition 2

Let $T = \{:\text{-}G_1, \ldots, :\text{-}G_m\}$ be a finite set of atomic goals. Let H :- B be a program clause. The *instantiation set* of H :- B by T is

$$\begin{aligned}&[\![H \text{ :- } B]\!]_T := \\ &\quad \{(H \text{ :- } B)\mu : \exists \text{ :-}G \in T \text{ such that} \\ &\quad G \text{ and } H \text{ are unifiable with mgu } \mu\}.\end{aligned}$$

Let $P = \{C_1, \ldots, C_k\}$ be a program. The *instantiation set* of P by T is

$$[\![P]\!]_T := \bigcup_{i \in \{1,\ldots,k\}} [\![C_i]\!]_T.$$

Of course, from testing a program, no more information than is already contained in the program can be gained. Hence, for each finite set of atomic goals T and each program clause $C = (H \text{ :- } B)$, either $[\![C]\!]_T = \emptyset$ or lca $([\![C]\!]_T) \leq C$.

Definition 3

Let $T := \{:\text{-}G_1, \ldots, :\text{-}G_m\}$ be a finite set of atomic goals. Let $R := \{C_1, \ldots, C_q\}$ be the definition of the predicate p/n, in a program P.

(1) T is a *cover for a program clause* C_i, or T covers C_i, if lca $([\![C_i]\!]_T) \simeq C_i$.
(2) T is a *cover for the predicate* p/n, or T covers p/n, if T is a cover for each $C \in R$.
(3) T is a *cover for a program* P, or T covers P, if T is a cover for each predicate in P.

The above coverage notion defines a structural coverage, but it is not based on control flow. It is more related to data flow (Rapps and Weyuker, 1985), since instances of clauses, i.e. substitutions of variables by terms, are central to the coverage notion. It also reflects the logical structure of the program P to be tested, since by clause instances its Herbrand base $\mathcal{B}_P$ is considered (in the case of ground instances). For illustration, consider the following example.

Example 1

Let P be the following program:

```
is_list([]).                          % C1
is_list([_|Tail]) :- is_list(Tail).   % C2
```

Let

T = {?-is_list([]), ?-is_list([a]), ?-is_list([a, b])}.

The sets of top level instances of P are then

$[\![C1]\!]_T$ = {is_list([])},
$[\![C2]\!]_T$ = {is_list([a]) :- is_list([]); is_list([a, b]) :- is_list([b])}.

The least common anti-instances are

lca($[\![C1]\!]_T$) = is_list([])
lca($[\![C2]\!]_T$) = is_list([a|Tail]) :- is_list(Tail).

So the clause C1 of P is covered, but clause C2 is not.

This is information about 'how much' of a clause is covered by a set of test inputs, i.e. which instance of the clause is covered. The following definition states this formally.

Definition 4

Let C be a program clause and T be a finite set of goals. T is a cover for the instance C' of C, or T covers the instance C' of C if the lca of T is a variant of C'. The instance C' of C covered by T is called the *coverage* of C by T. The coverage of a program P by T is the set of covered instances of the clauses of P.

Definition 4 essentially means that if T is a cover for an instance C' of a clause C, where $C' < C$, then, loosely speaking, T covers C *up to* C'. From the instance C' of C covered by T, information can be gained for extending the instance covered. Here, extending the instance is subject to the instance ordering $\leq$. In Example 1 the instance

is_list([a|Tail]) :- is_list(Tail).

of clause

is_list([_|Tail]) :- is_list(Tail).

was covered. From the definition of the lca, it can be seen that choosing any additional test input

?- is_list([*x*|*tail*]),

where x is different from the constant a and *tail* is any term, will lead to a cover of the clause.

The coverage notion has, of course, the property that augmenting a finite test input set T_0 by an additional test input ?-G does not decrease the coverage, i.e. for a program $P = \{C_1, \ldots, C_m\}$ with

$$\mathrm{Cov}_0 := \{\mathrm{lca}([\![(C_i)]\!]_{T_0}) : i = 1, \ldots, m\}$$

and

$$\mathrm{Cov}_1 := \{\mathrm{lca}([\![(C_i)]\!]_{T_1}) : i = 1, \ldots, m\}$$

where $T_1 := T_0 \cup \{?\text{-}G\}$, it is true that

$$\text{lca}([\![(C_i)]\!]_{T_0}) \leq \text{lca}([\![(C_i)]\!]_{T_1}) : i = 1, \ldots, m$$

This follows immediately from Definitions 1 and 2.

For any program there is a cover which consists of the heads of its program clauses. Consider a program which appends two lists.

```
append([], L, L).
append([H|T], L, [H|T1]) :- append(T, L, T1).
```

The first clause expresses that appending any list L to the empty list [] yields the list L. The second clause expresses that appending any list L to a non-empty list yields a list which has the same head H as the first list and a tail T1 which is the result of appending L to the first list's tail T. A cover for this program is

```
{?- append([], L, L), ?- append([H|T], L, [H|T1])}.
```

Such a cover is called a *trivial cover*. A trivial cover does not extract any specific information out of a program, e.g. in the above example there are no concrete lists specified to be concatenated.

A test input set for a program P is called *non-trivial* if it contains only goals ?-G such that there exists no clause (H :- B) in P with $G \neq H$. A goal from a non-trivial test input set for P is called a *non-trivial test input* for P. As a consequence, non-trivial test inputs for predicates consisting only of clauses with ground heads, must be non-ground. In many cases, program clauses with ground heads are used to represent *explicit* data, e.g. facts of the form arc(a, b) representing an edge in a directed graph. 'Testing' such explicit data is often not useful.

Program clauses with non-ground heads are interesting for testing, since they contain general knowledge about the relation they encode. A special case of a non-trivial test input for non-ground clauses is a ground goal. Next, the question of whether there always exists a cover of ground goals for an arbitrary program is investigated. Fortunately this question can be answered with 'yes', provided that a certain (reasonable) condition holds.

Theorem 2

Let P be a program such that its Herbrand universe $\mathcal{H}_P$ contains at least one constant and one n-ary function symbol ($n > 0$). Then for P there exists a cover T such that T contains only ground goals.

Proof

It will be shown that $\text{lca}([\![C]\!]_{\mathcal{G}(\{?\text{-}H\})}) \simeq C$, i.e. $\text{lca}([\![C]\!]_{\mathcal{G}(\{?\text{-}H\})}) \leq C$ and $C \leq \text{lca}([\![C]\!]_{\mathcal{G}(\{?\text{-}H\})})$ (remember that $\mathcal{G}(\{?\text{-}H\})$ is the set of ground instances of ?-H).

Let $C = (H\text{:-}\ B)$ a clause in P. Consider the set $\mathcal{G}(\{?\text{-}H\})$ of ground instances of $\{?\text{-}H\}$. Since $[\![C]\!]_{\mathcal{G}(\{?\text{-}H\})}$ contains only instances of C,

$$\text{lca}([\![C]\!]_{\mathcal{G}(\{?\text{-}H\})}) \leq C.$$

$\mathcal{H}_P$ contains one constant, say c, and one n-ary function symbol, say f, with $n > 0$. Without loss of generality, let $n = 1$. Hence,

(1) for each variable X occurring in H, there exist $t, s \in [\![C]\!]_{\mathcal{G}(\{?\text{-}H\})}$ and substitutions σ and ρ, such that $t = C\sigma$ where $X/c \in \sigma$ and $s = C\rho$ where $X/f(c) \in \rho$. It follows that $\mathrm{lca}([\![C]\!]_{\mathcal{G}(\{?\text{-}H\})})$ is of the form $C\theta$, where θ is a substitution such that $X/X' \in \theta$ for a variable X';

(2) for any pair of distinct variables X and Y occurring in H, there exist $s, t \in [\![C]\!]_{\mathcal{G}(\{?\text{-}H\})}$ and substitutions σ and ρ, such that $t = C\sigma$ where $X/c \in \sigma$ and $s = C\rho$ where $Y/f(c) \in \rho$.

From (1) and (2) it follows that $C \leq \mathrm{lca}([\![C]\!]_{\mathcal{G}(\{?\text{-}H\})})$.

Definition 3 provides the general notion of test coverage for logic programs. It defines the *qualitative* coverage notion in the sense of a decidable relation on the set of programs and test input sets. Definition 4 gives the *quantitative* coverage notion in the sense of defining the extent of testing with respect to the instance ordering $\leq$ on the set of program clauses. Theorem 2 shows that any program containing non-ground clause heads indeed possesses a non-trivial cover. A consequence is, that, using the coverage notion given above, any program can be covered by test inputs, in contrast to traditional coverage notions for imperative programs (e.g. branch coverage), where 'dead' code cannot be accessed. The reason is that the coverage notion just described refers to the program's logic. But Theorem 2 provides no feasible constructive method for the generation of such a cover. Essentially it states that *exhaustive testing* provides a cover.

Defining an instance ordering on the set of terms, and considering least upper bounds of sets of terms under the instance ordering, has also been used for optimization of logic programs (Marriott *et al.*, 1990). Their approach was to generate a *most specific version* of a logic program. This is obtained by replacing the clauses C of a program P by specific instances C', such that the resulting program P' has the same success set as P, i.e. $S^+(P) = S^+(P')$, but a possibly increased set of finitely failed goals. Hence P' can be more efficient than P since non-successful derivations may be detected more quickly.

From the coverage *notion* (Definitions 3 and 4), a traditional coverage *measure* can be defined in a straightforward fashion. Let P be a program consisting of n clauses, and denote the set of goals for P by *Goals*. Let $T \subseteq Goals$ be a set of test inputs for P. Then

$$\mathrm{cv}_P^{(c)} : \mathcal{P}(Goals) \to \mathbb{R}^+, \quad \mathrm{cv}_P^{(c)}(T) := \frac{|\{C \in P : T \text{ covers } C\}|}{n}$$

is a coverage measure. Of course the program P is covered, within the meaning of Definition 3, if $\mathrm{cv}_P^{(c)}(T) = 1$. The coverage measure $\mathrm{cv}_P^{(c)}$ is related to the clauses of the program P, indicated by the superscript '$^{(c)}$'. It measures the relative number of clauses covered by T. Similarly, a coverage measure $\mathrm{cv}_P^{(r)}$ can be defined, which is related to the predicates of P, indicated by the superscript '$^{(r)}$'. Let $\{R_1, \ldots, R_k\}$ be the set of definitions of predicates in a program P, i.e. P defines k predicates. Notice that R_i is a set of program clauses for $i = 1, \ldots, k$, hence, R_i can be interpreted as a program. Then

$$\mathrm{cv}_P^{(r)} : \mathcal{P}(Goals) \rightarrow \mathbb{R}^+, \quad \mathrm{cv}_P^{(r)}(T) := \frac{\sum_{i=1}^{k} \lfloor \mathrm{cv}_{R_i}^{(c)}(T) \rfloor}{k}$$

is also a coverage measure. It measures the relative number of predicates covered by T.

Esssential for this work are Definitions 3 and 4, which, in contrast to the measures $\mathrm{cv}_P^{(c)}$ and $\mathrm{cv}_P^{(r)}$, can be utilized for *test input construction* instead of only *measuring* test coverage. Constructive methods for cover generation will be handled in Section 5.

4. DECLARATIVE INSTRUMENTATION

Although coverage computation is possible for an arbitrary program and an arbitrary set of goals, the *construction* of a non-trivial cover is not possible in general. Construction with no more knowledge than the program itself must refer to the minimal Herbrand model $\mathcal{M}^+(P)$. For coverage-oriented test input generation, the structure of $\mathcal{M}^+(P)$ must be examined. This requires the identification of the subset of terms of the Herbrand universe $\mathcal{H}_P$ which is given by the arguments of the atoms of $\mathcal{M}^+(P)$. For example, for the append program, the subset of $\mathcal{H}_P$ which is given by the arguments of append/3 should be the set of lists, i.e. terms of the form [], [a], [a, b], etc. Inferring these subsets from the program is not possible in general (Yardeni and Shapiro, 1991).

Additional information is required for automatic test input generation. Declarative instrumentation with types and modes of a logic program enables automatic test input generation which is capable of covering a program. For instrumentation purposes, types as subsets of the Herbrand universe $\mathcal{H}$ are introduced. Types have been investigated for logic programming for different purposes. The main motivation is run-time type checking (Dart and Zobel, 1992), approximating the denotational semantics of a logic program (Yardeni and Shapiro, 1991) and debugging (Naish *et al.*, 1989). For a comprehensive discussion of types and logic programming see the book by Pfenning (1992). In the following, a type scheme suitable for the needs of generating test inputs is described. It is derived from the work of Dart and Zobel (1992).

The first-order language, restricted to terms, is augmented by type functions and type variables. The set of *type function symbols* is denoted by $\mathcal{T}$, and the set of *type variable symbols* is denoted by $\mathcal{W}$, where $\mathcal{F} \cap \mathcal{V} = \mathcal{F} \cap \mathcal{W} = \mathcal{F} \cap \mathcal{T} = \mathcal{T} \cap \mathcal{V} = \mathcal{T} \cap \mathcal{W} = \mathcal{V} \cap \mathcal{W} = \emptyset$. The set $\mathcal{T}$ is grouped according to the arities in the same way as the set $\mathcal{F}$, i.e. the pair τ/n of a type function symbol τ with arity n is called the *type functor* τ/n. An element of $\mathcal{T}_0$ is called a *type constant*. A type language is denoted by $\mathcal{L}(\mathcal{F}, \mathcal{T}, \mathcal{V}, \mathcal{W})$.

For better readability and distinction from function and variable symbols, type function symbols are denoted by φ, ρ and type variable symbols by ν. In logic programs, lower case Greek letters are used for type variables and *italic* typeface for type functions. As usual in Prolog, type function symbols will begin with a lower case letter. Type terms are defined analogously to terms. The set of type terms is denoted by $\mathcal{L}(\mathcal{F}, \mathcal{T}, \mathcal{V}, \mathcal{W})$. The set $\mathcal{L}(\mathcal{F}, \mathcal{T}, \mathcal{W})$ of *pure type terms* consists of variable-free type terms. The set $\mathcal{L}(\mathcal{F}, \mathcal{T})$ of *ground pure type terms* consists of type variable-free pure type terms.

A type rule, or a type definition, is a rule of the form $\varphi(\nu_1, \ldots, \nu_n) \rightarrow \Phi$, where φ is an n-ary type function, $\nu_1, \ldots, \nu_n$ are distinct type variables and Φ is a set of pure type terms. In addition, no type variables different from $\nu_1, \ldots, \nu_n$ occur in Φ.

It is assumed that there are special type constants in $\mathcal{T}_0$. These are ϵ and η denoting the type $\emptyset$ and the Herbrand universe $\mathcal{H}$. It is further assumed that the constants of $\mathcal{H}$ are partitioned into subsets S_β called *base types*, i.e. $\mathcal{F}_0 = \cup_{i=1}^{n} S_{\beta_i}$ where the $= S_{\beta_i}$ are pairwise disjoint. A base type is referred by the base type symbol β. The set of base type symbols is denoted by $\mathcal{T}_B$. Examples of base types are: S_{integer}, which is the set of constants denoting integers; S_{float}, which is the set of constants denoting floating point numbers; and S_{atom}, which is the set of the other constants of $\mathcal{H}$. The availability of base types is not necessary for the development of the type system, but it is convenient as a basis for the definition of other types.

A type function φ/n is *defined* by a set T of type definitions if there exists a type rule $\varphi(\nu_1, \ldots, \nu_n) \rightarrow \Phi \in T$. A type term τ is *defined* by a set T of type rules if each type function occurring in τ is defined by T. It is assumed that each type function occurring in a set T of type definitions is either η, ϵ, a base type symbol β, or a type function defined by T. Each type function defined in T has exactly one type rule in T. By analogy with substitutions, applied to terms and formulae, a type replacement is defined for syntactic manipulations. Examples for type rules are $list \rightarrow \{[], [\eta|list]\}$ and $list(\nu) \rightarrow \{[], [\nu|list(\nu)]\}$, which define lists. The second is parametric; it defines lists of elements of type ν.

If the view is restricted to non-parametric type rules, these rules define a *regular term grammar*. Type constants are non-terminals and constants as well as structures in $\mathcal{H}$ are terminals. Such type rules are production rules, and the ground terms derivable from them are the type *defined* by the type rules. The semantics of types is defined via associated programs. For $\varphi(\tau_1, \ldots, \tau_n) \in \mathcal{L}(\mathcal{F}, \mathcal{T})$ defined in a set T of type definitions, where $\varphi(\nu_1, \ldots, \nu_n) \rightarrow \Phi \in T$, and $\sigma = \{\nu_1/\tau_1, \ldots, \nu_n/\tau_n\}$, the ground pure type term $\varphi_\tau(_1, \ldots, \tau_n)$ is associated with the definite program $P_{\varphi(\tau_1, \ldots, \tau_n)_T}$, defining the predicate $\varphi_{\tau_1, \ldots, \tau_n}$. Each $\vartheta \in \Phi\sigma$ is associated with a clause in the following way.

- If $\vartheta = f(\rho_1, \ldots, \rho_k)$, $f \in \mathcal{F}$ and $\rho_i = \tilde{\rho}_i(\omega_{i_1}, \ldots, \omega_{i_l})$ then the associated clause is

$$\varphi_{\tau_1, \ldots, \tau_n}(f(X_1, \ldots, X_k)) :- \tilde{\rho}_{1,\omega_{1_1}, \ldots, \omega 1_l}(X_1), \ldots, \tilde{\rho}k,\omega_{k_1}, \ldots, \omega k_l(X_k)$$

- If $\vartheta = \rho(\omega_1, \ldots, \omega_k)$ and $\rho \in \mathcal{T}$, then the associated clause is

$$\varphi_{\tau_1, \ldots, \tau_n}(X) :- \rho_{\omega_1, \ldots, \omega_k}(X).$$

$P_{T,\varphi(\tau_1, \ldots, \tau_n)}$ consists of the clauses for $\varphi_{\tau_1, \ldots, \tau_n}$ and the recursively associated clauses for each $\tilde{\rho}_i(\omega_1, \ldots, \omega_{i_l})$, respectively $\rho(\omega_1, \ldots, \omega_k)$. A special treatment is needed for the type symbols η and ϵ. Since η denotes the set of all terms $\mathcal{H}$, the associated clause is the fact $\eta(X)$. Since ϵ denotes the empty type, there is no clause associated with ϵ.

A ground pure type term τ can be seen as an instantiation of a type rule in a set T of type definitions, and the definite program P_{τ_T} will be used for the definition of type semantics. For a set T of type definitions, the type represented by a ground pure type term τ via T is denoted by $[\![\tau]\!]_T$, where

$$[\![\tau]\!]_T := \begin{cases} \{f(t_1, \ldots, t_n) : t_i \in [\![\tau_i]\!]_T\} & : \tau = f(\tau_1, \ldots, \tau_n), f \in \mathcal{F}_n \\ \{t : \varphi_{\tau_1, \ldots, \tau_n}(t) \in \mathcal{M}^+(P_{\varphi(\tau_1, \ldots, \tau_n)_T})\} & : \tau = \varphi(\tau_1, \ldots, \tau_n), \varphi \in \mathcal{T}_n \\ S_\beta & : \tau = \beta \in \mathcal{T}_B \end{cases}$$

This definition implies that for a $\varphi(\tau_1, \ldots, \tau_n) \in \mathcal{L}(\mathcal{F}, \mathcal{T})$

$$[\![\varphi(\tau_1, \ldots, \tau_n)]\!]_T = \bigcup_{\nu \in \Phi\sigma} [\![\nu]\!]_T$$

where $\varphi(\nu_1, \ldots, \nu_n) \rightarrow \Phi \in T$, $\sigma = \{\nu_1/\tau_1, \ldots, \nu_n/\tau_n\}$, and $\tau_1, \ldots, \tau_n$ are defined in T. Bodies of type rules represent unions of types; hence this type scheme allows additive polymorphism.

Parametric or polymorphic types (Mycroft and O'Keefe, 1984) can be considered regular in the sense that they are equivalent to regular types which will be defined as above without having type variables and only having type symbols (Dart and Zobel, 1992). But parametric types are a notational convenience. One can define a type and if there is a need, define types *btree(integer)*, *btree(float)*, etc.

5. COVERAGE GENERATION

In order to get a practical and tractable test input generation scheme, consider the initial definition of test coverage from Definitions 3 and 4 in Section 3. The following definition relates covers and coverage to types.

Theorem 2 states that (under reasonable conditions) there always exists a cover of ground goals. If types are considered, then there might not exist a cover of goals of the declared type. For illustration, consider the following example.

Example 2

Let

$$T = \{list(\nu) \rightarrow \{[], [\nu|list(\nu)]\}\}$$

be a set of type definitions. Let τ = p(*list*(a)). Using only symbols from T and τ, a cover for t = p([X|Y]) would be

$$C_0 = \{?\text{- p([a])}, ?\text{- p([a, a])}, ?\text{- p([[], a])}\}$$

But there does not exist a cover C for t such that for each $c \in C$, $c \in [\![\text{p}(list(\text{a}))]\!]_T$. Notice that $C_0 \ni$?- p([[], a]) $\notin [\![$?- p(*list*(a))$]\!]_T$.

Types restrict the set $\mathcal{G}(C)$ of ground instances of a clause C. As Example 2 shows, this has an impact on the coverage of typed programs. Types also restrict the *maximum coverage* (in terms of the instance relation) of a clause.

Definition 5

Let $\tau \in \mathcal{L}(\mathcal{F}, \mathcal{T})$ be defined by a set T of type definitions and t be a term such that $\mathcal{G}(t) \cap [\![\tau]\!]_T \neq \emptyset$. The term

$$\max_{\tau}(t) := \mathrm{lca}(\mathcal{G}(t) \cap [\![\tau]\!]_T)$$

is called a *maximum typed coverage* of t with respect to τ. Let $C = (H :- B)$, $H = p(t_1, \ldots, t_n)$ be a clause and $d = type(p/n) := p(\tau_1, \ldots, \tau_n)$ be a type declaration for p/n where τ_i is defined by T, $i = 1, \ldots, n$. The clause

$$\max_{p(\tau_1,\ldots,\tau_n)}(C) := \mathrm{lca}(\{(H :- B)\sigma : H\sigma \in \mathcal{G}(H) \cap [\![p(\tau_1, \ldots, \tau_n)]\!]_T\})$$

is called a *maximum typed coverage* of C with respect to d.

A maximum typed coverage is well-defined since by Theorem 1 each non-empty set of terms has an lca that is unique modulo variants. Hence, a maximum typed coverage is called *the* maximum typed coverage.

Example 3

As in the previous example, take $\tau = \mathrm{p}(list(\mathrm{a}))$ and $t = \mathrm{p}([\mathrm{X}|\mathrm{Y}])$. Since

$$[\![list(\mathrm{a})]\!]_T = \{[], [\mathrm{a}], [\mathrm{a}, \mathrm{a}], [\mathrm{a}, \mathrm{a}, \mathrm{a}], \ldots\}$$

$$\max_{p(list(\mathrm{a}))} = \mathrm{p}([\mathrm{a}|\mathrm{X}]) < \mathrm{p}([\mathrm{X}|\mathrm{Y}])$$

The maximum typed coverage $\max_\tau(C)$ defines the coverage of C that can be achieved by ground goals. Actually interesting is the *cover* for this coverage, i.e. the set G of ground goals such that $\mathrm{lca}([\![C]\!]_G) = \max_\tau(C)$.

Definition 6

Let R be a definition for the predicate p/n, let $type(p/n) := p(\tau_1, \ldots, \tau_n)$ be a type declaration for p/n with respect to a set of type definitions T, and $C = (H :- B)$ be a clause of P. Suppose $C' = H' :- B' < C$ is a coverage for C, i.e. clause C is not covered. The goal $?\text{-}G$ is a *strict extension* of C' if $G \in [\![p(\tau_1, \ldots, \tau_n)]\!]_T$ and $G \leq H$ and $G \not\leq H'$.

Example 4

Consider the append/3 predicate. Its type declaration is

$$type(\mathrm{append/3}) := \mathrm{append}(list(\eta), list(\eta), list(\eta))$$

Let

C' = append([A|B], B, [A|B]) :- append(B, B, B).

be a coverage of the second clause C of append/3,

C = append([A|B], C, [A|D]) :- append(B, C, D).

?-append([any], [any], [any|any]).

is a strict extension of C'.

Syntactically, a type declaration can be identified with a ground pure type term in the sense that *type(p/n)* := $p(\tau_1, \ldots, \tau_n)$ is identified with $p(\tau_1, \ldots, \tau_n)$. As mentioned before in Section 3, clauses can syntactically also be identified with terms. In this sense, a type declaration for a predicate *p/n* means that ground goals for *p/n* are intended to be in $[\![?\text{-}p(\tau_1, \ldots, \tau_n)]\!]_T$ where $\tau_1, \ldots, \tau_n$ are defined by T.

Now the problem of extending a coverage for a program clause can be reduced to the following problem.

> Given a ground pure type term τ defined by a set T of type definitions and two terms t_1, t_2, such that $t_2 \leq t_1$, find a ground term $t_3 \in [\![\tau]\!]_T$ with $t_3 \leq t_1$.

Strict extension then is finding such a t_3 with $t_3 \not\leq t_2$. Of course this is only possible if $\mathcal{G}(t_1) \cap [\![\tau]\!]_T \neq \emptyset$. For the existence of a cover for typed terms (or clauses), the condition of Theorem 2 should hold for each *corresponding type* of each sub-term. In Example 3, the corresponding type of the sub-term X of [X|Y] is 'a'. Formally, corresponding types of sub-terms are defined next.

Definition 7

Let $\tau \in \mathcal{L}(\mathcal{F}, \mathcal{T})$ be defined by T and let t be a term. $[\![\tau(t)]\!]_T := \mathcal{G}(t) \cap [\![\tau]\!]_T$ is called the *restriction* of $[\![\tau]\!]_T$ to t. Let s be a sub-term of t, i.e. s is a term occurring in t.

$$[\![\tau(t)]\!]_T(s) := \begin{cases} [\![\tau(t)]\!]_T & \text{if } t = s \\ \{\tilde{s} : f(\cdots\tilde{s}\cdots) \in [\![\tau(t)]\!]_T\} & \text{if } t = f(\cdots s\cdots) \end{cases}$$

is called the *corresponding type* of s for t with respect to τ.

Example 5

Consider $\tau = list(\alpha)$ where α is an arbitrary type, $t = [X, Y, Z|V]$. Then $[\![\tau(t)]\!]_T(Y) = [\![\alpha]\!]_T$ and $[\![\tau(t)]\!]_T(V) = [\![list(\alpha)]\!]_T$. For $[\![\tau(t)]\!]_T(s)$, there does not always exist a $\tilde{\tau}$ defined by T such that $[\![\tau(t)]\!]_T(s) = [\![\tilde{\tau}]\!]_T$. If T is

$$\alpha \rightarrow \{[], [a|\alpha], [b|\alpha]\}$$

$\tau = \alpha$, and $t = [X, Y]$, then $[\![\tau(t)]\!]_T(X) = \{a, b\}$. Notice, that if $t = [a, Y]$, then $[\![\tau(t)]\!]_T(a) = \{a, b\}$.

Notice that the corresponding type of a sub-term s for a term t with respect to τ is always the finite union of types given by pure type terms ϖ_i, $i = 1, \ldots, m$, occuring in the right-hand sides of the type rules defining τ. Hence,

$$[\![\tau(t)]\!]_T(s) = [\![\tilde{\varphi}(\nu_1, \ldots, \nu_n)]\!]_{\tilde{T}}$$

where

$$\tilde{T} = T \cup \{\varphi(\nu_1, \ldots, \nu_n) \rightarrow \{\varpi_1, \ldots, \varpi_m\}\}$$

Definition 8

For a ground pure type term τ, a term t and sub-term s of t, the right-hand side of the type rule defining $[\![\tau(t)]\!]_T(s)$ is denoted by $\tau[t](s)$.

From the above definition, it is clear that $[\![\tau(t)]\!]_T(s) = [\![\tau[t](s)]\!]_T$. For the rest of this text, it is assumed that when a corresponding type of a sub-term of a term is encountered, the underlying set of type definitions is augmented by the type rule according to Definition 8. Also, for the rest of this text, two sub-terms s_1 and s_2 occurring in different positions in a term t and neither s_1 is a sub-term of s_2 in t nor s_2 is a sub-term of s_1 in t, are called *distinct*. For example, if $t = f(X, X)$, then X as the first argument of t is distinct from X as the second argument of t. The following lemma exposes the conditions under which there exists a strict extension for a coverage.

Lemma 1: Strict extension

Let $\tau \in \mathcal{L}(\mathcal{F}, \mathcal{T})$ and let $t_1 \geq t_2$ be terms sharing no common variables such that $[\![\tau(t_1)]\!]_T \neq \emptyset \neq [\![\tau(t_2)]\!]_T$. Let σ be a substitution such that $t_1\sigma = t_2$. There exists a term $t_3 \in [\![\tau]\!]_T$ with $t_3 \leq t_1$ and $t_3 \not\leq t_2$ if and only if one of the following conditions holds:

(1) there exists an $X/\tilde{u} \in \sigma$ such that there exist two distinct variable sub-terms u_1, u_2 of $\tilde{u}$ where $u_1 = u_2 = Y$ and there exist $s_1 \in [\![\tau(t_1)]\!]_T(u_1)$, $s_2 \in [\![\tau(t_1)]\!]_T(u_2)$ with $s_1 \neq s_2$;

(2) there exists $\{X/\tilde{u}, Y/\hat{u}\} \subseteq \sigma$ such that there exist two variable sub-terms u_1 of $\tilde{u}$, u_2 of $\hat{u}$ where $u_1 = u_2 = Z$ and there exist $s_1 \in [\![\tau(t_1)]\!]_T(u_1)$, $s_2 \in [\![\tau(t_1)]\!]_T(u_2)$ with $s_1 \neq s_2$;

(3) there exists an $X/\tilde{u} \in \sigma$, where $\tilde{u}$ is not a variable, and for some non-variable sub-term $\tilde{s}$ of $\tilde{u}$ there exists an $\hat{s} \in [\![\tau(t_1)]\!]_T(\tilde{s})$ having a different functor than $\tilde{s}$.

Proof

$\Rightarrow$ *Proof by structural induction.* The base case, where t_1 is a variable and t_2 is a variable or constant or t_1 and t_2 are constants, is trivial. Now let t_1 be a variable and t_2 be an arbitrary term; let $t_3 \leq t_1$, $t_3 \not\leq t_2$ and $t_3 \in [\![\tau]\!]_T$. Then t_3 must differ in the functor of one non-variable sub-term s of t_2, or there are two occurrences s_1, s_2 of the same variable in t_2 and in t_3 there are two different sub-terms u_1, u_2 at the corresponding positions. In the first case, $[\![\tau(t_2)]\!]_T(s)$ contains at least two terms which have a different

functor. In the second case, $[\![\tau(t_2)]\!]_T(u_1) \cup [\![\tau(t_2)]\!]_T(u_2)$ contains at least two elements. Now let t_1 be a structure $f(t'_1, \ldots, t'_n)$. Then, since $t_2 \leq t_1 \geq t_3$, $t_2 = f(t''_1, \ldots, t''_n)$ and $t_3 = f(t'''_1, \ldots, t'''_n)$. Since $t_3 \not\leq t_2$, there are two cases:

(a) there exists a $k \in \{1, \ldots, n\}$ such that $t'''_k \not\leq t''_k$. Then by the induction hypothesis, one of the conditions 1–3 holds;
(b) for all $k \in \{1, \ldots, n\}$ $t'''_k \leq t''_k$. This can only be true if there exists an occurrence u_1 of a variable Z in t''_i and u_2 of the same variable Z in t''_j where $i \neq j$, $i = 1, \ldots, n$, $j = 1, \ldots, n$, such that the corresponding sub-terms s_1 of t'''_i and s_2 of t'''_j are different terms. In this case, since t_3 is ground, $[\![\tau(t_2)]\!]_T(u_1) \cup [\![\tau(t_2)]\!]_T(u_2)$ contains at least two elements s_1, s_2 where $s_1 \in [\![\tau(t_2)]\!]_T(u_1)$ and $s_2 \in [\![\tau(t_2)]\!]_T(u_2)$.

$\Leftarrow$ *Obvious*. Simply take t_3 as t_2, replace the sub-term u_1 by s_1 and u_2 by s_2 (cases 1 and 2), or replace the sub-term $\tilde{s}$ by $\hat{s}$ (case 3). Then $t_3 \leq t_1$ by construction, and $t_3 \not\leq t_2$.

Example 6

Consider $T = \alpha \rightarrow \{h(\eta, \eta), r(\eta, \eta)\}$, $\tau = f(\alpha, g(\alpha))$, and $t = f(X_0, g(Y_0))$. Table IV lists terms u, extensions of u, and the corresponding conditions of Lemma 1.

The construction of a cover for the maximum typed coverage of a clause will be done in terms of strict successive extensions starting with an arbitrary ground term.

Definition 9

Let $C = (H :- B)$, where $H = p(t_1, \ldots, t_n)$, be a clause and let type $(p/n) := p(\tau_1, \ldots, \tau_n)$ be a type declaration for p/n where τ_i is defined by a set T of type definitions, $i = 1, \ldots, n$. Let $\tau = p(\tau_1, \ldots, \tau_n)$. The sequence $\mathrm{cvg}_\tau(C) := \langle C_0, \ldots, C_k \rangle$ of instances of C, such that

$C_0 :=$ some $H_0 :- B_0$ such that $H_0 \in [\![\tau]\!]_T$, $H_0 = H\sigma$, and $B_0 = B\sigma$

$C_{n+1} := \mathrm{lca}(\{C_n\} \cup [\![C]\!]_{\{?\text{-}G_n\}})$ where $?\text{-}G_n$ is a strict extension of H_n

is called a *strictly increasing coverage sequence* for C with respect to τ. The corresponding sequence $\mathrm{cov}_\tau(C) := \langle ?\text{-}G_0, \ldots, ?\text{-}G_k \rangle$ where $G_0 := H_0$ is called a *strictly increasing cover sequence* for C with respect to τ.

Table IV. Strict extension of terms

u	extension of u	by condition
$f(h(X, X), g(Y))$	$f(h(a, b), g(h(a, a)))$	1
$f(h(X, Y), g(Y))$	$f(h(a, a), g(h(a, a)))$	2
$f(h(X, Y), g(r(Y, Y)))$	$f(h(a, a), g(r(b, a)))$	2
$f(h(X, Y), g(r(Y, Y)))$	$f(h(a, a), g(h(a, a)))$	3
$f(h(a, Y), g(r(a, b)))$	$f(h(b, a), g(r(a, b)))$	3

Notice that $[\![C]\!]_{\{?\text{-}G_i\}}$ consists of one element. Since G_i is a ground instance of H, there exists a substitution σ_i such that $H\sigma_i = G_i$. It is

$$[\![C]\!]_{\{?\text{-}G_i\}} = \{C\sigma_i\}$$

and

$$\text{lca}(\{C_n) \cup [\![C]\!]_{\{?\text{-}G_i\}}) = \text{lca}(\{C_n, C\sigma_i\})$$

Hence, there is a one-to-one correspondence of $\text{cvg}_\tau(C)$ and $\text{cov}_\tau(C)$. Notice also that $\text{cvg}_\tau(C)$ does not only depend on the choice of C_0 but also on the choices of the strict extensions $?\text{-}G_i$. But $\text{cvg}_\tau(C)$ is indeed finite. By Definition 6, $C_{n+1} \geq C_n$ and $C_{n+1} \neq C_n$. Hence, the number of anti-instances of C_{n+1} is less than the number of anti-instances of C_n. It follows that for any choice of C_0 and any choice of strict extensions of C_n (if such exist), there exists a $k \in \mathbb{N}$ such there does not exist a strict extension of C_k.

Example 7

Consider the type rules

$\alpha \rightarrow \{a, b\}$
$\beta \rightarrow \{c, d\}$
$\gamma \rightarrow \{e\}$
$list(\nu) \rightarrow \{[], [\nu|list(\nu)]\}$

Let C = f([A|B], C, D, [D|E]) and τ = type (f/4) = f($list(\alpha)$, α, β, $list(\beta)$). Then

$$\text{cvg}_\tau(C) = \langle \text{f([a], a, c, [c]), f([A|B], A, C, [C|D]), f([A|B], C, D, [D|E])} \rangle$$

is a strictly increasing coverage sequence with its corresponding cover sequence

$$\text{cov}_\tau(C) = \langle \text{f([a], a, c, [c]), f([b, a], b, d, [d, c]), f([b], a, c, [c])} \rangle$$

Let D = g([A|B], C, D, [E|F]) and ρ = type(g/4) = g($list(\alpha)$, α, β, $list(\gamma)$). Then

$$\text{cvg}_\rho(D) = \langle \text{g([a], a, c, [e]), g([A|B], A, C, [e|D]), g([A|B], C, D, [e|E])} \rangle$$

is a strictly increasing coverage sequence with its corresponding cover sequence

$$\text{cov}_\rho(D) = \langle \text{g([a], a, c, [e]), g([b, a], b, d, [e, e]), g([b], a, c, [e])} \rangle$$

Notice that the last element C_3 of $\text{cvg}_\tau(C)$ is a variant of C and the last element D_3 of $\text{cvg}_\rho(D)$ is not a variant of D.

Theorem 3

Let $C = (H :- B)$, where $H = p(t_1, \ldots, t_n)$, be a clause and let type$(p/n) = \tau$ be a type declaration for p/n where τ is defined by a set T of type definitions and $\mathcal{G}(H) \cap [\![\tau]\!]_T \neq \emptyset$. Then for any strictly increasing coverage sequence $\text{cvg}_\tau(C) = \langle C_0, \ldots, C_k \rangle$, $C_k \simeq \max_\tau(C)$.

Proof

Take some $\text{cvg}_\tau(C) = \langle C_0, \ldots, C_k \rangle$ and let $C_k = H_k :- B_k$. Let $\text{cov}_\tau(C) = \langle ?\text{-}G_0, \ldots, ?\text{-}G_k \rangle$ be a corresponding cover sequence to $\text{cvg}_\tau(C)$. Now take

$$J := \{[\![C]\!]_{\{?\text{-}G_0\}}, \ldots, [\![C]\!]_{\{?\text{-}G_k\}}\}$$

$$I := \{(H :- B)\sigma : H\sigma \in G(H) \cap [\![\tau]\!]_T\}$$

By definition, $C_k = \text{lca}(J)$, and $J \subseteq I$, hence

$$C_k = \text{lca}(J) \leq \text{lca}(I) = \max_\tau(C)$$

Assume that $C_k \not\simeq \max_\tau(C)$. Then there exists a $C' = (H' :- B') \in I$ with $C' \not\leq C_k$. By definition of J and I, $H \in G(H) \cap [\![\tau]\!]_T$ and $H' \not\leq H_k$, i.e. $?\text{-}H'$ is a strict extension of C_k. This contradicts the definition of $\text{cvg}_\tau(C)$, hence $C_k \simeq \max_\tau(C)$.

The previous theorem completes the treatment of constructing maximum typed covers. It states that constructing a strictly increasing cover sequence for a clause C will result in a maximum typed cover of C. Algorithms have been developed for the construction of maximum typed covers (Jack, 1996) and an environment for testing logic programs has been implemented which will be described in the following section.

6. THE PROTEST SYSTEM

This section describes the main features of a test environment based on the testing concepts developed in the previous sections. The test environment is called PROTest (PROgram Test Environment) (Belli and Jack, 1994). As the methods for test coverage and test input generation have been developed for logic programming, this test environment is designed for testing programs written in a typical logic programming language, Prolog. It is implemented partly in Prolog and partly in C and uses most of the programs described in the previous sections. It has a graphical user interface implemented under X-Windows. The test environment demonstrates the practicability of the proposed methods for test input generation. The environment is uniform in the sense that it incorporates testing concepts in the program development process.

For a Prolog program, the test environment performs a declarative (structure) check, automatically generates test inputs for structural coverage, receives a test program, runs the program according to the test program using the generated test inputs and finally generates a test report for the user.

The system consists of five components: structure checker; test input generator; test coverage analyser; test driver; and a test report generator (see Figure 3).

The structure checker analyses the source code which is an *instrumented* Prolog program. Instrumentation of a Prolog program means the association of the predicates with formal information concerning the predicate's arguments. The programmer provides the instrumentation for a program. There are two categories of such formal information: types and modes. Types are sets of terms that define the domains of the arguments of a predicate. Structure checking comprises verification of the program for type and mode correctness and generation of a structure report.

The test input generator uses the instrumentation of a program for generation of test inputs. The underlying generation principle is structural induction over the type declarations. In addition, the declarative test coverage notion described in Section 3 is used to control selection of test inputs. Test cases consist of test inputs and expected outputs. The expected outputs must be constructed from the specification of the program. This is not incorporated in PROTest, but expected outputs can be supplied by an external oracle

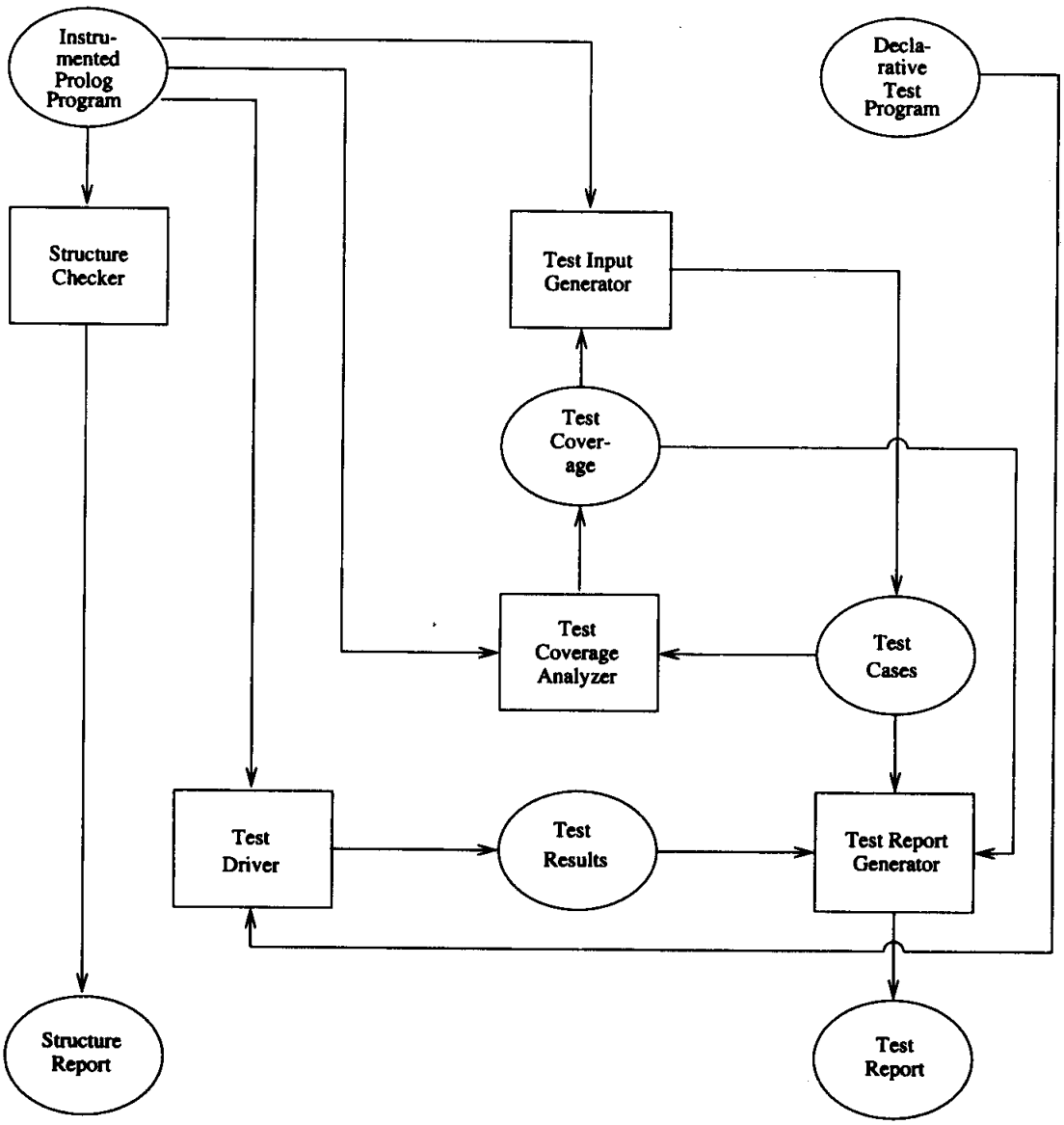

Figure 3. System architecture of PROTest.

to the system and can then be automatically checked against the actual outputs of the test run.

Test coverage information is provided by the test coverage analyser. This information includes instances of program clauses obtained by *anti-unifying* them with the actual set of test inputs. Test coverage is declarative information and can be determined from the source code and the test inputs, i.e. without program execution. Program clauses are covered if the test coverage instances are variants of them. Thus, test coverage does not only provide a measure of coveredness, i.e. the ratio of covered and uncovered clauses, but also directives for the generation of new test cases from the existing ones.

From a software engineering point of view, it is desirable to incorporate testing issues in the design and implementation of programs as a constructive element. Therefore, PROTest includes a built-in test language. This test language is Prolog-based and enables the development of test programs uniformly with the implementation of the programs to be tested. PROTest test programs include, apart from test procedures, the formulation of complex test result evaluation.

The test driver runs the dynamic test using the test cases by means of executing the test program for the instrumented Prolog program. The test results obtained by the test run are passed to the test report generator.

In addition to the results from the dynamic test, the test report generator uses the static test coverage obtained during test input generation and the set of test inputs to generate a test report for the user. Depending on the specification by the user, a test report can be comprehensive and contain statistics on the covered clauses and predicates, the test coverage instances and the test cases. The test report can also be selective, i.e. include a selection of the above items.

6.1. Structure checker

Apart from syntax checking of a Prolog program as performed by any interpreter or compiler, type and mode declarations can be used for checking the general consistency of a program. However, all of these checks are *static*, i.e. performed on the source code. In particular, the type and mode declarations describe the *structure* of a program. The task of statically checking a program is performed by the structure checker of PROTest. Two classes of program structure are handled by PROTest, data structure and data flow structure. The intended data flow is described in terms of instantiations of variables, between the literals of program clauses (mode declarations). The data structure is described as the *structure of the arguments* of the predicates in a program clause (type definitions and type declarations). The static checks refer to the consistency of type and mode declarations. This means that it will be checked whether

- the type definitions are complete, i.e. each type functor used appears exactly once in the left-hand side of a type definition. It is also checked that the arguments T_1, ..., T_n of a type structure $\varphi(T_1, \ldots, T_n)$ appearing as a left-hand side of a type definition $\varphi(T_1, \ldots, T_n) \rightarrow \Phi$ are distinct variables, and no variables different from T_1, ..., T_n occur in the right-hand side Φ;
- there is exactly one type declaration for each predicate defined in the program to be tested;

- the type declarations are meaningful, i.e. if for all clauses $C = (H :- B)$ of a predicate p/n, declared ?-type $p(t_1, \ldots, t_n)$, the set

$$\{(H :- B)\mu : H\mu \in \mathcal{G}(H) \cap [\![p(t_1, \ldots, t_n)]\!]_T\} \neq \emptyset.$$

  This is a weak type check, since it only depends on the heads of the clauses. A strong type check also considers the bodies of the program clauses, i.e. if the p/n is declared ?-type δ and the literals of the body $B = b_1, \ldots, b_k$ are declared ?-type β_1, ..., ?-type β_k, then the set

$$\mathcal{G}(H :- B) \cap [\![\delta :- \beta_1, \ldots, \beta_k]\!]_T \neq \emptyset$$

- the mode declarations are consistent with the predicates defined in the program to be tested, i.e. the declared data flow for the predicates is possible by the arrangement of the literals of a program clause.

The results of a static check are combined in a structure report.

6.2. Test case generator

After having checked a program for static type and mode errors (and corrected it) the next step for testing is to run the program with some test cases, i.e. perform a dynamic test. To overcome the nuisance of generating test cases (pairs of test inputs and intended test outputs) by hand, PROTest includes a test input generator. Test inputs will be generated automatically; moreover, the generated test inputs are partitioned (Weyuker and Jeng, 1991) into several classes derived by the mode declarations. As the test concept is implementation-based, only test inputs are generated automatically.

The test cases are partitioned by the mode declarations. For each mode of the predicate p, a set of test inputs can be generated. This leads to a classification of test cases.

In practice, it is not necessary to test all predicates but a subset of them, because a program may include facts constituting a predicate which represent only data. PROTest provides the facility to select the predicates for which test inputs are to be generated. The user is provided with a list of the predicates defined in the program to be tested and can select by mouse click the predicates to be tested. Test predicate selection is shown in Figure 4.

The generated test inputs will be used to run the program. The outputs will then be checked against the type and mode declarations. PROTest is a tool for implementation-based testing. It utilizes only a partial formal specification given by types and modes. If the expected outputs, which must be derived from the specification of the program, are supplied to the system, PROTest automatically checks the actual outputs of the test run for correctness.

The output of automatically generated test inputs, which may be edited, i.e. modified, deleted or extended, is shown in Figure 5. In this state of a test session, the user may supply expected outputs for the test inputs: each generated test input is associated with a variable for this purpose.

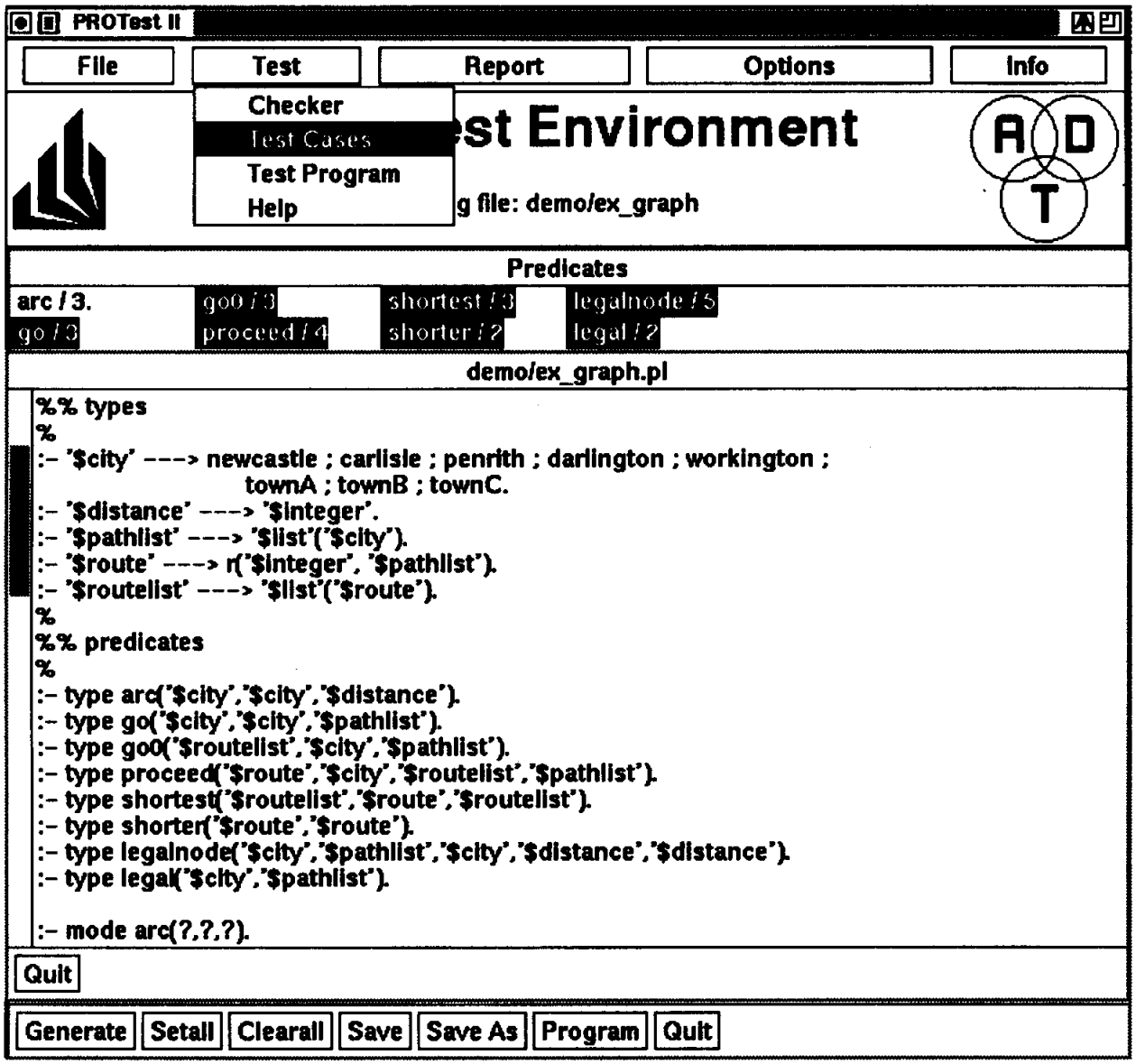

Figure 4. Test predicate selection in PROTest.

6.3. Test coverage analyser

Test coverage is determined via anti-unification and the test coverage analyser is utilized for the generation of test inputs and a test report. While in the first task, the analyser is used incrementally, in the second it is used for analysing a whole set of test inputs. For generating test inputs, the coverage analyser explicitly utilizes the type system (cf. Section 4). Because of this dual use, the test coverage analyser is designed as a core program with an interface prepared for the needs of generating test inputs and generating test reports. The test coverage analyser implements the results given in Section 3.

6.4. Test driver

The dynamic part of testing in PROTest is performed by the test driver. A dynamic test consists of running the program to be tested with test inputs. Hence, such a test is possible only if test inputs are available. Because of the possible complex structure of

PROTest II

File | Test | Report | Options | Info

Prolog Test Environment

Working file: demo/ex_graph

Edit Test Cases

```
% Test cases for demo/ex_graph.pl
% Generated by PROTest II

% Test goals for go / 3
% ----------------------------------------------------
tc(go(newcastle, newcastle, []), A).
tc(go(carlisle, carlisle, [newcastle]), A).
tc(go(carlisle, newcastle, []), A).

% Test goals for go0 / 3
% ----------------------------------------------------
tc(go0([], newcastle, []), A).
tc(go0([r(0, [])], carlisle, [newcastle]), A).

% Test goals for proceed / 4
% ----------------------------------------------------
tc(proceed(r(0, []), newcastle, [], []), A).
tc(proceed(r(1, [newcastle]), carlisle, [r(0, [])], [newcastle]), A).
tc(proceed(r(0, [carlisle,newcastle]), newcastle, [], []), A).

% Test goals for shortest / 3
% ----------------------------------------------------
tc(shortest([r(0, [])], r(0, []), [r(0, [])]), A).
tc(shortest([r(1, [newcastle]),r(0, [])], r(1, [newcastle]), [r(1, [newcastle]),r(0, [])]), A).
tc(shortest([r(0, []),r(0, [])], r(1, [newcastle]), [r(0, [])]), A).
tc(shortest([r(0, [])], r(0, []), []), A).
tc(shortest([r(1, [newcastle]),r(0, [])], r(1, [newcastle]), [r(0, [])]), A).
```

Select | Save | Save As | Program | Quit

Figure 5. Automatically generated test inputs in PROTest.

test inputs, the actual dynamic test needs to be structured. The structure of a dynamic test can be specified in PROTest by a test program. The language used for test programs is based purely on Prolog; in fact, it consists of special built-in predicates.

In general, a test program specifies conditions for running individual test inputs or sets of test inputs. The sets of test inputs to be operated on the program to be tested are specified as subsets of the available set of test inputs. The typical situation for this would be that test inputs have been generated automatically for the relevant predicates of a program under test. The test inputs can be classified according to type and mode properties. The type properties are well-typedness and its complement, ill-typedness. This means that if for a test input goal ?-G, where

$$G = p(t_1, \ldots, t_n), \quad type(p/n) = p(\tau_1, \ldots, \tau_n), \quad \text{and } \mathcal{G}(G) \cap [\![type(p/n)]\!]_T \neq \emptyset$$

then ?-G is well-typed. This property may be important, as has been pointed out in

Section 5. Mode properties are given directly by the mode declarations, i.e. each mode of a predicate builds up a mode class.

Test inputs can also be classified by modes as described in Section 5. Such classes can be specified in a test program for program runs with the appropriate test inputs. Classes may be composed by forming logic expressions from the above mentioned basic classes. If there are expected outputs available (these can be supplied by the user), then the test program can specify comparison of expected and actual outputs. Specific actions can then be performed dependent on the result of a comparison.

Since a test program is written in a Prolog-based language, classification and comparison are described in a declarative way by means of built-in predicates. In order to avoid unnecessary overhead to regular Prolog, only two predicates have been defined for this task. These predicates are test/4 and success/1. Their semantics are described informally in the following.

test(*functor*, *type_class*, *mode_class*, *Result*)

This predicate takes the *functor* of a predicate to be tested, *type_class* and *mode_class* as its input arguments and *Result* as its output argument; *type_class* can be well_typed, ill_typed or any; *mode_class* can be either a single mode specification or a list of mode specifications. The predicate succeeds whenever the arguments are according to the above requirements.

success (*Result*).

This predicate is applicable only if expected outputs are available. It takes *Result* as input argument and succeeds if the expected outputs coincide with the actual outputs.

With the above predicates, a test program can be written entirely in the form of a Prolog program. Hence, the full power of Prolog is also available in test programs. A view of a test program in PROTest is shown in Figure 6.

6.5. Test report generator

Test reports are generated by the test report generator, which uses the test coverage analyser. There are two sorts of test reports to be generated, as follows.

(1) A *brief test report* for a Prolog program based on a test program contains

- the name of the file of the Prolog program to be tested, the name of the file of the test program, and the name of the file containing the test cases;
- the *ratio* of the number of tested predicates and total predicates;
- the number of test cases;
- the test *coverage*, consisting of the *ratio* of
  - * the covered clauses and the total clauses of the tested predicates, i.e. the coverage measure $cv_P^{(r)}(T)$ as defined in Section 3,
  - * the covered clauses and the total clauses of the program, i.e. the coverage measure $cv_P^{(c)}(T)$ as defined in Section 3, and

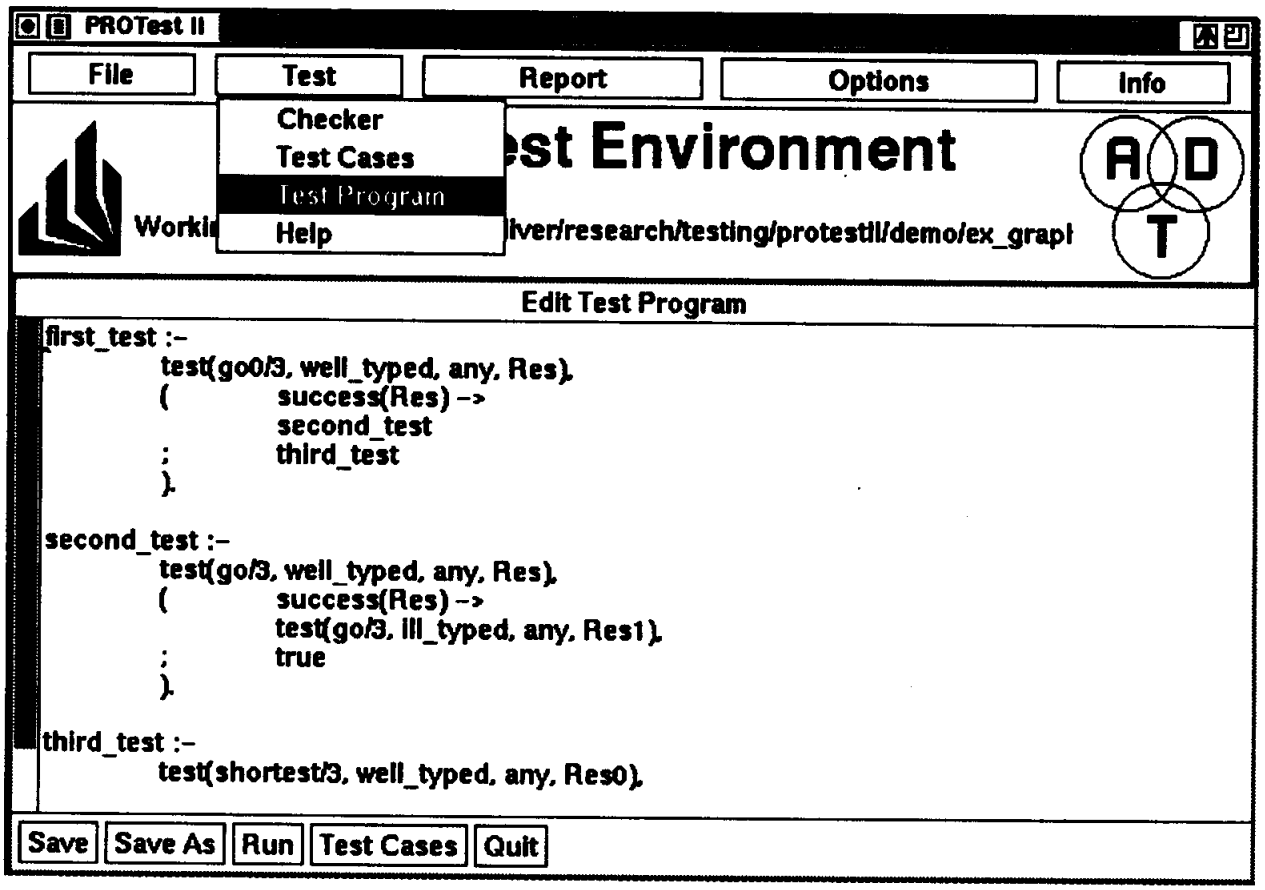

Figure 6. A test program in PROTest.

* the covered predicates and the tested predicates,
* the covered predicates and the total predicates.

(2) A *full test report* contains all the information of a brief test report and for each uncovered clause C its coverage, i.e. the instance C' of C which is covered.

A view of the full test report for the sample program is given in Figure 7.

7. PRACTICAL RESULTS

The PROTest system has been used for testing software developed in a project dealing with concurrent robot programming (Pollmann, 1996). The hardware of the robot programming system includes a transputer system in order to meet performance requirements for operating multiple robots. The program to be tested using PROTest is an off-line Prolog program for robot task scheduling for a variable number of transputers. The schedule consists of a finite set of dependent robot tasks with various processing times aiming to minimize the schedule length. Since the problem of computing such a schedule is *NP*-hard (Garey and Johnson, 1979), a heuristic approach utilizing best-first search is implemented.

The program considered has a total of 58 predicates with a total of 101 clauses. PROTest was first used to generate well-typed test inputs for maximum typed coverage. It generated a total of 275 test inputs. The coverage results are given in Table V. Notice that this table points out the maximum coverage which can be obtained with test inputs

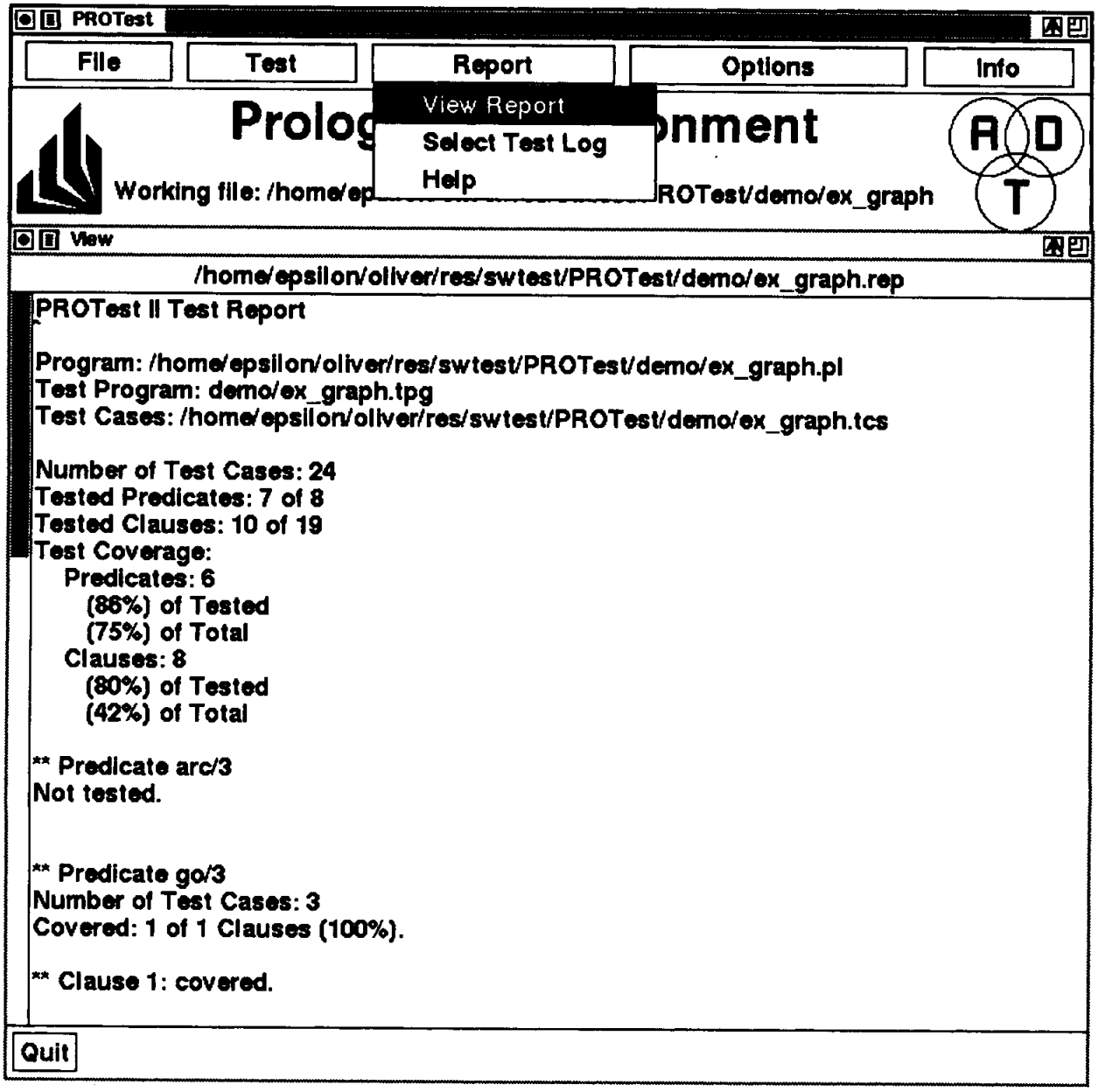

Figure 7. A test report in PROTest.

Table V. Maximum typed coverage results for the task schedule program

	Total	Tested	Covered	Covered of Total (%)	Covered of Tested (%)
Predicates	58	55	34	59	62
Clauses	101	98	72	71	73

that conform to the intended data structures of the program. In order to construct a 100% covering test input set, PROTest was used to generate additional ill-typed test inputs. It generated a total of 67 ill-typed test inputs which covered, together with the above mentioned 275 well-typed test inputs, the task schedule program. Of the ill-typed test

inputs, 38 produced elements from the success set of the program. These test inputs called 17 of the 22 predicates which required ill-typed test inputs (see Table VI). Manual inspection and comparison with the specification revealed faults in the clauses.

The test inputs generated by PROTest point out potential faults in the program. It generates ill-typed test inputs, which must not, when run on a correct program, produce answers from the success set of the program. If these inputs produce elements from the success set, the user is pointed to the faulty clauses.

8. CONCLUSIONS

In this paper, an implementation-based test concept for a declarative and relational programming paradigm has been presented. The results of this work are as follows.

- A *coverage measure* for test completeness validation which can be used as an *operable constructive tool.* The coverage notion is based purely on data aspects and thus is *independent of some specific computational model.* It is developed completely within the theory behind logic programming: predicate logic and unification theory. The computation of the coverage measure for a given test input set and a program is *computationally feasible* and even *statically computable.*
- An effective and efficient scheme for *automatic test input generation* and *selection.* With this scheme a 100% data covering test input set, a *maximum typed cover*, can be generated. The test input generation algorithm uses the program instrumentation as its basic construction mechanism and the coverage measure as the driving and selecting component in order to achieve an *expressive* and *data covering test input set.* Automatic test input generation is a novel research area in the field of logic programming.
- The implementation of the test concept in a *test environment.* The algorithms developed for analysing a program and its instrumentation, test coverage computation, test input generation, and test input selection have been implemented and shown *feasible for practical application.*

The program instrumentation is a specialized application of type and mode schemes, formerly introduced for interpreter and compiler optimization. The coverage measure reflects the logic programming paradigm. The abstract model on which it is defined consists of the set of program clause instances which form a complete lattice, and coverage is defined in a natural way as least upper bounds on subsets of the instances space. The adequacy of the coverage notion has been shown by mathematical characterization of its relevant properties.

Table VI. Ill-typed test input generation for the task schedule program

	Total	Tested	Need ill-typed to cover	% of total	% of tested
Predicates	58	55	22	38	40
Clauses	101	98	27	27	28

The test coverage notion alone is not sufficient for expressive test input generation. Hence, program instrumentation is utilized in connection with coverage for test input generation. Because of rigorous mathematical foundation, these concepts allow for automatic test input generation. The mathematical properties of this interaction of concepts have been carefully analysed. On the basis of this analysis, algorithms for effectively determining the test coverage for a program and a given set of test inputs have been developed. Together with algorithms implementing the program instrumentation framework, algorithms for effectively generating maximum typed covers have been established.

While classical structural testing is commonly based on data flow or control flow, the approach presented in this paper is based on the program logic which is at a higher level of abstraction. This approach allows the generation of a test input set that covers the program logic. This can be achieved by *formal program instrumentation* and the utilization of a test coverage measure as a goal driven constructive tool. For non-imperative programming especially, *data coverage* is an appropriate measure for *test completeness* and together with formal instrumentation it gives test input selection criteria as well as *test termination criteria*. An important result is that the process and expressiveness of program testing is reflected by the programming paradigm.

Acknowledgements

This work has been supported by Mitsubishi Electric Europe under the project 'Modelling and implementation of a flexible robot cell'. The authors would like to thank Richard Hamlet for his encouragement and fruitful discussions concerning this work. The authors also thank the referees for valuable hints and suggestions for improving this paper.

References

Angluin, D. and Smith, C. H. (1983) 'Inductive inference: theory and methods', *ACM Computing Surveys*, **15**(3), 237–269.

Belli, F. and Jack, O. (1993) 'Implementation-based analysis and testing of Prolog programs', In T. Ostrand and E. Weyuker (Eds), *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, Cambridge, Massachusetts, U.S.A., June, ACM Press, New York, U.S.A., pp. 70–80.

Belli, F. and Jack, O. (1994) 'An environment for self-testing of logic programs', In F. Anger (Ed.), *Proceedings of the Seventh International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Austin, Texas, U.S.A., Gordon and Breach, New York, U.S.A., pp. 29–36.

Belli, F. and Jack, O. (1995) 'A test coverage notion for logic programming', *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, Toulouse, France, October, pp. 133–142.

Bobrow, D. G. (1985) 'If Prolog is the answer, what is the question? or What it takes to support AI programming paradigms', *IEEE Transactions on Software Engineering*, **11**(11), 1401–1408.

Bougé, L., Choquet, N., Fribourg, L. and Gaudel, M.-C. (1986) 'Test sets generation from algebraic specifications using logic programming', *Journal of Systems and Software*, **6**(4), 343–360.

Boyer, R. S. and Moore, J. S. (1979) *A Computational Logic*, Academic Press, New York, USA.

Dart, P. W. and Zobel, J. (1992) 'Efficient run-time checking of typed logic programs', *Journal of Logic Programming*, **14**(1 and 2), 31–69.

Denney, R. (1991) 'Test-case generation from Prolog-based specifications', *IEEE Software*, **8**(2), 49–57.

Ducassé, M. and Noyé, J. (1994) 'Logic programming environments: dynamic program analysis and debugging', *Journal of Logic Programming*, **19 & 20**, 351–384.

Duran, J. W. and Ntafos, S. C. (1984) 'An evaluation of random testing', *IEEE Transactions on Software Engineering*, **10**(4), 438–444.

Garey, M. and Johnson, D. (1979) *Computers and Intractability*, W. H. Freeman, San Francisco, California, U.S.A.

Hekmatpour, S. and Ince, D. (1988) *Software Prototyping, Formal Methods and VDM*, International Computer Science Series, Addison-Wesley, Wokingham, U.K.

Huet, G. (1976) 'Resolution d'equations dans des languages d'ordre 1, 2, ..., ω', Thèse d'état, Université de Paris VII, Paris, France.

Jack, O. (1996) *Programming Complex Systems*, Vol. 10, *Software Testing for Conventional and Logic Programming*, Walter de Gruyter, Berlin, Germany.

Kowalski, R. A. (1979) 'Algorithm = logic + control', *Communications of the ACM*, **22**(7), 424–436.

Lassez, J.-L., Maher, M. and Marriott, K. (1988) 'Unification revisited', In J. Minker (Ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kauffman, San Francisco, California, U.S.A., pp. 587–625.

Leibrandt, U. and Schnupp, P. (1984) 'An evaluation of Prolog as a prototyping system', In R. Budde, K. Kuhlenkamp, L. Mathiassen and H. Zullighoven (Eds), *Approaches to Prototyping*, Springer-Verlag, Berlin, Germany, pp. 424–433.

Lloyd, J. W. (1987) *Foundations of Logic Programming*, Springer-Verlag, Berlin, Germany.

Luo, G., von Bochmann, G., Sarikaya, B. and Boyer, M. (1992) 'Control-flow based testing of Prolog programs', *Proceedings of the Third International Symposium on Software Reliability Engineering*, pp. 104–113.

Marriott, K., Naish, L. and Lassez, J.-L. (1990) 'Most specific logic programs', *Annals of Mathematics and Artificial Intelligence*, **1**(2), 303–338.

Mycroft, A. and O'Keefe, R. A. (1984) 'A polymorphic type system for Prolog', *Artificial Intelligence*, **84**(23), 295–307.

Naish, L., Dart, P. W. and Zobel, J. (1989) 'The NU-Prolog debugging environment', In G. Levi and M. Martelli (Eds), *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon, Portugal, MIT Press, Cambridge, Massachusetts, U.S.A., pp. 521–536.

Padawitz, P. (1992) *Deduction and Declarative Programming*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, U.K.

Pfenning, F. (Ed.) (1992) *Types in Logic Programming*, MIT Press, Cambridge, Massachusetts, U.S.A.

Pollmann, A. (1996) 'An approach to logic/object-oriented concurrent robot programming and performance aspects', PhD thesis, Universität-GH Paderborn, Paderborn, Germany.

Prather, R. E. and Myers, J. P. Jr (1987) 'The path prefix software testing strategy', *IEEE Transactions on Software Engineering*, **13**(7), 761–766.

Rapps, S. and Weyuker, E. J. (1985) 'Selecting software test data using data flow information', *IEEE Transactions on Software Engineering*, **11**(4), 367–375.

Ruggieri, S. (1996) 'Decidability of logic program semantics and applications to testing', In H. Kuchen and S. D. Swierstra (Eds), *Programming Languages: Implementation, Logics, and Programs, Proceedings of the 8th International Symposium (PLILP '96)*, Aachen, Germany, September, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, pp. 347–362.

Weyuker, E. J. and Jeng, B. (1991) 'Analyzing partition testing strategies', *IEEE Transactions on Software Engineering*, **17**(7), 703–711.

Yardeni, E. and Shapiro, E. (1991) 'A type system for logic programs', *Journal of Logic Programming*, **10**(1, 2, 3 and 4), 125–154.