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., Implementation-Based Analysis and Testing of Prolog Programs, Proceedings of the 1993 ACM International Symposium on Software Testing and Analysis (ISSTA), pp. 70-80

Implementation-Based Analysis and Testing of Prolog Programs

F. Belli O. Jack

University of Paderborn
Department of Electrical and Electronics Engineering
Germany

ABSTRACT

In this paper, we describe the PROTest II (**PRO**log **Test** Environment, Version 2) system to test logic programs in an interactive support environment. Logic programs are augmented with declarative information about the types and modes of the arguments of a predicate. Modes correspond to in, out, and in-out parameters. With this information PROTest II statically checks the types of Prolog programs, generates test cases, executes Prolog programs, and produces reports summarizing results including information about new test coverage metrics. Thus, PROTest II enables both static analysis and dynamic testing uniformly using a Prolog-based test language DTL/1. The strength of PROTest II stems from its idea of defining coverage in real logic programming terms, rather than adapting imperative programming ideas.

1 INTRODUCTION

Testing of programs has been of great interest in the past decades, and many results have been presented [13, 3, 17]. However, most of these systematic approaches are applicable to *imperative* programming but not for *declarative* programming, such as logic programming.

The logic programming-related issues in the area of software testing which have been studied concentrate on debugging [29, 28] and termination checking [30].

Generating test data form Prolog-based specifications also has been investigated [5, 11, 14, 16]. These investigations rely on a specification of imperative programs. The view is seeing Prolog as a *specification language* for imperative programs. The issue of testing logic programs, seen as an *implementation language*, has been investigated rarely [24].

There exist some differences between *specification-based* and *implementation-based* testing. With specification-based testing, both test input data and the expected outputs can be generated from a specification. With implementation-based testing, only a set of test input data can be generated from an implementation, but the expected outputs can not be derived from the implementation. In this case, the existence of an *oracle* (in the human mind) must be assumed, and checking the test results against the oracles has to be done.

Instrumentation [10] of Prolog programs by adding *types* and *modes* enable efficient implementation-based fault analysis and test input generation.

In this paper we will survey our approach PROTest II (**PRO**log **Test** Environment, Version 2) which allows a uniform, implementation-based test. *Uniform* means that the test language will be built into the Prolog programming environment. The test language is based on Prolog to achieve a seamless production environment. White-box testing (or glass testing) means that we consider the structure of the tested program when we determine the test procedure. Black-box testing has no such consideration.

For the description of our concept we use the terminology according to [23, 13, 21, 18, 20, 19].

2 PROTEST II SYSTEM OVERVIEW

The objective of PROTest II is to build a fully automated test environment. It is a prototype system and is embedded in a product assurance environment, presently including the test environment PROTest II and the reliability assessment environment PRORool (**PRO**log based environment for **R**eliability determination of **o**bject-oriented logic programs) [2]. The latter provides an approach to reliability prediction of Prolog programs, introducing measures describing Prolog program complexity. It also implements several software reliability models. The test results from the test environment serve as inputs for the reliability assessment environment to determine the complexity measures.

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.

PROTest II is hosted on a SUN SPARC station under the X-Window System. The window interface program is written in C and the main program is written in Prolog.

The system consists of five components: *structure checker*, *test input generator*, *test coverage analyzer*, *test driver* and a *test report generator*.

The structure checker analyzes 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 of a program. There are two categories of such formal information: types

ACM-ISSTA'93-6/93/Cambridge, MA, USA

and modes. Types are sets of terms which define the domains of the arguments of a predicate. These are regular sets described by type declarations. Modes are states of instantiation of the arguments of a predicate described by mode declarations. Structure checking comprises verification of the program for type and mode correctness and generation of a structure report.

Test cases consist of test inputs and expected outputs. As mentioned in the last section, with implementation-based testing, only the test inputs can be derived. For the construction of expected outputs the user or programmer acts as an oracle. The test input generator uses the instrumentation of a program for generation of test inputs. The underlying generation principle is a structural induction over the type declarations. In addition, a declarative test coverage notion is utilized to obtain a meaningful selection of test inputs to be generated.

Test coverage information is provided by the test coverage analyzer. 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 II includes a built-in test language, DTL/1 (Declarative 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. DTL/1 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.

Figure 1 depicts the system level architecture of PROTest II.

According to the above briefly explained overall structure of PROTest II, in the following sections we describe our approach to testing of logic programs. First, we define our fault model we use for Prolog testing.

3 A FAULT MODEL FOR PROLOG

A *fault* is a textual problem with the program resulting from a mental mistake by a programmer, and the mental mistake is defined as an *error* [21]. Only those faults are considered here which are not detectable by an ordinary compiler.

The fault model we use is not based on a special computational model for logic programming, such as the Prolog model [23], but relies on the declarativeness of logic programming. The focus is on the construction and derivation of *term structures* through predicates. These term structures

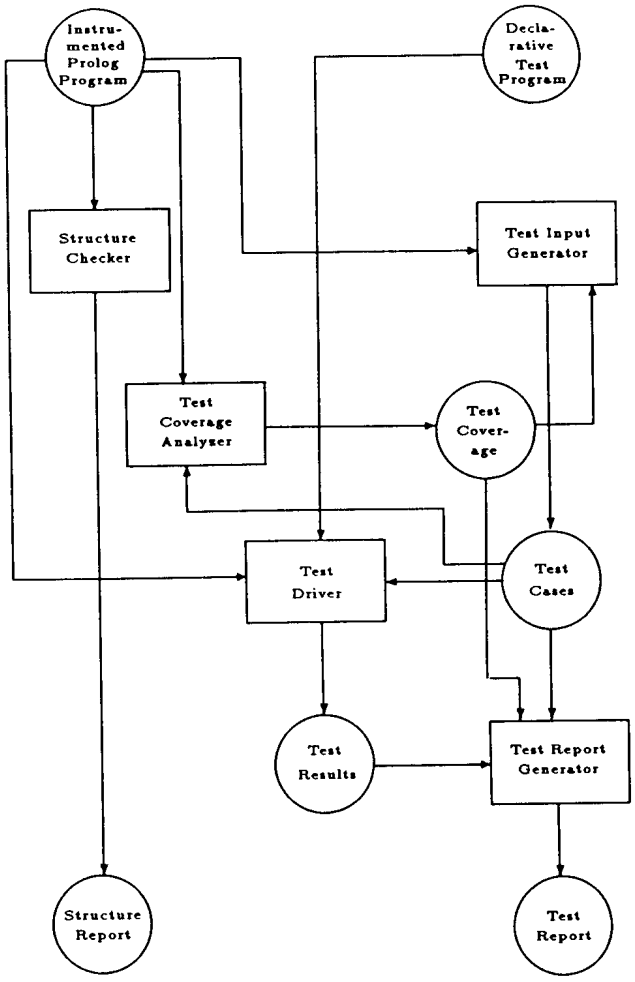

Figure 1: The system level architecture of PROTest II

are categorized by regular subsets of the Herbrand universe called *types*.

The following are the fault types we consider:

1. wrong typing,
2. wrong subtyping,
3. wrong parameter passing.

This classification subsumes other fault models as specified in [24, 1, 15].

4 ADDING TYPES AND MODES TO PROLOG

Unification provides a uniform mechanism for parameter passing, data selection and data construction [23]. Thus, Prolog does not include, in contrast to many imperative programming languages like Pascal or C, explicit information about the types of arguments of a predicate. This is theoretically elegant, but, from a software engineering point of view, disadvantages appear especially concerning program testing. Moreover, it is not always clear which of the arguments of a predicate are intended to be input and which to be output (or both). The following example illustrates this.

4.1 Example Consider the standard append predicate, which is intended to concatenate lists:

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

The "normal" use of append is a call (query) of the form

?- append(*list1* , *list2* ,L).

Provided that *list1* and *list2* are lists, the answer then is L = *list1 list2*. But also other calls are possible, e.g. ?- append(L1,L2,L3). For the testing of the append predicate, one needs to know the types of it's arguments (which must be lists) and the intended constellations of input and output arguments. The above mentioned call can hardly be useful for the practice, because all arguments are outputs.

Information about argument types and input-output constellations in a formal scheme is essential in our approach for automatic test case generation and structure checking of Prolog programs. To be useful, these schemes must be easy to handle and must not require extensive redundancy. A brief description of such schemes, *type* and *mode schemes* will be given in the following.

TYPES

Type schemes have been suggested for several purposes [26, 27, 25]. The type scheme here is based on the Mycroft/O'Keefe scheme [27] and the the scheme of [9].

Our type scheme involves declarations of types and predicates. Type declarations can be given as *construction* or *union*. They can also include arguments. A type declaration is a goal, defined by the following (we use extended BNF here):

⟨type declaration⟩ ::= ⟨type construction⟩ | ⟨type union⟩

⟨type construction⟩ ::= "type" ⟨name⟩ "==>" ⟨constructor⟩ [";" ⟨constructor⟩]*

⟨type union⟩ ::= "type" ⟨name⟩ "==>" ⟨constructor⟩ ["+" ⟨constructor⟩]*

⟨name⟩ ::= ⟨functor⟩ ["(" ⟨variable⟩ ["," ⟨variable⟩]* ")"]

⟨constructor⟩ ::= ⟨term⟩

A ⟨type construction⟩ is also called a *parametric type rule*.

4.2 Example The construction of a type called `list(T)` which is a list containing elements of type T will be defined as follows:

```
:- type list(T) ==> [] ; [T|list(T)].
```

4.3 Example The declaration of a type called number which is an integer or a float is done as follows:

```
:- type number = integer + float.
```

The above described scheme enables the declaration of types of *terms* (which are arguments of predicates). These types are *regular* subsets of the Herbrand universe (of a program). The declaration of the types of arguments of *predicates* is a goal of the following form:

⟨pred type declaration⟩ ::= "predtype" ⟨functor⟩ "(" ⟨type id⟩ ["," ⟨type id⟩]* ")"

⟨type id⟩ ::= ⟨term⟩

A variable free ⟨type id⟩ is called a *pure type term*.

4.4 Example The type declaration of the arguments of the predicate append (intended to concatenate integer lists) is

```
:- predtype append(list(integer),
                   list(integer),
                   list(integer)).
```

The scheme behind this declarations is called a *polymorphic* [27] or *parametric* [9] type system, which is described formally in [4, 9].

Several *type checking*, *comparison* and *unification* algorithms [27, 9, 33] can be adopted for this type scheme and will be used in section 5.

MODES

Next, a much simpler but also concise and useful concept to be utilized in our approach is described.

The declaration of the intended input-output constellations of the arguments of a predicate will be done by a mode scheme, similar to mode declarations used by some Prolog compilers, such as Quintus Prolog, for object code optimization purposes. Modes are states of instantiation of the arguments of a predicate. These mode declarations are goals of the following form:

⟨pred mode declaration⟩ ::= "predmode" ⟨functor⟩ "(" ⟨mode id⟩ ["," ⟨mode id⟩]* ")"

⟨mode id⟩ ::= "+" | "−" | "?"

The informal semantics of ⟨mode id⟩ is:

- + The argument is *not* a variable (intended to be *input*).
- − The argument is a variable (intended to be *output*).
- ? The argument is an arbitrary term (can be input as well as output).

In contrast to type declarations, more than one mode may be declared for a predicate.

4.5 Example The mode declarations for the append predicate will be

```
:- predmode append(+, +, ?).
:- predmode append(?, ?, +).
```

The first declaration specifies the append predicate to be used for concatenating two lists, the second specifies it to be used for construction of composite lists of a given one.

Types and modes can be considered as an *approximation* of the *intended interpretation* [23] thus as an *instrumentation* of a logic program. These type and mode schemes enable both static structure checking of a program and automatic test case generation, as will be described in the next sections. An early unformalized approach on this basis was given in [6].

5 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. However, all of these checks are *static*, i.e. performed on the source code. Especially, the type and mode declarations describe the *structur* of a program. The task of statically checking a program i performed by the structure checker of PROTest II. Structur means the intended data flow — in terms of instantiations o variables — between the literals of program clauses and th *structure of the arguments* of the predicates in a progran clause. Types are *regular* subsets of the Herbrand univers of a program as mentioned above. Types are *not* genera *constraints* for terms such as the "ordered" constraint fo lists.

5.1 Example Consider the following sort program.

```
sort([], []).
sort([X|Y1], Y2) :-
partition(X, Y1, Z1, Z2),
sort(Z1, Z11),
sort(Z2, Z22),
append(Z11, [Y1|Z22], Y2).

partition(X, [], [], []).
partition(X, [Y|Z], [Y|Z1], Z2) :-
Y < X,
partition(X, Z, Z1, Z2).
partition(X, [Y|Z], Z1, [Y|Z2]) :-
Y >= X,
partition(X, Z, Z1, Z2).
```

The *type declarations* will be

```
:- type numlist = list(number).
:- predtype sort(numlist,numlist).
:- predtype partition(number,numlist,numlist,numlist).
:- predtype append(numlist,numlist,numlist).
```

Using the type declarations, one can check the program clauses whether the types of the arguments passed by the atoms of the clauses are correct or not. This kind of static type checking might reveal programming errors in the second clause of sort. The second argument of the last atom in the body of this clause, append(Z11,[Y1|Z22],Y2), is constructed by passing two variables and combining them to the first and second argument of a list term. Y1 is passed from the head of the clause, and Z22 from the second argument of the third atom in the body, sort(Z2,Z22). The type declarations imply that Y1 must have a type list(number), since the first argument of sort has type numlist (= list(number)). Moreover, if [X|Y1] is of the type numlist then Y1 has to be of the type numlist. But then the second argument of the append atom, [Y1|Z22], does not obey the declarations of the type numlist. Replacing Y1 in [Y1|Z22] by X, the program is *type correct*, with respect to the types declared.

In the last section, the definition of types which are (regular) subsets of the Herbrand universe was given. Now types will be associated with arguments of a predicate, which will lead to the definition of a *type correct clause*.

A predicate p with arity n is *typed* $p(\alpha_1, \ldots, \alpha_n)$ if the i-th argument of p is associated with a type α_i. A program P is *typed* if each predicate in P is typed.

The types α_i will be given by pure type terms which determine the type via the mapping T_M, which is the usual mapping used for the definition of fixpoint semantics (see [23, 9]).

Let A be an atom $p(t_1, \ldots, t_n)$ and θ be a substitution such that $A\theta$ is ground. The instance $A\theta$ is *type correct* with respect to a typed program P, where p is typed $p(\alpha_1, \ldots, \alpha_n)$, if each $t_i\theta \in \alpha_i$, $1 \leq i \leq n$. A ground instance $C\sigma$ of a clause C is *type correct* with respect to a typed program P if each atom of that instance is type correct with respect to P.

Let E be an expression and θ a substitution. The *restriction* $\theta_{|E}$ of θ to E is the substitution

$$\{X_1/t_1, \ldots, X_r/t_r\} \subset \theta$$

where $X_1, \ldots, X_r$ are the variables occurring in E.

Now we are in position to define a type correct clause.

5.2 Definition (Type correct clause) A clause C is *type correct* with respect to a typed program P, if

- for each atom in C a type correct ground instance exists,
- for each atom A of C and type correct ground instance $A\theta$ of A there exists a ground substitution σ such that $C(\theta_{|A}\sigma)$ is type correct (with respect to P).

A program P is *type correct* if each clause is type correct with respect to P.

Note that, even if a typed program is type correct, a goal ?-Q may succeed with an answer θ such that $Q\theta$ is not type correct, e.g. if Q itself is not type correct.

5.3 Example By definition 5.2 neither of the clauses

```
a(X) :- b(X).
b(X) :- a(X).
```

typed

```
:- predtype a(number).
:- predtype b(integer).
```

is type correct, even there exist ground substitutions such that the head and body are type correct (e.g. {X/1}). The error is "wrong subtyping", i.e. possible parameter passing of arguments from supertypes to subtypes.

A slightly more complex example to illuminate definition 5.2 is the typed sort program form example 5.1.

5.4 Example The second clause of the sort program in example 5.1 is not type correct. To see this, consider the substitution

```
θ  = {X/1, Y1/[2,3], Y2/[1,2,3], Z1/[], Z2/[2,3],
      Z11/[], Z22/[2,3]}.
```

The instance by θ is then

```
sort([1|[2,3]], [1,2,3]) :-
    partition(1, [2,3], [], [2,3]),
    sort([], []),
    sort([2,3], [2,3]),
    append([], [[2,3]|[2,3]], [1,2,3]).
```

The instance of the head is type correct, but the instance of the body is not, because the second argument of

```
append([], [[2,3]|[1,2,3]], [1,2,3])
```

is not in the type numlist. Each ground substitution θ of the second sort clause such that the instance

```
append(Z11, [Y1|Z11], Y2)θ
```

is type correct yields the partition and the head instance being not type correct. For each atom in this clause there exist type correct ground instances, but there exists no ground instance yielding *all* atoms being type correct. This indicates an error in parameter passing.

PROTest II checks clauses of a typed program for type correctness adopting type unification [9].

6 TEST CASE GENERATION

After having checked a program for static type 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) per hand, PROTest II includes a test input generator. Test inputs will be generated automatically; moreover, the generated test inputs are partitioned [31] in several classes derived by the mode declarations [32]. Test outputs will be generated semi-automatically.

Following, we introduce this concept.

Let P be a logic program and $def(P)$ be the set of predicates defined in P.

A *test input* for a program P is a goal

$?\text{-}p(t_1, \ldots, t_n)$

where $p \in def(P)$ and $t_1, \ldots, t_n$ are the arguments to p.

Note that goals which are a conjunction of atoms are not considered as test inputs. Further, goals including atoms not defined in P are also not test inputs because it is obvious that they will fail, and thus no information can be obtained from such goals.

An *intended output* for a test input $?\text{-}Q$ and a program P is a set a instances of Q.

The intended output may be empty which means that the (test input) goal is intended to fail.

The *output* of a test case $?\text{-}Q$ and a program P is the set of instances of Q derived by the computed answers via SLD-refutations [23] of $?\text{-}Q \cup P$.

The output of a test case for a program may be infinite. We will handle this at the end of this section.

A *test case* for a program P is a tuple (ζ, η) where ζ is a test input and η is the intended output for ζ.

6.1 Example A test input for the sort program of example 5.1 is

```
?- sort([3,5,1,7,10,0], S).
```

The intended output for this input is

```
{sort([3,5,1,7,10,0], [0,1,3,5,7,10])}.
```

Another test input is

```
?- partition(5, X, Y, Z).
```

The intended output is {}.

In the following a program is assumed to include type and mode declarations.

For the generation of test inputs, first the mode declarations are considered. These imply which arguments of a test input

$$\zeta = ?\text{-}Q,\ Q = p(t_1, \ldots, t_n),$$

have to be variables and which have to be nonvariables. Although nonvariables may contain variables, for test input generation only arguments are considered which are either variables or ground. The arguments of Q which have to be nonvariables with respect to a mode declaration for the predicate p are generated according to the type declarations. The definition of types by a regular language enables such generation. Test cases will be generated by structural induction using the type declarations. In order to generate a set of test inputs, i.e. goals, with the property that each test input unifies with the head of a program clause, the type declarations will be used to generate the arguments of these test inputs. The test coverage analyzer (see section 8) determines the next test input to be generated. This ensures the termination of the test input generation procedure. This general concept of utilizing a separate module — the test coverage analyzer — for efficient test input generation is similar to the concept of test case generation for procedural programming introduced in [7], where system performance is evaluated to generate new test cases.

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 such as the representation of a graph by facts `arc(x,y)`. Thus, the predicate `arc` represents only data and it is irrelevant to test it. PROTest II provides the facility to select the predicates for which test inputs are to be generated.

The generated test inputs will be used to run the program. The outputs will then be checked against the type and mode declarations. In addition to these checks, the user has the option to provide the intended outputs for each or some selected inputs, i.e. the outcomes of the execution he/she expects. These intended outputs complete the generated test inputs to test cases. Thus, we have test oracles by means of these test cases of which inputs have been generated automatically.

At this point, we have to make a pragmatic decision because the output as well as the intended output for a test input may be infinite. In this case, the programmer wishes a finite subset to be specified by the user as the intended output. Accordingly, also for the output only a finite subset is to be considered. This takes place in the test program in a well-defined test language we describe in the following section.

7 TEST LANGUAGE

PROTest II includes a *built-in test language* DTL/1 (Declarative Test Language) based on Prolog. This enables a *declarative* testing style. The test driver accepts test programs written in DTL/1 and a set of test inputs or test cases and performs a dynamic test. We sketch DTL/1 without giving its entire syntax [4].

Generally, DTL/1 consists of predicates delivering following services:

- *Parametrized* call of test inputs,
- *Validation* of test results.

PARAMETRIZED CALLS

The formal syntax of a *parametrized* call of a set of test inputs is:

⟨parametrized call⟩ ::= "exec" "(" (⟨pred id⟩ | ⟨term⟩)
  [⟨call parameters⟩] ")"

⟨pred id⟩ ::= (⟨functor⟩ "/" ⟨arity⟩)

⟨arity⟩ ::= ⟨non negative integer⟩

⟨call parameters⟩ ::= ["," ⟨maxsolutions⟩] [","
  ⟨mode spec⟩]

⟨maxsolutions⟩ ::= ⟨non negative integer⟩ | "spec" |
  "nolimit"

⟨mode spec⟩ ::= "mode" "(" ⟨mode id⟩ [","
  ⟨mode id⟩]* ")"

⟨non negative integer⟩ ::= "0" | "1" | "2" ...

A parametrized call with ⟨pred id⟩ = p/n and without ⟨parameters⟩ results in calling *all* test inputs available for the predicate p with arity n. If there are intended outputs — given by the user — then these will be looked up for the number of answers specified, and outputs up to this number will be generated via backtracking for each test input. This mechanism enables handling of predicates and goals with an infinite set of answers. A parametrized call with ⟨pred id⟩ = $p(t_1, \ldots, t_n)$ will execute the test inputs for predicate p/n *which are unifiable with* $p(t_1, \ldots, t_n)$.

The ⟨parameters⟩ define by ⟨maxsolutions⟩ the maximum number of solutions to be generated via backtracking for the test input (switching off the above described mechanism) and by ⟨mode spec⟩ the execution of the test inputs of the partition according to ⟨mode spec⟩. If ⟨maxsolutions⟩ is `nolimit` then no restriction on the number of answers is given, and if ⟨maxsolutions⟩ is `spec` then the maximum number is that specified in the intended output — if given any; otherwise the test driver will report an error.

VALIDATION OF TEST RESULTS

The formal syntax of the *Validation* of test results is:

⟨success check⟩ ::= "successful" "(" ⟨pred id⟩ | ⟨term⟩ ["," ⟨mode spec⟩] ")"

⟨type check⟩ ::= "type_correct" "(" ⟨pred id⟩ | ⟨term⟩ ["," ⟨mode spec⟩] ")"

`successful` checks the output of a test input — specified by its arguments — whether it equals the intended output. It fails if the according test inputs have not been executed.

`type_correct` checks the output of a test input — specified by its arguments — whether it is type correct with respect to type declarations by implementing the concepts of section 4.

7.1 Example A typical test program for the `sort` program of example 5.1 will be

```
exec(partition/2, _, mode(+,+,-,-)),
successful(partition/2, _, mode(+,+,-,-)) ->
        exec(sort/2)
;
        exec(append/3)).
```

8 TEST COVERAGE ANALYSIS

The notions of test coverage for imperative programming based on the execution of branches is not adequate for Prolog [12]. Prolog's execution model (the procedural semantics) is based on SLD resolution with the central concept of unification [23]. Thus, we have to base a coverage notion on this execution model. This coverage notion is *declarative* and thus can be used for static analysis as well as for dynamic analysis. The main idea is, to use the concept of unification because unification determines program clauses to be selected in a derivation of a goal and a program. If a clause is selected in a derivation, it will be activated, from a procedural point of view. Unification of a goal with the head of a clause can be interpreted as procedure activation. We say, a set of goals (as test inputs) covers a program, if these goals activate each program clause. This definition is very general as it ignores the structure of the clauses concerning their arguments and possible instantiations. In the following, we will briefly complete and refine this definition.

In PROTest II, the test coverage analyzer performs the determination of the coverage of a set of test inputs for a program.

Following, we briefly describe our coverage notion for Prolog. First, we review the concept of anti-unification. We follow the standard terminology of [23] enriched by the terminology introduced in [22].

An expression E' is an *instance* of an expression E, if there exists a substitution σ such that $E' = E\sigma$. If E' is an instance of E, this will be denoted by $E \geq E'$. E' is a *variant* of E, if E' is an instance of E and E is an instance of E'. If E' is a variant of E, this will be denoted by $E' \sim E$.

It is obvious that $\sim$ is an equivalence relation and $\geq$ is a partial order on the set of expressions modulo $\sim$.

Let S be a set of expressions. An expression E is a *common anti-instance* of S, if $E \geq E' \quad \forall\, E' \in S$. E is a *least common anti-instance* (lca) of S if $\tilde{E} \geq E$ for all common anti-instances $\tilde{E}$ of S.

If a least element is added to the set of expressions, then $\geq$ forms a lattice [22] (a greatest element is a variable). The lca of a set S of expressions is simply the least upper bound of S under the relation $\geq$. Thus, it follows immediately that an lca is unique modulo $\sim$.

8.1 Example Let
S = {`p(1, X, [a|R]), p(Y, 2, [a,b|S])`}. Then `p(V, W, [a|R])` is an lca of S.

Let T = {`p(1, X, [a|R]), q(1, X, [a|R])`}. Then `W` is an lca of T. In this case every common anti-instance of T is a least common anti-instance of T.

The concept of the least common anti-instance is inspired by the unification concept of logic programming: unification leads to greatest lower bounds of terms under the instance order, lcas are least upper bounds. For this reason, the calculation of an lca is called *anti-unification*. This gives the clue to the coverage notion for logic programs, which we will call *cover*.

8.2 Definition (Cover) Let T be a finite set of goals of the form $\{?\text{-}Q_1, \ldots, ?\text{-}Q_m\}$.
Let $C_1, \ldots, C_q$ be the definition of the predicate p, in a program P and $C_1^{(T)}, \ldots, C_q^{(T)}$ the sets of the top level instances of the program clauses $C_1, \ldots, C_q$ used in the computation of answers to the goals $?\text{-}Q_i \in T$.
T is a *cover for a program clause* C_i, or T *covers* C_i, if the lca of $C_i^{(T)}$ is a variant of C_i.
T is a *cover for the predicate* p, or T *covers* p, if T is a cover for every program clause in the definition of p.
T is a *cover for a program* P, or T *covers* P, if T is a cover for every predicate in P.

It is obvious that if a set of instances $C^{(T)}$ of a Program clause C is not empty, then the lca of $C^{(T)}$ is an instance of C.

The idea behind this definition is, that if a set of goals

- leads to activation of *every* program clause,
- and the activations of each program clause are *as general as possible*, i.e. the full scope of the terms of the arguments which lead to unification with a program clauses head is reached,

then this set covers a program.

There are algorithms to compute the lca of a finite set of expressions [22]. A Prolog version of such an algorithm, computing the lca `LCA` of two terms `Term1` and `Term2` is the following.

```
anti_unify(Term1, Term2, LCA) :-
    differences(Term1, Term2, LCA, Diffs, []),
    sort(Diffs, OrdDiffs),
```

```
    anti_unify_1(OrdDiffs).

differences(Term1, Term2, Anti, Diffs0, Diffs) :-
    ( nonvar(Term1), nonvar(Term2),
    functor(Term1, F, N), functor(Term2, F, N) ->
    functor(Anti, F, N),
    differences_1(N, Term1, Term2, Anti,
                          Diffs0, Diffs)
    ;    Term1 == Term2 ->
    Anti = Term1,
    Diffs0 = Diffs
    ;    Diffs0 = [t(Term1,Term2,Anti)|Diffs]
    ).

differences_1(N, Term1, Term2, Anti,
                        Diffs0, Diffs) :-
    ( N > 0 ->
    arg(N, Term1, Term1Arg),
    arg(N, Term2, Term2Arg),
    arg(N, Anti, AntiArg),
    differences(Term1Arg, Term2Arg, AntiArg,
                Diffs1, Diffs),
    M is N- 1,
    differences_1(M, Term1, Term2, Anti,
                          Diffs0, Diffs1)
    ;    Diffs0 = Diffs
    ).

anti_unify_1([]).
anti_unify_1([t(A,B,V)|OrdDiffs]) :-
    anti_unify_2(OrdDiffs, A, B, V).

anti_unify_2([], _, _, _).
anti_unify_2([t(M,N,W)|OrdDiffs], A, B, V) :-
    ( M == A, N == B -> W = V ; true ),
    anti_unify_2(OrdDiffs, M, N, W).
```

8.3 Example Consider the following program

```
p([]).
p([X|Y]) :- p(Y).
```

Let

$$T = \{\text{?- p([]),?- p([1]),?- p([1,2])}\}.$$

The sets of top level instances are then

$$\{\text{p([]).}\}$$

and

$$\{\text{p([1]) :- p([]).}, \text{p([1,2]) :- p([2]).}\}.$$

The least common anti-instances are

```
p([])
```

and

```
p([1|X]) :- p(X).
```

So the first program clause is covered, but the second not. Adding the goal

```
?- p([2,1])
```

to T, the sets of the top level instances will be

$$\{\text{p([1]) :- p([]).}, \text{p([1,2]) :- p([2]).}, \text{p([2,1]) :- p([1]).}\},$$

and the least common anti-instances are

```
p([])
```

and

```
p([X|Y]) :- p(Y).
```

Thus, these goals cover the program.

The above example shows that the notion of a coverage delivers more information than just determining whether a set of goals covers a program clause or not. Moreover, by means of coverage we can virtually evaluate the coveredness of a program clause concerning its arguments and possible instantiations. The lca of the set of top level instances of the clause

```
p([X|Y]) :- p(Y).
```

was

```
p([1|X]) :- p(X).
```

Using the information, one can construct additional goals in order to get a cover. Any goal having the lists first argument different from 1 will lead to a cover. This gives rise for a more detailed coverage notion.

8.4 Definition (Cover of an Instance) 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 coverage notions will be used for test input generation (see section 6) and the generation of test report which will be described in the next section.

9 TEST REPORT AND A SAMPLE SESSION

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

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

- the name of the file of 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 type correct clauses and the total clauses, and 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,
  - the covered clauses and the total clauses of the program, and
  - the covered predicates and the tested predicates.
  - the covered predicates and the total predicates,
- the number of failed *tests*, the number of successful *tests*, the number of failed *goals*, and the number of successful *goals*.

A *full test report* consists in addition to a brief test report of

- the according numbers of the test cases, the type correct clauses, the ratios for the covered clauses, the number of the failed and successful tests and goals for *each predicate*, and
- the least common anti-instance of each clause of each non covered predicate.

```
% types

:- type town ==> newcastle ; carlisle ; penrith ;
                 darlington ; workington ;
                 townA ; townB ; townC.
:- type distance = integer.
:- type pathlist = list(town).
:- type route ==> r(distance, pathlist).
:- type routelist = list(route).

% predicates

:- predtype a(town,town,distance).
:- predtype go(town,town,pathlist).
:- predtype go0(routenlistrouteliste,
                        town,pathlist).
:- predtype proceed(route,town,routelist,pathlist).
:- predtype shortest(routelist,route,routelist).
:- predtype shorter(route,route).
:- predtype legalnode(town,pathlist,town,
                      distance,distance).
:- predtype legal(town,pathlist).

:- predmode a(?,?,?).
:- predmode go(+,+,+).
:- predmode go0(?,+,?).
:- predmode proceed(?,+,?,?).
:- predmode shortest(?,?,?).
:- predmode shorter(?,?).
:- predmode legalnode(?,?,?,+,?).
:- predmode legal(?,?).
```

Figure 2: Instrumentation for the sample program

To illustrate the test environment we sketch a typical test session including a structure check, test case generation and selection, test program execution and test report generation. As a sample program, we use a graph searching program taken from [8]. This program calculates shortest paths between two nodes of a graph. The sample graph describes connections between cities, its edges are labeled with distances. The program consists of 11 predicates and 24 clauses. It is depicted in Figure 3. We have omitted the clauses for the standard `findall` predicate here. The instrumentation, i.e. the type and mode declarations are shown in Figure 2.

As mentioned in section 2, PROTest II is embedded in a product assurance environment and includes tools for program development. For a program under development, the user may perform a structure check and generate test cases at any time. The structure check for the sample program comes up with no type and mode inconsistencies and a "Test Cases" menu offers the selection of predicates to be tested. The clauses of the predicate a/3 represent the graph and consist of facts with arguments defining connections and distances of cities. Thus this predicate is not selected for testing, as well as the standard `findall` predicate. The remaining predicates are selected by the user as depicted in Figure 4.

The test input generator generates 37 test cases for theses predicates which may be edited, i.e. modified, deleted, or extended. 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 as shown in Figure 5.

The user can develop and implement a test program which should properly be integrated in the development and implementation of the Prolog program. The sample test pro-

```
/* Calculation of shortest paths in a graph */

/* A: some distances, defining a graph */
a(newcastle,carlisle,58).
a(carlisle,penrith,23).
a(townB,townA,15).
a(penrith,darlington,52).
a(townB,townC,10).
a(workington,carlisle,33).
a(workington,townC,5).
a(workington,penrith,39).
a(darlington,townA,25).

/* B: program */
go(START,DEST,ROUTE) :-
   go0([r(0,[START])],DEST,R),
   reverse(R,ROUTE).

go0(ROUTES,DEST,ROUTE) :-
   shortest(ROUTES,SHORTEST,RESTROUTES),
   proceed(SHORTEST,DEST,RESTROUTES,ROUTE).

proceed(r(_,ROUTE),DEST,_,ROUTE) :-
   ROUTE = [DEST|_].
proceed(r(DIST,[LAST|TRAIL]),DEST,ROUTES,ROUTE) :-
   findall(
           r(D1,[Z,LAST|TRAIL]),
           legalnode(LAST,TRAIL,Z,DIST,D1),
           LIST),
   append(LIST,ROUTES,NEWROUTES),
   go0(NEWROUTES,DEST,ROUTE).

shortest([ROUTE|ROUTES],SHORTEST,[ROUTE|REST]) :-
   shortest(ROUTES,SHORTEST,REST),
   shorter(SHORTEST,ROUTE),
   !.
shortest([ROUTE|REST],ROUTE,REST).

shorter(r(M1,_),r(M2,_)) :- M1 < M2.

legalnode(X,TRAIL,Y,DIST,NEWDIST) :-
   (
     a(X,Y,Z)
     ;
     a(Y,X,Z)
   ),
   legal(Y,TRAIL),
   NEWDIST is DIST + Z.

legal(_,[]).
legal(X,[H|T]) :-
   X \== H,
   legal(X,T).
```

Figure 3: The sample program, taken from [8, p. 163]

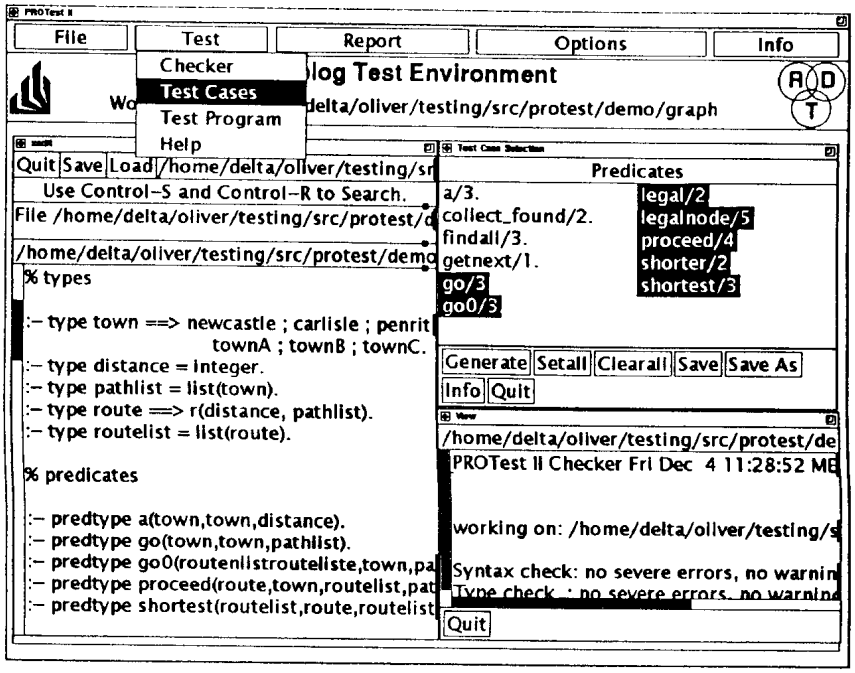

Figure 4: Structure checker and test case selection

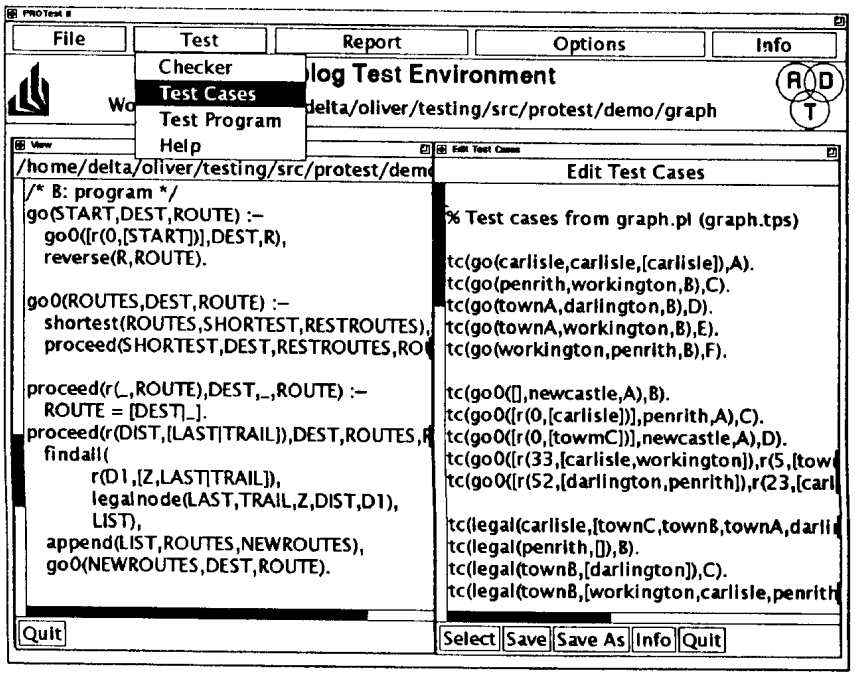

Figure 5: Test case editing

gram is displayed in Figure 6. This test program first tests the predicate go0/3 which is the main predicate and in case of success tests the top level (user interface) predicate go/3. In case a failure occurs, the remaining selected predicates will be tested.

The user can run the dynamic test as soon as the test inputs are generated and a test program is implemented (and verified syntactically correct).

The test report of the sample test run concludes that 4 of 6 tested predicates (67%) are covered but the coverage of the clauses is less (25%) as shown in Figure 7. A closer look at the full test report reveals that neither of the two shorte clauses is covered by 6 test cases out of the 37 generated one which activate these clauses. The test coverage instances

```
shortest([r(V0,[V1|V2])|V3],r(V0,[V1|V2]),V3).

shortest([r(V0,[V1|V2])|V3],V4,[r(V0,[V1|V2])|V5]) :-
    shortest(V3,V4,V5),
    shorter(V4,r(V0,[V1|V2])),
    !.
```

show that the declared types for **shortest/3** are more strict

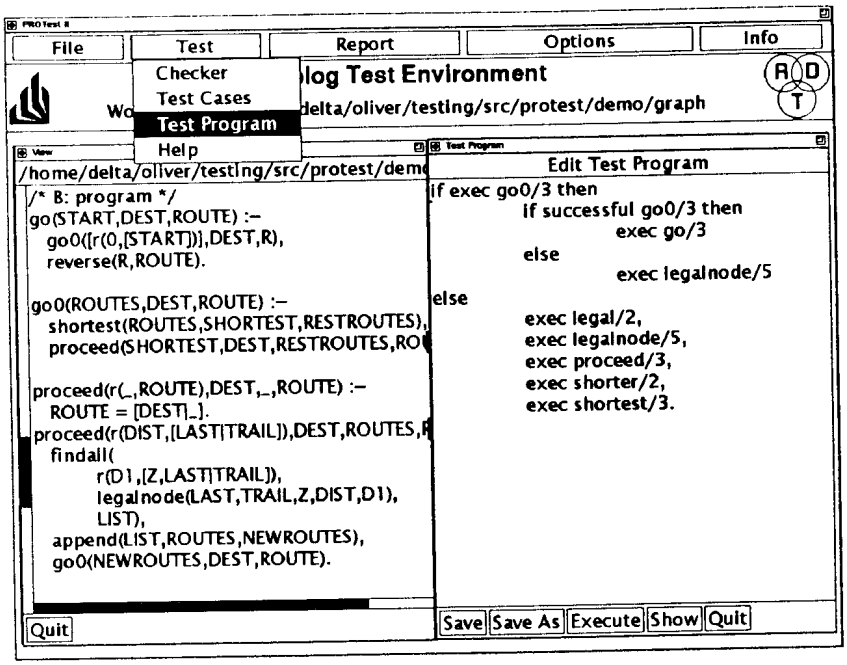

Figure 6: Test program editing

than the types which can be constructed by the clauses of this predicate. Thus, no set of type correct test inputs can cover this predicate. We conclude that the program to be tested is faulty. The corresponding test cases could also localize the fault being in the predicate `shortest/3`.

10 CONCLUSION

We have described basic notions and a rudimentary test theory for logic programming, particularly materialized and integrated into a Prolog programming and test environment. The major concepts are type and mode declarations for programs. Types are represented in terms of parametric type rules which are equivalent to regular types. This provides a concise framework, and the concepts and algorithms can be applied to manipulate regular sets. Thus, types and modes approximate the intended interpretation of a program. Therefore, the framework also enables more readable logic programs. The built-in test language based on Prolog leads to a uniform declarative test environment, and the concept of a test coverage provides detailed analysis and selection of the set of test inputs.

The ongoing research is to extend this environment to an object-oriented extension of Prolog which will be developed on the basis of *pure* Prolog with negation. Another research direction is to incorporate other type schemes which consider *implicit* type information of logic programs and *type inference* [9, 33]. The investigation of other concepts, such as *well typedness* [9] will also be considered.

ACKNOWLEDGEMENT

This work has been supported by Heinz-Nixdorf-Institut, Paderborn, Project number ZIT 521 and ZIT 538.

REFERENCES

[1] ANDERT JR., E. P. Expert system software verification, validation and test. In *Proccedings Second International Conference On Software Quality* (October 1992), American Society for Quality Control, pp. 122–127.

[2] AZEM, A., BELLI, F., AND JEDRZEJOWICZ, P. Reliability assessment of Prolog programs. In *Proceedings European Safety and Reliability Conference* (1993), Elsevier Science Publishers. (to appear).

[3] BAGGI, D., AND SHOOMAN, M. L. An axiomatic driver for pseudo-exhaustive software testing. In *Digest of Papers COMPCON '78* (New York, Feb. 1978), IEEE.

[4] BELLI, F., AND JACK, O. Type/mode-based testing of logic programs. Tech. Rep. 1993/1, Universität Paderborn, FB Elektrotechnik, FG Angewandte Datentechnik, 1993.

[5] BOUGÉ, L., CHOQUET, N., FRIBOURG, L., AND GAUDEL, M.-C. Application of PROLOG to test sets generation from algebraic specifications. In *Proceedings of the Second International Joint Conference on Theory and Practice of Software Development* (Berlin, 1985), Springer Verlag, pp. 261–275. LNCS 186.

[6] BRUYNOOGHE, M. Adding redundancy to obtain more reliable and more readable Prolog programs. In *Proceedings of the First International Logic Programming Conference* (1982), pp. 129–138.

[7] CHANG, K., CROSS, J., CARLISLE, H., AND BROWN, D. A framework for intelligent test data generation. *Journal of intelligent Robotic Systems*, 5 (1992), 147–165.

[8] CLOCKSIN, W., AND MELLISH, C. *Programming in Prolog.* Springer Verlag, Berlin Heidelberg, 1987.

[9] DART, P. W., AND ZOBEL, J. Transforming typed logic programs into well-typed logic programs. Tech. Rep. 88/11, Department of Computer Science, University of Melbourne, 1988.

[10] DEMILLO, R. A. Progress toward automated software testing. In *Proceedings 13th International Conference on Software Enginnering* (1991), pp. 180–183.

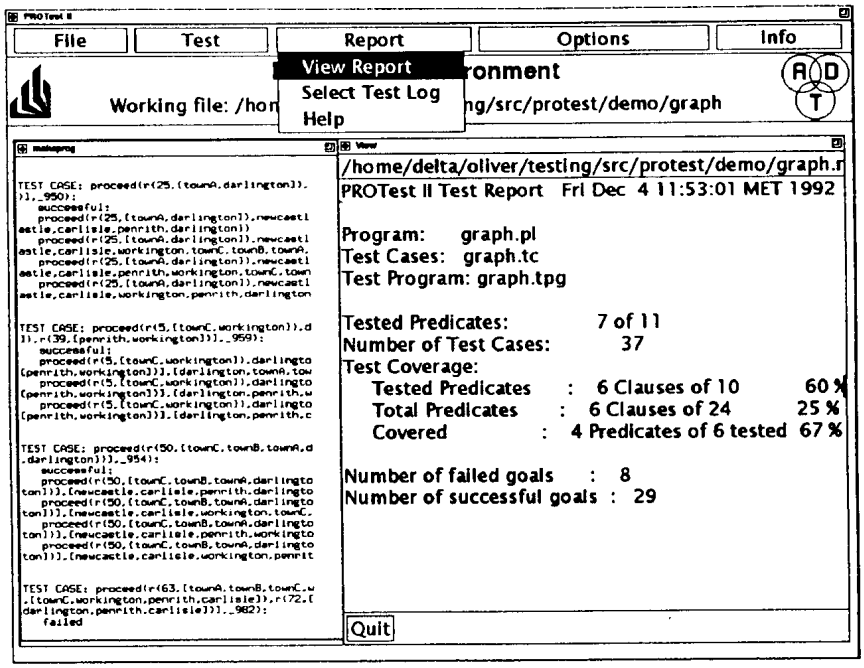

Figure 7: Test report

[11] DENNEY, R. Test-case generation from Prolog-based specifications. *IEEE Software 8*, 2 (Mar. 1991), 49–57.

[12] GANNON, J. Testing tools using formal specification and coverage metrics. *GI Softwaretechnik-Trends 6*, 1 (1986), 5–11.

[13] GOODENOUGH, J. B., AND GERHARD, S. L. Toward a theory of test data selection. *IEEE Transactions on Software Engineering SE-1*, 2 (June 1975), 156–173.

[14] GORLICK, M. M., KESSELMAN, C. F., MAROTTA, D. A., AND PARKER, S. D. MOCKINGBIRD: A logical methodology for testing. *Journal of Logic Programming*, 8 (1990), 95–119.

[15] GUPTA, U., Ed. *Validating and Verifying Knowledge-Based Systems*. IEEE Computer Society Press, 1991.

[16] HOFFMAN, D. M., AND STROOPER, P. Automated module testing in Prolog. *IEEE Transactions on Software Engineering 17*, 9 (1991), 934–943.

[17] HOWDEN, W. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering* (1986), 997–1005.

[18] THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC. *IEEE Standard for Software Test Documentation*, 1983. ANSI/IEEE Std 829-1983.

[19] THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC. *IEEE Standard for Software Verification annd Validation Plans*, 1986. ANSI/IEEE Std 1012-1986.

[20] THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC. *IEEE Standard for Software Unit Testing*, 1987. ANSI/IEEE Std 1008-1987.

[21] THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC. *IEEE Standard Glossary of Software Engineering Terminology*, 1991. IEEE Std 610.12.1990.

[22] LASSEZ, J.-L., MAHER, M., AND MARRIOTT, K. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kauffman, 1988, pp. 587–625.

[23] LLOYD, J. W. *Foundations of Logic Programming*. Springer Verlag, Berlin Heidelberg, 1987.

[24] LUO, G., V.BOCHMANN, G., SARIKAYA, B., AND BOYER, M. Control-flow based testing of Prolog programs. In *Proceedings of the Third International Symposium on Software Reliability Enginnering* (1992), pp. 104–113.

[25] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (1987), 295–307.

[26] MISHRA, P. Towards a theory of types in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming, Atlantic City* (1984), pp. 289–298.

[27] MYCROFT, A., AND O'KEEFE, R. A. A polymorphic type system for Prolog. *Artificial Intelligence 84*, 23 (1984), 295–307.

[28] PEREIRA, L. Rational debugging for logic programs. In *Third International Conference on Logic Programming* (1986), E. Shapiro, Ed., pp. 203–210.

[29] SHAPIRO, E. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachsetts/London, England, 1982.

[30] VAN GELDER, A. Efficient loop detection in Prolog using the tortoise-and-hare technique. *Journal of Logic Programming* (1987), 23–31.

[31] WEYUKER, E. J., AND JENG, B. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering 17*, 7 (July 1991), 703–711.

[32] WILDEN, D. Structured testing of Prolog programs — test case selection through partitioning. Master's thesis, Universität-GH Paderborn, 1992. (in German).

[33] YARDENI, E., AND SHAPIRO, E. A type system for logic programs. *Journal of Logic Programming 91*, 10 (1991), 125–153.