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., PROTest II: Testing Logic Programs, Proc. 2nd International Conference on Software Quality (ASQC), pp.128-135

PROTest II: TESTING LOGIC PROGRAMS

Fevzi Belli Oliver Jack

University of Paderborn
Department of Electrical and Electronics Engineering
Germany

Abstract

Testing of programs has been of great interest in the past decades, and many results have been presented [7, 1, 8]. However, all these systematic approaches are applicable to *imperative* programming but not for *declarative* programming, such as logic programming. In this paper, we describe an approach to testing logic programming materialized in a *test environment*, including a *test language*. The environment consists of *structure analysis* of logic programs, *automatic test case generation*, *automatic test execution*, *test coverage determination*, and *generation of test reports*. Our concept refines a *type* and *mode* scheme for Prolog.

Key words: Logic Programming, Prolog, white-box testing, test environment, validation of rule based expert systems.

1 Basic problems

Logic Programming in an object-oriented environment allows the development of rule based expert systems in a natural way [2]. Although formal aspects of knowledge- based systems constitute a well-researched area, pragmatic aspects of their development are still somehow underdeveloped, e.g. the methods and tools to systematically test Prolog-based expert systems are mostly limited to low level debugging facilities to trace run-time errors ("Testing in the small"). Also only few programming environments consider the advent of methods to handle static-semantic errors, e.g. type constraints, or structural conventions, by a compiler or an interpreter [13].

The following problems emerge when validating a real knowledge-based expert system:

Test Problem Testing large systems requires tools of its own kind. Methods to formally verify large systems become tedious, and moreover error prone, thus not reliable. The then arising question is: *To which extent can we apply the systematic test approaches known from Software Engineering to the logic programming to develop reliable expert systems*, e.g. the Test Theory of Goodenough and Gerhart, or the Domain Theory of Howden, or the Mutation Theory of DeMillo/Lipton etc., just to mention a few of those sound techniques [11, 2]?

Reliability Problem Once one has decided to apply testing to validate a system, he/she would likely have the problem of determining when to terminate testing. A probabilistic approach is given by quantifying the reliability on the assumption that a reliability growth model is available [3] Such a model materializes the idea that the reliability of this program is increased by the successive steps to test a program in order to fix the bugs. In this case, the iterative procedure testing the program, fixing the bugs, and measuring the reliability will be terminated as soon as the measured value of the reliability is satisfactory — or the test budget has been run out and the system cannot be released. The question which arises also here is: *To which extent can we apply the well-known software reliability growth models from the literature to the area of expert systems?*

Tool Problem The very next problem which emerges with the test and reliability problems is in how far the solutions can efficiently be applied. This goal usually requires the development and employment of some special purpose software to involve computers as an assistant (tools). To achieve a comfortable utilization, one has to integrate the tools of different applications having a uniform interface to the user (common tools). Again the question arises: *To which extent can we deploy the lessons we learned from Software Engineering to design and construct tools which support the development of complex systems to the field of expert systems?*

In the next sections we will survey our approach (PROTest II) to solve the first and last problem, i.e. Test Problem and Tool Problem, respectively. PROTest II allows a uniform, white-box 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.

The Reliability Problem is a very tough one as it is still a very controversially discussed subject even in the conventional Software Engineering. We will exclude the Reliability Problem of expert systems in this paper; however it is the subject of ongoing research in our group.

For the description of our concept we use the terminology according to [10].

2 The overall structure of PROTest II

PROTest II (as object-oriented extension of PROTest [2]) consists of a *static* and a *dynamic* component. The *static*

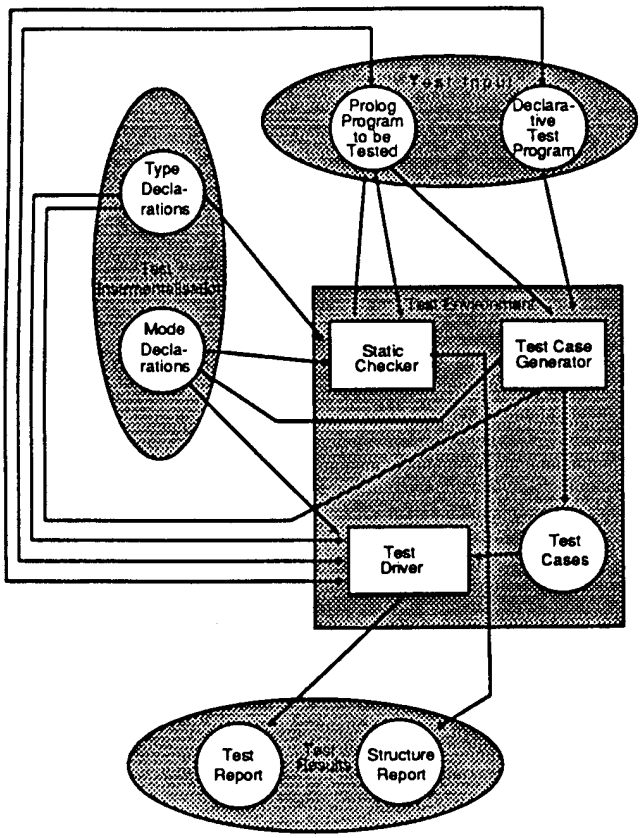

Figure 1: The overall arcitecture of PROTest II

component includes a *type* and *mode* system, tools for *structure checking* of Prolog programs, a *test case generator*, and a *test language*.

The *dynamic* component includes a *test driver* which is an interpreter for the *test language* and a *test reporter* which aggregates the results of the test case execution. Figure 1 depicts the overall architecture of PROTest II in form of a simple Petri net.

3 Adding "Types and Modes" to Prolog

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. Moreover, it is not always clear which of the arguments of a predicate are intended to be input and which to be output. The following example illustrates this.

Example 1. Consider the standard *append* predicate, which is intended to concatenate lists:

$append([], A, A) \leftarrow$
$append([A|B], C, [A|D]) \leftarrow append(B, C, D)$

The "normal" use of append is a call of the form

$\leftarrow append(\langle list1 \rangle, \langle list2 \rangle, L)$

Provided that ⟨list1⟩ and ⟨list2⟩ are lists, the answer then is L = ⟨list1 list2⟩. But also other calls are possible, e.g. $\leftarrow append(L1, L2, L3)$. For the testing of the *append* predicate, one has at least to know about the types of it arguments (which must be lists) and the intended constellations of input and output arguments. The above call is meaningless, because all arguments are output. □

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 extential redundancy. A brief description of such schemes, *type* and *mode schemes* will be given now.

Types

Type schemes have been suggested for several purposes [12, 13, 11]. The type scheme here is based on the Mycroft/O'Keefe scheme [13] and the the scheme of [6].

Our type scheme involves declarations of types and predicates. Type declarations can be given in two forms: *construction* and *union*. Type declarations can also be given arguments. The construction of types is done in the following way:

← *type* ⟨name⟩ ==> ⟨constructors⟩.

where ⟨constructors⟩ is

⟨constructor⟩[; ⟨constructor⟩] * .

Example 2. The construction of a type called $list(T)$ which is a list containing elements of type T will be given as follows:

$\leftarrow type\ list(T)$ ==> $[]; [T|list(T)]$. □

The union of types is declared in the following way:

← *type* ⟨name⟩ = ⟨names⟩[+⟨name⟩] * .

where ⟨names⟩ is

⟨name⟩[+⟨name⟩] * .

Example 3. The declaration of a type called number which is an integer or a float is done as follows:

$\leftarrow type\ number = integer + float.$ □

Note that this is essentially the same as

$\leftarrow type\ number$ ==> $integer; float.$

For technical reasons there are two ways of declaring types. The first is used for declarations of which right hand side contains the term on left hand side, such as $list(T)$, the second is used for the other types declarations, such as number.

The above described scheme enables the declaration of types of *terms* (which are arguments of predicates). The declaration of the types of arguments of *predicates* is done in the following way:

← *predtype* ⟨functor⟩(⟨type⟩, ..., ⟨type⟩).

Example 4. The declaration of the types of the *append* predicate intended to concatenate integer lists is

$\leftarrow$ *predtype* $append(list(integer), list(integer), list(integer))$. □

The scheme behind this declarations is called a *polymorphic* [13] or *parametric* [6] type system. It is given formally by the following definitions.

Definition 1. A *type* is a subset of the Herbrand universe. The *type symbols* α, β, μ and ϕ are used to represent types. The symbol μ represents the Herbrand universe and ϕ represents the empty type $\emptyset$. □

Clearly, a partial order on the set of types is given by set inclusion $\subseteq$, and the set of types forms a lattice. The least type is ϕ and the greatest type is μ.

To have a practical scheme, we have to express types in an easy way to handle and concise language. For this, a regular language is used. Types will then be defined by *parametric type rules* [6]. In order to do so, *parametric type term* is defined first.

In addititon to variables, constants and functions of first-order theory, we consider *type variables*, denoted by ν, and *type functions*, denoted by δ. A type function with arity 0 is called a *type symbol*.

Definition 2. A *parametric type term* is one of the following.

- A constant, a variable and a type variable,
- $f(\alpha_1, \ldots, \alpha_n)$, if f is an n-ary function and α_i, $1 \leq i \leq n$, are parametric type terms,
- $\delta(\alpha_1, \ldots, \alpha_n)$, if δ is an n-ary type function and α_i, $1 \leq i \leq n$, are variable free parametric type terms.

A *pure parametric type term* is a variable free parametric type term. □

At this point it is assumed for convenience, as mentioned above, that the set of constants (of the Herbrand universe) is partitioned in non-empty subsets called *base types* which are used as a starting point for the definition of other types. These base types are *integer*, *float* and *atom* (as used in Prolog). These represent constants for integers, foats and (Prolog) atoms. This assumption is not necessary but overcomes type definitions in terms of enumeration of possibly infinite constants.

Definition 3. A *parametric type rule* is a rule of the form

$\delta(\nu_1, \ldots, \nu_n) \rightarrow \sigma$

where δ is an n-ary type function, $\nu_1, \ldots, \nu_n$ are type variables and σ is a set of pure parametric type terms. In addition, no type variables different from $\nu_1, \ldots, \nu_n$ occur in σ. □

Example 5. The following is a parametric type rule.

$list(\nu) \rightarrow \{[], \nu.list(\nu)\}$

Here "." in $\nu.list(\nu)$ is the usually infix notated list constructor function. □

Next, the connection between *pure parametric type terms* and *types* with respect to a *set of parametric type rules* is given by a mapping.

Definition 4. Let M be a set of such parametric type rules that each type function in M has a parametric type rule in M or is μ, ϕ, or a base type. A mapping T_M from pure parametric type terms to the Herbrand universe is defined as follows.

- $T_M(\alpha) = \alpha$ where α is a constant,
- $T_M(f(\alpha_1, \ldots, \alpha_n)) =$ $\{f(t_1, \ldots, t_n) \mid t_i \in T_M(\alpha_i),\ 1 \leq i \leq n\}$,
- $T_M(\mu) =$ the Herbrand universe,
- $T_M(\phi) = \emptyset$,
- $T_M(\alpha) =$ the type represented by α where α is a base type,
- $T_M(\delta(\alpha_1, \ldots, \alpha_n)) = \cup_{\beta \in \sigma'} T_M(\beta)$ where $\delta(\nu_1, \ldots, \nu_n) \rightarrow \sigma \in M$ and σ' is σ with ν_i is replaced by α_i, $1 \leq i \leq n$.

The type represented by a pure parametric type term given by a set of parametric type rules is defined to be the least fixpoint (of that pure parametric type term) of the mapping T_M (for least fixpoints see [10]). □

It should be mentioned that parametric type rules may generate infinte terms, e.g. $\alpha \rightarrow f(\alpha)$. Such infinte terms are excluded from the types corresponding to pure parametric type terms since types are subsets of the Herbrand universe (which consists if finite terms).

Example 6. Let M be

$\{\alpha \rightarrow \{1\}, list(\nu) \rightarrow \{[], \nu.list(\nu)\}\}$.

Then $T_M(list(\alpha)) = \{[],\ 1.[],\ 1.1.[],\ \ldots\}$. In the commonly used *Edinburg* syntax [5] this would be

$\{[],\ [1],\ [1,1],\ \ldots\}$.

□

Parametric or polymorphic [13] types 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 [6]. But to have parametric types is a notational covenience. One can define a type

$btree(\nu) \rightarrow \{nil, t(btree(\nu)), \nu, t(btree(\nu))\}$,

and if there is a need, define types btree(integer), btree(float), etc.

Several *type checking*, *comparison* and *unification* algorithms [13, 6, 15] can be adopted for this type scheme and will be used in section 4.

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 predicates will be done by a mode scheme, similar to mode declarations used by some Prolog compilers, such as Quintus Prolog, for optimization purposes. These declarations have the form

$\leftarrow$ *predmode* ⟨functor⟩(⟨mode⟩, ..., ⟨mode⟩).

where ⟨mode⟩ is

$+ \mid - \mid ?$.

They will be defined as follows:

- $+$ 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.

Example 7. The mode declarations for the append predicate will be

$\leftarrow$ *predmode* $append(+,+,?)$
$\leftarrow$ *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* [10] thus as an *instrumentalisation* of a logic program. These type and mode schemes enable 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 [4].

4 Static checker

Besides syntax checking of a Prolog program as performed by any interpreter or compiler performs, type and mode declarations can be used for more subtle checking. These checks are *static*, i.e. performed on the source code. Especially, the type declarations describe the *structure* of a program.

Example 8. Consider the following sort program.

$sort([],[]) \leftarrow$
$sort([X|Y1],Y2) \leftarrow$
  $partition(X,Y1,Z1,Z2)$,
  $sort(Z1,Z11)$,
  $sort(Z2,Z22)$,
  $append(Z11,[Y1|Z22],Y2)$.

$partition(X,[],[],[]) \leftarrow$
$partition(X,[Y|Z],[Y|Z1],Z2) \leftarrow$
  $Y < X$,
  $partition(X,Z,Z1,Z2)$
$partition(X,[Y|Z],Z1,[Y|Z2]) \leftarrow$
  $Y \geq X$,
  $partition(X,Z,Z1,Z2)$.

The *type declarations* will be

$\leftarrow$ *type* $numlist = list(number)$
$\leftarrow$ *predtype* $sort(numlist, numlist)$
$\leftarrow$ *predtype* $partition(number,$
  $numlist, numlist, numlist)$
$\leftarrow$ *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. This kind of static type checking might reveal a programming error 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, $sort(Z2,Z22)$. From the type declarations follows that $Y1$ must have a type $list(number)$, since the first argument of *sort* has type $numlist(= list(number))$ and if $[X|Y1]$ has type *numlist* then $Y1$ has type *numlist*. But then the second argument of the *append* atom, $[Y1|Z22]$, does not have the declared 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 the association of types with arguments of a predicate is given.

Definition 5. 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 given in definition 4.

Definition 6. Let A be an atom $p(t_1,\ldots,t_n)$ and θ be a substitution such that $A\theta$ is ground. An 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 $t_i\theta \in \alpha_i$, $1 \leq i \leq n$. An instance of a conjunction $A_1,\ldots,A_m$ of atoms is *type correct* with respect to a typed program P, if each A_i is type correct with respect to P. □

Definition 7. Let $A \leftarrow B$ be a clause in a typed program P. $A \leftarrow B$ is *type correct* if for every instance $(A \leftarrow B)\theta$ of $A \leftarrow B$ such that $A\theta$ is type correct then $B\theta$ is type correct. 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 $\leftarrow Q$ may succeed with an answer θ such that $Q\theta$ is not type correct, e.g. if Q itself is not type correct.

Example 9. The second clause of the *sort* program in example 8 is not type correct. To see this, consider the substitution

$$\theta = \{X/1, Y1/[2,3], Y2/[1,2,3], Z1/[], Z2/[2,3], Z11/[], Z22/[2,3]\}.$$

The instance by θ is then

$sort([1|[2,3]],[1,2,3]) \leftarrow$
$\quad partition(1,[2,3],[],[2,3]),$
$\quad sort([],[]),$
$\quad sort([2,3],[2,3]),$
$\quad append([],[[2,3]|[2,3]],[1,2,3]).$

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*. □

Checking clauses of a typed program for type correctness can be done by adopting type unification [6].

5 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 per hand, PROTest II includes a test case generator. Test inputs will be generated automatically; moreover, the generated test inputs are partitioned in several classes derived by the mode declarations [14].

Following, we precise this concept.

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

Definition 8. A *test input* for a program P is a goal $\leftarrow Q$ where $Q = p(t_1,\ldots,t_n)$ and $p \in def(P)$. □

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.

Definition 9. An *intended output* for a test input $\leftarrow Q$ 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.

Definition 10. The *output* of a test case $\leftarrow Q$ and a program P is the set of instances of Q derived by the computed answers via SLD-refutations [10] of $\leftarrow Q \cup P$. □

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

Definition 11. 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. □

Example 10. A test input for the sort program of example 8 is

$\leftarrow sort([3,5,1,7,10,0],S).$

The intended output is

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

Another test input is

$\leftarrow partition(5,X,Y,Z).$

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 give the directive which arguments of a test input $\zeta = \leftarrow Q$, $Q = \mathrm{p}(t_1,\ldots,t_n)$, 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. Elements of a type are generated randomly.

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 gives a classification of test cases.

In practice, it is not necessary to test all predicates, e.g. a program can consist of facts constituting a predicate such as the representation of a graph by facts $arc(x,y) \leftarrow$. 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 should be generated. After generation of the test inputs, the user has to supply with the corresponding intended outputs, if he/she whishes comparisons in addition to the ones automatically generated according to types and modes. At this point, we have to make a pragmatic decision. As mentioned above, the output as well as the intended output for a test input may be infinite. In this case, there is simply a finite subset to be specified by the user. This is for the intended output. According, also for the output only a finite subset is to be considered. This takes place in the test program according to a test language which will be described in the following section.

6 Test language

PROTest II includes a *built-in test language* DTL/1 (**Declarative Test Language**) based on Prolog. This enables a *declarative* testing style.

DTL/1 consists of predicates delivering following services:

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

Parametrized calls

A *parametrized* call of a test input is the following.

$exec(\langle\text{atom}\rangle[,\langle\text{parameters}\rangle]),$

where $\langle$atom$\rangle$ is

$p/n \mid p(t_1,\ldots,t_n)$

and $\langle$parameters$\rangle$ is

$maxsolutions[,mode(m_1,\ldots,m_n)].$

A parametrized call with $\langle$atom$\rangle = p/n$ and without $\langle$parameters$\rangle$ results in calling *all* test inputs available for the predicate p with arity n. The intended outputs will be looked up for the number of answers specified, and outputs

upto 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 ⟨atom⟩ = $p(t_1, \ldots, t_n)$ 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 $mode(m_1, \ldots, m_n)$ the execution of the test inputs of the partition according to $mode(m_1, \ldots, m_n)$. If *maxsolutions* is a variable then no restriction on the number of answers is given, and if *maxsolutions* = *spec* then the maximum number is that specified in the intended output.

Validation of test results

Validation of the test results will be performed as follows:

successful(⟨atom⟩[, ⟨parameter⟩]),

where ⟨atom⟩ is

$p/n \mid p(t_1, \ldots, t_n)$

and ⟨parameter⟩ is

$mode(m_1, \ldots, m_n)$.

successful checks the output of a test input whether it equals the intended output. The parameters are the same as for *exec*. It fails if the according test inputs have not been executed.

type_correct(⟨atom⟩[, ⟨parameter⟩]),

where ⟨atom⟩ is

$p/n \mid p(t_1, \ldots, t_n)$

and ⟨parameter⟩ is

$mode(m_1, \ldots, m_n)$.

type_correct ckecks the output of a test input whether it is type correct with respect to type declarations by implementing the concepts of section 3.

Example 11. A typical test program for the sort program of example 8 will be

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

□

7 Test coverage determination

The notions of test coverage for imperative programming based on the execution of branches is not adequate for Prolog. Prolog's execution model (the procedural semantics) is based on SLD resolution with the central concept of unification [10]. Thus, we have to base a cover notion on this execution model. We follow the standard terminology of [10] enriched by the terminology introduced in [9].

Instances and anti-instances

Definition 12. 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 $E' \sim E$. □

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

Definition 13. 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 $E' \geq E$ for all common anti-instances E' of S. □

If a least element is added to the set of expressions, then $\geq$ forms a lattice [9] (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 immediatly from definition 13 that an lca is unique modulo $\sim$.

Example 12. 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. □

Covers

Definition 14. Let T be a finite set of goals of the form $\{\leftarrow G_1, \ldots, \leftarrow G_m\}$, where $G_i = \mathrm{p}(t_{i1}, \ldots, t_{in})$.

Let $C_1, \ldots, C_q$ be the definition of the predicate 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 $\leftarrow G_i \in T$.

T is a *cover for a program clause* C_1, 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 from the above definitions 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.

This definitions are of practical use, because there are algorithms to compute the lca of a finite set of expressions [9].

Example 13. Consider the following program

$p([]) \leftarrow$
$p([X|Y]) \leftarrow p(Y)$.

Let $T = \{\leftarrow p([]),\ \leftarrow p(1),\ \leftarrow p([1,2])\}$. The sets of top level instances are then $\{p([])\}$ and $\{p([1]) \leftarrow p([]),\ p([1,2]) \leftarrow p([2])\}$. The least common anti-instances are $p([])$ and $p([1|X]) \leftarrow p(X)$. So the first program clause is covered, but the second not. Adding the goal $\leftarrow p([2,1])$ to T, the sets of the top level instances will be $\{p([1]) \leftarrow p([]),\ p([1,2]) \leftarrow p([2]),\ p([2,1]) \leftarrow p([1])\}$, and the least common anti-instances are $p([])$ and $p([X|Y]) \leftarrow p(Y)$. Thus, these goals cover the program. □

The above example shows that the notion of a cover supports more information than just determine whether a set of goals covers a program clause or not. The lca of the set of top level instances of the second clause was $p([1|X]) \leftarrow p(X)$. This gives information *to which extend* the clause is covered. 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 cover notion.

Definition 15. 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 cover notions will be used for the generation of test report which will be described in the next section.

8 Test report

A *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 test and goals for *each predicate*, and
- the least common anti-instance of each clause of each non covered predicate.

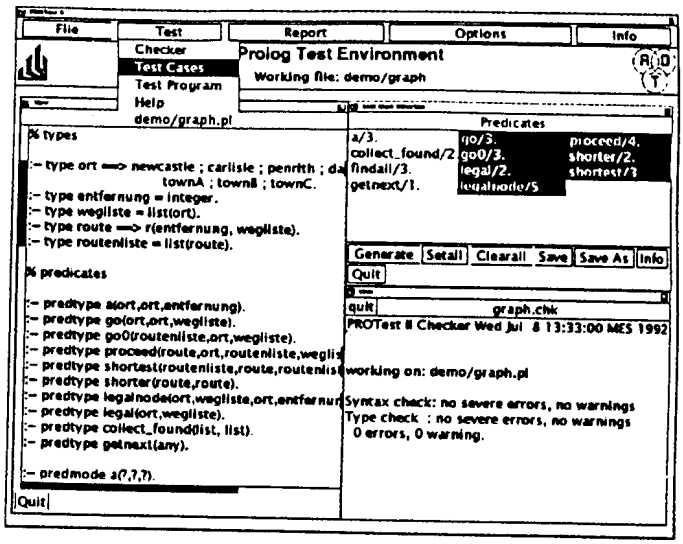

Figure 2: Static checker and test case selection

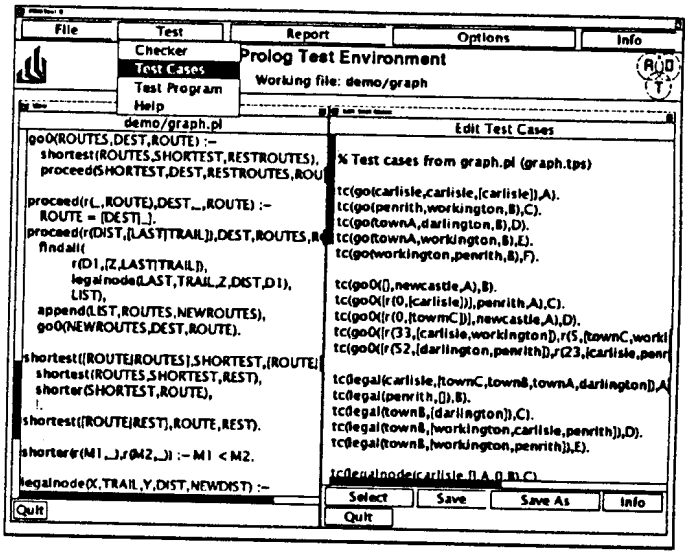

Figure 3: Test case editing

9 A sample session

A typical session contains following stages:

- selection of predicates to be tested (Figure 2),
- editing the test cases (Figure 3),
- editing the test program (Figure 4),
- test report (Figure 5).

10 Conclusions

We have decribed basic notions and a rudimentary test theory for logic programming, particulary materialized and informly integrated into a Prolog production and test environment. The main 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

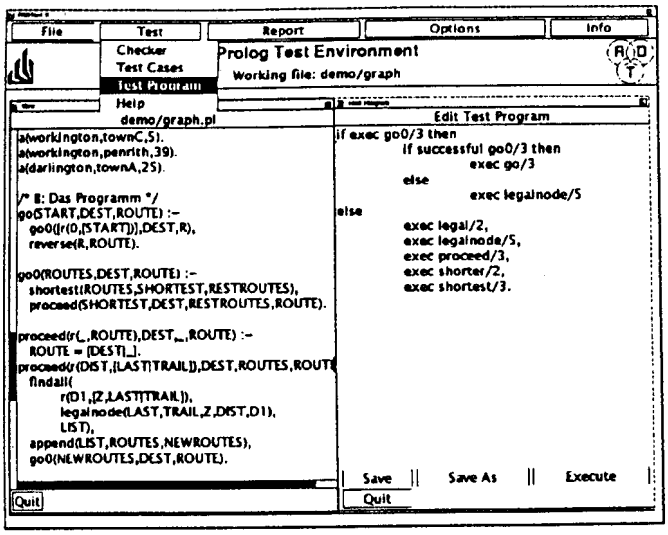

Figure 4: Test program editing

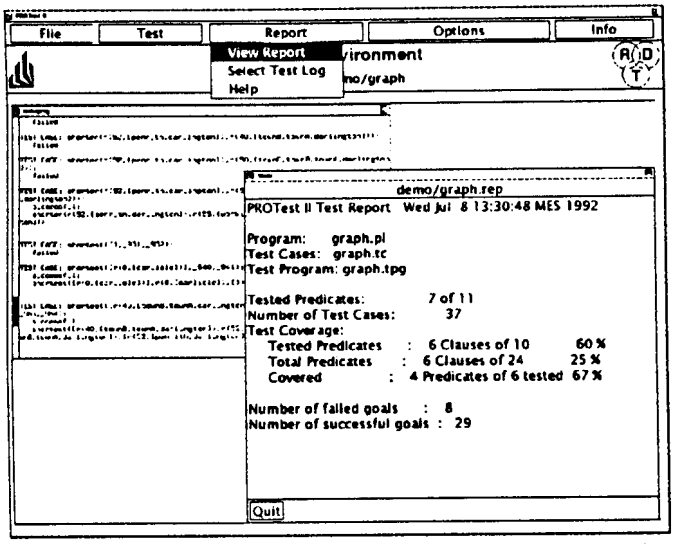

Figure 5: Test report

a concise theoretical framework, and the concepts and algorithms to manipulate regular sets can be applied. Types and modes can be seen as an approximation of the intended interpretation of a program. Thus, the framework also enables better readable logic programs. The built-in test language based on Prolog leads to a uniform test environment, and the concept of a test cover 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* [6, 15]. The investigation of other concepts, such as *well typedness* [6] will also be considered.

Acknowledgments

The project will be supported by the Heinz-Nixdorf-Institute, Paderborn. Our thanks are due to Dittmar Wilden and Lee Naish for the fruitful hints and discussions.

References

[1] D.L. Baggi and Martin L. Shooman. An axiomatic driver for pseudo-exhaustive software testing. In *Digest of Papers COMPCON '78*, New York, Feb. 1978. IEEE.

[2] F. Belli, H. Bonin, H. Gerdes, W. Filipowicz, and P. Jedrzejowicz. Some aspects on the development and validation of FIREX — a knowledge-based system for the transport of dangerous goods and fire department consulting. In *Proceedings 2nd Intern. Conference of IEA/AIE-89*, pages 680–689, June 1989.

[3] Fevzi Belli and Piotr Jedrzejowicz. Fault-tolerant programs and their reliability. *IEEE Transactions on Reliability*, 39(2):184–192, June 1990.

[4] M. Bruynooghe. Adding redundancy to obtain more reliable and more readable Prolog programs. In *Proceedings of the First International Logic Programming Conference*, pages 129–138, 1982.

[5] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, Berlin Heidelberg, 1987.

[6] Philip W. Dart and Justin Zobel. Transforming typed logic programs into well-typed logic programs. Technical Report 88/11, Department of Computer Science, University of Melbourne, 1988.

[7] John B. Goodenough and Susan L. Gerhard. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[8] W.E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, pages 997–1005, 1986.

[9] J.-L. Lassez, M.J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kauffman, 1988.

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

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

[12] Prateek Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming, Atlantic City*, pages 289–298, 1984.

[13] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 84(23):295–307, 1984.

[14] Dittmar Wilden. Strukuriertes Testen von Prolog-Programmnen — Testfallauswahl durch Partitionierung. Master's thesis, University of Paderborn, 1992.

[15] Eyal Yardeni and Ehud Shapiro. A type system for logic programs. *The Journal of Logic Programming*, 91(10):125–153, 1991.