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: Azem, A., Belli, F., Jack, O., Jedrzejowicz, P., Testing and Reliability of Logic Programs, Proc. 4th IEEE International Symposium on Software Reliability Engineering (ISSRE), pp.318-327

Testing and Reliability of Logic Programs

Alireza Azem[a] Fevzi Belli[a] Oliver Jack[a] Piotr Jedrzejowicz[b]

[a]University of Paderborn, Dept. of Electrical and Electronics Engineering, Paderborn
[b]Merchant Marine Academy, Dept. of Computer Science, Gdynia

Abstract

The systematic approaches to testing and reliability determination of programs e.g. [8, 9, 17, 20] are applicable to *imperative* programming but not immediately to *declarative* programming, such as logic programming, which is of great importance to develop knowledge-based systems. In this paper, we describe an approach to implementation-based testing and reliability determination of logic programs materialized in a *product assurance environment*, presently limited to two major components: the *test environment* PROTest and the *reliability assessment environment* PRORool, with the results of the former serving as input for the latter. The test environment consists of structure analysis of logic programs, automatic test case generation and execution, test coverage determination, and generation of test reports. The reliability assessment environment provides an approach to *reliability prediction* and *estimation* of Prolog programs, introducing two measures describing Prolog programs *complexity*, which are used to determine the program reliability. It implements also several well-known software reliability models for comparison purposes.

Key Words: White-Box Testing, Software Reliability, Software Complexity Measures, Logic Programming, Prolog

Introduction

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 and determine their reliability are mostly limited to low level debugging facilities. The following problems emerge when validating a real knowledge-based system:

Test Problem. Testing large systems requires methods of its own kind. Methods to formally validate and verify [3] 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 knowledge-based 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 [8, 9, 16]?

Reliability Problem. Once one has decided to apply testing to validate a system, one would likely have the problem of determining when to terminate testing. Further, with the increasing complexity, cost and size of software sytems the measurement, prediction and estimation of software product characteristics and their current status during development have become increasingly important. Reliability is one of the most important of those characteristics and has proved to be the most readily quantifiable of the attributes of software quality [18]. One possible (probabilistic) approach to determine reliability in order to meet the mentioned needs is given by quantifying it on the assumption that a reliability growth model is available (e.g. [17]). Such a model materializes the idea that the reliability of the program is increased by successive steps to test the program in order to remove the faults. In this case, the iterative procedure of testing the program, removing the faults, and determining the reliability will be terminated as soon as the achieved 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 knowledge-based 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 (*tool*). 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 knowledge-based systems?* The tool problem leads to an environment to assure the production of software, i.e. to assure that cost and schedule limits will not be injured, the required quality will be achieved, etc.

In the following the above mentioned problems will be discussed for the case of logic programming materialized in Prolog, which is of great importance to

develop knowledge-based systems. The concepts discussed below base on previous work [1, 2], refining the test concepts and introducing a fully new approach to determine reliability, consisting of prediction as well as estimation. The tool problem is presently limited to the test and reliability aspects.

PART A: TEST PROBLEM

The logic programming-related issues in the area of software testing which have been intensively researched concentrate on debugging and termination checking [21, 23].

Generating test data form Prolog-based specifications also has been investigated [6]. 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.

There exist some significant 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 cannot 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* of Prolog programs by adding *types* and *modes* enable efficient implementation-based fault analysis and test input generation. In the following we will survey our approach to the test problem for Prolog programs. For the description of our concept we use the terminology according mainly to [15, 8, 10].

1 PROTest

PROTest (RROlog Test environment) 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 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. In some Prolog systems a type scheme is provided [19]. 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. The test input generator uses the instrumentation of a program for generation of test inputs.

Test coverage information is provided by the test coverage analyzer. Program clauses are covered if the test coverage instances are variants of them. 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.

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 in this section.

Fault Model

A *fault* is textual problem with the program resulting from a mental mistake by a programmer, and the mental mistake is defined as an *error* [10]. 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 [15], but relies on the declarativeness of logic programming. The focus is on the construction and derivation of *term structures* through predicates. These term structures 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.

Types

Our type scheme [2, 16] 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 1 The construction of a type called $list(T)$ which is a list containing elements of type T will be given as follows:

← *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 2 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:

$\leftarrow$ *predtype* ⟨functor⟩(⟨type⟩, ..., ⟨type⟩).

Example 3 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 these declarations is called a *polymorphic* or *parametric* type system. It is defined formally for PROTest in [2].

Parametric or polymorphic 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. 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.

Modes

Next, a much simpler but also concise and powerful concept to select meaningful test cases 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 [2], 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

+ | − | ?.

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 4 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* [15, 21] thus as an *instrumentation* 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 section.

2 Components of PROTest

Static 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 *structure* of a program. The task of statically checking a program is performed by the structure checker of PROTest.

Example 5 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

$\leftarrow$ *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. □

Test Case Generation

After having checked a program for static type errors (and, if necessary, 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 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 [24].

Following, we precise this concept (see also [2]).

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

Definition 1 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 2 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 3 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 [15] of $\leftarrow Q \cup P$. □

The output of a test case for a program may be infinite. We will handle this further in this section.

Definition 4 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 6 A test input for the sort program of example 5 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 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 wishes 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. Accordingly, also for the output only a finite subset is to be considered. This takes place in the test program according to the ***built-in test language*** DTL/1 (Declarative Test Language), which is based on Prolog and is included in PROTest. DTL/1 will be utilized to perform integration test and to log the observed test outcomes as parameters of the reliability determination.

PART B: RELIABILITY PROBLEM

3 Reliability Determination

Software reliability is defined as the probability of failure free operation of a computer program for a specified time in a specified environment [18]. Alternative ways of expressing reliability is determining the failure intensity or the hazard rate. Software reliability models (e.g. [7, 11, 14, 17, 20, 22]) specify the dependence of the failure process on fault detection and localization, fault removal and the operational environment.

The problem of reliability determination is intensively researched and thus well-understood for the conventional programming paradigms, e.g. procedural programming deploying an imperative style. The declarative way of logic programming, in general, and Prolog [4], in particular, offers a powerful method for the construction of software e.g. knowledge-based systems, database applications etc.

PRORool (PROlog based environment for Reliability dertermination of Object-Oriented and Logic programs) provides at the present a reliability assessment environment introducing an approach for reliability determination of Prolog programs, implementing also several well-known software reliability models for comparison purposes.

It is assumed that Prolog programs can be divided into *segments*. A segment corresponds to a set of clauses providing an overall functionality. We note that the notion "segment" corresponds to the notions "module", "component", "part", or "path" we synonymously use in conventional programming.

In the following we will survey our approach to determination of Prolog programs reliability and the reliability assessment environment. To account for special features of logic programming and Prolog, two complexity measures are introduced. The *structural complexity measure* reflects the structural, i.e. static, program characteristics. The *operational complexity measure* reflects its operational, i.e. dynamic, characteristics (see also [12, 5]).

Structural Complexity Measure

The introduction of the structural complexity measure is based on the underlying assumption that software failures are more likely to occur within program segments (or programs) that are more complex. Structural complexity, in case of Prolog programs, proved to be in our experience a better indicator of the program's state than conventional measures like "faults per lines of code" or "failure rate per class of statements", which are used in some other failure modelling methods (see for example [13]). The reason for the above statement is that the structural complexity measure is directly based on the segments of a Prolog program identifying potential sources of failures.

Consider a Prolog program consisting of n segments. The structural complexity of a given segment is defined as the sum of the structural complexities of all its clauses. The structural (static) complexity of the clause i belonging to segment k is defined as

$$W_{ki} = \begin{cases} \sum_{j=1}^{n_{ki}} G_{kij} & : \text{clause } i \text{ is a rule} \\ X_{ki} & : \text{clause } i \text{ is a fact} \end{cases} \quad (1)$$

with

k	index refering to segment k
G_{kij}	static complexity of predicate j in the body of clause i
n_{ki}	number of predicates in the body of clause i
X_{ki}	static complexity of fact i.

The structural complexity of the predicate j in the body of clause i is determined through

$$G_{kij} = \frac{1}{n_{ki}} M_{kij} A_{kij} \quad (2)$$

$$A_{kij} = \begin{cases} 1 & : * \\ 2^{S_{kij}} - 1 & : S_{kij} > 0 \end{cases} \quad (3)$$

with

*	predicate j is a fact or a recursive call of clause i or $S_{kij} = 0$
M_{kij}	disallocation factor: total number of predicates in the body of all clauses representing the predicate j; if a predicate in the body of a clause representing predicate j is a recursive call of j, it contributes the increment 1 to the total number; $M_{kij} = 1$ if predicate j is a fact or a built-in predicate
S_{kij}	total number of arguments of the clauses representing the predicate j.

The structural complexity of a fact i in segment k is determined through

$$X_{ki} = X'_{ki} X''_{ki} \quad (4)$$

$$X'_{ki} = \begin{cases} 1 & : S'_{ki} = 0 \\ 2^{S'_{ki}} - 1 & : S'_{ki} > 0 \end{cases} \quad (5)$$

$$X''_{ki} = \begin{cases} 1 & : S''_{ki} = 0 \\ \prod_{l=1}^{S''_{ki}} (2^{S''_{kil}} - 1) & : S''_{ki} > 0 \end{cases} \quad (6)$$

with

S'_{ki}	number of variables within the arguments of fact i
S''_{ki}	number of arguments of fact i, which are not variables and not ground
S''_{kil}	number of arguments of the top level functor of term l within the arguments of fact i, which are variables.

The structural (static) complexity of segment k is then given through

$$W_k = \sum_{i=1}^{n_k} W_{ki} \tag{7}$$

with

n_k number of clauses in segment k

and the overall program structural complexity through

$$W = \sum_{k=1}^{n} W_k \tag{8}$$

Operational Complexity Measure

The operational profile of a program, which considers its dynamic characteristics, influences at least one important reliability characteristic. This characteristic is the relative change of failure intensity per failure experienced. The failures that tend to be experienced during a period of execution are associated with the related input states. During testing activity each failure that is experienced will affect the failure intensity (in general a failure experienced generates some repair activity and the result of the repair is a decrement in failure intensity). Some conclusions about this can be drawn from analyzing which program segments would be accessed and to which extent. The operational complexity of a segment k is defined through

$$W'_k = \frac{B_k}{1+\beta_k}(H_k + R_k) \tag{9}$$

with

- B_k utilization factor reflecting access frequency of segment k during program operation
- β_k maximal backtracking degree (maximal number of choice points for deriving goals) of segment k during testing
- H_k total number of accesses of segment k during testing, i.e. total number of all clause executions of segment k (not considering recursive calls)
- R_k total number of all direct recursive clause executions of segment k during testing (only the direct recursive calls are counted here)

and the overall program operational complexity through

$$W' = \sum_{k=1}^{n} W'_k \tag{10}$$

The utilization factor B_k, representing the utilization of the segment k due to the user behavior during the program operation, can be identified during program operation by tracking the segment accesses for a given period of time during program operation and normalizing the access number of each segment through the minimum segment access count.

4 Complexity Measures Versus Prolog Programs Reliability

The introduction of the two complexity measures discussed above is based on the underlying assumption that software failures are more likely to occur within program segments (or programs) that are more complex.

Looking more closely to the defintion of the structural complexity of a predicate, the ratio $\frac{1}{n_{ki}}$ (see (2)) represents a relative contribution of each predicate within the body of clause i. The disallocation factor M_{kij} relates to the fact that the functionality of a predicate j in the body of clause i is disallocated elsewhere (this feature is comparable to procedural abstraction in conventional programming languages). Furthermore, the value of A_{kij} (see (3)) reflects the feature that Prolog does not distinguish between data types and that each argument within a predicate can be associated with a value of either the right type or the wrong one. In the latter case a failure is to be expected. With value of each argument being either of the right type or not, there are $2^{S_{kij}}$ possible combinations of value settings with only one where all arguments are of the right type and $2^{S_{kij}} - 1$ cases having at least one argument not of the right type.

In case of Prolog programs the three introduced features of the operational profile may prove useful when trying to model its affect on failure behavior: the *backtracking degree*, the extent of recursive and non-recursive executions, and the access frequency of program segments during program operation. The more trials the program segments perform to find alternative solutions through backtracking, the more chances exist for the program to find solutions and not to fail, so backtracking does affect the operational profile, i.e. the higher the backtracking degree, the lower the failure contribution.

As mentioned above, the access frequency of program segments during program operation is considered through the operational profile. This feature accounts for different user behaviors and different utilization of program segments during operation. Consider a system with a faulty segment, which is never used during operation, so the user observes a high program reliability. Another user may need the functionality of that faulty segment frequently, so this user observes a poor program reliability.

5 Determining Program Reliability

Determining the reliability of a computer program can be accomplished by identifying the values of the parameters of the respective reliability model through either prediction or estimation. Prediction is needed whenever system engineering studies are required before failure data are available (before testing) or are not adequate to enable a meaningful estimation. Estimation allows to determine the reliability change during time and/or to control whether the reliability objective is achieved.

Reliability Prediction

The basic execution time model as proposed in [18] is considered. The initial program failure intensity is given through [18]

$$\lambda_0 = fK\omega_0 \tag{11}$$

with

- f linear execution frequency of the program (the average instruction rate divided by the number of object instructions in the program)
- K fault exposure ratio (represents the fraction of time that the program run results in a failure)
- ω_0 number of inherent faults (faults existing before testing begins).

In the traditional approach, it is assumed that the number of inherent faults is linearly related to program size and that the value of the fault exposure ratio is determined from similar programs. In case of Prolog programs, it is believed that better predictions can be obtained by using correlation between the number of faults and structural complexity, as well as between failure exposure ratio and the operational complexity:

$$\omega_0 = \gamma_0 W \tag{12}$$

$$K = \sum_{k=1}^{n} K_k \frac{W'_k}{W'} \tag{13}$$

$$K_k = \xi W'_k \tag{14}$$

with

- γ_0 inherent faults per unit of overall structural complexity (fairly constant for similar programs)
- K_k fault exposure ratio of segment k
- ξ constant parameter determined from similar projects.

Substituting (14) in (13) yields

$$K = \frac{\xi}{W'} \sum_{k=1}^{n} W_k'^2 \tag{15}$$

The values of W'_k are not known before testing has commenced. It is suggested to use either a single uniform value averaged from similar projects or predict values of W'_k basing on correlation between W_k (or W) and W'_k for similar projects.

Taking advantage of additional information available during the early stages of testing an alternative way of determining K is suggested. The early stages of testing is ment to be the period when test data and the available information are not sufficient to estimate the reliability characteristics of the software product and yet some information and possibilities for improving pre-testing predictions are already at hand. Consider the situation after i faults have been already removed (i being a sufficiently small integer). The program failure intensity is then given through

$$\lambda_i = fK\omega_i \tag{16}$$

$$\omega_i = \omega_0 - i \tag{17}$$

with ω_i representing the number of faults remaining after i faults are removed.

K is then calculated as before except that instead of the predicted values of W'_k one may use the real values obtained from averaging over the test runs that have already been performed.

Reliability Estimation

Accounting for an exponential distribution of the time to failure of an individual fault

$$f_a(t) = \phi \exp(-\phi t) \tag{18}$$

with a per fault hazard rate $z_a(t)$ being a constant, i.e.

$$z_a(t) = \phi \tag{19}$$

a Poisson type model considering this feature was proposed in [7]. Further, for a Poisson type model the following relationships apply (see [18])

$$\mu(t) = \omega_0 F_a(t) \tag{20}$$

$$\lambda(t) = \omega_0 f_a(t) = \frac{\mathrm{d}\mu(t)}{\mathrm{d}t} \tag{21}$$

with

- t time
- i failure number
- $\mu(t)$ expected number of failures at time t
- $\lambda(t)$ failure intensity
- ω_0 mean value of the random variable representing the inherent faults
- $F_a(t)$ time to failure distribution of an individual fault
- $f_a(t)$ time to failure density of an individual fault.

In this paper we suggest that, in case of Prolog programs, the constant ϕ can be related to the program characteristics and be best estimated using the previously defined complexity measures. Consider the relative segment structural and operational complexity measures

$$U_k = \frac{W_k}{\sum_{l=1}^{n} W_l} = \frac{W_k}{W} \tag{22}$$

$$U'_k = \frac{W'_k}{\sum_{l=1}^{n} W'_l} = \frac{W'_k}{W'} \tag{23}$$

Value of U_k is known before testing has commenced and can be taken as constant through program operation (except large changes to the program are made due to the removal of faults; in this case U_k must be recomputed). Value of U'_k can be estimated during operation (by, for example, averaging numbers of previous accesses, backtrackings, etc. for a given user or a paticular type of applications). U'_k is, in general, variable in time and for a user/application type. Denoting by $U_k(t_i)$ the value of the relative segment structural complexity estimated at the given time of failure i, i.e. at the time t_i, we propose to define a simple measure of the "program failure potential" at time t_i as

$$V(t_i) = \sum_{k=1}^{n} U_k(t_i) U'_k(t_i) Y_k(t_i) \tag{24}$$

with

$$Y_k(t_i) = \frac{N_{fk}(t_i)}{N_f(t_i)} \tag{25}$$

and

$N_{fk}(t_i)$ total number of test runs up to the moment t_i which led to a failure encountered within segment k

$N_f(t_i)$ total number of test runs up to the moment t_i which led to a failure.

$Y_k(t_i)$ relates U_k to program operation. The reasoning for doing so can be described as follows. Suppose a segment k having a high value of U_k which causes no failures during the program operation (or testing), so if one takes only U_k as is into consideration one would wrongly consider a bad state for segment k although no faults due to this segment were detected during operation (or testing). In contrary, a segment with a low value of U_k, which is highly faulty, would not be sufficiently considered in the failure modeling of the program. One must also keep in mind, that the fact of a high or low value for U_k does not automatically imply a faulty or not faulty state of segment k respectively. A high value of U_k solely represents that a faulty state of segment k is more probable; this can be validated, however, during the program operation, better during the fault detection and correction process (i.e. during testing).

For a representative set of Prolog programs, it has been observed that as faults are detected and corrected during testing, the term $\frac{N(t_i)V(t_i)}{N_f(t_i)}$ approaches a value that is changing in a small interval. $N(t_i)$ denotes the total number of test runs up to t_i. Denoting the total execution time up to the last detected and corrected fault with t_t, an estimate for ϕ is given by

$$\text{Est } \phi = \rho \frac{N(t_t)V(t_t)}{N_f(t_t)} \tag{26}$$

where ρ is a constant factor for a class of Prolog programs.

It can be shown that

$$\mu(t) = \omega_0(1 - \exp(-\phi t)) \tag{27}$$

$$\lambda(t) = \omega_0 \phi \exp(-\phi t) \tag{28}$$

Using "Maximum Likelihood Estimation (MLE)" (see for example [18, 22]), the estimator for ω_0 is the solution to the following relationship

$$\frac{N_f(t_t)}{\omega_0} = 1 - \exp(-\phi S) \tag{29}$$

with

$$S = \sum_{i=1}^{N_f(t_t)} t'_i \tag{30}$$

and

t'_i execution time between failures $i-1$ and i.

The reliability approach discussed above was applied to several Prolog programs. Following table summarizes the results for the failure intensity λ of one of the programs after detecting and correcting 15 faults:

Real Life	A	B	C
$0.70s^{-1}$	$0.26s^{-1}$	$0.35s^{-1}$	$0.37s^{-1}$

with A representing estimation after the execution time model [18], B representing prediction after (11) to (15) and C representing prediction after (16) and (17) (with $N = 20$). All predictions are underestimated compared with real life data. It can be seen that using the feauture oriented approach proposed in this paper yields better predictions (cases B and C).

For estimation, with $\phi = 0.055s^{-1}$ (determined after data in [25], setting $\rho = 1s^{-1}$) the failure intensity of the program was determined to $\lambda = 0.37s^{-1}$, which is comparable to the MLE estimate of $\lambda = 0.36s^{-1}$.

PART C: TOOL PROBLEM

6 Product Assurance Environment

To give an adequate answer to the problems discussed in the introduction a product assurance environment, embedding PROTest and PRORool, was developed under Unix and X Windows V11 R5. It serves as an interface between PROTest and PRORool. The cooperation between the test and the reliability assessment environment is, thus, possible in an integrated uniform environment. The results of the test environment PROTest can immediately serve as input for the reliability assessment environment PRORool. The reliability models of Musa [17, 18], Goel and Okumoto [7], Jelinski and Moranda [11], Schick and Wolverton [20] and the reliability prediction and estimation approach proposed in this paper are implemented in PRORool to date; for the numerical computations needed to determine model parameters, "Maximum Likelihood Estimation" (MLE) method was implemented.

7 A Sample Session

A sample session of using PROTest and PRORool is explained in the following.

PROTest

A typical session of PROTest contains following stages:

- selection of predicates to be tested,
- editing the test cases,
- editing the test program,
- test report (Figure 1).

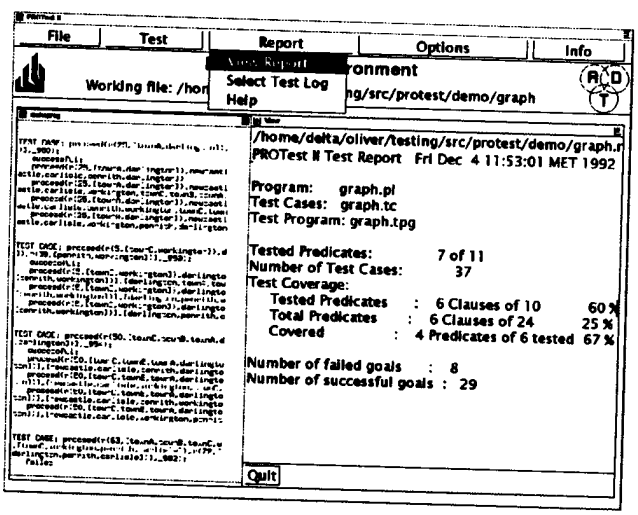

Figure 1: Test report

PRORool

A typical session of PRORool contains:

- selection of the input data set,
- selection of the model type,
- reliability parameter calculation,
- viewing the results (Figure 2).

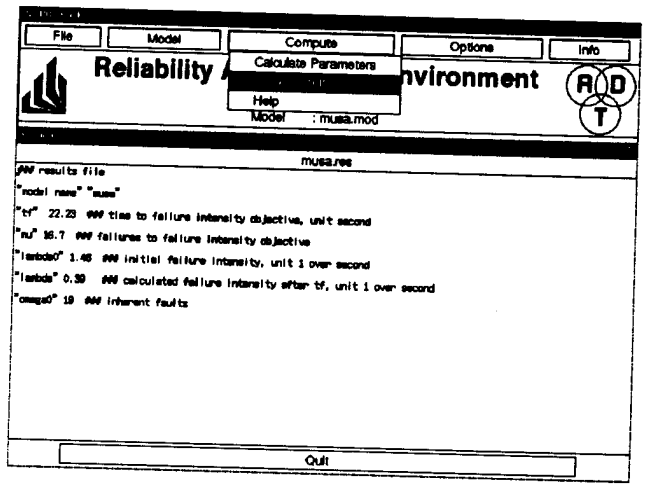

Figure 2: Viewing results after parameter calculation

8 Conclusion

In this paper an approach to testing and reliability prediction and estimation for logic programming in Prolog, materialized and integrated into a product assurance environment, has been proposed. The main testing concepts are type and mode declarations for Prolog programs. Types are represented in terms of parametric type rules which are equivalent to regular types. This provides 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 proposed reliability prediction approach accounts for certain specific characteristics of logic programming, and specifically, of Prolog programs. Such a feature oriented refinement should lead, eventually, to improve the quality of predictions and furthermore to simplify predictions with respect to software reliability characteristics. Several exemplary Prolog programs analyzed so far seem to support the above statement.

The ongoing and further research concentrate on extending the product assurance environment and its underlying concepts of testing and reliability determination in order to support an object-oriented extension of Prolog which will be developed on the basis of *pure* Prolog with negation.

Acknowledgement

This paper was supported by Heinz-Nixdorf-Institut, Paderborn, Project number ZIT 521 and ZIT 538. Our thanks are due to Armin Bühler (University of Bielefeld), Lee Naish (University of Melbourne, Australia), Dittmar Wilden and Michael Winter (University of Paderborn) and to the anonymous reviewers for valuable hints and discussions.

References

[1] A. Azem, F. Belli, and O. Jack. Implementation-Oriented Testing and Reliability Determination of Prolog Programs. In *Proceedings of the fifth International Conference on Software Engineering and Knowledge Engineering, San Francisco Bay*, pages 526–533, June 1993.

[2] F. Belli and O. Jack. PROTest II: Testing Logic Programs. In *Proc. 2nd Int. Conf. Software Quality, Research Triangle Park, N.C.*, pages 128–135, 1992.

[3] B.W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1):75–88, January 1984.

[4] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 3rd edition, 1987.

[5] Saumya K. Debray and Nai-Wei Lin. Automatic complexity analysis of logic programs. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 599–613, Paris, France, 1991. The MIT Press.

[6] Richard Denney. Test-case generation from Prolog-based specifications. *IEEE Software*, 8(2):49–57, Mar. 1991.

[7] A.L. Goel and K. Okumoto. Time-Dependent Error-Detection Rate Model for Software Reliability and other Performance Measures. *IEEE Trans. Reliability*, R-28(3):206–211, Aug. 1979.

[8] 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.

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

[10] The Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard Glossary of Software Engineering Terminology*, 1991. IEEE Std 610.12.1990.

[11] Z. Jelinski and P. Moranda. Software Reliability Research. In W. Freiberger, editor, *Statistical Computer Performance Evaluation*, pages 465–484. Academic Press New York, 1972.

[12] S. Kaplan. Algorithmic complexity of logic programs. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 780–793, Seatle, 1988. ALP, IEEE, The MIT Press.

[13] D.E. Knuth. An Imperial Study of FORTRAN Programs. *Software Practice and Experience*, 2:105–134, April-June 1971.

[14] B. Littlewood and J.L. Verrall. A Bayesian Reliability Model with a Stochastically monotone Failure Rate. *IEEE Trans. Reliability*, R-23(2):108–114, June 1974.

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

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

[17] J.D. Musa. A Theory of Software Reliability and its Applications. *IEEE Trans. Software Eng.*, SE-1(3):312–327, Sept. 1975.

[18] J.D. Musa, A. Ianino, and K. Okumoto. *Software Reliability - Measurement, Prediction, Applications.* McGraw-Hill, 1987.

[19] PDC Prolog 3.20, User's Guide. Prolog Development Center, Copenhagen, 1990.

[20] G.J. Schick and R.W. Wolverton. An Analysis of Competing Software Reliability Models. *IEEE Transactions on Software Engineering*, SE-4(2):104–120, Mar. 1978.

[21] Ehud Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, Massachsetts/London, England, 1982.

[22] M.L. Shooman. *Software Engineering, Design, Reliability and Management.* McGraw-Hill, New York, 1983.

[23] Allen Van Gelder. Efficient loop detection in Prolog using the tortoise-and-hare technique. *Journal of Logic Programming*, pages 23–31, 1987.

[24] Dittmar Wilden. Strukturiertes Testen von Prolog-Programmnen — Testfallauswahl durch Partitionierung. Master's thesis, Universität-GH Paderborn, 1992.

[25] M. Winter. Zuverlässigkeit logischer Programme - Eine probabilistische Untersuchung. Technical report, University of Paderborn, Dept. of Electrical and Electronics Engineering, Software and Knowledge Engineering, July 1992.