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., Jedrzejowicz, P., Reliability Prediction & Estimation of Prolog Programs, IEEE Transactions on Reliability, Vol. 43(4), 542-549

Reliability Prediction & Estimation of Prolog Programs

Alireza Azem
University of Paderborn, Paderborn
Fevzi Belli
University of Paderborn, Paderborn
Piotr Jedrzejowicz
Merchant Marine Academy, Gdynia

Key Words — **Software reliability, software complexity measures, logic programming, Prolog**

Reader Aids —
General Purpose: Present a prediction/estimation approach
Special math needed for explanations: Probability theory
Special math needed to use results: Same
Results useful to: Software & reliability engineers

Summary & Conclusions — **This paper presents an approach to reliability prediction & estimation of Prolog programs, and introduces 2 complexity measures for Prolog programs. The structural complexity measure refers to the program's static characteristics: size & number of clauses, clause arguments (number & types), and clause types (facts or rules). The operational complexity measure refers to the program's dynamic characteristics: execution frequency of program components, user behavior, and backtracking & recursion. Values of the two measures are used to: 1) predict Prolog- program reliability before testing and in the early testing stages, and 2) estimate the reliability as a function of time, in order to determine whether the reliability objective is achieved. The feature-oriented reliability determination approach leads to improvements in the accuracy of software-reliability predictions & estimations.**

1. INTRODUCTION

Logic programming provides for formulating & solving problems in a declarative manner. Hence, problems are solved by describing what has to be done instead of how it has to be done. Declarative programming is a powerful method for constructing software for knowledge-based systems, database applications, *etc*. It also allows modeling the programming process and the program development process in the same way [19].

Software reliability is the probability of failure-free operation of a computer program for a specified time in a specified environment [18]. Equivalent ways of expressing reliability are the failure intensity or failure rate. Software reliability models [3, 6, 9, 13, 14, 16, 17, 20-22] specify the dependence of the failure process on fault introduction, fault removal, and the environment. Reliability of a computer program can be determined by identifying the values of the model parameters through either prediction or estimation. Prediction occurs before adequate failure data are available. Estimation occurs after such failure data are available and can determine 1) the reliability change during time, and/or 2) whether the reliability objective is achieved.

This paper proposes an approach to prediction & estimation of Prolog program reliability. Section 2 briefly introduces the constructs of Prolog programs. Section 3 describes the fault model. To account for special features of logic programming in general, and Prolog programs in particular, two complexity measures are introduced: *structural* complexity measures the program's static characteristics, *operational* complexity measures its dynamic characteristics. Sections 4 & 5 introduce these measures. Section 6 highlights the correlations between Prolog programs' complexity & reliability. Section 7 shows the relations between reliability parameters and both complexity measures, in order to predict & estimate reliability. Section 8 introduces some field results.

Standard notation is given in "Information for Readers & Authors" at the rear of each issue.

2. CONSTRUCTS OF A PROLOG PROGRAM

A subset of formulae of the first-order predicate logic, the Horn clauses, has a procedural meaning. This enables a programmer to express problems and, more importantly, problem solving in a declarative manner leaving the task of actually generating the solution to the Prolog programming system. Prolog, however, is not a pure first-order predicate logic, as it is not restricted to Horn clauses but includes built-in predicates (cut, assert, *etc*) which are outside the scope of the first-order predicate logic [2, 5, 15]. A Prolog program is a set of *clauses* which describe the functionality of the program in a declarative manner. Two forms of clauses are distinguished: facts & rules (*goals* are considered as rules with an empty head). Clauses (facts & rules) provided through the programmer can further be distinguished from built-in predicates embedded within the underlying system. The following general notation of a rule is used (a fact is a special form of a rule with an empty right part):

$$C_i(A_{i1}, \ldots, A_{in_i}) :- C_{i1}(A_{i1,1}, \ldots, A_{i1,ni_1}), \ldots, C_{im}(A_{im,1}, \ldots, A_{im,n_{im}}). \quad (1)$$

Notation

C_x functors (names)
A_x arguments of the Prolog clauses.

Assumption

1. Prolog programs can be divided into segments. A segment corresponds to a set of clauses providing an overall functionality. ◄

The simplest realization of a segment is a predicate: the set of all clauses having the same functor with the same number of arguments, *viz*, the same *arity*. Some Prolog systems (*eg*, SB-Prolog, Quintus Prolog) allow a modular program structure; modules are program entities 1) with clearly defined import & export interfaces, and 2) providing a defined functionality. The segments, then, are the modules.

3. FAULT MODEL

A fault is a 'textual problem with the program' resulting from a mental mistake by a programmer; the mental mistake is defined as an error [7, 8]. This paper considers only those faults which are not detectable by an ordinary compiler. A failure is a departure of operation from requirements [7, 18]. The fault model used here 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 following fault types are considered:

- wrong typing,
- wrong subtyping,
- wrong parameter passing.

Other features of logic programming are related to software failures. For example, the SLD-resolution [15] using a depth-first search combined with a fixed order for trying clauses given by their ordering in the program, is incomplete. This means that a logic programming system with a depth-first search using a fixed order for trying program clauses does not always guarantee to find the success branch [15]. This addresses *intrinsic* software faults due to the fundamental techniques used in logic programming, but does not address residual software faults due to residual design/coding faults in the program [4]. This paper considers only residual faults.

4. STRUCTURAL COMPLEXITY MEASURE

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

$$W_{ki} = \begin{cases} \sum_{j=1}^{n_{ki}} G_{kij}, & \text{clause } i \text{ is a rule} \\ X_{ki}, & \text{clause } i \text{ is a fact.} \end{cases} \tag{2}$$

Notation

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

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

$$G_{kij} = M_{kij} \cdot A_{kij} / n_{ki}, \tag{3}$$

$$A_{kij} = \begin{cases} 1, & \text{predicate } j \text{ is a fact or is a recursive call of clause } i, \text{ or } S_{kij}=0 \\ 2^{S_{kij}} - 1, & S_{kij} > 0 \end{cases} \tag{4}$$

Notation

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 by:

$$X_{ki} = X'_{ki} \cdot X''_{ki}, \tag{5}$$

$$X'_{ki} = \begin{cases} 1, & S'_{ki}=0 \\ 2^{S'_{ki}} - 1, & \text{otherwise} \end{cases} \tag{6}$$

$$X''_{ki} = \begin{cases} 1, & S''_{ki}=0 \\ \prod_{l=1}^{S''_{ki}} (2^{S''_{kil}} - 1), & \text{otherwise} \end{cases} \tag{7}$$

Notation

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 complexity of segment k is then:

$$W_k = \sum_{i=1}^{n_k} W_{ki}, \tag{8}$$

Notation

n_k	number of clauses in segment k.

The program structural complexity is:

$$\mathrm{W} = \sum_{\mathrm{k}=1}^{\mathrm{n}} \mathrm{W}_{\mathrm{k}}. \tag{9}$$

5. OPERATIONAL COMPLEXITY MEASURE

The operational complexity measure can be calculated for each Prolog program segment and considers the following features of its operational profile:

- the backtracking degree (number of choice points for deriving a given goal)
- the extent of recursive & non-recursive executions
- the access frequency of the analyzed program segments.

The operational complexity of a segment k is:

$$W'_k = \frac{B_k}{1 + \beta_k} \cdot (H_k + R_k) \tag{10}$$

Notation

B_k utilization factor reflecting access frequency of segment k during program operation

β_k maximal backtracking degree of segment k during testing

H_k total number of accesses of segment k during testing (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 direct recursive calls are counted).

The program operational complexity is:

$$W' = \sum_{k=1}^{n} W'_k. \tag{11}$$

The B_k can be identified during program operation by applying the procedure:

- track the program operation by counting the segment accesses for a given period of time (comparable to determining $H_k + R_k$ for a segment k during testing); denote the accesses (including also the recursive calls) of each segment k by D_k
- choose a reference D' by determining the minimum of the segment access counts: $D' \equiv \min_k(D_k)$
- set $B_k = D_k/D'$. (12)

B_k represents the utilization of segment k (with D' as a reference) due to user behavior during program operation.

6. COMPLEXITY MEASURES *vs* PROLOG-PROGRAM RELIABILITY

The two complexity measures in sections 4 & 5 are based on the underlying assumption that software failures are more likely to occur within program segments (or programs) that are more complex. With this assumption in mind it is obvious that complexity measures can be best validated by determining how much they correlate with program reliability. Intuitively, more complex software is generally less reliable. Our experience, as well as the logical analyses of the proposed measures, support this assumption.

How software complexity influences its reliability is, however, a complex question. It seems that structural complexity can be related to the number of *inherent* faults (faults existing before testing begins) in a program. Similarly, operational complexity can be related to the relative change of failure intensity per failure experienced. In our experience, structural complexity, for Prolog programs, is a better indicator of the number of inherent faults than are conventional measures like 'faults per line of code' or 'failure rate per class of statements', which are used in some other failure modeling methods [12]. The reason for this is that the structural complexity measure is based on directly identifying potential sources of failure within Prolog-program segments & clauses. Looking more closely at the definition of predicate structural-complexity, $1/n_{ki}$ in (3), represents the relative contribution of each predicate within the body of clause i. The M_{kij} relates to the fact that the functionality of predicate j in the body of clause i is disallocated elsewhere (this feature is comparable to procedural abstraction in conventional programming languages). Furthermore, the A_{kij} in (4) reflects the features that: 1) Prolog does not distinguish between data types, and 2) 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 anticipated. With 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
- $2^{S_{kij}} - 1$ cases having at least one argument not of the right type.

The operational profile of a program is the: 1) set of run types that the program can execute, and 2) probabilities with which they will occur [18]. A run type is identified by its input state. An input state is characterized by a set of values of the input variables. An input variable is a variable that exists external to the program and is used by the program in executing its function. For an airline reservation system, *destination* might be an input variable. The operational profile which considers program 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 occur during execution are associated with the related input states. During testing, each failure affects the failure intensity (in general a failure 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 components (parts, paths, segments, *etc*) would be accessed and to what extent. For Prolog programs the three features of the operational profile can be useful when trying to model its affect on failure behavior:

- the backtracking degree,
- the extent of recursive & non-recursive executions,
- the access frequency of program segments during program operation.

Backtracking is affected by the operational profile, as it depends on the run types. Some run types do, and some do not, enforce backtracking. 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 the higher the backtracking degree, the lower is the failure contribution. Assuming a linear reciprocal relation, the effect of backtracking for each segment can be quantified as:

$$[1 + \text{backtracking degree}]^{-1}.$$

If there is no backtracking (backtracking degree is zero) then the result is 1 (the backtracking has no effect). Example 1 illustrates backtracking degree.

Example 1

A segment is defined by:

$a(X):-b,d(X),e.$

$b.$

$e.$

$d(Y):-f,h(Y).$

$d(Y):-j(Y).$

$f.$

$j(3).$

$h(1).$

$h(2).$

a, b, d, e, f, h, j represent functors of clauses.

Having the goal ?-*a*(2)., the system tries to compute an answer to this goal in the following manner: the variable *X* is unified with 2 and the goals *b*, *d*(2), *e* must be derived, in order that goal ?-a(2). is successful. The *b* can be unified with the fact b., and for deriving goal *d*(2) it is necessary to derive either both goals *f* & *h*(2), or goal *j*(2). Goal *f* can be unified with fact *f*., and for deriving goal *h*(2) the system tries first to match the goal with the first appearance of a clause with the same functor and same *arity*, which is h(1). As the goal cannot be unified with this fact, the system backtracks to the next choice point, *ie*, it tries to find an alternative way to derive goal *h*(2), which can be matched with fact *h*(2). Goal e can also be matched with fact *e*.; so goal ?-*a*(2). is successful. The backtracking degree is the number of choice points for deriving goal ?-*a*(2)., and as there exists 1 choice point in this case, the backtracking degree is 1. Consider now the goal ?-*a*(3). The system backtracks once for deriving *h*(3), which is not successful, and once for deriving *d*(3), which is successful through deriving *j*(3). The backtracking degree is then 2 in this case. ◄

As mentioned in section 5, the access frequency of program segments during program operation is considered through the operational profile. This feature accounts for various user behaviors and various use 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 needs the functionality of that faulty segment frequently, so this user observes a poor program reliability.

7. DETERMINING PROGRAM RELIABILITY

" This section discusses the relations between reliability parameters and both segment complexity measures (structural & operational), in order to predict & estimate the reliability of Prolog programs.

7.1 Reliability Prediction

The basic execution-time model [18] is considered. The execution time component for this model assumes that failures occur as a nonhomogeneous Poisson process. The initial program failure intensity is [18]:

$$\lambda_0 = f \cdot K \cdot \omega_0, \tag{13}$$

Notation

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 (the fraction of time that the program-run results in a failure)
ω_0	number of *inherent* faults.

Our prediction approach is a modification & refinement of the methodology in [18], by concentrating on features of a particular class of software systems (Prolog programs). We believe that the knowledge of structural & operational characteristics of Prolog programs, embedded in the structural & operational complexity measures, helps to predict better than the traditional & general approach in [18]. The total number of failures ν_0 can be predicted [18]:

$$\nu_0 = \omega_0/B, \tag{14}$$

Notation

ν_0	total number of failures
ω_i	number of faults remaining after *i* faults are removed
ω_0	number of *inherent* faults
γ_0	*inherent* faults per unit of overall structural complexity
B	fault reduction factor: [net fault reduction]/[failures experienced as time of operation $\rightarrow \infty$].

There is evidence that B typically results from new fault spawning alone [1]. We believe that a project-independent value of B can be found for a class of applications. The traditional approach assumes that the number of *inherent* faults is linearly related to program size. For Prolog programs, we believe that better predictions can be obtained by using correlation between the number of faults and the program structural complexity W. Hence, the number of *inherent* faults for a Prolog program is estimated from:

$$\omega_0 = \gamma_0 \cdot W. \tag{15}$$

We assume that γ_0 is fairly constant for nearly similar Prolog software projects and can be determined beforehand from other projects. The state of the art assumes that the value of the fault exposure ratio must be determined from similar programs. We believe that for Prolog programs a better approach can be used. The fault exposure ratio represents the fraction of time that the program run results in a failure. It accounts for the facts: 1) programs are not generally executed in straight line, but have many loops and branches, and 2) each instruction can be executed in many different machine states, with a failure usually occurring for only a few of them [17, 18]. Recall that the operational complexity represents the access frequency. As the fault exposure ratio is related to the access (execution) frequency it is correlated with the operational complexity. Thus,

$$K = \sum_{k=1}^{n} K_k \cdot (W_k'/W'), \tag{16}$$

$$K_k \equiv \xi \cdot W_k', \tag{17}$$

$$K = (\xi/W') \cdot \sum_{k=1}^{n} W_k'^2, \tag{18}$$

Notation

K_k fault exposure ratio of segment k
ξ constant parameter determined from similar projects.

The W_k' are not known before testing begins. We suggest using either 1) a single uniform value averaged from similar projects, or 2) predicted values of W_k' based on correlation between W_k (or W) and W_k' for similar projects.

With the number of *inherent* faults and initial failure intensity available, one can predict reliability characteristics such as, additional failures to failure intensity objective and additional execution time to failure intensity objective.

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 are the period when test data and the available information are not sufficient to estimate accurately the reliability characteristics of the software product, and yet some information and possibilities for improving pre-testing predictions are already at hand.

Let i faults have been removed (i being a sufficiently small integer). The program failure intensity is:

$$\lambda_i = f \cdot K \cdot \omega_i, \tag{19}$$

$$\omega_i = \omega_0 - i. \tag{20}$$

K is then calculated as before except that instead of the predicted values of W_k', use the values obtained from averaging over the existing test runs.

Reliability prediction is required before testing or before failure data are available. This means that the parameters needed to determine reliability must be determined for system engineering studies due to, *eg*, anticipated program size or data from similar projects. Other well-known reliability prediction models obtain such parameters in a similar way, *eg*, for prediction due to [17, 18] the parameters f, K, B, ω_0, *etc* must be determined from similar programs or from experience.

7.2 Reliability Estimation

Assumptions

1. Whenever a failure occurs, the fault causing the failure is removed instantaneously and perfectly (without introducing any other problems).
2. The total number of *inherent* faults in the program is a Poisson r.v.
3. Failures occur *s*-independently and randomly according to the per fault hazard rate which is the same for all faults.◄

According to the software reliability model classification [18], these assumptions hold true for Poisson models.

Notation (Non-processes)

t time
$f(t)$, $F(t)$, $R(t)$ pdf, Cdf, Sf of failure time
$z(t)$ hazard rate.

Notation (Processes)

i failure number
t_i (execution) time to failure i from $t=0$
t_i' (execution) time between failures $i-1$ and i
$z(t_i' \mid t_{i-1})$ hazard rate in the time interval t_i'
ω_0 mean number of *inherent* faults
$f_a(t)$, $F_a(t)$ pdf, Cdf for an individual fault
$z_a(t)$ hazard rate for an individual fault
ϕ constant $z_a(t)$
$\mu(t)$ mean number of failures in $[0,t]$
$\lambda(t)$ $d\mu(t)/dt$: failure intensity at t
$U_k(t_i)$, $U_k'(t_i)$ value of the relative segment [structural, operational] complexity estimated at t_i.

A software reliability model describes software failures as a random process, which is characterized in either failure times or the number of failures at fixed times. For a Poisson model the following relationships apply [18]:

$$z(t_i' \mid t_{i-1}) = \omega_0 \cdot f_a(t_{i-1} + t_i'), \tag{21}$$

$$\mu(t) = \omega_0 \cdot F_a(t), \tag{22}$$

$$\lambda(t) = \omega_0 \cdot f_a(t). \tag{23}$$

For an exponential distribution of failure time of an individual fault,

$$f_a(t) = \phi \cdot \exp(-\phi \cdot t). \tag{24}$$

A model considering this feature was proposed in [6].

We suggest that, for Prolog programs, the constant ϕ can be related to the program characteristics and be best estimated using previously defined complexity measures. Consider the relative segment structural and operational complexity measures:

$$U_k = W_k \Big/ \sum_{l=1}^{n} W_l = W_k/W, \tag{25}$$

$$\mathbf{U}_k' = W_k' \Big/ \sum_{l=1}^{n} W_l' = W_k'/W'. \tag{26}$$

The U_k is known before testing has commenced and can be taken as constant through program operation — except when major changes to the program are made due to the removal of faults (in this case U_k must be recomputed). The U_k' can be estimated during operation — by averaging numbers of previous accesses, backtracking, *etc* for a given user or a particular type of applications. U_k' is, in general, a function of time and of user/application type. We propose a measure of program-failure-potential:

$$V(t_i) = \sum_{k=1}^{n} U_k(t_i) \cdot U_k'(t_i) \cdot Y_k(t_i), \tag{27}$$

Notation

$V(t_i)$	program failure potential
$Y_k(t_i)$	$N_{fk}(t_i)/N_f(t_i)$
$N_{fk}(t_i)$	total number of test runs, up to t_i, which led to a failure within segment k
$N_f(t_i)$	total number of test runs, up to t_i, which led to a failure.

The $Y_k(t_i)$ relates U_k to program operation. Let segment k, having a high U_k, cause no failures during the program operation (or testing). Thus if one considers only U_k by itself, one would wrongly consider a bad state for segment k although no faults due to this segment were detected during operation (or testing). On the other hand, a segment with a low 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 & correction process (during testing).

For a representative set of Prolog programs, as faults are detected & corrected during testing, $\Psi(t_i)$ approaches a value that is changing in a small interval.

$$\Psi(t_i) = N(t_i) \cdot V(t_i)/N_f(t_i), \tag{28}$$

Notation

$N(t_i)$	total number of test runs up to t_i
t_{tot}	total execution time up to the last detected and corrected fault
ρ	constant factor for a class of Prolog programs.

$$\phi = \rho \cdot N(t_{tot}) \cdot V(t_{tot})/N_f(t_{tot}). \tag{29}$$

The determination of ϕ after (29) is based on the following considerations. ϕ is the contribution of each fault to the program hazard rate. For Prolog programs ϕ can be related to program characteristics using the proposed complexity measures. The $V(t_i)$ represents the fault-proneness of the program (with a fault causing a failure) at t_i (after having detected & corrected i faults). As $V(t_i)$ considers relative structural & operational complexities due to (27) for the test period t_i, with $N(t_i)$ test runs, the contribution of all test runs to failures is $N(t_i) \cdot V(t_i)$ and the per fault contribution is that term divided by the number of failures (with a fault causing a failure) up to t_i. The number of failures up to t_i is exactly the number of test runs up to t_i which led to a failure, $N_f(t_i)$. As faults are detected & corrected during testing, the $\Psi(t_i)$ in (28) approaches a value that is changing in a small interval and is nearly constant. Ref [11, 24] regard (29) as a useful & meaningful to determine ϕ for Prolog programs. Thus,

$$F_a(t) = 1 - \exp(-\phi \cdot t), \tag{30a}$$

$$\mu(t) = \omega_0 \cdot (1 - \exp(-\phi \cdot t)), \tag{30b}$$

$$\lambda(t) = \omega_0 \cdot \phi \cdot \exp(-\phi \cdot t). \tag{30c}$$

The maximum likelihood estimator for ω_0 is the solution to (31):

$$N_f(t_{tot}) = \omega_0 \cdot \left[[1 - \exp(-\phi \cdot \sum_{i=1}^{N_f(t_{tot})} t_i')\right]. \tag{31}$$

Using (30c) the additional time to failure intensity objective can be determined from:

$$\lambda(t_f)/\lambda(t_p) = \exp(\phi(t_p - t_f)), \tag{32}$$

$$t_\Delta = t_f - t_p = (1/\phi) \cdot \ln[\lambda(t_p)/\lambda(t_f)], \tag{33}$$

Notation

t_f	execution time to failure intensity objective
t_p	execution time up to present failure intensity
$\lambda(t_f)$	failure intensity objective
$\lambda(t_p)$	present failure intensity
t_Δ	additional execution time to $\lambda(t_f)$.

For estimation, only one parameter ρ must be determined from similar projects. This is acceptable due to the good results which were achieved through the estimation approach discussed above.

8. SOME RESULTS

We applied our approach to several Prolog programs. Some results are given in this section.

One of the programs investigated in detail in [24] was a medium-size Prolog program of about 110 KB of executable code; it was a small database program providing functions like Insert, Update, Delete for handling data records.

For prediction (see section 7.1), the following data were used.

$n = 5$ (number of segments),

$W = 230294$,

$\gamma_0 = 10^{-4}$,

$\bar{W}_k = 2.5$, for all k (from a similar program),

$\xi = 1.5$ ($\xi = 1, \ldots, 1.7$ from similar programs).

The results were:

$K = 0.38$,

$\lambda_0 = 1.73\ s^{-1}$.

For a conventional program analyzed in [10], the results were:

$f = 0.02\ s_{-1}$ (after two different methods, $K = 0.03$ and

$K = 0.18$ was determined).

Using clauses instead of assembly instructions for determining f, for the Prolog program an average of,

$f = 0.2\ s^{-1}$

was determined. The results are:

f	γ_0	ω_0	$\bar{W}_k$	ξ	K	λ_0
$0.2\ s^{-1}$	10^{-4}	23	2.5	1.5	0.38	$1.73\ s^{-1}$

Using (19) & (20) for prediction, and considering data from the early stages of testing (total number of test runs = 45) following results were achieved:

N	i	ω_i	λ_i	K
10	5	18	$1.58\ s^{-1}$	0.44
20	8	15	$1.02\ s^{-1}$	0.34

A comparison of λ after detecting & correcting 15 faults yielded:

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

Notation

A	estimation after the execution time model [17, 18]
B	prediction after (13) - (18)
C	prediction after (19) & (20), with N=20.

All predictions are underestimated compared with real data. Real failure intensity corresponds to the actual total execution time divided by the actual total number of failures. This is an instantaneous interpretation of failure intensity at each failure detection point, which is used here to compare the accuracy of predictions & estimations with the actual program state. Using our feature oriented approach yields better predictions: cases *B* & *C*.

For estimation (see section 7.2), with $\phi = 0.055\ s^{-1}$ (determined after data in [24], setting $\rho = 1\ s^{-1}$), the failure intensity of the program was $\lambda = 0.37\ s^{-1}$. The following table summarizes the results of the estimation of failure intensity after the reliability models of Jelinski & Moranda [9], Musa [17, 18], and Goel & Okumoto [6], as well as the real data, and the failure intensity using our estimation approach.

Real Life	A	B	C	D
$0.70\ s^{-1}$	$0.26\ s^{-1}$	$0.31\ s^{-1}$	$0.36\ s^{-1}$	$0.37\ s^{-1}$

Notation

A	estimation using the execution time model [17, 18]
B	estimation using [9]
C	estimation using [6]
D	estimation using our approach.

Our feature-oriented approach is better than the others.

ACKNOWLEDGMENT

This paper was partly supported by the University of Paderborn and the Heinz-Nixdorf-Institut Paderborn, Project numbers ZIT 521 & ZIT 538. We are pleased to thank Michael Winter (University of Paderborn) and the anonymous referees for their valuable remarks & discussions.

REFERENCES

[1] V.R. Basili, B.T. Perricone, "Software errors and complexity: An empirical investigation, *Communications ACM*, vol 27, num 1, 1984, pp 42-52.

[2] F. Belli, O. Jack, A. Schmidt, "Object-oriented extension of Prolog: Theoretical background and a case study", *Technical Report*, 1991 Nov; University of Paderborn, Dept. of Electrical & Electronics Eng'g.

[3] B.W. Boehm, J.R. Brown, M. Lipow, "Quantitative evaluation of software quality", *Proc. 2nd Int'l Conf. Software Eng'g*, 1976 Oct, pp 592-605.

[4] I. Chen, F.B. Bastani, "Effect of artificial-intelligence planning-procedures on system reliability", *IEEE Trans. Reliability*, vol 40, 1991 Aug, pp 364-369.

[5] W.F. Clocksin, C.S. Mellish, *Programming in Prolog* (3rd ed), 1987; Springer-Verlag.

[6] A.L. Goel, K. Okumoto, "Time-dependent error-detection rate model for software reliability and other performance measures", *IEEE Trans. Reliability*, vol R-28, 1979 Aug, pp 206-211.

[7] IEEE Std 982.1-1988, *IEEE Standard Dictionary of Measures to Produce Reliable Software*, 1989 Aug.

[8] IEEE Std 610.12.1990, *IEEE Standard Glossary of Software Engineering Terminology*, 1991.

[9] Z. Jelinski, P. Moranda, "Software reliability research", *Statistical Computer Performance Evaluation* (W. Freiberger, Ed), 1972 Aug, pp 465-484; Academic Press.

[10] I. Kamundi, "Measurement of software reliability by micro and macro models - A parameter determination and computer aided reliability prediction", *Technical Report*, 1991 Apr; University of Paderborn, Dept. of Electrical & Electronics Eng'g.

[11] I. Kamundi, "An approach to validation of assumptions made by metric-based reliability modelling for logic programs", *Technical Report*, 1993 Apr; University of Paderborn, Dept. of Electrical & Electronics Eng'g.

[12] D.E. Knuth, "An empirical study of FORTRAN programs", *Software Practice and Experience*, vol 1, 1971 Apr-Jun, pp 105-133.

[13] B. Littlewood, J.L. Verrall, A Bayesian reliability model with a stochastically monotone failure rate", *IEEE Trans. Reliability*, vol R-23, 1974 Jun, pp 108-114

[14] B. Littlewood, J.L. Verrall, "Likelihood function of a debugging model for computer software reliability", *IEEE Trans. Reliability*, vol R-30, 1981 Jun, pp 145-148.

[15] J.W. Lloyd, *Foundations of Logic Programming* (2nd ed), 1987; Springer-Verlag.

[16] P.B. Moranda, "Prediction of software reliability during debugging", *Proc. Ann. Reliability & Maintainability Symp*, 1975, pp 327-332.

[17] J.D. Musa, "A theory of software reliability and its applications", *IEEE Trans. Software Eng'g*, vol SE-1, 1975 Sep, pp 312-327.

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

[19] B. Peuschel, W. Schäfer, S. Wolf. "A knowledge-based software development environment supporting cooperative work", *Int'l J. Software Engineering and Knowledge Engineering*, vol 2, 1992 Mar, pp 79-106.

[20] G.J. Schick, R.W. Wolverton, "An analysis of competing software reliability models", *IEEE Trans. Software Engineering*, vol SE-4, 1978 Mar, pp 104-120.

[21] M.L. Shooman, "Probabilistic models for software reliability prediction", *Statistical Computer Performance Evaluation* (W. Freiberger, Ed), 1972 Aug, pages 485-502; Academic Press.

[22] M.L. Shooman, "Structural models for software reliability prediction", *Proc. 2nd Int'l Conf. Software Eng'g*, 1976, pp 268-280; San Francisco.

[23] M.L. Shooman, *Software Engineering, Design, Reliability and Management*, 1983; McGraw-Hill.

[24] M. Winter, "Zuverlässigkeit logischer programme - Eine probabilistische untersuchung [Reliability of logic diagrams - A probabilistic approach]" (*Technical Report*, 1992 Jul; University of Paderborn, Dept. of Electrical and Electronics Eng'g.

AUTHORS

Dipl.-Ing. Alireza Azem; Dept. of Electrical & Electronics Eng'g; University of Paderborn; 33095 Paderborn Fed. Rep. GERMANY.

Alireza Azem (born 1963) is a PhD student at the University of Paderborn. He received the MS from the Ruhr University of Bochum in Electrical & Electronics Engineering. His research interests are software reliability, logic programming, and object-oriented programming. He is a member of the German Association for Informatics (GI) and the working group Software Reliability of the German Engineers Association (VDI).

Prof. Dr.-Ing. habil. Fevzi Belli; Dept. of Electrical & Electronics Eng'g; University of Paderborn; 33095 Paderborn; Fed. Rep. GERMANY.

Fevzi Belli (born 1948) is a Professor of Computer Science at the University of Paderborn and is a faculty member of the University of Maryland, European Division. His research interests are software fault tolerance, software reliability, programming techniques, and knowledge-based systems. His degrees (MS, PhD, Habilitation) are from the Technical University of Berlin in Electrical & Electronics Engineering and Computer Science. He is a member of IEEE Computer Society, ACM, German Association for Informatics (GI) and its working group Software Reliability and Fault Tolerant Computing, and the German Engineers Association (VDI).

Prof. Dr. habil. Piotr Jedrzejowicz; Dept. of Systems Analysis & Informatics; Merchant Marine Academy; ul. Morska; 83 Gdynia, POLAND.

Piotr Jedrzejowicz (born 1942) is a Professor of Computer Science and the former Head of the Merchant Marine Academy in Gdynia. He served as Head of the Dept. of Systems Analysis & Informatics of the Academy. His degrees (MA, PhD, Habilitation) are from the University of Gdansk in Operations Research and Maritime Economics. His research interests are reliability optimization, modeling, and simulation of complex systems, and decision-support systems.

Manuscript received 1993 September 22.

IEEE Log Number 92-14871

◄TR►