Determining the Reliability of Prolog Programs

ALIREZA AZEM AND FEVZI BELL

Department of Electrical and Electronics Engineering, University of Paderborn, 33095 Paderborn, Germany

SUMMARY

In this paper an approach to reliability prediction and estimation of Prolog programs is proposed. Two complexity measures describing Prolog programs are introduced. Values of the two measures are used, subsequently, to predict the reliability of Prolog programs before testing and in the early testing stages, and further, to estimate the reliability as a function of time, in order to determine whether the reliability objective is achieved. The proposed reliability determination approach is based on previous work (Azem et al., 1993), extending the prediction approach used therein through modification of the complexity measures and providing an estimation approach. It leads to improvements in the quality of predictions and estimations with respect to software reliability characteristics. The proposed approach is implemented in a reliability assessment environment, which also includes several well-known software reliability models for comparison purposes.

KEY WORDS software reliability; software complexity measures; logic programming; Prolog; reliability assessment environment

1. INTRODUCTION

Logic programming provides a way of formulating and solving problems in a declarative manner. Hence, problem solving is performed by describing what has to be done, instead of how it has to be done. The declarative way of programming offers a powerful method for the construction of software, e.g. for knowledge-based systems, database applications, etc. It also allows the programming process and the program development process to be modelled in the same way (Peuscher et al., 1992).

Software reliability is defined as the probability of failure-free operation of a computer program for a specified time in a specified environment (Musa et al., 1987). Alternative ways of expressing reliability are possible in terms of the failure intensity or the hazard rate. Software reliability models (for example Boehm et al., 1976; Goel and Okumoto, 1979; Jelinski and Moranda, 1972; Littlewood and Verrall, 1974; Moranda, 1975; Musa, 1975; Schick and Wolverton, 1978; Shooman, 1972; Shooman, 1976; Yamada et al., 1984; Adams, 1991) specify the dependence of the failure process on fault introduction, fault removal and the environment. Determining the reliability of a computer program can be accomplished by identifying the values of the parameters of the respective models through either prediction or estimation. Prediction is needed whenever system engineering studies are required before failure data are available or are not adequate to enable a meaningful
estimation. Estimation allows the reliability change over time to be determined and/or a
decision to be made whether or when the reliability objective is achieved.

Complexity measures are used to determine reliability in this paper. Software measures
in general aim to measure a wide range of attributes, including complexity, size, etc.
Measures are used either numerically to characterize some attribute of an existing entity
or to predict/estimate some attribute of a future entity involving a mathematical model.
A measure is valid if it accurately characterizes the proposed attribute; a prediction/
estimation procedure is valid if it makes accurate predictions/estimations. It is often an
implicit assumption that a measure should be part of a prediction/estimation procedure
to be useful. Validating a prediction/estimation procedure in a given environment is the
process of establishing the accuracy of the procedure by empirical means, by comparing
model performance with known data points in the given environment (Fenton, 1991).
Complexity is commonly used to capture the totality of all attributes in a term that is an
indicator of those attributes, which are normally associated with complex systems such
as poor reliability and maintainability. It is generally accepted that certain desirable
attributes (e.g. modularization, low coupling, etc.) will lead to a high degree of the
desirable external attributes, such as reliability and maintainability. Thus programs and
modules that exhibit poor values for the desirable internal attributes are likely to have
more faults and take longer to produce and maintain. The complexity measures introduced
in this paper address this issue by considering attributes of Prolog programs related to
program structure as well as program operation.

This paper proposes an approach to the prediction and estimation of the reliability of
Prolog programs. The proposed feature-oriented reliability determination approach is
based on previous work (Azem et al., 1993), extending the prediction approach used
therein through modification of the structural and operational complexity and providing
an estimation approach. It leads to improvements in the quality of predictions and
estimations with respect to software reliability characteristics. Section 2 briefly introduces
the constructs of Prolog programs. To account for special features of logic programming
in general, and Prolog programs in particular, two different complexity measures are
introduced. The structural complexity measure reflects the program's static characteristics.
The operational complexity measure reflects its dynamic characteristics. These measures
are introduced in Sections 4 and 5, respectively. Section 6 highlights the correlations
between the complexity of Prolog programs and their reliability. In Section 7, the relations
between reliability parameters and both complexity measures, in order to predict/estimate
reliability, are shown. The approach for reliability prediction and estimation is
implemented in a reliability assessment environment, which is introduced in Section 8,
implementing also several well-known software reliability models for comparison purposes.
In Section 9 the results of some experience are introduced.

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
programming system, i.e. the Prolog system. Prolog is, however, not a pure first-order
predicate logic, as it is not restricted to Horn clauses but also includes built-in predicates
(e.g. cut, assert, etc.), which are outside the scope of the first-order predicate logic (see
Clocksin and Mellish, 1987; Lloyd, 1987). A Prolog program is a set of clauses that describe the functionality of the program in a declarative manner. Two forms of clauses are distinguished: facts and rules (goals), which are questions or queries to the system, are considered as rules with an empty head. Clauses (facts and rules) provided through the programmer can further be distinguished from built-in predicates embedded within the underlying system. The following notation for a rule is used in general (a fact is a special form of rule with an empty right part):

\[ C_i(A_{i1}, \ldots, A_{im}) \leftarrow C_{i1}(A_{i1,1}, \ldots, A_{i1,n_{i1}}), \ldots, C_{im}(A_{im,1}, \ldots, A_{im,n_{im}}) \]  

(1)

with \( C_x \) being the functors (names) and \( A_x \) being the arguments of the Prolog clauses.

It is assumed that 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 which is the set of all clauses having the same functor (name) with the same number of arguments, i.e. the same arity. Some Prolog systems (e.g. SB-Prolog, Quintus Prolog) allow a modular program structure, with modules being program entities with clearly defined import and export interfaces and providing a defined functionality. These segments, then, are the modules.

3. FAULT MODEL

A fault is a textual problem with a program, resulting from a mental mistake by a programmer; the mental mistake is defined as an error (IEEE, 1991; IEEE, 1989). Only those faults are considered in this paper which are not detectable by a compiler. A failure is a departure of operation from requirements (Musa et al., 1987; IEEE, 1989). The fault model used here is not based on a special computational model for logic programming, such as the Prolog model (Lloyd, 1987), but relies on the declarative nature of logic programming. The following fault types are considered:

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

There are other features of logic programming which are related to software failures. For example, the selective linear derivation (SLD) resolution (Lloyd, 1987), 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 the finding of the successful branch (Lloyd, 1987). 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 (see Chen and Bastani, 1991). In this paper, only residual faults are considered.

4. STRUCTURAL COMPLEXITY MEASURE

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 clause \( i \) belonging to segment \( k \) is defined as
\[ W_{ki} = \begin{cases} \sum_{j=1}^{n_{ki}} G_{kij} & \text{if clause } i \text{ is a rule} \\ X_{ki} & \text{if clause } i \text{ is a fact} \end{cases} \]  

(2)

where \( k \) is the index referring to segment \( k \); \( G_{kij} \) is the static complexity of predicate \( j \) in the body of clause \( i \); \( n_{ki} \) is the number of predicates in the body of clause \( i \); and \( X_{ki} \) is the static complexity of fact \( i \).

The structural complexity of the predicate \( j \) in the body of clause \( i \) is determined by

\[ G_{kij} = \frac{1}{n_{ki}} M_{kij} A_{kij} \]  

(3)

\[ A_{kij} = \begin{cases} 1 & : \ast \\ 2^{S_{kij}} - 1 & : S_{kij} > 0 \end{cases} \]  

(4)

where \( \ast \) indicates that predicate \( j \) is a fact or a recursive call of clause \( i \) or \( S_{kij} = 0 \); \( M_{kij} \) is the disallocation factor: the 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; \( S_{kij} \) is the total number of arguments of the clauses representing the predicate \( j \).

The structural complexity of a fact \( i \) in segment \( k \) is determined by

\[ X_{ki} = X'_{ki} X''_{ki} \]  

(5)

\[ X'_{ki} = \begin{cases} 1 & : S'_{ki} = 0 \\ 2S'_{ki} - 1 & : S'_{ki} > 0 \end{cases} \]  

(6)

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

(7)

where \( S'_{ki} \) is the number of variables within the arguments of fact \( i \), \( S''_{ki} \) is the number of arguments of fact \( i \), that are not variables and not ground (ground terms are terms in which variables do not occur), \( S''_{ki} \) is the 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 by

\[ W_k = \sum_{i=1}^{n_k} W_{ki} \]  

(8)
with $n_k$ being the number of clauses in segment $k$ and the overall program structural complexity is given by

$$W = \sum_{k=1}^{n} W_k$$  \hspace{1cm} (9)

**Example 1**

Consider a program consisting of the one segment, denoted segment 1, given below. The structural complexity of the program is to be determined.

```prolog
/* **********************************************/
The predicate city/1 finds out the city, where a customer lives.
**********************************************/
customer('Meyer', 'Hans', 'Hamburg', '1234567').
customer('Bergman', 'Jack', 'Los Angeles', '7654321').
city(X) :- customer(X, _, Y, _), write(Y), nl.
```

For the determination of the structural complexity of the program, the structural complexities of all clauses are needed. The numbers after the slashes denote the number of arguments of each clause.

$$W = W_1 = W_{\text{customer}/4} + W_{\text{customer}/4} + W_{\text{city}/1}$$

$$X_{\text{customer}/4} = 1 \hspace{0.5cm} X''_{\text{customer}/4} = 1 \hspace{0.5cm} X_{\text{customer}/4} = 1$$

$$W_{\text{customer}/4} = X_{\text{customer}/4} = 1$$

$$W_{\text{city}/1} = G_{\text{customer}/4} + G_{\text{write}/1} + G_{\text{nl}/0}$$

$$G_{\text{customer}/4} = \frac{1}{2} \times 1 \times 1 \hspace{0.5cm} G_{\text{write}/1} = \frac{1}{2} \times 1 \times 1 \hspace{0.5cm} G_{\text{nl}/0} = \frac{1}{2} \times 1 \times 1$$

$$W_{\text{city}/1} = \frac{1}{2} + \frac{1}{2} + \frac{1}{2} = 1$$

$$W = W_1 = 1 + 1 + 1 = 3$$

**5. OPERATIONAL COMPLEXITY MEASURE**

The operational complexity measure accounts for different use of program segments. Consider a system with a faulty segment that is never used during operation: 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. The operational complexity measure can be calculated for each Prolog program segment and takes into account the following features of its operational profile:

1. the backtracking degree (i.e. number of choice points for deriving a given goal);
2. the extent of recursive and non-recursive executions;
3. the access (execution) frequency of the analysed program segments.

The operational complexity of a segment $k$ is defined by
\[ W'_k = \frac{B_k}{1 + \beta_k} (H_k + R_k) \]  

where \( B_k \) is the utilization factor reflecting the access frequency of segment \( k \) during program operation; \( \beta_k \) is the maximal backtracking degree of segment \( k \) during testing; \( H_k \) is the total number of accesses of segment \( k \) during testing, i.e. the total number of all clause executions of segment \( k \) (not considering recursive calls); and \( R_k \) is the total number of all direct recursive clause executions of segment \( k \) during testing (only the direct recursive calls are counted here).

The overall program operational complexity is obtained from

\[ W' = \sum_{k=1}^{n} W'_k \]  

The utilization factor \( B_k \) can be identified during program operation by applying the following procedure:

(a) 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) and denote the accesses (including also the recursive calls) of each segment \( k \) with \( D_k \);

(b) choose a reference \( D' \) by determining the minimum of the segment access counts:
\[ D' = \min (D_k) \]

(c) conclude that

\[ B_k = \frac{D_k}{D'} \]  

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

**Example 2**

Consider a segment, denoted segment 1, given below (see also Example 3). The operational complexity of the segment is to be determined, having the query `?- a`.

\[ a :- b, d. \]
\[ b. \]
\[ d :- e. \]
\[ d :- f. \]
\[ f. \]

Denoting \( B_1 = 1 \) yields the following:

\( \beta_1 = 1 \) (one choice point due to \( d \), where both \( e \) and \( f \) are called)

\( R_1 = 0 \) (no recursions)

\( H_1 = 5 \) (\( a, b, d, e \) and \( f \) are called)

\( W' = \frac{1}{4} \times 5 = 2.5 \)
6. COMPLEXITY MEASURES VERSUS PROLOG PROGRAMS
RELIABILITY

The introduction of the two complexity measures (Sections 4 and 5) is 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, complexity measures can be best validated by determining to what extent they correlate with program reliability. Intuitively, the more complex software is, the less reliable it is. The authors' experience as well as the logical analyses of the proposed measures support this assumption.

The structural complexity is an indicator for faults, but not failures (a fault is a defective part in a program, which, if executed, may cause a failure, i.e. a deviation from expected behaviour). Structural complexity cannot be related directly to failure occurrence, as the failure occurrence process (due to its random nature), failure rate and reliability depend also on program operation. The combination of operational complexity and the structural complexity is thus the idea followed in this paper to bring both program operation and the indicator of faults together, in order to determine program reliability. There are various program attributes and characteristics that may influence the program failure occurrence process and program reliability. Measures of structural and operational complexity thus consider several program characteristics. The aim of the structural and operational complexity measures is to establish a basis for considering these characteristics, when program failures and reliability are the points of interest.

To identify exactly how software complexity influences its reliability is, however, a complex question. At this stage 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 the authors' experience, structural complexity, in the case of Prolog programs, proved to be a better indicator of the number of inherent faults than conventional measures such as 'faults per lines of code' or 'failure rate per class of statements', which are used in some other failure modelling methods (see, for example, Knuth, 1971; Shooman, 1976). The reason for the above statement is that the structural complexity measure is based on directly identifying potential sources of failure within a Prolog program's segments and clauses. Looking more closely at the definition of the structural complexity of a predicate, the ratio $1/n_{ki}$ (see (3)) represents the relative contribution of each predicate within the body of clause $i$. The disallocation factor $M_{kj}$ 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 (4)) 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 the 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 setting where all arguments are of the right type and $2^{s_{kij}} - 1$ cases having at least one argument not of the right type.

The operational profile of a program is defined as the set of run types that the program can execute along with the probabilities with which they will occur (Musa et al., 1987). 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 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 since 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 analysing which program components (parts, paths, segments, etc.) would be accessed and to what extent. In the case of Prolog programs, the three introduced features of the operational profile may prove useful when trying to model its affect on failure behaviour: the backtracking degree, the extent of recursive and non-recursive executions, and the access frequency of program segments during program operation.

Backtracking is affected by the operational profile, as it is dependent 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 is, the lower is the failure contribution. Assuming a linear reciprocal relation, the effect of backtracking for each segment can be considered through $1/(1+\text{backtracking degree})$. The additive term 1 in the denominator accounts for the feature that if there is no backtracking, i.e. the backtracking degree is zero, then the fraction would have a value of 1 (i.e. the backtracking has no effect). To illustrate the meaning of backtracking degree, consider the following example.

Example 3

Suppose the following segment defines three rules and six facts, where $a$, $b$, $d$, $e$, $f$, $h$ and $j$ represent functors of clauses. The first rule states, for example, that the logical statement $a$ with a variable argument $X$ is true when $b$, $d$ (with the argument $X$) and $e$ are true.

\begin{verbatim}
 a(X):- b(X),e.
b.
e.
d(Y):- f,h(Y).
d(Y):- j(Y).
f.
j(3).
h(1).
h(2).
\end{verbatim}

Having the goal $\text{?- } a(2)$. the system tries to compute an answer to this goal in the following manner: the variable $X$ is unified (i.e. temporarily bounded with a value) with 2 and all the goals $b$, $d(2)$ and $e$ must be derived, in order that goal $\text{?- } a(2)$, is successful. $b$ can be unified with the fact $b$. and for deriving goal $d(2)$ it is necessary to derive either both the goals $f$ and $h(2)$ or the goal $j(2)$. The goal $f$ can be unified with the fact $f$. and for deriving the 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 then to the next choice point upwards from the derivation, i.e. it tries to find an alternative way for deriving 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 the goal ?- a(2). and as there exists 1 choice point in this case, the backtracking degree is 1. Consider now the goal ?- a(3). It can easily be seen that 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.

7. 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 or are not adequate to enable a meaningful estimation. Estimation allows the reliability change over time to be determined and/or provides a means for controlling whether or when the reliability objective is achieved. It is assumed that all detected faults are removed. The faults that are not detected are considered through parameters such as inherent faults (see below). In this section, the relationships between reliability parameters and both segment complexity measures (structural and operational) are discussed in order to predict and estimate the reliability of Prolog programs.

7.1. Reliability Prediction

The basic execution time model as proposed by Musa et al. (1987) is considered. The execution time component for this model assumes that failures occur as a non-homogeneous Poisson process. The initial program failure intensity is given by (Musa et al., 1987)

\[ \lambda_0 = fK\omega_0 \]  

where \( f \) is the linear execution frequency of the program (the average instruction rate divided by the number of object instructions in the program), \( K \) is the fault exposure ratio (which represents the fraction of time that the program run results in a failure), and \( \omega_0 \) is the number of inherent faults (i.e. faults existing before testing begins).

The prediction approach proposed in the following is in fact a modification and refinement of the methodology proposed by Musa et al. (1987) by concentrating on features of a particular class of software systems (Prolog programs). It is believed that the knowledge of structural and operational characteristics of Prolog programs, embedded in the proposed structural and operational complexity measures, would help to achieve better predictions than the traditional and general approach described by Musa et al. (1987). The total number of failures \( \nu_0 \) may be predicted, before testing, from the number of inherent faults \( \omega_0 \) and the fault reduction factor \( B \) (Musa et al., 1987):
\[ \nu_0 = \frac{\omega_0}{B} \] (14)

The fault reduction factor \( B \) is the ratio of net fault reduction to failures experienced as the time of operation approaches infinity. There exists evidence that the values of \( B \) typically result from new fault spawning alone (Basili and Perricone, 1984). It is believed that a project independent value of \( B \) can be found for a class of applications. In the traditional approach, it is assumed that the number of inherent faults is linearly related to program size. In the case of Prolog programs, it is believed that better predictions can be obtained by using a correlation between the number of faults and the overall program structural complexity \( W \). Hence, the number of inherent faults for a Prolog program can be estimated from

\[ \omega_0 = \gamma_0 W \] (15)

where \( \gamma_0 \) is the number of inherent faults per unit of overall structural complexity.

The value of \( \gamma_0 \) is assumed to be fairly constant for nearly similar Prolog software projects and can be determined beforehand from other projects. The present state of the art assumes that the value of the fault exposure ratio must be determined from similar programs. It is believed 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 fact that programs are not generally executed in a straight line, but have many loops and branches and that each instruction can be executed in many different machine states, with a failure usually occurring for only a few of them (Musa, 1975; Musa et al., 1987). 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. The value of \( K \) can be predicted by

\[ K = \sum_{k=1}^{n} K_k \] (16)

\[ K_k = \xi W'_k \] (17)

where \( K_k \) is the fault exposure ratio of segment \( k \) and \( \xi \) is a constant parameter determined from similar projects.

Substituting (17) in (16) and using (11) yields

\[ K = \xi \sum_{k=1}^{n} W'_k = \xi W' \] (18)

The values of \( W'_k \) are not known before testing has commenced. It is suggested that either an average value from similar projects or predicted values of \( W'_k \) depending on \( W_k \) or \( W \) be used.

With values of the number of inherent faults and initial failure intensity available, one can predict various reliability characteristics such as, for example, failures additional to the failure intensity objective and additional execution time to the failure intensity objective (the desired value of the failure intensity).
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 meant 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 removed ($i$ being a sufficiently small integer). The program failure intensity is then given by

$$\lambda_i = fK\omega_i$$  \hspace{1cm} (19)

$$\omega_i = \omega_0 - i$$  \hspace{1cm} (20)

with $\omega_i$ 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 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, for example, expected program size, data from similar projects, etc. Also, other well-known reliability models providing a prediction approach obtain such parameters in a similar way, for example for prediction due to Musa (1975) and Musa et al. (1987) the parameters $f$, $K$, $B$, $\omega_0$, etc. must be determined from similar programs or from experience.

### 7.2. Reliability Estimation

The proposed reliability estimation procedure is based on the following assumptions:

1. Whenever a failure occurs, the fault causing the failure is removed instantaneously;
2. The total number of inherent faults (faults existing before testing begins) in the program is a Poisson random variable;
3. Failures occur independently and randomly according to the per fault hazard rate, which is the same for all faults.

According to the software reliability model classification suggested by Musa et al. (1987), these assumptions imply a Poisson type model.

It is worth mentioning that the cumulative failure probability function is non-decreasing with time. Time is thereby the major basis for failure occurrences. The cumulative failure probability must be non-decreasing with time, whereas the density function must not, as it is decreasing, for example, for an exponential distribution.

A software reliability model describes software failure as a random process, which is characterized by either the times of failure or the number of failures at fixed times. In the following, the (execution) time to failure $i$ and the (execution) time between failures $i - 1$ and $i$ are denoted by $T_i$ and $T'_i$, respectively. The realizations of $T_i$ and $T'_i$ are denoted by $t_i$ and $t'_i$, respectively. Denoting the expected number of failures (with the number of failures being a random process) at time $t$ with $\mu(t)$, the failure intensity $\lambda(t)$ is defined by
\[ \lambda(t) = \frac{d\mu(t)}{dt} \]  

(21)

For a Poisson type model, the following relationships apply (see Musa et al., 1987)

\[ z(t_i|_{t_{i-1}}) = \omega_i f_a(t_{i-1} + t_i) \]  

(22)

\[ \mu(t) = \omega_i F_a(t) \]  

(23)

\[ \lambda(t) = \omega_i f_a(t) \]  

(24)

where \( i \) is the failure number, \( z(t_i|_{t_{i-1}}) \) is the hazard rate in the time interval \( t_i \), \( \omega_i \) is the mean value of the random variable representing the inherent faults, \( F_a(t) \) is the failure probability of an individual fault, and \( f_a(t) \) is the failure density of an individual fault.

The per fault hazard rate \( z_a(t) \) and the failure density of an individual fault are

\[ z_a(t) = \phi \]  

(25)

and

\[ f_a(t) = \phi \exp(-\phi t) \]  

(26)

with an exponential distribution of the time to failure of an individual fault, \( \phi \) being a constant. A model considering this feature was proposed by Goel and Okumoto (1979). In this paper it is suggested that, in the case of Prolog programs, the constant \( \phi \) 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

\[ U_k = \frac{W^k}{\sum_{l=1}^{n} W_l} = \frac{W^k}{W} \]  

(27)

\[ U'_k = \frac{W'_k}{\sum_{l=1}^{n} W'_l} = \frac{W'_k}{W'} \]  

(28)

The value of \( U_k \) is known before testing has commenced and can be taken as constant through program operation (except if large changes to the program are made due to the removal of faults, in which case \( U_k \) must be recomputed). The value of \( U'_k \) can be estimated during operation (for example, by averaging numbers of previous accesses, backtracking, etc. for a given user or a particular type of application). \( U'_k \) 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 \( t_i \), a simple measure of the ‘program failure potential’ at time \( t_i \) is proposed as

\[ V(t_i) = \sum_{k=1}^{n} U_k(t_i) U'_k(t_i) Y_k(t_i) = \sum_{k=1}^{n} V_k(t_i) \]  

(29)
with

\[ Y_k(t_i) = \frac{N_{tk}(t_i)}{N_i(t_i)} \]  

(30)

where \( V_k(t_i) \) is the failure potential of segment \( k \), \( N_{tk}(t_i) \) is the total number of test runs up to the moment \( t_i \) which led to a failure encountered within segment \( k \), and \( N_i(t_i) \) is the 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 \( U_k \) causes no failures during program operation (or testing). If one takes only \( U_k \) into consideration, one would wrongly consider segment \( k \) to be potentially problematic, even if no faults due to this segment were detected during operation (or testing). On the other hand, a segment that is highly faulty, but with a low value of \( U_k \), would not be sufficiently considered in the failure modelling of the program. One must keep in mind that a high or low value for \( U_k \) does not automatically imply a faulty or non-faulty state of segment \( k \), respectively. A high value of \( U_k \) represents solely that a faulty state of segment \( k \) is more probable; this can be validated, however, during program operation, or 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 \( N(t_i)V(t_i)/N_i(t_i) \) 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 with \( t_i \) the total execution time up to the last detected and corrected fault, \( \phi \) is given by

\[ \phi = \rho \frac{N(t_i)V(t_i)}{N_i(t_i)} \]  

(31)

where \( \rho \) is a constant factor for a class of Prolog programs. The access frequency of program components during testing and operation, which is the testing view as well as the user-oriented view of the program behaviour, is considered through the parameters of the operational complexity such that the assumption of a constant \( \rho \) is reasonable. The determination of \( \phi \) after (31) is based on the following considerations. Owing to (25), \( \phi \) is the contribution of each fault to the program hazard rate. As mentioned before, it is suggested that in the case of Prolog programs, \( \phi \) can be related to program characteristics using the proposed complexity measures. The failure potential \( V(t_i) \) represents the fault-proneness of the program (with a fault causing a failure) at the time \( t_i \), i.e. after having detected and corrected \( i \) faults. As \( V(t_i) \) considers relative structural and operational complexities due to (29) for the test period \( t_i \) with \( N(t_i) \) test runs, the contribution of all tests runs to failures is \( N(t_i)V(t_i) \) and the per fault contribution is the latter 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 identical with the number of test runs up to \( t_i \) that led to a failure, i.e. \( N_i(t_i) \). As mentioned before, it has also been observed that, as faults are detected and corrected during testing, the term \( N(t_i)V(t_i)/N_i(t_i) \) approaches a value that is changing in a small interval and is nearly constant. Using this, combined with the considerations described above, (31) was found to be a meaningful approach for determining \( \phi \) for Prolog programs, achieving good results (Winter, 1992; Kamundi, 1993).
The expected number of failures and the failure intensity for the Poisson process are

\[ \mu(t) = \omega_0 (1 - \exp (-\phi t)) \]  
(32)

\[ \lambda(t) = \omega_0 \phi \exp (-\phi t) \]  
(33)

The expected number of failures (failures to be experienced) is to be distinguished from the failure potential. The latter is used to determine \( \phi \), which is itself used to determine the expected number of failures. What the user observes are failures \( \mu(t) \), which are dependent on the inherent faults \( \omega_0 \) (the total number of faults in the program) as well as \( \phi \), which reflects the test phase (test profile), including fault detection and correction.

Using 'maximum likelihood estimation' (MLE) (see, for example, Musa et al., 1987; Shooman, 1983), the estimator for \( \omega_0 \) is the solution to the following relationship:

\[ \frac{N_f(t_i)}{\omega_0} = 1 - \exp (-\phi S) \]  
(34)

with

\[ S = \sum_{i=1}^{N_f(t_i)} t'_i \]  
(35)

Using (33) the additional time to failure intensity objective can be determined through the following relationships:

\[ \frac{\lambda(t_i)}{\lambda(t_p)} = \exp (\phi (t_p - t_i)) \]  
(36)

\[ t_\Delta = t_i - t_p = \frac{1}{\phi} \ln \frac{\lambda(t_p)}{\lambda(t_i)} \]  
(37)

where \( t_i \) is the execution time to the failure intensity objective, \( t_p \) is the execution time up to the present failure intensity, \( \lambda(t_i) \) is the failure intensity objective, \( \lambda(t_p) \) is the present failure intensity, and \( t_\Delta \) is the additional execution time to the failure intensity objective.

For estimation, only one parameter \( \rho \) must be determined from similar projects. This is acceptable due to the good results that were achieved through the estimation approach discussed above.

8. RELIABILITY ASSESSMENT ENVIRONMENT

The approach for reliability prediction and estimation is implemented in the reliability assessment environment PRORool (PROlog based environment for Reliability determination of Object-Oriented and Logic programs), which also implements several well-known software reliability models for comparison purposes. Companion utilities for PRORool provide automatic calculations for determining the complexity measures of Prolog programs. The reliability models of Musa (1975), Musa et al. (1987), Goel and Okumoto
(1979), Jelinski and Moranda (1972) and Schick and Wolverton (1978), 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, the MLE method was implemented. A typical session of PRORool contains the following.

(1) Selection of the input data set (Figure 1). The data columns in the figure are in the following order: test run number, number of failures, execution time of the test run, cumulative execution time up to the test run, time of failure, time elapsed to remove the fault due to a failure and the calendar time. A value of –1 for times means that the respective test run did not result in a failure.

(2) Selection of the model type.

(3) Reliability parameter calculation.

(4) Viewing the results (Figure 2). The calculated reliability parameters are shown in the figure. They are commented upon using the comment character ‘#’, giving the meaning and the unit (if needed). The second line states the model name, for example, and the last line gives the number of inherent faults calculated for the input data set.

9. SOME RESULTS

For the establishment of values for constant parameters, empirical analysis of several Prolog programs was undertaken. These are the parameters $\gamma_0$, $\xi$, $r$ (average instruction rate) and $\bar{W}_k$ (average value) for prediction as well as $\rho$ for estimation. The average
instruction rate $r$ (instructions executed in time) used for the determination of $f$ is actually a constant, but not $f$ itself, so the parameter $r$ is the point of interest. Five programs with executable code between 50 and 500 kB (kilo Byte) were analysed. Most of them were written more than once by different programmers, the parameter values were then averaged. The results are shown in Table I, where $\overline{W}_k^*$ denotes $\overline{W}_k$ per 1000 units of $W$.

The instruction rate depends on the machine and the environment, i.e. operating system, reduced instruction set computer (RISC) or complex instruction set computer (CISC) machine type, the actual load, number of processes running, etc. The programs analysed were run under normal circumstances several times on the same machine and environment to obtain an average instruction rate. To determine $f$, the parameter $r$ is to be divided by the number of object (machine) instructions $l$ of a program. The parameter $l$ can be determined using compiler options to produce machine code instead of directly executable code.

<table>
<thead>
<tr>
<th>No.</th>
<th>$W$</th>
<th>$\gamma_0$</th>
<th>$\xi$</th>
<th>$\overline{W}_k^*$</th>
<th>$\rho$ (s$^{-1}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>67325</td>
<td>$10^{-4}$</td>
<td>0.011</td>
<td>0.0105</td>
<td>0.39</td>
</tr>
<tr>
<td>2</td>
<td>85550</td>
<td>$10^{-4}$</td>
<td>0.012</td>
<td>0.0108</td>
<td>0.39</td>
</tr>
<tr>
<td>3</td>
<td>175367</td>
<td>$10^{-4}$</td>
<td>0.012</td>
<td>0.0109</td>
<td>0.38</td>
</tr>
<tr>
<td>4</td>
<td>253636</td>
<td>$10^{-4}$</td>
<td>0.013</td>
<td>0.0109</td>
<td>0.35</td>
</tr>
<tr>
<td>5</td>
<td>271572</td>
<td>$10^{-4}$</td>
<td>0.013</td>
<td>0.0114</td>
<td>0.34</td>
</tr>
</tbody>
</table>
Table II. Parameters for prediction

<table>
<thead>
<tr>
<th>$f$</th>
<th>$\gamma_0$</th>
<th>$\omega_0$</th>
<th>$\bar{w}_k^*$</th>
<th>$\xi$</th>
<th>$K$</th>
<th>$\lambda_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5 s$^{-1}$</td>
<td>$10^{-4}$</td>
<td>23</td>
<td>2.5</td>
<td>0.012</td>
<td>0.15</td>
<td>1.73 s$^{-1}$</td>
</tr>
</tbody>
</table>

Table III. Parameters for prediction in early testing stages

<table>
<thead>
<tr>
<th>$N$</th>
<th>$i$</th>
<th>$\omega_i$</th>
<th>$\lambda_i$</th>
<th>$K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>18</td>
<td>1.58 s$^{-1}$</td>
<td>0.18</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>15</td>
<td>1.02 s$^{-1}$</td>
<td>0.14</td>
</tr>
</tbody>
</table>

One of the programs investigated in detail in (Winter, 1992) was a medium sized Prolog program of about 110 kB of executable code, serving as a database program providing functions like Insert, Update, Delete, etc. for handling data records. For prediction (see Section 7.1), with $W = 230,294$, using Table I (averaging parameters), the following can be determined: $\gamma_0 = 10^{-4}$, $\xi = 0.012$, $\bar{W}_k^*$ = 2.5 for all $k$. With $n = 5$ (number of segments), $K = 0.15$ can be obtained. $f$ was determined to be $f = 0.5$ s$^{-1}$. With $\gamma_0 = 10^{-4}$, $\omega_0 = 23$ is obtained. With these values of $f$, $K$ and $\omega_0$, $\lambda_0 = 1.73$ s$^{-1}$ is determined. Table II shows these results at a glance.

Using (19) and (20) for prediction and considering data from the early stages of testing (with 45 being the total number of test runs) the results achieved are shown in Table III.

A comparison of $\lambda$ after detecting and correcting 15 faults is shown in Table IV with $A$ representing estimation using the execution time model (Musa, 1975; Musa et al., 1987), $B$ representing prediction using (13)–(18), $C$ representing prediction using (19) and (20) (with $N = 20$) and $D$ representing prediction using the approach of Azem et al. (1993). Real life failure intensity corresponds to the actual total execution time divided by the actual total number of failures. This is an instantaneous interpretation of failures in time (failure intensity) at each failure detection point, which is used here to compare the quality of predictions/estimations with the actual program state. It can be seen that using the feature oriented approach proposed in this paper yields better predictions (cases $B$ and $C$).

For estimation (see Section 7.2), with $N = 45$, $N_f = 15$, $t = 21.38$ s ($t_{cum}$, see Table V), $V = 499.2338 \times 10^{-4}$ from Table VI, and $\rho = 0.37$ s$^{-1}$ from Table I (average), $\phi = 0.055$ s$^{-1}$ is obtained and the failure intensity of the program is determined to be

Table IV. Comparison of predictions

<table>
<thead>
<tr>
<th>Real life</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.70 s$^{-1}$</td>
<td>0.26 s$^{-1}$</td>
<td>0.35 s$^{-1}$</td>
<td>0.37 s$^{-1}$</td>
<td>0.24 s$^{-1}$</td>
</tr>
</tbody>
</table>
\( \lambda = 0.37 \text{ s}^{-1} \). Table V shows the time values of the test runs, where \( t_{\text{exec}} \) denotes the execution time, \( t_{\text{cum}} \), the cumulative execution time, \( t_{\text{fail}} \), the execution time between the last and present failure, \( t_{\text{corr}} \), the time for removing a fault, and \( t_{\text{used}} \), the total elapsed time.

Table VII summarizes the results of the estimation of failure intensity using the reliability models of Jelinski and Moranda (1972), Musa (1975), Musa et al. (1987) and Goel and Okumoto (1979) as well as the real life data and the failure intensity using the estimation approach for Prolog programs proposed in this paper. Case A represents estimation after the execution time model (Musa, 1975; Musa et al., 1987); B after Jelinski and Moranda (1972), C after Goel and Okumoto (1979) and D after the estimation approach for Prolog programs proposed in this paper. It can be seen that using the feature oriented estimation approach yields a better estimation (case D).

Acknowledgement

This paper was partly supported by the University of Paderborn and the Heinz-Nixdorf-Institut Paderborn, Project numbers ZIT 521 and ZIT 538. Our thanks are due to Michael Winter (University of Paderborn) and the anonymous reviewers for valuable remarks and suggestions.

References


Table V. Time values of the test runs

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N_t$</th>
<th>$t_{exec}$ (s)</th>
<th>$t_{cum}$ (s)</th>
<th>$t_{fail}$ (s)</th>
<th>$t_{corr}$ (min)</th>
<th>$t_{used}$ (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0.10</td>
<td>0.10</td>
<td>—</td>
<td>—</td>
<td>5.87</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.05</td>
<td>0.15</td>
<td>0.15</td>
<td>3</td>
<td>0.05</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0.33</td>
<td>0.48</td>
<td>0.33</td>
<td>3</td>
<td>6.12</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0.38</td>
<td>0.86</td>
<td>—</td>
<td>—</td>
<td>7.69</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0.34</td>
<td>1.20</td>
<td>0.72</td>
<td>9</td>
<td>6.33</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0.39</td>
<td>1.59</td>
<td>—</td>
<td>—</td>
<td>10.46</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>0.40</td>
<td>1.99</td>
<td>—</td>
<td>—</td>
<td>16.35</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>0.38</td>
<td>2.37</td>
<td>1.17</td>
<td>25</td>
<td>13.79</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>0.38</td>
<td>2.75</td>
<td>—</td>
<td>—</td>
<td>14.82</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>0.42</td>
<td>3.17</td>
<td>0.80</td>
<td>4</td>
<td>55.69</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>0.46</td>
<td>3.63</td>
<td>—</td>
<td>—</td>
<td>59.39</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>0.34</td>
<td>3.97</td>
<td>0.80</td>
<td>3</td>
<td>13.18</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>0.33</td>
<td>4.30</td>
<td>—</td>
<td>—</td>
<td>13.72</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>0.40</td>
<td>4.70</td>
<td>0.73</td>
<td>37</td>
<td>23.78</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
<td>0.51</td>
<td>5.21</td>
<td>—</td>
<td>—</td>
<td>24.02</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>0.55</td>
<td>5.76</td>
<td>1.06</td>
<td>43</td>
<td>19.50</td>
</tr>
<tr>
<td>17</td>
<td>8</td>
<td>0.45</td>
<td>6.21</td>
<td>—</td>
<td>—</td>
<td>21.86</td>
</tr>
<tr>
<td>18</td>
<td>8</td>
<td>0.51</td>
<td>6.72</td>
<td>—</td>
<td>—</td>
<td>60.91</td>
</tr>
<tr>
<td>19</td>
<td>8</td>
<td>0.46</td>
<td>7.18</td>
<td>—</td>
<td>—</td>
<td>52.37</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>0.69</td>
<td>7.87</td>
<td>—</td>
<td>—</td>
<td>81.75</td>
</tr>
<tr>
<td>21</td>
<td>9</td>
<td>2.08</td>
<td>9.95</td>
<td>4.19</td>
<td>17</td>
<td>11.92</td>
</tr>
<tr>
<td>22</td>
<td>9</td>
<td>0.61</td>
<td>10.56</td>
<td>—</td>
<td>—</td>
<td>10.60</td>
</tr>
<tr>
<td>23</td>
<td>10</td>
<td>0.34</td>
<td>10.90</td>
<td>0.95</td>
<td>5</td>
<td>7.80</td>
</tr>
<tr>
<td>24</td>
<td>10</td>
<td>0.56</td>
<td>11.46</td>
<td>—</td>
<td>—</td>
<td>8.02</td>
</tr>
<tr>
<td>25</td>
<td>10</td>
<td>0.60</td>
<td>12.06</td>
<td>—</td>
<td>—</td>
<td>58.71</td>
</tr>
<tr>
<td>26</td>
<td>10</td>
<td>0.50</td>
<td>12.56</td>
<td>—</td>
<td>—</td>
<td>74.20</td>
</tr>
<tr>
<td>27</td>
<td>10</td>
<td>0.38</td>
<td>12.94</td>
<td>—</td>
<td>—</td>
<td>52.34</td>
</tr>
<tr>
<td>28</td>
<td>10</td>
<td>0.49</td>
<td>13.43</td>
<td>—</td>
<td>—</td>
<td>21.34</td>
</tr>
<tr>
<td>29</td>
<td>10</td>
<td>0.56</td>
<td>13.99</td>
<td>—</td>
<td>—</td>
<td>22.91</td>
</tr>
<tr>
<td>30</td>
<td>11</td>
<td>0.44</td>
<td>14.43</td>
<td>3.53</td>
<td>23</td>
<td>40.42</td>
</tr>
<tr>
<td>31</td>
<td>12</td>
<td>0.49</td>
<td>14.92</td>
<td>0.49</td>
<td>8</td>
<td>23.74</td>
</tr>
<tr>
<td>32</td>
<td>12</td>
<td>0.50</td>
<td>15.42</td>
<td>—</td>
<td>—</td>
<td>28.18</td>
</tr>
<tr>
<td>33</td>
<td>12</td>
<td>0.39</td>
<td>15.81</td>
<td>—</td>
<td>—</td>
<td>38.66</td>
</tr>
<tr>
<td>34</td>
<td>12</td>
<td>0.53</td>
<td>16.34</td>
<td>—</td>
<td>—</td>
<td>28.23</td>
</tr>
<tr>
<td>35</td>
<td>12</td>
<td>0.44</td>
<td>16.78</td>
<td>—</td>
<td>—</td>
<td>22.78</td>
</tr>
<tr>
<td>36</td>
<td>13</td>
<td>0.45</td>
<td>17.23</td>
<td>2.31</td>
<td>7</td>
<td>20.16</td>
</tr>
<tr>
<td>37</td>
<td>13</td>
<td>0.44</td>
<td>17.67</td>
<td>—</td>
<td>—</td>
<td>23.22</td>
</tr>
<tr>
<td>38</td>
<td>14</td>
<td>0.44</td>
<td>18.11</td>
<td>0.88</td>
<td>4</td>
<td>22.93</td>
</tr>
<tr>
<td>39</td>
<td>14</td>
<td>0.46</td>
<td>18.57</td>
<td>—</td>
<td>—</td>
<td>77.18</td>
</tr>
<tr>
<td>40</td>
<td>14</td>
<td>0.38</td>
<td>18.95</td>
<td>—</td>
<td>—</td>
<td>7.25</td>
</tr>
<tr>
<td>41</td>
<td>14</td>
<td>0.46</td>
<td>19.41</td>
<td>—</td>
<td>—</td>
<td>49.54</td>
</tr>
<tr>
<td>42</td>
<td>14</td>
<td>0.45</td>
<td>19.86</td>
<td>—</td>
<td>—</td>
<td>23.61</td>
</tr>
<tr>
<td>43</td>
<td>14</td>
<td>0.46</td>
<td>20.32</td>
<td>—</td>
<td>—</td>
<td>62.34</td>
</tr>
<tr>
<td>44</td>
<td>14</td>
<td>0.56</td>
<td>20.88</td>
<td>—</td>
<td>—</td>
<td>23.33</td>
</tr>
<tr>
<td>45</td>
<td>15</td>
<td>0.50</td>
<td>21.38</td>
<td>3.27</td>
<td>5</td>
<td>25.21</td>
</tr>
</tbody>
</table>
Table VI. Program parameters

<table>
<thead>
<tr>
<th>Segment No. $k$</th>
<th>$U_k'$</th>
<th>$U_k$</th>
<th>$N_{tk}$</th>
<th>$V_k \times 10^4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.240639</td>
<td>0.001193</td>
<td>3</td>
<td>0.57416</td>
</tr>
<tr>
<td>2</td>
<td>0.128341</td>
<td>0.007049</td>
<td>4</td>
<td>2.41247</td>
</tr>
<tr>
<td>3</td>
<td>0.288766</td>
<td>0.000894</td>
<td>1</td>
<td>0.17210</td>
</tr>
<tr>
<td>4</td>
<td>0.154256</td>
<td>0.003503</td>
<td>3</td>
<td>1.08072</td>
</tr>
<tr>
<td>5</td>
<td>0.187999</td>
<td>0.987361</td>
<td>4</td>
<td>494.99435</td>
</tr>
</tbody>
</table>

Table VII. Comparison of estimations

<table>
<thead>
<tr>
<th>Real life</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.70 s(^{-1})</td>
<td>0.26 s(^{-1})</td>
<td>0.31 s(^{-1})</td>
<td>0.36 s(^{-1})</td>
<td>0.37 s(^{-1})</td>
</tr>
</tbody>
</table>


