ESPOO 2006

VTT WORKING PAPERS 63

```
l1->mode = md[0];

/* if caller wants to read */
if (md[0] == 'r' && (fd = dup (fileno (stdout))) >= 0)
{
  if ((l1->fp = freopen (temp_na  "wb", stdout)))
  {
    l1->exit_status = system (
    if (dup2 (fd, fileno (std
      l1->fp = fopen (temp
  }
  close (fd);
}
/* if caller wants to
else if (md[0] == 'w'                      (1 + strlen (cm))))
{
  strcpy (l1->command, cm
  l1->fp = fopen (tem
}

if (l1->fp)
{
```

Methods and problems of software reliability estimation

Ilkka Karanta

VTT

ISBN 978-951-38-6622-8 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1459-7683 (URL: http://www.vtt.fi/publications/index.jsp)

Copyright © VTT 2006

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 3, PL 1000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 4374

VTT, Bergsmansvägen 3, PB 1000, 02044 VTT
tel. växel 020 722 111, fax 020 722 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 1000, FI-02044 VTT, Finland
phone internat. +358 20 722 111, fax +358 20 722 4374

VTT, Vuorimiehentie 3, PL 1000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 6027

VTT, Bergsmansvägen 3, PB 1000, 02044 VTT
tel. växel 020 722 111, fax 020 722 6027

VTT Technical Research Centre of Finland, Vuorimiehentie 3, P.O. Box 1000, FI-02044 VTT, Finland
phone internat. +358 20 722 111, fax +358 20 722 6027

Published by

Series title, number and report code of publication

VTT Working Papers 63
VTT–WORK–63

Author(s)
Karanta, Ilkka

Title

Methods and problems of software reliability estimation

Abstract

There are many probabilistic and statistical approaches to modelling software reliability. Software reliability estimates are used for various purposes: during development, to make the release decision; and after the software has been taken into use, as part of system reliability estimation, as a basis of maintenance recommendations, and further improvement, or a basis of the recommendation to discontinue the use of the software. This report reviews proposed software reliability models, ways to evaluate them, and the role of software reliability estimation. Both frequentist and Bayesian approaches have been proposed. The advantage of Bayesian models is that various important but nonmeasurable factors, such as software complexity, architecture, quality of verification and validation activities, and test coverage are easily incorporated in the model.

Despite their shortcomings – excessive data requirements for even modest reliability claims, difficulty of taking relevant nonmeasurable factors into account etc. – software reliability models offer a way to quantify uncertainty that helps in assessing the reliability of a software-based system, and may well provide further evidence in making reliability claims.

ISBN
978-951-38-6622-8 (URL: http://www.vtt.fi/publications/index.jsp)

Series title and ISSN		Project number
VTT Working Papers 1459-7683 (URL: http://www.vtt.fi/publications/index.jsp)		13107
Date December 2006	Language English	Pages 57 p.
Name of project SAFIR/PPRISMA06	Commissioned by VYR	
Keywords Software reliability, Bayesian software, modelling, reliability models, Poisson models, Bayesian models, errors, faults, failures, software life-cycle	Publisher VTT Technical Research Centre of Finland P.O. Box 1000, FI-02044 VTT, Finland Phone internat. +358 20 722 4404 Fax +358 20 722 4374	

Preface

This report has been prepared as a part of the PPRISMA project. PPRISMA (Principles and Practices of Risk-Informed Safety MAnagement) was carried out in SAFIR, the The Finnish Research Programme on Nuclear Power Plant Safety, which lasted from 2003 to 2006. The research programme was founded by Valtion Ydinjätehuoltorahasto.

The manuscript was commented by Jan-Erik Holmberg and Urho Pulkkinen, which is gratefully acknowledged. Jan pointed out that usually operational data does not come from a single source but from several installations that possibly operate in different circumstances; he also noted that most modern embedded software consists of at least two components – the platform and the application – which have different reliability characteristics.

Contents

Preface 5

List of symbols, concepts and abbreviations 8

1. Introduction 11

2. Some modelling considerations 15
   - 2.1 Errors, faults, failures and reliability 15
   - 2.2 Reliability characteristics of software 17
   - 2.3 Reliability and the software lifecycle 19
   - 2.4 Sources of software reliability evidence 20

3. Software reliability growth models 21
   - 3.1 General considerations 21
     - 3.1.1 Reliability model classification 21
     - 3.1.2 Frequentist versus Bayesian models for likelihood 22
     - 3.1.3 Standard assumptions 22
   - 3.2 Nonhomogenous Poisson models 23
     - 3.2.1 Goel and Okumoto's model 24
     - 3.2.2 Musa-Okumoto model 25
     - 3.2.3 Schneidewind's model 25
     - 3.2.4 Musa's basic execution time model 26
     - 3.2.5 S-shaped model 28
     - 3.2.6 Inflection S-shaped growth curve model 29
     - 3.2.7 K-stage Erlangian (Gamma) growth curve model 29
     - 3.2.8 Duane model 30
     - 3.2.9 Pham-Nordmann-Zhang model 30
   - 3.3 Other frequentist models 31
     - 3.3.1 Jelinski-Moranda de-eutrophication model 31
     - 3.3.2 Geometric model 32
     - 3.3.3 Hyperexponential model 32
     - 3.3.4 Weibull model 34
     - 3.3.5 Barghout-Littlewood-Abdel-Galy model 35
   - 3.4 Bayesian models 35
     - 3.4.1 Bayesian versions of frequentist reliability models 36
     - 3.4.2 Littlewood-Verrall model 36
     - 3.4.3 Other Bayesian models 37
   - 3.5 Special-purpose models 38

4. Software reliability models 40
4.1 Homogenous Poisson model 40
4.2 Other frequentist models 41
4.3 Bayesian reliability models 41

5. Application to the operational data problem...42
5.1 Requirements on the data 42
5.2 Evaluation and comparison criteria for software reliability models 43
5.2.1 U-plot 44
5.2.2 Y-plot 45
5.2.3 Bayes factor..45
5.2.4 Prequential predictive ordinate 46
5.2.5 Prequential likelihood and prequential likelihood ratio 46
5.2.6 Akaike information criterion...47
5.3 Some comparisons...47
5.4 Selection of model...48

6. Summary and conclusions 50

References 52

List of symbols, concepts and abbreviations

cumulative distribution function $F(x)$	For a random variable x, the probability $P[X \leq x]$
cumulative failure function $\mu(t)$	the average cumulative failures associated with each point of time: $\mu(t) = E[M(t)]$. $\mu(0) = 0$.
cumulative number of failures $M(t)$	the total number of failures experienced up to time t
failure	the inability of a system or component to perform its required functions within specified performance requirements
failure intensity function $\lambda(t)$	the derivative (rate of change) of the cumulative failure function
failure rate function $\frac{P(t \leq T \leq t + \Delta t \mid T > t)}{\Delta t}$	the probability that a failure per unit time occurs in the interval $[t.t + \Delta t]$, given that a failure has not occurred before t
fault	an incorrect step, process or data definition in a software artefact
finite failure model	a model for which the expected number of failures, $\mu(t)$, remains finite as $t \to \infty$. This means that $\lim_{t \to \infty} \mu(t) = a$ for some $a > 0$.
hazard rate $z(t)$	$\lim_{\Delta t \to 0} \frac{P(t \leq T \leq t + \Delta t \mid T > t)}{\Delta t}$, the limit of failure rate function at 0.
I	the number of time intervals in the observation range
infinite failure model	a model for which the expected number of failures, $\mu(t)$, grows without bound as $t \to \infty$.

Kolmogorov distance	For two vectors $(x_1,...,x_M)$ and $(y_1,...,y_M)$, $\max_i(\lvert x_i - y_i \rvert)$
m , $m(t_1,t_2)$, m_i	the number of failures in the time interval $[t_1,t_2]$ or the ith time interval
mean failure function	same as cumulative failure function
MTTF	mean time to failure
$n(t)$, $n(t_1,t_2)$, n_i	the number of faults detected so far, in time interval $[t_1,t_2]$ (the same fault can give rise to many failures, and faults can be detected even if they haven't yet caused a failure), or in the ith time interval.
N	see total number of faults
$\overline{N}$	the total number of faults in the software initially
NHPP	nonhomogenous Poisson process
$\nu(t)$	The number of undetected faults at time t
observation range	the period of time from which the reliability data stems.
reliability growth model	a model that predicts, given a history of failures and corrective actions, the probability that the software under consideration will work failurelessly for the given time period under the given operational conditions. Typically applied to make the decision to release the software (when a satisfactory level of reliability has been reached).
reliability model	a model that predicts the mean time to failure [Hamlet 1992]. Typically applied after debugging, when the program has been tested,

	and no failures have been observed.
T	the time to failure
τ_i	The time between the *i-1*th and *i*th software failure
total number of faults N	the total number of faults in the software (usually meaningful only if we assume that new faults aren't introduced to the software by maintenance)

1. Introduction

Dependability is defined [Laprie 1992] as the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers. Dependability has several aspects [Sommerville 2001]:

- availability, or readiness for usage. Informally, the probability that the system will be up and running and able to deliver useful services at any given time.

- reliability, or continuity of service. Informally, the probability, over a given period of time, that the system will correctly deliver services as expected by the user.

- safety, or avoidance of catastrophic consequences on the environment. Informally, the likelihood (usually judgmental) that the system will cause damage to people or its environment.

- security, or prevention of unauthorized access and/or handling of information. Informally, the likelihood (usually judgmental) that the system can resist accidental or deliberate intrusion.

This report concentrates on reliability, leaving availability, safety and security issues aside.

When estimating the reliability of software, two main paths can be taken. One is to analyze the code by means of static analyzers (see e.g. [Khoshgoftaar and Munson 1990]), model checkers, theorem provers, compilers etc. Another is to analyze the software from an external point of view; here, the main sources of input are

- testing; software testing, including unit, integration and system testing, provides much data about the program's reliability if test coverage is good, tests are systematically conducted, and at least part of the testing is done under realistic operating conditions

- expert opinion; this preferably comes from experts that haven't participated in the development of the software (to avoid myopy), but are experienced in software development (including architecture and code analysis issues), have access to the software's source code and documentation, may carry out static analysis, and have a good understanding of the requirements and operating conditions of the software

- operational data (failure reports etc.); if high-quality operational data is available from a sufficiently long time period, it provides a solid ground for stochastic analysis of reliability.

Reliability estimates of software can be used in a number of ways, among them:

- to estimate the reliability of a total system of which the software is a part
- to allocate resources during development and maintenance
- to estimate maintenance costs.

We consider the problem of arriving at a reliability estimate for a piece of software. The assumed end use of the reliability estimates is risk and reliability analysis of software-based systems, e.g. nuclear power plants. We assume that informed expert opinion exists about a software's reliability, and try to update this reliability estimate with information contained in operational data records.

Probability estimates can be arrived at in three, mutually non-exclusive ways:

- *Subjective degrees of belief* tell how much trust we (or experts, or whoever made the estimate) put on that the event actually occurs. This is often the best way of arriving at probability estimates when data is scarce or nonexistent, or when the situation involves very complex elements.

- *Propensities* tell how probable the event is based on some formal model(s). The model is usually mathematical or physical. For example, after making a model for a die, we may arrive at the conclusion that the probability of the die arriving on each of its faces is equal. This is often the best way to arrive at probability estimates when data is scarce but detailed modelling is feasible.

- *Frequencies* tell how many times (relatively speaking) the event actually occurred in a dataset. This is often the best way to arrive at probability estimates when data is abundant.

In Bayesian statistics, there are two kinds of probabilities of an event. *Prior probability* expresses the probability estimate before a new event (measurement, arrival of data or such) takes place. *Posterior probability* expresses the probability estimate after the event.

Subjective probabilities (expert opinions) are often used as prior probabilities, which are updated by likelihood derived from operational data to yield the posterior probabilities. This is the case in the two-stage process described by Helminen (2005) (see also [Helminen and Pulkkinen 2003]), which consists of

1. a judgmental reliability estimate by experts. This is used as a prior probability in the Bayesian setting.

2. utilizing prior probability, operational data and test results in finding the posterior reliability estimates.

Figure 1 illustrates the process in IDEF0 notation.

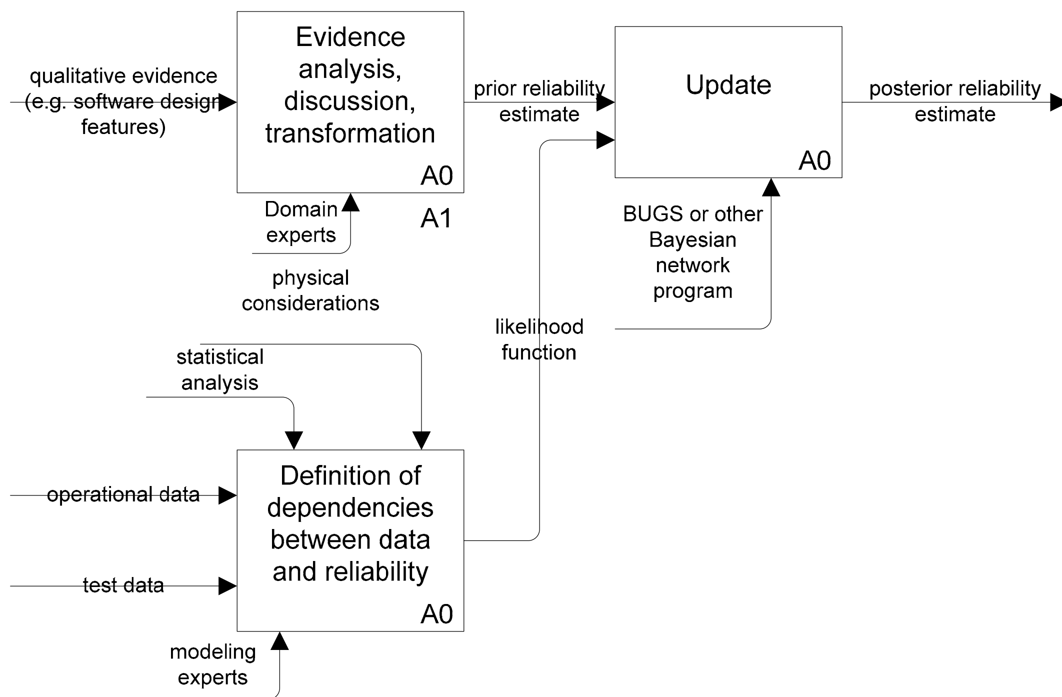

Figure 1.The process of Bayesian reliability assessment described in [Helminen 2005].

To state the problem in Bayesian terms,

$$p(t_b, t_e, m_f \mid data) = \frac{p(data \mid t_b, t_e, m_f)\, p(t_b, t_e, m_f)}{p(data)} \quad (1)$$

Here the event t_b, t_e, m_f means the prediction that the software will fail m_f times in the time period starting from time t_b and lasting until time t_e; if the different failure modes would be counted separately, a separate index term would be inserted to the

event. Note that the reliability of the system is a special case of this formulation: it is the probability that $m_f = 0$.

The event *data* means that the operational data is what it is; if the data would be in time series form, it might indicate that software failed m_d times in the time period starting at time $t_{d,b}$ and lasting until time $t_{d,e}$; there might of course be data for several time periods. The data might alternatively consist of failure events with timing information.

The prior probability $p(t_b, t_e, m_f)$ is, as stated above, arrived at through expert judgment. The prior probability for the data, $p(data)$, can be thought of as a scaling factor, for discounting exceptional circumstances.

This report mainly focuses on finding an estimate for the likelihood function $p(data \mid t_b, t_e, m_f)$. In what follows, the information about t_b, t_e, m_f is usually embedded in models: we try to find the likelihood of the data given that we have some model with which to predict the number of failures. Then, the problem essentially boils down to finding a suitable model class, and estimating the parameters of the model by maximizing the likelihood function wrt. the parameters.

2. Some modelling considerations

2.1 Errors, faults, failures and reliability

First, we consider what we would actually want to model, and what we mean by software failure. Below, we follow the IEEE terminology [IEEE 90] unless stated otherwise.

A *fault* (the term *bug* can be used interchangeably) in a software artefact is an incorrect step, process, or data definition. Under suitable circumstances, a fault may cause a *failure*, or "the inability of a system or component to perform its required functions within specified performance requirements"; that is, a deviation from the stated or implied requirements. Between failures, the fault is dormant.

Failures have different consequences, some being harmless and others perhaps fatal. Failures can be classified into severity classes, e.g. catastrophic, major, minor. The definitions of severity classes vary from system to system. An example classification is given in Table 1 [Donnelly et al. 1996].

Table 1. Severity classification based on service degradation.

Severity classification	**Definition**	**Example**
Catastrophic	Entire system failed, no functionality left	The robot doesn't function at all
Severe	A high-priority customer feature is not working	The robot must be operated manually
Significant	Customers must change how they use the system	The robot's settings cannot be changed remotely; if they are to be changed, they must be configured from the control panel
Minor	Problem is not noticeable by customers	Some maintenance functions are not currently performed

An *error* is a "discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. Examples include exceptional conditions raised by the activation of existing software faults, and incorrect computer status due to an unexpected external interference. The term is especially useful in fault-tolerant computing to describe an intermediate stage in between faults and failures." [Lyu 1996b]

Besides software faults, failures may also be caused by improper input either by a human actor or some component of the system (e.g. a measurement device), and by failing hardware or other equipment.

How often faults manifest themselves as failures depends on how the software is used. The *operational profile* of a system is "the set of operations that the software can execute along with the probability with which they will occur" [Lyu 1996b].

Reliability is "The ability of a system or component to perform its required functions under stated conditions for a specified period of time". Stated in slightly different terms [Sommerville 2001], it is the *probability* that the system or components performs its required functions under stated conditions for the specified period of time:

$$p(\neg fail, t_b, t_e) \tag{2}$$

This probability is, however, the object of interest mainly when the *probability of failure-free survival of mission* is of concern; in such situations, there is a natural mission time. Examples are airplane flights, space flights etc. Other measures are more appropriate in other situations, and include

- The *mean time to next failure* (MTTF). This is often used to determine if the software is reliable enough that it can be released.

- The expected number of failures in the time interval of interest. This is also called *rate of occurrence of failures* (ROCOF). It is the appropriate measure for a system which actively controls some potentially dangerous process [Littlewood and Strigini 1993].

- *Probability of failure on demand*. This is suitable for a safety system, which is only called on to act when another system gets into a potentially unsafe condition. An example is an emergency shutdown system of a nuclear reactor [Littlewood and Strigini 1993].

- *Availability*. This could be used in circumstances when the losses incurred as a result of system failure depend on the length of time that the system is unavailable. Examples: airline reservation systems, telephone switches [Littlewood and Strigini 1993].

2.2 Reliability characteristics of software

From the reliability point of view, software differs from hardware in many ways. For example, software doesn't age in the way a hardware component does. Thus, duplicating a system doesn't lead to increase in reliability unless the duplicate is designed and programmed separately.

Some salient features of software failures are the following (see also [Keene 1994] and [Herrmann 1999]):

- Each software is, at least to an extent, unique. Even minor differences in the program code might mean large differences in the behaviour of the software. Therefore experience with the reliability of other software is of very limited use at best.

- Software faults are caused by hidden design flaws rather than wear-and-tear or physical failure. Therefore software faults are static: they exist from the day the software was written (or revised) until the day they get fixed. They also tend to be unique for each software regarding places of occurrence, mode and severity.

- Software reliability does not depend on time as such. Rather, it depends on the amount and quality of corrections, and on how what kind of input combinations (possibly together with some kind of state such as amount of available memory) the software is subjected to.

- software faults causing a failure are rare. This is so because usually bugs have been caught in the testing phase. A research study [Pham and Zhang, 1999] states a typical commercial software application of 350000 lines of code can contain over 2000 programming errors, that is, an average of six software faults for every 1000 lines of code written. However, as these figures show, commercial software may still contain many errors due to their complexity.

- External environment conditions don't affect software reliability. Internal factors of the software-hardware system, such as amount of memory, clock speed etc. may affect the reliability of software.

- the number of faults in a program tends to decrease with age. Faults are usually corrected after they have been detected and identified. The most frequently occurring faults (in terms of failure frequency) are naturally detected first, and therefore disappear earliest. Also the significance of the fault affects repair time: a more significant fault is prioritized and corrected promptly, whereas an inconsequential bug may be left to stay in the system for the whole of its lifecycle. A change in operating conditions may activate a previously dormant fault. New faults can usually be introduced only through updates of the software; updating the software might happen e.g. to correct detected faults, to accommodate the use of new equipment, etc. Usually the amount of new code is kept to a minimum, the bulk of efforts going to improving existing code. Thus, one would expect at least the expected number of failures to decrease with time

- software faults manifest themselves only under particular conditions. For example, when the fault is in the consequent of an `if` clause, the error can be manifested only when the conditions stated by the clause are true. It seems reasonable to assume that these operational conditions wouldn't occur completely randomly but would be correlated; that is, if an operational condition giving rise to an error had occurred in the recent past, it would be likely that the condition would occur in the near future. Naturally, it is sometimes hard to pinpoint the exact conditions when the software fails

- a single software fault can give rise to several system errors or failures. This may happen until the bug has been identified and corrected. Between the failures the fault induces, it will probably give no sign of its existence

- there is a time lag from the failure to the correction of the underlying fault. This lag is stochastic in nature, and depends on the nature of the fault, the maintainability characteristics of the program, the abilities of program developer(s) tasked with the repair, etc.

- in practice it has been observed [Hamlet 1992] that the mean time to failure (measured in number of runs) of a software system in large systems is inversely proportional to program size; this would indicate that the number of faults per line is roughly constant.

- When the software is deployed (the operational phase), it is usually installed in many places (e.g. devices). Although usually the software itself is identical for each of these, operational conditions (operational profile) differ from place to place. Therefore failure data, if collected, comes from different sources. This also leads to many phenomena that are of interest from the modelling point of

view: for example, when a bug is found and corrected, the reliability of all installations should improve approximately at the same time due to maintenance release.

- Two identical programs behave exactly in the same way. Therefore reliability cannot be increased by redundancy (many clones of the same program doing the same thing) but by diversity (different programs doing the same thing).

2.3 Reliability and the software lifecycle

The methods and needs of software reliability assessment and prediction vary by the phase of software development lifecycle [Asad et al. 2004; Hamlet 1992]:

- at the requirements and design phases, when no implementation is available, early prediction models can be used. Reliability must be analyzed based the architecture and stated requirements.

- at the implementation and testing phases, software reliability assessment is needed to make the stopping decision concerning testing and debugging: when the mean time to failure is long enough, the software can be released. Models most applicable here are *reliability growth models*.

- When the software is released, it is ordinary to assume that all observed faults have been debugged and corrected. Thus, after release, a *reliability model* is used to predict the mean time to failure that can be expected. The resulting reliability estimate may be used in system reliability estimation, as a basis of maintenance recommendations, and further improvement, or a basis of the recommendation to discontinue the use of the software.

Thus, when the software is in operational use, the model to be used depends on maintenance policies and occurrence of failures:

- If no failures are detected in the software, or if the software is not maintained, a reliability model is most appropriate

- If failures are detected and the software is updated, a reliability growth model is in order.

2.4 Sources of software reliability evidence

As mentioned in the introduction, reliability can be defined as the probability, over a given period of time, that the system will correctly deliver services as expected by the user. Thus, the best measure of a program's reliability is its operational record: when and how it failed; this includes also information about the failure's severity, downtimes etc.

Another source of data is the program's test records. These, however, are not as good sources of indicating operational reliability. This is so because in most development projects, new features are introduced to the software almost throughout testing. Therefore the records might imply something about the reliability of a mature feature and something else about the reliability of a less well-developed feature.

Expert judgment about the program, often acquired in software inspections, may give valuable insight. Sometimes, in the lack of reliable operating records, it is the only usable source. Expert opinions, too, have their shortcomings. One is that humans are notoriously bad forecasters [Kahneman et al. 1982], making many kinds of errors in the process.

Also results of static analyses of the program may give valuable information about the program's reliability. For example, software metrics can be used [Hudepohl et al. 1996]. This topic isn't, however, pursued further in this report.

Knowledge of how the software was constructed, such as whether a strict development process was followed or how verification and validation were conducted, may give valuable evidence on the software's reliability. However, it is difficult to well express this evidence in quantitative terms.

System architecture is also a potent source of information for reliability considerations. Almost ubiquitously in modern software systems, at least two components can be distinguished: the platform (operating system and other infrastructure, such as a virtual machine), and the application program itself. This issue is not pursued further in this report, but the reader is referred to [Goševa-Popstojanova et al. 2001; Goševa-Popstojanova et al. 2005; Wang et al. 2006].

The operational profile, if known, can be a valuable input to the reliability estimation process [Haapanen et al. 1997]. If it is known, models such as [Bai 2005] can be used. However, determining an operational profile for a program is usually tedious, and often difficult to do beforehand. For further details on operational profile, see [Musa et al. 1996].

Testing and maintenance policies also affect reliability modelling. However, their impact mainly stems from their possible incompleteness or inadequacy, and would have to be modelled separately.

3. Software reliability growth models

This chapter reviews some existing software reliability growth models. These models describe how observation of failures, and correcting the underlying faults – such as occurs in software development when the software is being tested and debugged – affect the reliability of software. These models are applicable also to assessing the reliability of software in operational use, when the latest reliability estimate given by the model is used.

3.1 General considerations

Before we consider actual models, it is in order to introduce some central concepts. Let $M(t)$ be the (random) number of failures experienced by time *t* (that is, from the start of using the system up to time *t*), and let its expectation be

$$E[M(t)] \stackrel{def}{=} \mu(t) . \tag{3}$$

When modelling software failures, it is natural to assume that $\mu(t)$ is a concave (though nondecreasing) function of time; this assumption is not mandatory, and is broken by e.g. the S-shaped model (see section 3.2.5). We can express the failure intensity function as

$$\lambda(t) = \frac{d\mu(t)}{dt} \tag{4}$$

3.1.1 Reliability model classification

The categorization in this chapter is based on the one presented by [Farr 1996], which in turn owes much to the one presented in [Musa and Okumoto 1983]. However, the present chapter considers also advances made since 1996, and has an emphasis on finding a (Bayesian) likelihood function.

Musa and Okumoto classified models in terms of five attributes:

1. Time domain. The possible values here are ordinary (calendar) time and execution time (that is, the amount of time that the software has been running).

2. Category. The total number of failures that can be experienced in infinite time, which can be either finite or infinite (the possible values).

3. Type. The distribution of the number of failures experienced by time t, such as Poisson, binomial etc.

4. Class. Functional form of the failure intensity expressed in terms of time. The most common distribution here is the exponential distribution, but also Weibull, Pareto and gamma distributions are common. This applies only to models in finite failure category.

5. Family. Functional form of the failure intensity function expressed in terms of the expected number of failures experienced. This applies only to models in infinite failure category.

Especially they classified models along the two dimensions of type and class (in the finite failure case), or type and family (in the infinite failure case).

3.1.2 Frequentist versus Bayesian models for likelihood

Finding the likelihood function of data given a model can be accomplished in two main ways. One is to use a likelihood function provided by an existing (frequentist) model; thus, a frequentist model is converted to a Bayesian one. An advantage of this approach is that the area is well-researched and a multitude of models, together with information on their applicability and accuracy, is available. Problems with this approach are that it might be difficult to construct a likelihood function, and that it might not be straightforward to utilize the experts' judgments in the construction. The other is to use a Bayesian model from the beginning. This approach has the advantage that the likelihood function is automatically a part of the model, and thus obtaining it is a non-issue.

3.1.3 Standard assumptions

There are some assumptions that are usually made, and they apply to all the models in this section unless otherwise stated.

1. The operational profile of the software remains constant. The software where the data comes from is operated in a similar manner as that in which reliability predictions are to be made.

2. Every fault has an equal chance of being encountered within a severity class as any other fault in that class.

3. The failures, when the faults are detected, are independent. This means that failures from fault A don't affect failure times from fault B.

4. After a failure, the fault causing it is corrected immediately and no new faults result from the correction.

Of these, especially the last one has received considerable criticism, and models taking into account delays in error correction and possibility of new errors have been formulated (e.g. [de Bustamante and de Bustamante 2003]).

An important assumption, often made implicitly, is that the software doesn't change during testing and usage, except that faults are fixed.

Furthermore, most models consider the software system as a whole, and individual properties of its constituent modules or components are not considered.

3.2 Nonhomogenous Poisson models

In these models, the number of failures experienced so far follows the nonhomogenous Poisson process (NHPP). The NHPP model class is a close relative of the homogenous Poisson model (see section 4.1); the difference is that here the expected number of failures is allowed to vary with time.

It has been shown [Pham et al. 1999] that a general class of NHPP models can be obtained by solving the differential equation

$$\frac{d\mu(t)}{dt} = b(t)\left[a(t - \mu(t)\right] \tag{5}$$

With suitably chosen $a(t)$ and $b(t)$. For all the models below that can be seen as resulting from equation (5), we list $a(t)$, $b(t)$ and $\mu(t)$.

Say we are interested in the number of failures in the time interval starting at time T and lasting for time t. The nonhomogenous Poisson probability (NHPP) density function is

$$P(n_d(t_b, t_e) = x) = e^{-(\mu(t_e) - \mu(t_b))} \frac{\left[\mu(t_e) - \mu(t_b)\right]^x}{x!} \text{ for } x = 0,1,\ldots \tag{6}$$

Or, expressed for the cumulative number of faults,

$$P(M(t)=x)=e^{-\mu(t)}\frac{[\mu(t)]^x}{x!}\text{ for }x=0,1,\ldots \tag{7}$$

A central issue in utilizing a nonhomogenous Poisson model is how to estimate the expected number of failures. A natural way to do this is to postulate a functional form for $\mu(t)$, possibly with some parameters, and then estimate the parameters from data.

3.2.1 Goel and Okumoto's model

This model [Goel and Okumoto 1979] captures many software reliability issues presented in section 2.2, without being overly complicated. It is similar to the Jelinski and Moranda de-eutrophication model (section 3.3) except that failure rate decreases continuously in time.

The following assumptions are made:

- $M(t)$, the cumulative number of failures, follows a Poisson process. Its mean value function $\mu(t)$ has the following property:

$$\mu(t+\Delta t)-\mu(t)\propto E[\nu(t)], \tag{8}$$

That is, the expected number of failures in a (shortish) time interval after time *t* is proportional to the expected number of undetected faults at time *t*.

- $\mu(t)$ is also assumed to be a bounded, nondecreasing function of time with

$$\lim_{t\to\infty}\mu(t)=N<\infty \tag{9}$$

- The number of faults $(n_1,n_2,...,n_k)$ detected in each of the respective time intervals $[(t_0=0,t_1),(t_1,t_2),...,(t_{i-1},t_i),...,(t_{k-1},t_k)]$ is independent for any collection of times $t_1<t_2<\ldots<t_k$.

The data required by Goel and Okumoto's model consists of

1. $(n_1,n_2,\ldots,n_k)$, the number of faults detected for each time interval
2. $(t_1,t_2,\ldots,t_k)$, the completion times of the time intervals for which the $(n_1,n_2,\ldots,n_k)$ are observed. The time intervals are assumed to be adjacent.

Goel and Okumoto show that under the assumptions above, the cumulative failure function must be of the form

$$\mu(t) = \overline{N}(1 - e^{-bt}) \tag{10}$$

In terms of equation 5, the parameters are $a(t) = \overline{N}$, $b(t) = b$.

3.2.2 Musa-Okumoto model

This model [Musa and Okumoto 1984] is similar to the Goel-Okumoto model (section 3.2.1), except that it attempts to consider that later fixes have a smaller effect on a program's reliability than earlier. In particular, is intensity function decreases exponentially as failures occur. The model is also called Musa-Okumoto logarithmic Poisson model because the expected number of failures over time is a logarithmic function. Thus the model is an infinite failure model.

The basic assumption of the model, beyond the assumption that the cumulative number of failures follows a Poisson process, is that failure intensity decreases exponentially with the expected number of failures experienced:

$$\lambda(t) = \lambda_0 e^{-\theta\mu(t)} \tag{11}$$

As data, the model requires either the actual times that the software failed, of the elapsed times between failures.

3.2.3 Schneidewind's model

Make the following assumptions [Schneidewind 1975]:

- The data used are the number of failures per time interval where all time periods are of equal length.
- The cumulative number of failures follows a NHPP with mean value $\mu(t)$.
- The failure intensity function is an exponentially decreasing function of time: $\lambda(t) = \alpha_0 e^{-\alpha_1 t}$.
- The number of faults n_i detected in each of the respective intervals are independent.

- The fault correction rate is proportional to the number of faults to be corrected (which is usually the number of faults detected so far).

Schneidewind proposes models for data in three forms:

1. Utilize all fault counts from the *I* time intervals in the observation range.
2. Ignore the fault counts completely from the first through the *s*-1 time periods, i.e., only use the data from period *s* through *I*.
3. Use the cumulative fault counts for the time intervals 1 to *s*-1 as the first data point, and the individual fault counts for periods *s* through *I* as the additional time points.

Schneidewind has developed criteria for the optimal selection of the parameter *s*. He uses different kinds of mean square criteria as objective functions. The actual optimization happens simply by computing the parameter estimates under different *s*, and from these, the predictions and finally, the value of objective function; the value of *s* is chosen that gives the smallest value for the objective function.

The functional form of the mean failure function $\mu(t)$ is

$$\mu(t) = \frac{\alpha_0}{\alpha_1}(1 - e^{-\alpha_1 t}), \tag{12}$$

where $\alpha_0, \alpha_1 > 0$. Note that the equation above also gives an estimate to the total number of faults: it is α_0 / α_1. The parameters can be estimated by maximum likelihood; the form of the estimates varies by the kind of data available/used (see above). The parameters of equation 5 are $a(t) = \frac{\alpha_0}{\alpha_1}$, $b(t) = \alpha_1$.

From the model, it is easy to compute different kinds of measures for software reliability, such as the expected number of faults in the ith period.

3.2.4 Musa's basic execution time model

This model is perhaps the most popular of the software reliability models [Farr 1996]. The times between failures are expressed in terms of computational processing units (CPU) rather than the amount of calendar time that has elapsed (the model contains a feature for converting from processing time to calendar time).

Musa himself [Musa et al. 1987] recommends the use of this model (as contrasted to Musa's logarithmic Poisson model) when the following conditions are met:

- Early reliability is predicted before program execution is initiated and failure data observed
- The program is substantially changing over time as the failure data are observed
- If one is interested in seeing the impact of a new software engineering technology on the development process.

The following assumptions are made [Farr 1996]:

- $M(t)$, the cumulative number of failures, follows a Poisson process. Its mean value function $\mu(t) = \alpha_0(1 - e^{-\alpha_1 t})$, where $\alpha_0, \alpha_1 > 0$. This mean value function is such that the expected number of failures is proportional to the number expected number of undetected faults at that time.
- The execution times between the failures are piecewise exponentially distributed, i.e. the hazard rate for a single fault is constant.
- Data: either the actual times that the software failed, T_1, T_2 , …, T_k or the elapsed time between failures x_1, x_2 , …, x_k

If conversion from execution time to calendar time is needed, then also the following four assumptions are made:

- The quantities of resources (number of testers, software maintenance personnel and computer resources) that are available are constant over a time segment for which the software is observed.
- Resource expenditures for the kth resource, $\Delta\chi_k$, associated with a change in MTTF from T_1 to T_2 can be approximated by $\Delta\chi_k \approx \theta_1\Delta t + \theta_2\Delta m$, where Δt is the increment of execution time, Δm is the increment of failures experienced, and θ_1 and θ_2 are parameters.
- Fault-identification personnel (testers etc.) can be fully utilized and computer utilization is constant.
- Fault-correction personnel utilization is established by the limitation of fault ueue length for any fault-correction person. Fault queue is determined by

assuming that fault correction is a Poisson process and that servers are randomly assigned in time.

For converting from execution time to calendar time, also the following data are needed:

- The available resources for both identification and correction personnel and the number of computer shifts.
- The utilization factor for each resource
- The parameters θ_1 and θ_2 needed in computing the resource expenditures (see above)
- The maximum fault queue length for a fault correction personnel
- The probability that the fault queue length doesn't exceed the maximum.

It can be shown that after (i.1) failures have occurred, the reliability function is

$$R(\Delta t \mid T_{i-1}) = e^{-\alpha_0 e^{-\alpha_1 T_{i-1}}(1-e^{-\alpha_1 \Delta t})} \tag{13}$$

and the conditional hazard rate is

$$z(\Delta t \mid T_{i-1}) = \alpha_0 \alpha_1 e^{-\alpha_0 T_{i-1})} \tag{14}$$

The conversion from execution time to calendar time is explained in [Musa et al. 1987].

Maximum likelihood estimation of the parameters is explained in [Farr 1996].

3.2.5 S-shaped model

The model was proposed by Yamada, Ohba and Osaki [1983]. It is a descendant of the Goel and Okumoto model (see section 3.2.1), the data requirements being similar and the assumptions being similar with one exception. Yamada et al. reasoned that due to learning and skill improvements of the programmers during the debugging phase of the development cycle, the error detection curve is often not exponential but rather S-shaped. Furthermore, the per-fault failure distribution is gamma distribution. Based on this, they proposed the mean value function

$$\mu(t) = N\left[1-(1+\beta t)e^{-\beta t}\right], \tag{15}$$

where $N, \beta > 0$. The parameter N can be interpreted as the total number of errors and β as the failure detection rate.

3.2.6 Inflection S-shaped growth curve model

The assumptions behind this model [Ohba 1984] are that

- the faults in a program are mutually independent
- the probability of failure detection at any time is proportional to the current number of detectable faults in the program
- this proportionality is constant
- the isolated faults can be entirely removed.

The mean value function is

$$\mu(t) = N\frac{1-e^{-\beta t}}{1+\psi e^{-\beta t}}, \tag{16}$$

Where N can be interpreted as the total number of errors and β as the failure detection rate (as in the Jelinski-Moranda model); ψ is an inflection parameter:

$$\psi = \frac{1-r}{r}, r > 0 \tag{17}$$

Where r is the inflection rate which indicates the ratio of the number of detectable faults to the total number of faults in the program (some faults are not detectable until some other faults are removed).

3.2.7 K-stage Erlangian (Gamma) growth curve model

This model class was proposed by Khoshgoftaar [1988]. The mean value function of this model is

$$\mu(t) = \alpha\left(1 - e^{-\beta t}\sum_{j=o}^{K-1}\frac{(bt)^j}{j!}\right). \tag{18}$$

The model is a generalization of the Goel and Okumoto model (section 3.2.1, set $K = 1$) and the S-shaped model (section 3.2.5, set $K = 2$). The parameters can be given the same interpretation as in the S-shaped model.

3.2.8 Duane model

In this model, the rate of occurrence of failures is in power law form in operating time:

$$\mu(t) = \alpha t^{\beta} \tag{19}$$

The model is an infinite failure model. No physical interpretation can be attached to the parameters α and β. The power law form was introduced by Duane [1964], and Crow [1974] added the assumption that the underlying failure process is NHPP.

As is easily seen from (19), this is an infinite failure model.

3.2.9 Pham-Nordmann-Zhang model

This model [Pham et al. 1999] integrates imperfect debugging with the learning phenomenon. Learning occurs if testing appears to improve dynamically in efficiency (and thus fault-detection rate improves) as testing progresses. This of course doesn't necessarily happen – for example, non-operational profiles used to generate test and business models can prevent it. However, changes in fault-detection rate are common during the testing process. Pham et al. claim also that in most realistic situations, fault repair is associated with a fault re-introduction rate due to imperfect debugging.

Their model is easiest to state in terms of equation 5: here, $a(t) = \overline{N}(1 + \alpha t)$, $b(t) = b \big/ 1 + \beta e^{-bt}$ and

$$\mu(t) = \frac{\overline{N}}{1 + \beta e^{-bt}}\left[(1 - e^{-bt})(1 - \frac{\alpha}{b}) + \alpha t\right] \tag{20}$$

Pham et al. compare their model with other NHPP models, and conclude that the inclusion of imperfect debugging and learning, as in their model, improves both the descriptive and the predictive properties of the model, and is worth the increased model complexity and number of parameters.

3.3 Other frequentist models

The functional form of all of the failure intensity function in all these models is exponential.

3.3.1 Jelinski-Moranda de-eutrophication model

The Jelinski-Moranda model is one of the earliest in the field [Jelinski and Moranda 1972]. It assumes that failures occur purely at random, and that all faults contribute equally to unreliability. Due to the latter, and the assumption that no fixes produce new failures, follows that the program's failure rate improves by the same amount by each fix.

The following assumptions are made:

- The initial number of faults in the software is *N*. No new faults are introduced to the software (e.g. through bug correction or other maintenance).
- The elapsed time between failures follows an exponential distribution with a parameter that is proportional to the number of remaining faults in the software. Let *t* be any time instance between the occurrence of the (*i*-1)st and *i*th failure occurrence. Then the mean time between failures at time *t* is $1/\phi(N-(i-1))$
- The rate of fault detection is proportional to the current fault content of the software.
- The fault detection rate remains constant over the intervals between fault occurrences.
- A fault is corrected instantaneously after first detecting it, without introducing new faults into the software.

To estimate the parameters ϕ and θ, either of the following kinds of data is needed:

- The elapsed time between failures $\tau_1, \tau_2, \ldots, \tau_m$
- The actual times of failure from the start of the utilization of the system, $T_1, T_2, \ldots, T_m$

Here naturally $\tau_i = T_i - T_{i-1}$ (with this notation, $T_0 = 0$).

The density function of the time between failures, given the time of latest failure, is

$$f(X_i \mid T_{i-1}) = \phi[N-(i-1)]e^{-\phi[N-(i-1)]X_i} \quad (21)$$

The number of failures experienced is binomially distributed.

The model makes the assumption that the rate of fault detection is proportional to the current fault content. This is unrealistic because not all faults are equal. The most frequently occurring faults are detected first.

3.3.2 Geometric model

The geometric model [Moranda 1979] is a variant of the Jelinski-Moranda model (section 3.3.1). The time between failures follows an exponential distribution. The fault detection rate follows a geometric progression and is constant between fault detections:

$$z(t) = D\phi^{i-1}, \quad (22)$$

Where $0 < \phi < 1$ and $t_{i-1} < t < t_i$ (t_i is the time of the ith failure).

The expected time between failures is

$$EX_i = \frac{1}{D\phi^{i-1}} \quad (23)$$

The cumulative failure function is

$$\mu(t) = \frac{1}{\beta}\ln(D\beta e^{\beta}t + 1) \quad (24)$$

As is readily seen from equation 24, the model is an infinite failure model.

3.3.3 Hyperexponential model

The basic idea in this class of models is [Ohba 1984] that the different sections (or classes) of the software experience an exponential failure rate; however, the rates vary over these sections to reflect their different natures. These different rates reflect e.g. work done by different development groups; code that has been done long time ago vs. recently implemented code; code implemented in different programming languages;

code that has been subjected to formal specification and verification vs. code that hasn't been; etc.

[Laprie et al. 1991] developed a variation of this class of models when the number of subsystems is 2. They considered in this way the hardware and the software component of the system.

The following assumptions are made:

- The software system consists of K sections or subsystems
- Each subsystem exhibits an exponential failure rate; denote the failure rate of the kth subsystem by β_k
- $\forall k = 1, \ldots, K : 0 < \beta_k < 1$
- The rate of fault detection in a subsystem is proportional to the current fault content within that subsystem
- The fault detection rate remains constant over the intervals between fault occurrence
- A fault is corrected instantaneously without introducing new faults into the software
- The fault rates of the subsystems are independent of each other
- The cumulative number of failures by time t, M(t), follows a Poisson process with mean value function $\mu(t) = N \sum_{i=1}^{K} p_i (1 - e^{-b_i t})$, where $\sum_{i=1}^{K} p_i = 1$, $\forall i = 1, \ldots, K : 0 < p_i < 1$, and the total expected number of faults, N, is finite (it might not be an integer).

The following data are assumed:

- The fault counts in each of the testing intervals, i.e., the n_i
- The completion time of each period that thesoftware is under observation, i.e. the t_i's

Note that if $K = 1$, we have Goel and Okumoto's model (see section 3.2.1).

The failure intensity function of the model is

$$\lambda(t) = N\sum_{i=1}^{K} p_i \beta_i e^{-\beta_i t} \tag{25}$$

This failure intensity function is strictly decreasing for $t > 0$. For parameter estimation, see [Farr 1996, section 3.3.5.4].

3.3.4 Weibull model

The per-fault failure distribution is a Weibull distribution. The Weibull distribution is perhaps the most popular distribution used for component failure in reliability engineering. The distribution of the number of the failures experienced by time t follows a binomial distribution (models described in section 3.2 followed a Poisson distribution).

The following assumptions are made:

- There is an infinite number of faults, *N*, at the beginning of the period in which the software is being observed.
- The time to failure of fault *a*, denoted as T_a, is distributed as a Weibull distribution: the density function with parameters $\alpha, \beta > 0$ is

$$f_a(t) = \alpha\beta t^a e^{-\beta t^a} \tag{26}$$

- The number of faults $n(t_i, t_{i+1})$ detected in each time interval $[t_i, t_{i+1}]$ are independent for any collection of times.
- Fault counts for each of the testing intervals are available.
- The start and completion times of the time intervals are known.

The failure intensity function is

$$\lambda(t) = Nf_a(t) = N\alpha\beta t^a e^{-\beta t^a} \tag{27}$$

And the cumulative failure function is

$$\mu(t) = NF_a(t) = N(1 - e^{-\beta t^a}) \tag{28}$$

3.3.5 Barghout-Littlewood-Abdel-Galy model

Barghout, Littlewood and Abdel-Galy [1997] propose a nonparametric model that is constructed in two stages:

1. attempt to fit a trend in the interfailure time data. The goal is to find a function g_i of the number of failures experienced, such that the sequence $Z_i = T_i / g_i$ transformed from interfailure times T_i is approximately trend-free. The idea is that when Z_i are approximately identically distributed, kernel estimation methods can be used. For example, a possible form of g_i is $g_i = \alpha_1 + \alpha_2 i$.

2. use a kernel estimator to estimate the distributions of the interfailure times. For any general kernel $K(z)$ define $K_h(z) = \frac{1}{h} K(\frac{z}{h})$, where h is a scaling parameter. A kernel function $K_h(z - z_j)$ is centred around each observation z_i. The kernel density estimator (the estimated probability density) is obtained by averaging these kernel functions: $\hat{f}(z) = \sum_{i=1}^{n} K_h(z - z_j)$. A density function for the interfailure time is obtained by making the reverse transform $T_i = Z_i g_i$.

Barghout et al. tried Gaussian, double exponential and log-normal kernels, of which the log-normal kernel showed the best results with the data sets used.

This model has several good properties. As a nonparametric model, no specific distribution has to be assumed; this makes the model adaptable, because software development projects vary widely and often there is no good reason for adopting a given probability distribution for the interfailure time.

3.4 Bayesian models

Bayesian models are based on Bayesian statistics, where relevant parameters have a prior distribution which is then updated by evidence through the likelihood function.

Bayesian models have several desirable characteristics for software reliability assessment:

- It is rather easy to incorporate evidence from many sources, e.g. experts, tests, operational data etc.

- Bayesian models work also when there are no positive instances (e.g. when no failures have been observed).

The traditional way of doing statistics, which does not use the Bayesian idea of priors, is often called frequentist.

3.4.1 Bayesian versions of frequentist reliability models

In principle, most statistical models can be made Bayesian by assuming some prior distribution(s) to its parameters and then finding a likelihood function implied by the model to yield a posterior probability distribution to the variables of interest. This is a popular approach in the software reliability modelling literature.

There have been many reformulations of the Jelinski-Moranda deeutrophication model (section 3.3.1) in Bayesian terms, e.g. [Langberg and Singpurwalla 1985], who assume that the parameters of the Jelinski-Moranda model are themselves random variables, or Csenki [1990], who assumes the number if initial faults to be s-independent and Poisson-distributed, and the per-fault failure rate s-independent and gamma-distributed.

Becker and Camarinopoulos [1990] propose a Bayesian model that allows the possibility that, after some debugging, the program contains no errors. Their failure intensity function is exponential. The idea is that, corresponding to each failure and the respective correction, an update of the failure rate takes place. This is facilitated by having conjugate priors for the failure rate. The resulting model includes an estimator for the probability that a program still contains errors, which is an upper bound for the failure probability.

3.4.2 Littlewood-Verrall model

This model [Littlewood and Verrall 1973] is perhaps the best-known Bayesian software reliability model. The distribution of failure times is assumed to be exponential, with the failure rate distributed as a gamma distribution in the prior.

The following assumptions are made:

- Successive execution times between failures, X_i's, are assumed to be independent exponential random variables with parameters θ_i, $i = 1,\ldots,n$.

- The θ_i's form a sequence of independent random variables, each with a gamma distribution with parameters σ and $\psi(i)$. $\psi(i)$ is an increasing function of *i*, and describes the quality of the programmer and the difficulty of the task; for a skilled programmer, $\psi(i)$ increases faster.

- The software is operated in a manner similar to the anticipated operational usage.

The marginal distributions for the times between failures, x_i's, are Pareto distributions

$$f(x_i \mid \sigma, \psi(i)) = \frac{\sigma(\psi(i))^{\sigma}}{[x_i + \psi(i)]^{\sigma+1}} \tag{29}$$

Their joint density is thus the product

$$f(x_1, x_2, \ldots, x_n) = \frac{\sigma^n \prod_{i=1}^{n} (\psi(i))^{\sigma}}{\prod_{i=1}^{n} [x_i + \psi(i)]^{\sigma+1}} \tag{30}$$

Littlewood and Verrall also derive a posterior distribution for the parameters θ_i. They suggest a linear and a quadratic form for $\psi(i)$: $\psi(i) = \beta_0 + \beta_1 i$ or $\psi(i) = \beta_0 + \beta_1 i^2$. The failure intensity functions for these forms are, respectively,

$$\lambda_{linear}(t) = \frac{\sigma - 1}{\sqrt{\beta_0^2 - 2\beta_1 t(\sigma - 1)}} \tag{31}$$

and

$$\lambda_{quadratic}(t) = \frac{\nu_1}{\sqrt{t^2 + \nu_2}} \left((t + (t^2 + \nu_2)^{1/2})^{1/3} - (t - (t^2 + \nu_2)^{1/2})^{1/3} \right) \tag{32}$$

Where $\nu_1 = (\sigma - 1)^{1/3} / (18\beta_1)^{1/3}$ and $\nu_2 = 4\beta_0^3 / (9(\sigma - 1)^2 \beta_1)$.

3.4.3 Other Bayesian models

Basu and Ebrahimi [2003] propose a model with exponentially distributed interfailure times, driven by a piecewise constant failure rate. The failure rate changes at each failure (reflecting the assumption that the software is then debugged and revised). A Markovian martingale process prior is assumed on the failure rate. This martingale process is driven by hyperparameters, for which prior distributions are specified. The posterior and predictive quantities of interest are estimated with Markov chain Monte Carlo (MCMC) sampling.

A problem with this model is that the failure rate is martingale, i.e. no improvement is assumed after a failure (and correction). A more realistic assumption would be a tendency towards improvement at each failure.

Cid and Achcar [1999] propose a non-homogenous Poisson process model, which is based on the exponentiated-Weibull form intensity function

$$\lambda(t) = \frac{\alpha\theta\left[1 - e^{-(t/\sigma)^{\alpha}}\right]^{\theta-1} e^{-(t/\sigma)^{\alpha}} \left(t/\sigma\right)^{\alpha-1}}{\sigma\left(1 - \left[1 - e^{-(t/\sigma)^{\alpha}}\right]^{\theta}\right)} \tag{33}$$

This intensity function is very flexible, and with different choices for the parameters α, θ and σ will give failure intensity functions with different behaviors, among them nonmonotonic ones. Nonmonotonicity might arise in the course of software development due to faults introduced by software updates, and in the course of use due to changing operating conditions. However, the nonmonotonic forms in this paper are a bathtub form and an unimodal form. These don't seem to reflect any behaviour that a software might possess.

Zegueira [2000] proposes a model with failure intensity functions λ_i that are constant between corrections, and decreasing from correction to correction. He asserts a joint distribution for each two consecutive failure rates, with a gamma prior distribution. The probability density of the next failure rate can be derived from the one for the previous failure rate, the joint density, and the observation of failure time, and thus can be applied sequentially. The prior likelihood function for the length of the first interval (the time to the first failure) has an exponential prior distribution, and the predictive density for length of the ith interval (the time from $i-1$ th failure to the ith one) is calculated as

$$f(t_i \mid n_1,\ldots,n_{i-1},t_1,\ldots,t_{i-1}) = \frac{\alpha\left(t_{i-1} + \beta_i\right)^{\alpha_i}}{\left(t_i + t_{i-1} + \beta_i\right)^{\alpha_i+2}} \tag{34}$$

3.5 Special-purpose models

Several models have been developed to handle some specific aspect or situation in the software lifecycle.

Sometimes running a software is stopped even though the software has not failed. Such non-failure stops can occur e.g. in systems with periodic tasks, which are based on handling of interrupts. Non-failure-stops can be viewed as a type of censored data. [Cai

1997] develops censored forms of several software reliability models, such as Jelinski-Moranda (section 3.3.1) and Littlewood-Verrall (3.4.2) models.

In finite-failure models, it would greatly help if the number of remaining faults could be estimated. [Campodónico and Singpurwalla 1994] develop a Bayesian model, based on a non-homogenous Poisson process (the logarithmic-Poisson model); their approach can be used with other software reliability models, too. [Cai 1998] develops a static frequentist model for this purpose, and also a Bayesian version of it.

Most of the existing software reliability growth models deal with time as a continuous variable (either in calendar time, clock time or in execution time). However, there are systems, such as bank transaction processing systems, where reliability should be measured in terms of transactions successfully handled. Furthermore, there are systems, such as rocket control software, where it is more natural to measure reliability in how many rockets can be successfully launched. Such systems require a discrete conception of time, in terms of number of runs of the software. [Cai 2000] develops a conceptual framework for modelling these kinds of situations, and proposes three methodologies, based on probabilistic, Bayesian and fuzzy notions, respectively.

Code coverage (the proportion of code that some test actually reaches) in testing may affect reliability estimates significantly: testing may reach saturation where no new parts of code are actually tested. Then, reliability estimates relying entirely on testing/execution time may over-estimate the program's reliability. [Chen et al. 2001] propose a scheme where test coverage information is collected, and execution time between test cases, which neither increases code coverage nor causes a failure, is reduced by a parameterized amount. They show that overestimation of reliability is corrected in two case studies.

4. Software reliability models

These models are meant to predict software reliability when the software is in operational use.

4.1 Homogenous Poisson model

When using a Poisson process to describe the number of failures in a time period, the following assumptions are made:

- The time intervals between two consecutive failures are independently and exponentially distributed.
- The expected value of the number of failures in a time interval depends only on the length of the interval and not its start point.

A simple Poisson distribution can be used as the likelihood function $p(data \mid t_b, t_e, m_f)$:

$$p(t_{d,b}, t_{d,e}, m_d \mid t_b, t_e, m_f) \sim Poisson(\lambda, \Delta t_d) \tag{35}$$

That is,

$$p(T_d, t_d, m_d \mid T_f, t_f, m_f) = e^{-\lambda \Delta t_d} \frac{(\lambda \Delta t_d)^{n_d}}{m_d!} \tag{36}$$

Here m_d failures occur in the data in the time period that lasts for $\Delta t_d = t_{d,e} - t_{d,b}$. Note that the probability distribution doesn't directly depend on $t_{d,b}$ or $t_{d,e}$; in other words, the model is homogenous with respect to time. Note also that the impact of the predicted failure distribution (given by the model) is wholly absorbed in the parameter λ. This is the model used in [Helminen, 2005].

The model has several drawbacks:

- it is unrealistic to suppose that the likelihood would be homogenous wrt. time, because the number of software faults causing the failures decreases with time, with the faults most commonly causing failures being corrected first (see section 2.2). Of course, if the software exhibits no failures, this point is moot.

- it is also unrealistic to assume that the failures would occur homogenously wrt. time in the sense that the time between failures would be independently exponentially distributed. On the contrary, it is presumable that the failures would occur in clusters, when the operational conditions that cause an error to activate occur.

4.2 Other frequentist models

Baker [1988] considers the situation where many copies of the program are run at many different user sites with the support of a software service organization. The organization provides both preventive service (fixes to known faults are provided to all user sites) and corrective service (for users that encounter faults). The number of users (installations) may vary with time. His special focus is the effect of service organization on software reliability.

A fixed number of bugs is assumed. Separate models for first discovery time (the time it takes to come across the bug at least at one site) and total discovery time (discovery of a particular fault at many user sites). He proposes an exponential distribution for the first discovery time, and obtains a formula for $n(t)$, the cumulative number of faults found so far.

4.3 Bayesian reliability models

Cukic and Chakravarthy [2000] propose a Bayesian framework for reliability assessment. They formulate a model directly for reliability, that is, the probability that the software will fail in the specified time. The priors are assumed to follow a beta distribution. The priors reflect the application of verification and validation activities.

The Bayesian approach has many good features. The most important of these, in this case, is that it allows incorporation of program executions observed in the operational environment, even when they are failure-free (that is, no failure observations are available).

5. Application to the operational data problem

Our problem, as stated in section 1, is to update the failure probabilities by operational data records.

A major problem with all models is that techniques like the ones described in section 3 can only support relatively modest reliability claims [Littlewood 2005]: to state that mean time to failure is x hours might require a total time on test of tens or hundreds times x hours. Littlewood claims that these limitations of the models are inherent, and will not be eliminated by newer and cleverer modelling.

In an earlier paper, Littlewood and Strigini [1993] show that when the time to next failure is exponentially distributed, and software has been tested without failures for time t_0, the reliability function is

$$R(t \mid \text{no failures in time } t_0) = \frac{t_0}{t + t_0} \tag{37}$$

That is, if we want a 50 % certainty that the program doesn't fail in t hours, we have to test it for t hours. There is no solution to this problem in sight, because the problem is inherent in the probabilistic formulation.

5.1 Requirements on the data

To properly analyze a data sample for reliability purposes, the data must meet some prerequisites.

There are two forms of data that are suitable for statistical analysis of reliability:

- Event data, or failure reports. This data should contain date and time information, information about the kind and severity, and downtime of each failure.

- Time series data, or number of failures per time interval. There should be information about the time periods that these cover, and the time intervals should preferably be of uniform length. There might be several data series, for example one for each kind of failure or one for each failure severity.

The data should cover a time period as completely as possible. For example, if the data consists of failure reports from March 2003 to September 2005, all the failure reports from this period should be included.

The metadata associated with the data should contain information about the units used, what equipment the failure data covers, and explanations about the kinds of failure and their severity.

5.2 Evaluation and comparison criteria for software reliability models

There are several ways in which a model's goodness can be evaluated [Iannino et al. 1983]:

- Predictive validity. This is the capability of the model to predict future failure behaviour during either the test or the operational phases from present and past failure behaviour in the respective phase. This can be further divided [Lyu and Nikora 1992] to
    - Accuracy, as measured by prequential likelihood (section 5.2.5)
    - Bias, as measured by the U-plot (section 5.2.1)
    - Trend, or systematic change of bias from small to large values of failure time, as measured by the Y-plot (section 5.2.2)
    - Noise, as measured by the relative change in the predicted failure rate.
- Capability. The ability of the model to estimate with satisfactory accuracy quantities needed by software managers, engineers, and users in planning and managing software development projects or controlling change in operational software systems. These quantities include, e.g., present reliability, expected date of reaching a reliability objective, and cost required to reach that objective.
- *Quality of assumptions*. If an assumption made by a model can be tested, the degree to which it is supported by actual data; if it is not possible to test an assumption, its plausibility from the viewpoint of logical consistency and software engineering experience. Also the clarity and explicitness of an assumption should be judged.

- *Applicability* means the usefulness of the model across different software products (size, structure, function), different development environments, different operational environments, and different life cycle phases.

- *Simplicity*. The simplicity and inexpensiveness of collecting the data that is required to particularize the model. Conceptual simplicity in that the expected audience of the model (software engineers, project managers, reliability specialists, officials) can understand the nature of the model and its assumptions, so that they can determine its applicability to the particular problem and the extent to which the model may diverge from reality in the intended application. Simplicity of implementation so that it may become a practical management and engineering tool.

- *Ease of measuring parameters* [Lyu and Nikora 1992]. This concerns the number of parameters a model requires and the difficulty in estimating them.

- *Insensitivity to noise* [Lyu and Nikora 1992]. The ability of a model to make accurate predictions even when failure data is incomplete or contains uncertainties.

The techniques for model evaluation reviewed in this section, such as the U-plot, Bayes factor, and prequential likelihood ratio, work only when there are (positive) instances of the event. In the case of software faults, this means that at least some software failures should have occurred and should have been observed. In the case of reliable programmable devices in operational use, this is often an unrealistic assumption. Thus, the these methods are better suited to assessing software reliability growth models that address mainly the development phase of the software lifecycle.

There are other methods of model validation and comparison, such as cross-validated likelihood [Basu and Ebrahimi 1998], which will not be discussed in this report.

5.2.1 U-plot

The U-plot (see, e.g., [Brocklehurst and Littlewood 1992]) is used to determine if the postulated cumulative distribution function, $\hat{F}(t)$, is close to the true distribution $F(t)$ (provided by observations). It is known that the random variable $U = F(t)$ has a uniform distribution over the interval [0,1]. Thus, if the realizations t_i (e.g. failure times) are observed and $u_i = \hat{F}(t_i)$ are calculated, u_i should be a realization of a uniform random variable. Any departure from uniformity indicates deviation of $\hat{F}(t)$ from $F(t)$.

To find the departures (if any) the sample distribution function of the transformed observations y_i is plotted. The plot is a step function, consisting of the numbers $u_1,...,u_M$ on the interval [0,1]. Then, plot an increasing step function, each step of height $1/(M+1)$ is plotted at each u_i on the abscissa.

The closer this plot is to the line of unit slope, the closer $\hat{F}(t)$ is to $F(t)$. On the other hand, any systematic departure from unit slope indicates a misspecification of the probability distribution (that is, a reasonably consistent bias).

This can be developed into an operational measure by finding the Kolmogorov distance (maximum absolute vertical deviation) between the perfect prediction line of slope 1 and the actual plot [Lyu and Nikora 1992].

5.2.2 Y-plot

The Y-plot measures the consistency of a model's bias; a model might be initially too pessimistic and eventually too optimistic concerning the number of faults in the software, for example.

This is the result of the sequence of transformations of $u_i = \hat{F}(t_i)$ as defined in the previous section as follows:

$$x_i = -\ln(1 - u_i) \tag{38}$$

$$y_i = \frac{\sum_{j=1}^{i} x_j}{\sum_{j=1}^{M} x_j} \tag{39}$$

Where $i \leq M$.

This can be made into an operational measure by finding the Kolmogorov distance $\max_i(|x_i - y_i|)$ between the variables defined above.

5.2.3 Bayes factor

The Bayes factor [Gelman et al. 1995] is the formal Bayesian model comparison criterion. Let two competing models be H_1 and H_2, respectively. The Bayes factor is the ratio of marginal likelihoods of the two models under comparison:

$$BF(H_1, H_2) = \frac{p(t_1, \ldots, t_m \mid H_1)}{p(t_1, \ldots, t_m \mid H_2)} \quad (40)$$

Here $p(t_1, \ldots, t_m \mid H_i) = \int p(t_1, \ldots, t_m \mid \theta_i, H_i) p(\theta_i \mid H_i) d\theta_i$.

Computing Bayes factor can be computationally demanding. For a review of estimating the marginal likelihoods based on draws from the posterior distributions, see [DiCiccio et al. 1997].

5.2.4 Prequential predictive ordinate

The prequential predictive distribution (PPO) $p(t_j \mid t_1, ..., t_{j-1}, H_i)$ is the distribution of failure time T_j conditional on only those failure times observed before the *i*th failure, and the assumed model [Basu and Ebrahimi 2003]. A higher value of PPO indicates that the observed value of T_j is more likely under the model (compared with some other model), and is preferred.

5.2.5 Prequential likelihood and prequential likelihood ratio

Prequential likelihood measures the accuracy of a model [Lyu and Nikora 1992]. Let the probability density function given by model A to the data be $f_A(t)$. Furthermore, let $t_1, ..., t_m$ be observed occurrences of failures (or whatever phenomenon we are trying to model). The prequential likelihood of a model is

$$PL_A = \prod_{j=1}^{M} f_A(t_j) \quad (41)$$

This product is usually quite close to zero, and a more perspicuous measure is obtained by taking the logarithm of the prequential likelihood.

The prequential likelihood ratio [Brocklehurst and Littlewood 1992] compares two models' abilities to predict a particular set of data. Let the probability density functions, given by models *A* and *B* to the data, be $f_A(t)$ and $f_B(t)$, respectively. The prequential likelihood ratio PLR_i^{AB} is defined as

$$PLR_i^{AB} = \prod_{j=1}^{i} \frac{f_A(t_j)}{f_B(t_j)} \quad (42)$$

This ratio should increase with increasing number of observations if model A is superior to model B, and decrease otherwise.

It is easy to see that prequential likelihood ratio is a simplified version of the Bayes factor (section 5.2.2): if the observations are independent and if the probability space is discrete, they coinside. In most cases, however, prequential likelihood ratio is easier to compute.

5.2.6 Akaike information criterion

The Akaike information criterion (AIC) can be expressed in the following manner [Khoshgoftaar and Woodcock 1991]:

$$\begin{aligned} &-2(\text{log likelihood function at maximum likelihood estimators}) \\ &+2(\text{number of parameters fitted when maximizing the likelihood function}) \end{aligned} \tag{43}$$

5.3 Some comparisons

In this section, some representative comparisons of software reliability models are reviewed. Other comparisons, not treated in this section, include [Schick and Wolverton, 1978], [Selby, 1990], [Pham et al. 1999], and [Pham 2003].

It would be most interesting to carry out a meta-analysis of the results of the individual studies. This, however, is beyond the scope and the resources of the present study.

[Brocklehurst and Littlewood 1992] was carried out on the CSR1 data set, collected from a single-user workstation at the Centre for Software Reliability. It represents some 397 user-perceived failures such as genuine software failures, usability problems, inadequate documentation etc.

The models they considered were Jelinski-Moranda, Goel-Okumoto, Musa-Okumoto, Duane, Littlewood (not described in this report), Littlewood nonhomogenous Poisson process (not described in this report), Littlewood-Verrall, and Keiller-Littlewood (no described in this paper).

The criteria they used were the U-plot (section 5.2.1) and prequential likelihood ratio (section 5.2.4). In this comparison, Littlewood-Verrall and Littlewood-Keiller models fared best.

[Basu and Ebrahimi 2003] was carried out on the Naval Tactical Data System (NTDS) dataset, originally introduced in Jelinski and Moranda [1972] and since then widely used in the literature. Basu and Ebrahimi used the interfailure data from production phase, which consists of 26 observations.

The models they considered were Jelinski-Moranda (section 3.3.1), Goel-Okumoto (3.2.1), Littlewood-Verrall (3.4.2), Homogenous Poisson Process (4.1), Musa-Okumoto (3.2.2), Weibull order statistic (3.3.4), Singpurwalla and Soyer (not in this report), and three models of their own, which they named single-alpha, exchangeable alpha and equal variance. They used log-marginal likelihoods of the models as the goodness criterion. Their single-alpha and exchangeable-alpha models fared best, with Jelinski-Moranda and Goel-Okumoto models being rather close.

[Khoshgoftaar and Woodcock 1991] utilized a data set from an IBM computer system project. The project involved more than 50000 lines of code, mostly in assembler with a small amount of C code. They considered the following models: Goel-Okumoto (section 3.2.1), S-shaped (3.2.5), K-stage Erlangian, K = 3 or K = 4 (3.2.7), and the Duane model (3.2.8).

The S-shaped model was consistently better than the other models throughout the different phases of the comparison (i.e. through different phases of a software development project), and was thus the winner of this comparison.

5.4 Selection of model

There are several ways a model can be selected.

One is to use the comparison of the different models presented in section 5.2 as a basis, and then proceed to select the actual model by some systematic method. For example, the methods of decision analysis [Clemen 1996] are suitable for this.

Another method is to find out how well the model matches real operational data, and then select the method that is best by some selected criterion, e.g. the Akaike information criterion. This was the approach of [Khoshgoftaar and Woodcock 1991].

A third approach is to find a set of models whose reliability estimates are accurate and stable enough, and conservatively set the current failure probability as the maximum of the individual failure probabilities given by the models [Stringfellow and Andrews 2002].

A fourth approach [Lyu and Nikora 1992] is to first select a basic set of models, and prune out those models whose prediction biases (as measured e.g. by the U-plot) do not tend to cancel out. Then, each model is separately applied to the failure data. The resulting models are combined by forming a linearly weighted sum of the probability distribution functions. These weights should be nonnegative and sum up to 1; some proper choices for the weights include equal weights (yielding ordinary mean), medium-oriented linear combination (where the model is selected whose prediction lies between optimistic and pessimistic values), weighting the component models by the predictive accuracy they have shown so far, etc.

6. Summary and conclusions

This report is about the problem of statistically forecasting the number of software failures in a given time interval, given a history of previous failures (including the information that none have occurred). An emphasis in the review has been put on the form of the likelihood function which represents the probability that the data is what it is, given that the model has a specific parametric form.

A multitude of models have been proposed in the literature, but each has its drawbacks, some being shared by most models. A common problem with the reviewed models is that none allow for non-existent failure data, i.e., a software usage history with a known duration of time in operational use with no detected failures. Another problem shared by the models is that they support only rather modest reliability claims. There is no solution to this problem in sight.

Existing software reliability models don't take application complexity or test coverage (the proportion of all possible or plausible inputs that have been actually tested) into account [Whittaker and Voas 2000]. Things are furthermore complicated because the software under scrutiny never runs alone but is a part of a system consisting of hardware, operating system, interfaces (e.g. device drivers and communication interfaces), and possibly other programs.

Most of the reviewed models also share the feature that they have been developed to model reliability growth. This is appropriate when the program has been developed by a well-disciplined team and development-time fault reports, or at least fault statistics, are available. However, from the software users' point of view, it is more realistic to assume that only operational records are available.

Some recommendations can be made on applying software reliability models:

- Models that take into account software architecture, software complexity, test coverage, conduct of verification and validation, and structured expert opinion should be given priority.

- In applications requiring high dependability, software reliability models should be used only in conjunction with other methods of ensuring sufficient quality – otherwise the amount of testing grows prohibitively large. These methods include, but are not limited to, formal methods, software inspections and reviews, static analysis of code, and systematic software testing.

- It would seem that Bayesian models hold more promise in them than traditional frequentist models. An advantage of Bayesian approaches is that they allow the incorporation of different kinds of information, including human judgment.

- One should not rely on a single model, but rather choose a set of models whose results are combined in one way or another.

References

Asad, C.A., Ullah, M.I. & Rehman, M.J.-U. An approach for software reliability model selection. *Proceedings of the 28th Annual International Computer Software and Applications Conference* (COMPSAC'04), Vol. 1, 534–539.

Bai, C.-G. Bayesian network based software reliability prediction with an operational profile. *The Journal of Systems and Software*, Vol. 77, No. 3 (2005), 103–112.

Baker, C.T. Effects of field service on software reliability. *IEEE Transactions on Software Engineering*, Vol. 14, No. 2 (February 1988), 254–258.

Barghout, M., Littlewood, B. & Abdel-Ghaly, A. A non-parametric approach to software reliability prediction. *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, IEEE Press 1997, 366–377.

Basu, S. & Ebrahimi, N. Estimating the number of undetected errors: Bayesian model selection. *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, 4–7 November 1998, IEEE Press 1998, 22–31.

Basu, S. & Ebrahimi, N. Bayesian software reliability models based on martingale processes. *Technometrics*, Vol. 45, No. 2 (May 2003), 150–158.

Becker, G. & Camarinopoulos, A. A Bayesian estimation method for the failure rate of a possibly correct program. *IEEE Transactions on Software Engineering*, Vol. 16, No. 11 (November 1990), 1307–1310.

Brocklehurst, S. & B. Littlewood, B. New ways to get accurate reliability measures. *IEEE Software*, Vol. 9, No. 4 (July 1992), 34–42.

de Bustamante, A.S. & de Bustamante, B.S. Multinomial-exponential reliability function: a software reliability model. *Reliability Engineering and System Safety*, Vol. 79 (2003), 281–288.

Cai, K.-Y. Censored software-reliability models. *Transactions on Reliability*, Vol. R-46, No. 1 (March 1997), 69–75.

Cai, K.-Y. On estimating the number of defects remaining in software. *Journal of Systems Software*, Vol. 40 (1998), 93–114.

Cai, K.-Y. Towards a conceptual framework of software run reliability modelling. *Information Sciences*, Vol. 126 (2000), 137–163.

Campodónico, S. & Singpurwalla, N.D. Bayesian analysis of the logarithmic-Poisson execution time model based on expert opinion and failure data. *IEEE Transactions on Software Engineering*, Vol. 20, No. 9 (September 1994), 677–683.

Chen, M.-H. Lyu, M.R. & Wong, W.E. Effect of code coverage on software reliability measurement. *IEEE Transactions on Reliability*, Vol. R-50, No. 2 (June 2001), 165–170.

Cid, J.E.R. & Achcar, J.A. Bayesian inference for nonhomogeneous Poisson processes in software reliability models assuming nonmonotonic intensity functions. *Computational Statistics and Data Analysis*, Vol. 32 (1999), 147–159.

Clemen, R.T. *Making hard decisions – an introduction to decision analysis*, 2nd edition. Duxbury, Pacific Grove 1996.

Crow, L.H. Reliability analysis for complex, repairable systems. In: *Reliability and biometry* (Proshan, F. & Serfling, R.J. eds.), SIAM, Philadelphia 1974, 379–410.

Csenki, A. Bayes predictive analysis of a fundamental software reliability model. *IEEE Transactions on Reliability*, Vol. R-39, No. 2 (June 1990), 177–183.

Cukic, B. & Chakravarthy, D. Bayesian framework for reliability assurance of a deployed safety critical system. *Proceedings of the Fifth International Symposium on High Assurance Systems Engineering* (HASE 2000), IEEE Press 2000, 321–329.

DiCiccio, T.J., Kass, R.E., Raftery, A. & Wasserman, L. Computing Bayes factor by combining simulation and asymptotic approximations. *Journal of the American Statistical Association*, Vol. 92 (1997), 903–915.

Donnelly, M., Everett, B. Musa, J. & Wilson, G. Best current practice of SRE. Chapter 6 of [Lyu 1996], 219–254.

Duane. J.T. Learning curve approach to reliability monitoring. *IEEE Transactions on Aerospace*, Vol. AS-2, No. 2 (1964), 563–566.

Farr, W. Software reliability modelling survey. Chapter 3 of [Lyu 1996], 71–117.

Gelman, A., Carlin, J.B., Stern, H.S. & Rubin, D.B. *Bayesian data analysis*. Chapman & Hall, London 1995.

Goel, A.L. & Okumoto, K. Time-dependent error-detection rate model for software and other performance measures. *IEEE Transactions on Reliability*, Vol. R-28, No. 5 (August 1979), 206–211.

Goševa-Popstojanova, K., Hamill, M. & Perugupalli, R. Large empirical case study of architecture-based software reliability. *Proceedings of the 16th International Symposium on Software Reliability Engineering (ISSRE 2005)*, IEEE Press 2005.

Goševa-Popstojanova, K., Mathur, A.P. & Trivedi, K.S. Comparison of architecture-based software reliability models. *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE 2001)*, IEEE Press 2001, 22–31.

Haapanen, P., Pulkkinen, U. & Korhonen, J. Usage models in reliability assessment of software-based systems. Finnish Center for Radiation and Nuclear Safety (STUK), technical report STUK-YTO-TR 128, April 1997.

Hamlet, D. Are we testing for true reliability? *IEEE Software*, Vol. 9, No. 4 (July 1992), 21–27.

Helminen, A. Case study on reliability estimation of computer-based device for probabilistic safety assessment. *VTT Research Report BTUO-051375*, 2.11.2005, 32 p.

Helminen, A. & Pulkkinen, U. Reliability assessment using Bayesian networks – case study on quantitative reliability estimation of a software-based motor protection relay. Finnish Center for Radiation and Nuclear Safety (STUK), technical report STUK-YTO-TR 198, June 2003.

Herrmann, D.S. *Software safety and reliability*. IEEE Computer Society Press, Los Alamitos, 1999.

Hudepohl, J.P., Aud, S.J., Khoshgoftaar, T.M., Allen, E.B. & Mayrand, J. Integrating metrics and models for software risk assessment. *Proceedings of the 7th International Symposium on Software Reliability Engineering*, 30 Oct.–2 Nov. 1996, IEEE Press 1996, 93–98.

Iannino, A., Musa, J.D. & Okumoto, K. Criteria for software reliability model comparisons. *ACM SIGSOFT Software Engineering Notes*, Vol. 8, No. 3 (July 1983), 12–16.

IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard 610.12-1990. The Institute of Electrical and Electronics Engineers, New York 1990.

Jelinski, Z. & Moranda, P.B. Software reliability research. In *Statistical Computer Performance Evaluation* (Freiberger, W. ed.). Academic Press, New York 1972, 465–484.

Kahneman, D., Slovic, P. & Tversky, A. (eds.). *Judgment Under Uncertainty: Heuristics and Biases*. Cambridge University Press 1982.

Keene, S.J. Comparing hardware and software reliability. *Reliability review*, Vol. 14 No. 4, December 1994, 5–21.

Khoshgoftaar, T.M. Nonhomogenous Poisson processes for software reliability growth. *8th Symposium in Computational Statistics*, August 1988 (Compstat '88), 11–12 (Cited in [Khoshgoftaar and Woodcock 1991]).

Khoshgoftaar, T.M. & Munson, J.C. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 2, February 1990, 253–261.

Khoshgoftaar, T.M. & Woodcock, T.G. Software reliability model selection: a case study. *Proceedings of the 1991 International Symposium on Software Reliability Engineering*, May 17–18, IEEE Press 1991, 183–191.

Langberg, N. & Singpurwalla, N.D. A Unification of Some Software Reliability Models. *SIAM Journal of Scientific and Statistical Computing,* Vol. 6, No. 3 (1985), 781–790.

Laprie, J.C. (ed.). *Dependability: basic concepts and terminology: in English, French, German, Italian and Japanese.* Springer, Wien 1992.

Laprie, J.C., Kanoun, K., Béounes, C. & Kaâniche, M. The KAT (knowledge-action-transformation) approach to the modeling and evaluation of reliability and availability growth. *IEEE Transactions on Software Engineering*, Vol. 17, No. 4, April 1991, 370–382.

Littlewood, B. Dependability assessment of software-based systems: state of the art. *Proceedings of the 27th International Conference on Software Engineering*, 2005 (ICSE 2005), 15–21 May, IEEE Press 2005, 6–7.

Littlewood, B. & Strigini, L. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, Vol. 36, No. 11 (November 1993), 69–80.

Littlewood, B. & Verrall, J. A Bayesian reliability growth model for computer software. *Journal of the Royal Statistical Society, series C*, Vol. 22, No. 3, 1973, 332–346.

Lyu, M.R. (ed.). *Handbook of Software Reliability Engineering*. IEEE Computer Society Press and McGraw-Hill, 1996.

Lyu, M.R. Introduction. Chapter 1 of [Lyu 1996], pages 3–25.

Lyu, M.R. & Nikora, A. Applying reliability models more effectively. *IEEE Software*, Vol. 9, No. 4 (July 1992), 43–52.

Moranda, P.B. Event-altered rate models for general reliability analysis. *IEEE Transactions on Reliability*, Vol. R-28, No. 5 (August 1979), 376–381.

Musa, J.D., Fuoco, G., Irving, N., Kropfl, D. & Juhlin, B. The operational profile. Chapter 3 of [Lyu 1996], pages 167–216.

Musa, J.D., Iannino, A. & Okumoto, K. *Software reliability – measurement, prediction, application*. McGraw-Hill, 1987.

Musa, J.D. & Okumoto, K. Software reliability models: concepts, classification, comparisons, and practice. *Electronic Systems Effectiveness and Life Cycle Costing*, Slowirzynski, J.K. (ed.), NATO ASI Series, F3, Springer Verlag, Heidelberg 1983, 395–424.

Musa, J.D. & Okumoto, K. A logarithmic Poisson execution time model for software reliability measurement. *Proceedings of the international Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, California 1984, 230–238.

Ohba, M. Software reliability analysis models. *IBM Journal on Research and Development*, Vol. 28, No. 5 (July 1984), 428–443.

Pham, H. Software reliability and cost models: perspectives, comparison and practice. *European Journal of Operational Research*, Vol. 149, No. 3 (September 2003), 475–489.

Pham, H., Nordmann, L. & Zhang, X. A general imperfect-software-debugging model with S-shaped fault-detection rate. *IEEE Transactions on Reliability,* Vol. 48, No. 2 (June 1999), 169–175.

Pham, H. &. Zhang, X. A software cost model with warranty and risk costs. *IEEE Transactions on Computers*, Vol. 48, No. 1 (January 1999), 71–75.

Schick, G.J. & Wolverton, R.W. An analysis of competing software reliability models. *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 2, March 1978, 104–120.

Schneidewind, N.F. Analysis of error processes in computer software. *ACM Sigplan Notices,* Vol. 10, No. 6, June 1975, 337-346.

Selby, R.W. Empirically based analysis of failures in software systems. *IEEE Transactions on Reliability*, Vol. 39, No. 4, October 1990, 444–454.

Sommerville, I. *Software engineering*, 6th edition. Addison-Wesley, 2001.

Stringfellow, C. & Amschler Andrews, A. An empirical method for selecting software reliability growth models. *Empirical Software Engineering*, Vol. 7 (2002), 319–343.

Wang, W.-L. Pan, D. & Chen, M.-H. Architecture-based software reliability modeling. *The Journal of Systems and Software*, Vol. 79 (2006), 132–146.

Whittaker, J.A. & Voas, J. Toward a more reliable theory of software reliability. *IEEE Computer*, Vol. 13, No. 12 (December 2000), 36–42.

Yamada, S., Ohba, M. & Osaki, S. S-shaped software reliability growth modeling for software error detection. *IEEE Transactions on Reliability*, Vol. R-32, No. 5, December 1983, 475–478.

Zegueira, R.I. A model for Bayesian software reliability analysis. *Quality and Reliability Engineering International*, Vol. 16, No. 3, May/June 2000, 187–193.

ISBN 978-951-38-6622-8 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1459-7683 (URL: http://www.vtt.fi/publications/index.jsp)