Université de Tunis III
Institut Supérieur de Gestion

MEMOIRE DE DEA EN INFORMATIQUE DE GESTION

Option :
Recherche Opérationnelle

Evaluation of Competing Software Reliability Growth Models

Encadré par:

Pr. BARKAOUI KAMEL

Présenté par:

BEN JEDDOU ROUKAYA

Année universitaire 1998 / 1999

Acknowledgements

I would like to express my sincere gratitude and appreciation to my supervisor **M. Kamel Barkaoui** for his valuable advises encouragement and thoughtful suggestions.

I would also like to thank:

- M. Foued Ben Abdellaziz and M. Mhamed Ali El Aroui for their help.
- All my teachers of management institute, to whom I owe my formation.

At last, I would like to thank and dedicate my work to all my family in Algeria and in Tunisia especially my mother for her encouragement, and my friends Wiem Chitioui, Talel Laathari, Mohamed Aly Fall and Mohamed Yahyaoui for their care and support.

Ben Jeddou Roukaya

Contents

1 Introduction 5
- 1.1 Problematic . . . 5
- 1.2 Literature review . . . 6
- 1.3 Outline of the dissertation . . . 7

2 Software Reliability 8
- 2.1 System reliability . . . 9
- 2.2 Software features . . . 9
  - 2.2.1 Software and Hardware discrepancies . . . 10
  - 2.2.2 Software reliability . . . 11
  - 2.2.3 Software life cycle. . . . 14
- 2.3 Error, Fault, and Failure concepts . . . 16
  - 2.3.1 Error . . . 16
  - 2.3.2 Fault . . . 17
  - 2.3.3 Failure . . . 21
- 2.4 Reliability metrics . . . 24
  - 2.4.1 Reliability function . . . 25
  - 2.4.2 Mean value function . . . 25
  - 2.4.3 Hazard rate . . . 26
  - 2.4.4 The residual function . . . 26
  - 2.4.5 Failure intensity . . . 26
  - 2.4.6 The mean time to failure . . . 27
- 2.5 Conclusion . . . 27

3 Reliability Modelling 28
- 3.1 Statistical testing . . . 29
- 3.2 Trend analysis . . . 29
- 3.3 Reliability growth models . . . 30
  - 3.3.1 Jelinski and Moranda model [1972] . . . 30
  - 3.3.2 Goel and Okumoto NHPP model [1979] . . . 32
  - 3.3.3 S-shaped model [1983] . . . 33

3.3.4 Log-logistic model 34
3.4 Estimation of models parameters 35
3.4.1 The maximum likelihood method 35
3.4.2 Newton-Raphson algorithm 36
3.5 Application Example 37
3.5.1 Software failure data 37
3.5.2 Goel and Okumoto model 38
3.5.3 S-shaped model 42
3.5.4 Log-logistic model 46
3.6 Goodness of fit 50
3.7 Conclusion 51

4 Multi-log-logistic Software Reliability Model 53

4.1 Introduction 54
4.2 Trend Analysis 55
4.3 Multi-log-logistic model 55
4.3.1 Bi-log-logistic 56
4.3.2 Generalization of the log-logistic model [11, Pages 14,18] 58
4.3.3 Decomposition Principal 59
4.3.4 Parameters Estimation 60
4.4 Application example 61
4.5 Model Classification Scheme 65
4.6 Conclusion 66

5 Conclusions 67

List of Figures

Figure 2.1: Software and hardware failure rates 10
Figure 2.2: Reliability perceptions 12
Figure 2.3: Software life cycle 15
Figure 2.4: Faulty program 17
Figure 2.5: Perfect program 18
Figure 2.6: Cost versus number of residual faults 18
Figure 2.7: Correction strategies 20
Figure 2.8: Failure concept 22
Figure 2.9: Software behavior 23

Figure 3.1: Failure intensity for the JM model 31
Figure 3.2: Failure intensity for the GO model 33
Figure 3.3: The S-shaped failure intensity 34
Figure 3.4: The log-logistic failure intensity 35
Figure 3.5: The actual and fitted m(t) 40
Figure 3.6: The fitted failure intensity 40
Figure 3.7: The actual and the fitted residual 41
Figure 3.8: The fitted software reliability 41
Figure 3.9: The hazard rate 42
Figure 3.10: The actual and fitted m(t) 44
Figure 3.11: The fitted failure intensity 44
Figure 3.12: The actual and fitted residual function 45
Figure 3.13: Reliability for the S-shaped model 46
Figure 3.14: The hazard rate for the S-shaped model 47
Figure 3.15: The actual and fitted m(t) 48
Figure 3.16: The fitted failure intensity 49
Figure 3.17: The residual function 49
Figure 3.18: The log-logistic reliability function 50
Figure 3.19: The log-logistic hazard rate 51

Figure 4.1: Laplace test - NTDS data 56
Figure 4.2: Bi-log-logistic Taxonomy 58
Figure 4.3: Laplace test with Musa data n=36 59
Figure 4.4: Fitted model (set I) 63
Figure 4.5: Fitted model (set II) 63
Figure 4.6: Fitted model (set I) 65
Figure 4.7: Fitted model (set II) 65
Figure 4.8: Fitted model (set III) 65

List of Tables

Table 2.1: Differences between hardware and software reliability 11
Table 2.2: Life cycle cost distribution 15
Table 2.3: Program specification 17
Table 2.4: Failure severity classification 23

Table 3.1: NTDS data 38
Table 3.1: Goodness of fit with NTDS data 52

Table 4.1: Goodness of fit 66
Table 4.2: Model classification Scheme 67

Chapter 1

Introduction

1.1 Problematic

Software reliability is a major problem in software engineering and reliability theory. Several approaches and methods have been implemented to obtain a reliable software such as:

- Formal specification: software specifications should be well defined,
- Structured programming: programming using only *while loops if* statements (without using *go to* statements) which force programmers to think carefully about their program,
- Fault-free and fault avoidance techniques,
- test strategies,
- ...

However, these precautions are not enough to get a reliable software, we should check it effectively. It means that, methods have to be used to reach reliability objectives and others to check if these objectives are really reached.

For management of software development, a quantitative measurement of software reliability is important in assessing software performance and in optimizing development and maintenance costs. So the aim of software reliability is to quantify the performance of software, which due to the inevitable presence of faults within it, is subject to failure.

Software is an improvable system where failures are due to design faults. Software malfunction differs considerably from usual mechanical concept of failure; there is no physical process that determines when software should fail.

Faults present in the software are of syntax or logic nature, that may cause the software breakdown or produce results contrary to its specifications.

The reliability is hence, open to improvement during time, owing to their corrections. In contrast by repairable system (hardware) where performances are at most equal to new system. Hence the notion of reliability growth.

1.2 Literature review

A number of survey papers and a book (Musa et al.. 1987) deal with software reliability problem: assessment and prediction.

The first attempt to evaluate the software reliability has been carried out at the beginning of the Seventies where mathematical models had been proposed, basically to assess software reliability (Jelinski and Moranda 1972) and (Littlewood and Verrall 1973).

During the last two decades other models have been developed with different assumptions or by changing the critical assumptions in the Jelinski and Moranda model.

Goel and Okumoto (1979) present a stochastic model for software reliability based on a Non Homogeneous Poisson Process (NHPP). Based upon their works two assumptions are challenged:

- The initial number of software faults is treated as a random variable, while in Jelinski-Moranda model, it is an unknown, fixed constant,
- The times between failures $(i-1)$ and i depends on the time to failure $(i-1)$, whereas they are assumed to be independent of each other in the Jelinski-Moranda model.

Yamada et al. (1983) proposed an S-shaped reliability growth model and prove that this model fits the observed data (the curve of the number of detected faults for the observed data is S-shaped) in comparison to other deterministic and stochastic models.

In 1985 appeared a paper by Goel dealing with assumptions, limitations and applicability of some reliability models during the software life cycle.

Yamada and Osaki (1985) described NHPP models, with the maximum likelihood as a method of statistical inference.

Singpurwalla and Wilson (1994) review the literature concerning software reliability modeling. Gaudoin (1994) recaps in his report the problem of software reliability assessment and initiate non-specialist to the methodology and mathematical tools.

Faults present in the software are of syntax or logic nature, that may cause the software breakdown or produce results contrary to its specifications.

The reliability is hence, open to improvement during time, owing to their corrections. In contrast by repairable system (hardware) where performances are at most equal to new system. Hence the notion of reliability growth.

1.2 Literature review

A number of survey papers and a book (Musa et al.. 1987) deal with software reliability problem: assessment and prediction.

The first attempt to evaluate the software reliability has been carried out at the beginning of the Seventies where mathematical models had been proposed, basically to assess software reliability (Jelinski and Moranda 1972) and (Littlewood and Verrall 1973).

During the last two decades other models have been developed with different assumptions or by changing the critical assumptions in the Jelinski and Moranda model.

Goel and Okumoto (1979) present a stochastic model for software reliability based on a Non Homogeneous Poisson Process (NHPP). Based upon their works two assumptions are challenged:

- The initial number of software faults is treated as a random variable, while in Jelinski-Moranda model, it is an unknown, fixed constant,
- The times between failures $(i-1)$ and i depends on the time to failure $(i-1)$, whereas they are assumed to be independent of each other in the Jelinski-Moranda model.

Yamada et al. (1983) proposed an S-shaped reliability growth model and prove that this model fits the observed data (the curve of the number of detected faults for the observed data is S-shaped) in comparison to other deterministic and stochastic models.

In 1985 appeared a paper by Goel dealing with assumptions, limitations and applicability of some reliability models during the software life cycle.

Yamada and Osaki (1985) described NHPP models, with the maximum likelihood as a method of statistical inference.

Singpurwalla and Wilson (1994) review the literature concerning software reliability modeling. Gaudoin (1994) recaps in his report the problem of software reliability assessment and initiate non-specialist to the methodology and mathematical tools.

The paper by Perrin (1995) presents a bi-logistic model useful in modeling many biological and socio-technical systems not well modeled by the simple logistic.

A recent paper proposed by Swapna S. Gokhale and Kistor S. Trivedi (1998) deals with a new log-logistic software reliability model. Using a step-step Laplace test, Swapna et al. deduce that the fault behavior exhibit an increasing / decreasing nature so they proposed a log-logistic model with an increasing / decreasing failure intensity.

Until now several models have been developed since we enumerate a fifty. Each model try to be better than the others by using different assumptions. In fact, any model is effective at all situations and for different software data. One model fits a particular set of data and maladjusted to others.

1.3 Outline of the dissertation

This dissertation consists of five chapters.

In **chapter 2** we describe software reliability topics while starting with general system reliability (hardware & software) and carrying out the differences between hardware and software. After looking to software features, reliability is then specified in the case of the software system. Basic concepts such as: Error, Fault, and failure are defined with the major software reliability metrics (Reliability function, MTTSF, Residual function...).

Chapter 3 focus on existing software reliability basic models such as: Goel-Okumoto, S-shaped , log-logistic...

These models are applied to observe software faults. Some measures of interest (fitted model, reliability function, residual function ...) are evaluated to check the prediction provided by each model. To select the appropriate model, we use the goodness of fit as criteria for evaluation between these competing models.

Chapter 4 deals with a generalization of the log-logistic model. Using the step-by-step Laplace test we deduce that the fault behavior can exhibit an iterative increasing / decreasing nature so we propose a multi-log-logistic model. To modelize a multi-log-logistic model we decompose, the observed data into two or more sets each set is evaluated by a single log-logistic model.

We reuse the goodness of fit as criteria for comparison and conclude that a multi-log-logistic model fits better the data than the log-logistic model. A new model comparison scheme is given, where we classify models by failure intensity nature.

Finally, a general conclusion and main perspective are proposed in **chapter 5**.

Chapter 2

Software Reliability

AIM

The aim of this chapter is to describe software reliability and software features. Difference between software and hardware, and basic concepts such as error, fault, and failure are also focused. Reliability metrics are defined.

CONTENT

2.1 System reliability
2.2 Software features
2.3 Error, Fault, and Failure concepts
2.4 Reliability metrics
2.5 Conclusion

2.1 System reliability

Occasionally, when we buy an electronic system such as: radio, TV, washing machine... and we put it into use, we found it did not work or it fails after few days; we know that our system is unreliable and it requires more tests to correct the residual defects, hence to reach the same results as those provided by a good one.

The break down or the failure can cause a user embarrassment and repair activity of the defect system can be expensive if the system is just a household electrical appliances. However the consequences are more critical if the system is a Bank Automated Teller (ATM.), airplane navigation, nuclear power, or an automated drug injection.

A malfunction of such systems can have consequences of a human and economic nature, varying from minor to catastrophic impact.

However to avoid catastrophic results that would cost human life or property, system should be tested and defects causing possible failures have to be removed, until it attains an acceptable level compared to its specifications and requirements.

The level at which a system test stops is called the reliability; for instance we say the reliability of an airplane engine at time t is 0.995 so the reliability can be considered as a test stop condition and a decision to put the product into use.

System reliability is the ability to maintain a system works, in accordance with its specification, during a certain time; in other words system reliability is the probability that a system remains operational during a specific time interval and under given environmental conditions.

Basically, computer systems are a combination of hardware and software, the presence of defects in either of these components could cause a system break down or failure. Systems failures are due to hardware, software or the interaction between them. In fact, the major part of failures are coming from software, because hardware reliability become more and more controllable than software reliability. Hence we can say that the system reliability is equivalent to software reliability. Although hardware and software reliability share many similarities and some differences, the analogies between them should not be carried too far [10].

2.2 Software features

Since the system reliability is closely related to software reliability, we should focus on some software particularities to increase the reliability of the whole

system. Differences between Software and hardware should to be pointed out to show that techniques used to asses and improve hardware reliability should to be moderate in the case of software reliability.

2.2.1 Software and Hardware discrepancies

The field of hardware reliability has been established for longer time. Hence, it seems reasonable that the first attempt for evaluating software reliability be inspired by hardware reliability techniques. However current approaches for evaluating software reliability are basically similar to those used for hardware reliability assessment should take into account the appropriate modifications for the inherent differences between software and hardware reliability [Table 2.1].

The major difference is related to failure, because the source of failures in software is design faults whereas the main factor in hardware has broadly been physical deterioration. As it is illustrated, the classical bathtub curve [Figure 2.1], hardware exhibits mixtures of decreasing and increasing failure rates [10].

- The decreasing failure rate is seen due to the fact that, as test or use time on the hardware system accumulates, failures, most probably due to design faults are encountered and their causes are fixed.
- The increasing failure rate is primary due to hardware component wear-out or aging. However there is no such thing as wear-out in software. It is true that software may become obsolete because of changes in the user and computing environment, but once the software is updated to reflect these changes, a modified version of the above software will be generated.

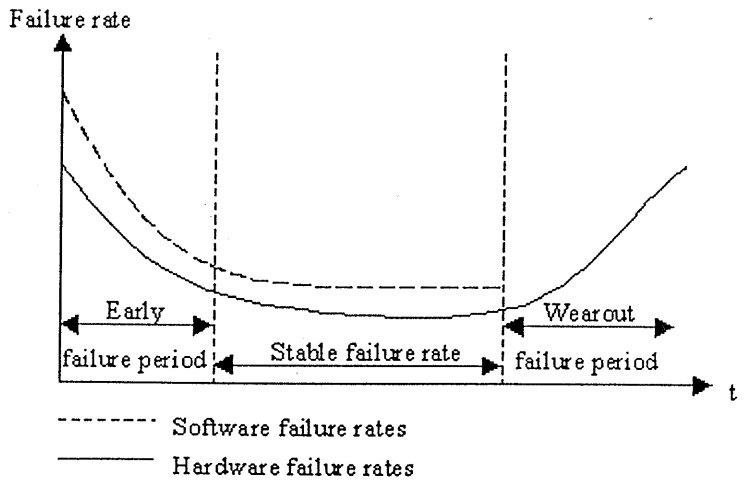


Figure 2.1 : Software & Hardware failure rates [16, Page 37]

Also hardware reliability may change during certain periods, such as initial burn-in or the end of useful life while software reliability tends to change continually during test periods since new faults are introduced either when new code is written or when repair action removes faults from the software. Software failure usually occurs only when a program exposed to an environment that it was not developed or tested for, however, hardware failure is due to wear and other physical causes.

It should also be noted that an assessed value of the software reliability measure is always relative to a given use environment, for instance, two users exercising two different sets of paths in the same software are probably to have different values of software reliability.

It should be pointed out that hardware and software reliability share many similarities. Both may be defined in the same way. Consequently, hardware and software component reliability may be combined to get system reliability. Both depend on the environment. In addition, software like hardware, exhibits a decreasing failure rate, improvement in quality, as the usage time on the system accumulates and design faults are fixed and removed.

Table 2.1: Differences between hardware and software reliability

Hardware reliability	Software reliability
Source of failures are physical deterioration.	Source of failures are design faults.
Failures are due to wearout and physical causes.	Failures are due to introduction and removal of faults during software development.
Classical bathtub curve.	There is no such curve for software.
failures are Chronic.	If a detected fault is fixed and correctly removed from software it is permanently eliminated.
Faults and failures are due to material bugs and wearout.	Faults and failures are due to human errors inherent to the programmation activity.
Hardwares are repairable systems.	Softwares are improvable systems.

2.2.2 Software reliability

Computer applications become more diverse and invade every field, so the dependability of computer systems has become a main interest for practitioners and theorists.. The dependability components are: maintainability,

availability, safety and reliability which is a matter of concern during this work.

"It is increasingly apparent that the most important characteristic of software is its reliability [15] ", because theorists as well as practitioners are aware of consequences caused by unreliable systems.

Due to the uncertainty in the various software functions, software reliability is expressed as a probabilistic measure, and can be defined as the probability that software faults do not cause failure during a specified period in a particular environment conditions.

The specified period can be a single run, a number of runs, calender time (day, month...) or execution time (CPU time) [10].

Reliability is a dynamic system characteristic which is a function of the number of software failures. Software faults cause software failures when the faulty code is executed with a particular set of inputs. Faults do not always manifest themselves as failures so the reliability depends on how the software is used, therefore reliability cannot be specified absolutely [15].

Each user runs a program in different ways so, faults which affect the reliability of the system for one user may never manifest themselves under a different mode of working for the other [Figure 2.2].

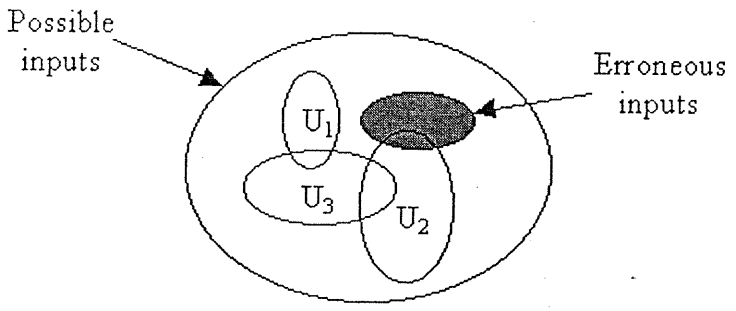


Figure 2.2: Reliability perceptions [15, Page 292]

The set of inputs produced by user2 intersects with the erronenous input set so, for some inputs from user2, the system will fail. User1 and user3, however, never choose inputs from the erroneous set so will always see the software as completely reliable.

Reliability can only be accurately specified if the software operational profile is also specified.

Operational profile : The operational profile reflects how the software will be used in practice. It consists of a classification of inputs and the probability of their occurrence [9].

The operational profile differs from one user to another. Two users with different operational profile can have different reliability perceptions for the same software.

Definition 1 ▶ *The operational profile is the probable mode of usage of the software, it can be given by discovering the different classes of inputs to the program and estimating their probability. The inputs which have the highest probability of being generated fall into a small number of classes.*

The software responds to these inputs by producing an output or a set of outputs. Some of these inputs cause system failures where erroneous outputs are generated by the program. the reliability is related to the probability that, in a particular execution of the program, the input to the system will be a member of the set of inputs which causes an erroneous output.

It is probable that there will be a small number of members belong to the erroneous input set which are much more likely to be selected than others. The reliability of the software is therefore related to that probability rather than the mean probability of selecting an erroneous input [10].

As reliability is related to the probability of an error occurring in operational use, a program may contain known faults but may still be seen as reliable by its users. They may use the system in such a way that they never select an erroneous input; the program always appears to be reliable. Furthermore, experienced users may work around known software faults and on purpose avoid using paths which they known to be faulty.

Increasing the reliability of a software usually slows it down. because if software is to be very reliable it must include extra, often redundant code to perform the necessary checking, what reduces the program execution speed and increases the amount of store required by the software and increase development cost. Although, reliability should be checked for the following reasons [15]:

1. unreliable software are avoided by users. If a company attains a reputation for unreliability because of a single product, this will influence future sales.

2. There are an increasing number of software systems coming into use where the human and economic costs of a catastrophic failure are unacceptable.

3. Software which is unreliable can have hidden errors which can violate system and user data without warning and which do not immediately manifest themselves.

2.2.3 Software life cycle.

Definition 2 ► *Software is the set of computer modules; which are composed of one or more program; associated with some application or product. It includes also the documentation necessary to develop, use, install and maintain theses programs.*

Definition 3 ► *A program is a set of data and machine instructions that when executes accomplishes a specific function. Program = Algorithms + structured data*

The software development process includes six major phases [Figure 2.3] which are [12]:

1. **Software Requirements & Objectives:** this initial phase gives a description and global evaluation of requirements which the software is supposed to satisfy and a specification of the functions that the software must carry out.

2. **Software Design:** the goal of the design phase is to define software functions so that they may be translated into one or more executable programs, and to establish software architecture. Close of this phase, all technical choice are made, functions are specified, modules regroupment are known, algorithms which will be implemented are identified.

3. **Implementation and unit testing:** This stage correspond to actual implementations coming from the design phase. Functions are translated into machine language, they become executable and hence experimentally checked, so possible weakness will be revealed. In case of failure attributed to the design phase, it is compulsory to update design's documentation at the same time as the faulty programs. From implementation phase the modules which regroup different functions must be compiled without mistake, and have been executed with test data which prove that the logic of program is good.

4. **Integration and tests of qualification:** The aim of this phase is to integrate all modules so as to assure the verification and validation of software, until it can work in its actual environment.. Close of this phase, we are able to ensure, by software execution, that the requirements of the software specified in phase1 are satisfied.

5. **Installation:** In this stage the system is delivered to the customer, installed, and putted into practical use.

6. **Operation and maintenance:** This is the longest life cycle phase. Information is fed back to previous development phases. This final phase is an operational stage where the software is used. Errors and omissions in the original software requirements are discovered, program and design errors are revealed. Maintenance stage involves correcting errors which were not discovered in earlier stages of the life cycle, improving implementation of the system units and enhancing the system's services as new requirements are discovered.

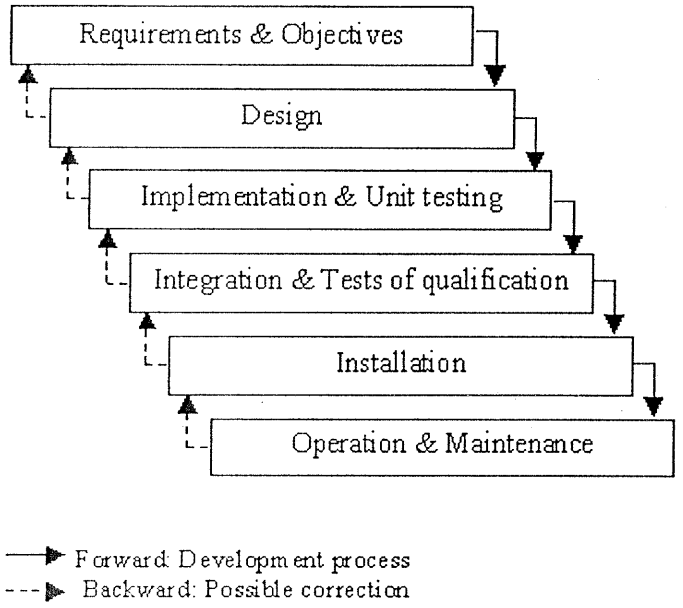


Figure 2.3: Software Life Cycle [12, Page 45]

A number of studies are made to allocate errors show that [1]:

- two-third of defaults are caused by analysis and design errors.
- one-third of defaults are attributed to implementation.

Regarding the cost of errors, these studies point out that the cost increase with the distance between the phase where the error is committed from the one which it has been detected [Table 2.2].

Table 2.2:Life cycle cost distribution [12, Page 107]

Requirements & Objectives	1			
Design	2.5	1		
Implementation & unit testing	6	2.5	1	
Integration & tests of qualification	25	10	4	1
Operation & maintenance	75	30	12.5	3

These figures show that errors committed in the initial phase of the life cycle are more expensive than in the other phases.

Example 4 *The above table shows that an error committed in the requirements and objectives phase will cost 75 times more expensive if it is removed during operation and maintenance phase.*

2.3 Error, Fault, and Failure concepts

Software failures are always caused by human *error* such as: the omission of a particular case, wrong interpretation of information, incoherent specification etc. In the software, the human errors are translated by a default. A default remain non apparent for long time. If some execution conditions are putted together, the default can be able to cause an execution *fault* of the functional unit corresponding to the program in which the default is revealed. This functional unit is not necessary the same that contain the default [12].

Eventually, in accordance with the nature of the fault, a *failure* and/or breakdown of the total system can take place.

2.3.1 Error

Definition 5 ▶ *An error is a programmer misapprehension of the user requirements.*

Errors can come up from many causes, but the well known sources are [10]:

1. **Transcription:** errors that occur between mind and paper or machine.
2. **Incomplete analysis:** errors are exhibited in a lack of recognition of all the possible conditions that can occur at a given point in the program.
3. **Knowledge:** these errors are generated when project personnel are inexperienced, basically in the design methodology, the programming, language, and the application area.
4. **Communication:** these errors arise when a misunderstanding take place between customer and designer, designer and coder, designer and tester or among designers, coders, testers.

2.3.2 Fault

Definition 6 ► *A fault is a defective, extra instruction, or missing instructions in the program. Software faults are not only program defects. Incomplete requirements and/or omissions in software documentation can lead to unexpected behavior.*

It should be noted out that the location of faults within the program are unknown, because the commission of errors by programmer - and hence the introduction of faults - is a very complex and unpredictable process. So the consequences of a faults are [12]:

- **critical:** if it interrupt the system's mission.
- **Serious:** if it provoke wrong answers.
- **Moderate:** if it provoke embarrassment of the user and waste of time.
- **Tolerable:** if the embarrassment is evident, we can ignore the corresponding default.

Here is an example to illustrate fault causes.

Example 7 *A seller needs a program able to calculate the total price (PT) of purchased quantities while respecting the attributed reduction in the table below:*

Table 2.3: Program specification

Purchase	Reduction
A$\leq$ 10 u	0%
10 <A$\leq$ 50 u	5%
50 <A$\leq$ 100 u	10%

The implementation of the above specification gives:

```
Line number        Code
    01        IF A< 10
    02        THEN PT← PU *A
    03             ELSEIF (A > 10 and A ≤ 50)
    04                  THEN PT ← PU * A * 0.95
    05                  ELSE PT ← PU * A * 0.7
```

Figure 2.4: Faulty Program

To make the detection of faults easy in the faulty program we should compared it to the "perfect" program.

Line number	Code
01	IF A ≤ 10
02	THEN PT← PU *A
03	ELSEIF (A > 10 and A ≤ 50)
04	THEN PT ← PU * A * 0.95
05	ELSEIF (A > 50 and A ≤ 100)
06	THEN PT ← PU * A * 0.9

Figure 2.5: Perfect Program

Remark 1 *The total price cannot be given if the purchased quantities are equal to ten.*

Remark 2 *For purchases greater than fifty, the program attributes a reduction of 30% instead of 10%, what can cause a loss for the seller.*

Remark 3 *For purchases greater than one-hundred, the total prices are not specified. This fault belongs to the faulty program as well as the perfect program.*

To achieve a high reliability software faults should be removed from a program. However, it is very difficult to detect and correct remaining faults and hence the cost of finding and removing residual faults tend to rise exponentially [15] [Figure 2.6].

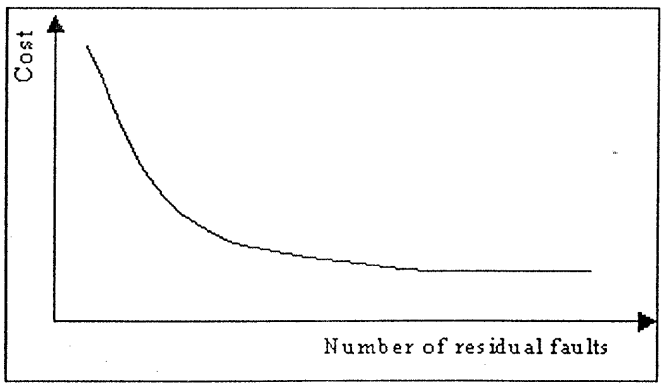


Figure 2.6: Cost versus Number of Residual Faults [15, Page 291]

Consequently, before looking for a development process which is based on fault discovery and elimination it is convenient to focus for fault avoidance process.

Fault Avoidance

Definition 8 ► *Fault avoidance is the use of development techniques which reduce the probability of introducing faults into a program.*

The development of software with a high-level reliability based on fault avoidance strategy depends on:

1. The description of a precise and formal specification,
2. The choice of a software design method,
3. The considerable use of reviews in the development phase which validate the software system,
4. The testing process to detect faults which are not discovered during the development reviews.

Fault Correction

When a failure take place, it is compulsory to make it right. hence we attempt to locate the fault that caused this failure and remove it.

Definition 9 ► *Fault correction is a modification of a program after which we hope that the input state that had so far caused a failure will now provide results conforms to specifications.*

There are three types of corrections [10] [Figure 2.7]:

1. **Good correction:** if the fault after correction (f_f) is a subset of the fault before correction (f_i), i.e. if we eliminate a part of f_i without introducing other faults. Close to this type of correction, the software reliability has a tendency to increase.
2. **Bad correction:** if f_i is a subset of f_f. In this case the correction introduce new faults instead of eliminate some of the old ones. if the correction is bad, the reliability has a tendency to decrease.
3. **General correction:** a part of the correction will be good and the other will be bad. It means, we remove a part of existent faults and we introduce others. If the introduced faults are less sensitive to appear than the removed faults, the correction can be satisfying.

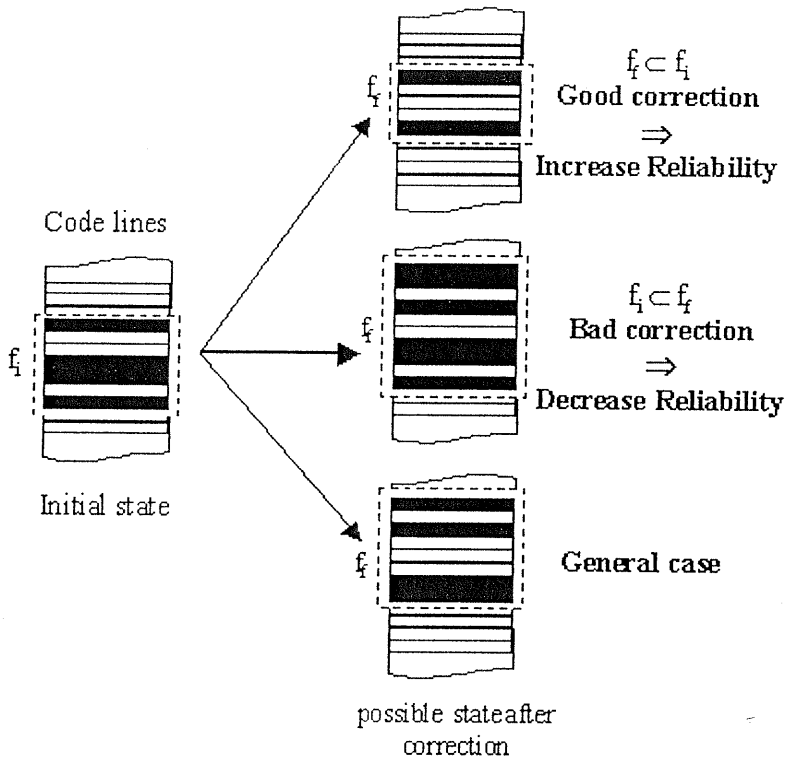


Figure 2.7: Correction strategies

It should be pointed out that [12]:

- The fault can be bogged down in a load of correct information, what can make its detection very difficult.
- The period elapsed between a fault introduction and its uncoverment can be very long (several years).
- A fault cannot necessary followed up by a failure.
- At the time of a failure, it could has elapse a certain time between the fault and the failure, what can make difficult the identification of the faulty module.
- Once the first faults are corrected, the discovery of the residual faults become very difficult; i.e. the relation between the failure and the fault is less and less evident.
- The error correction can bring streams of modifications in other modules that have functional dependency with the (these) faulty module(s), what make necessary a complete validation of software.

2.3.3 Failure

Definition 10 ▶ *A software failure is an unexpected behavior when the software is executed. This unexpected behavior is expressed by a discrepancy between the actual output state (set of values of output variables the run generates) provided by the execution of the program and the desired output state specified by the requirements.*

Failure concept

Software failures are caused by software faults when the faulty code is executed with a particular set of inputs.

For instance, in Figure 2.8, when the program is executed with input I the output is convenient with the result predicted by the software requirements and failure does not occurs. However, faults F_1 and F_2 manifest themselves when the input variables belong to the sets I_1, I_2, I_3, I_4 so the results provided by the software are different from results predicted by specification; and therefore the software failures take place.

So when a fault is executed under a particular conditions it causes failures. Moreover there can be different sets of conditions that cause failures or the conditions can be repeated; consequently a fault can be the source of more than one failure. However there cannot be multiple faults causing a failure [10], the entire set of defective instruction that is causing the failure is considered to be the fault (F_2). It can be noted that a fault cannot necessary followed up by a failure.

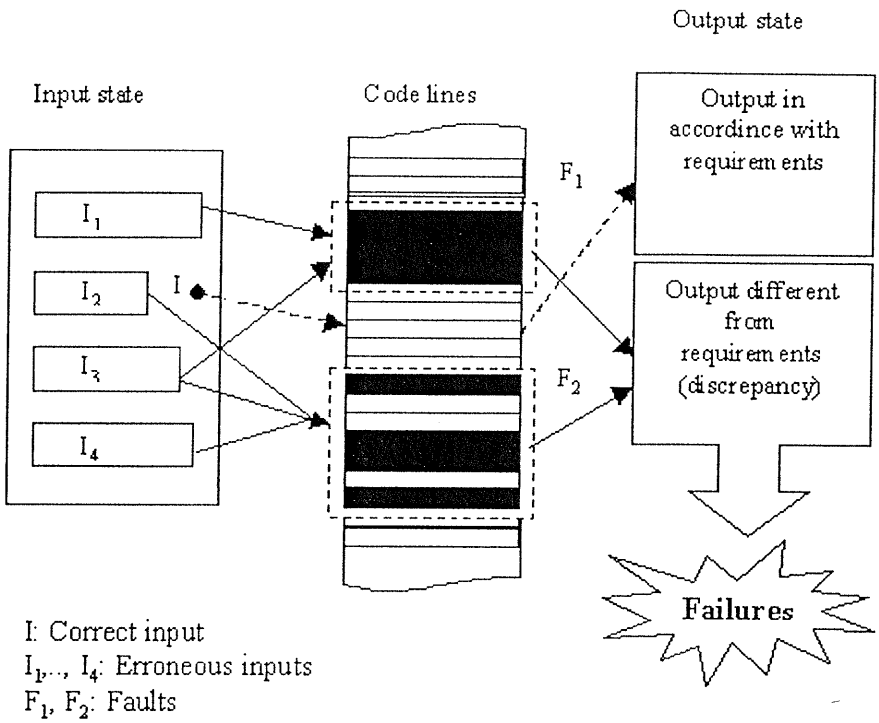


Figure 2.8: Failure concept

Failure process [2]

Failure process is a stochastic modeling of the software behavior during time by ignoring the internal structure of the software. The software functioning is observed from the start of testing process. Failures will occur randomly in time noted t_i. After each failure, an attempt will be made to identify and correct the fault that caused the software failure. corrections are made in negligible time.

This dynamic method, which is an evaluation of the software reliability, allows us to define three processes [Figure 2.9] which are:

- **Failure instant process:**

$T = \{T_i\}_{i \geqslant 1}$ where T_i is the instant of the ith failure.

- **Time between failure process:**

$X = \{X_i\}_{i \geqslant 1}$ where $X_i = T_i - T_{i-1}$ with $T_0 = 0$;

Xi is the time interval between the ith and the $(i-1)$ th failure.

- **Failure count process:**

$N = \{N_t\}_{t \geqslant 0}$ where N_t is the cumulative number of failures between the initial state ($T_0 = 0$) and the instant t.

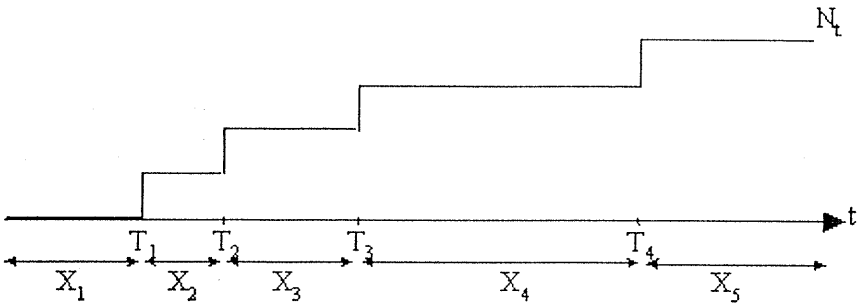


Figure 2.9: Software behavior [2, Page 4]

These processes are modelizes via analytical models.

Failure classification [15]

Failures usually differ in their influence on the software behavior, so they may be classified by severity [Table 2.4] and propose the reliability or failure intensity for each classification [Example 11].

Because when we give the reliability or failure intensity for each failure type we can ameliorate these measures for each category of failure as well as the whole system.

We can point out that combinations of these categories can occur such as a failure which is transient and recoverable.

Table 2.4: Failure severity classification [15, Page 397]

Failure class	Fault	Consequences on the software behavior
Permanent	Critical	This kind of failure occurs with all inputs and interrupt the software.
Non recoverable	Serious	Answers provided by the software are wrong which cause a software degradation and need external intervention.
Recoverable	Moderate	Software can recover from failure without external intervention.
Transient	Tolerable	This failure class occurs with certain inputs and can be ignored since transient failures are the less severe.

Example 11 *Consider a bank automated teller system with about 1000 transactions per day. We give the cost of one failure (C) for each class. We assume that the failure intensity (λ) for the overall classes of failures is equal to 0.75 failure*

per hour. Reliability is specified in term of failure per transactions for each class of failure. The number of observed failures is shown and the failure intensity is calculated for each class.

FC	Example	C	N	R	λ
P	No bar code data of a magnitude card can be read.	$C > 100$	1	1 *in* 1000 t	0.04
NR	Card cannot be return to user.	$10 < C \leqslant 100$	4	1 *in* 250 t	0.17
R	Loss of bar code data.	$1 < C \leqslant 10$	8	1 *in* 200 t	0.33
T	Bar code data cannot be read in a particular card.	$C \leqslant 1$	5	1 *in* 2000 t	0.21
		Total	18		0.75

Remark 4 *This example shows in addition to reliability and failure intensity, the cost impact for each class; for instance the effort in term of cost needed to put right a failure of permanent type is the most important compared to the other classes of failures which explain failure severity.*

2.4 Reliability metrics

Software reliability metrics have evolved from hardware reliability metrics. However, hardware metrics cannot be used without modification because of the differing nature of software and hardware failures. Software failures are transient, they are only exhibited from some inputs so the system can remain operational in the presence of these failures. Unlike software, hardware failures tends to be chronic [2], and the system stops working until repairing the manufactural defect.

This distinction prove that commonly used hardware reliability metrics such as: mean time to failure, rate of failure occurrence... are less useful for quantifying software reliability. Reliability evolves during software life, it can be evaluated from the failure process examination. Reliability metrics are, hence, functions describing the failure process behavior during time.

Some of these metrics which have been used to evaluate software reliability are [11]:

2.4.1 Reliability function

At time t, the reliability is the probability that the software is executed without failure during a specified time interval started from t.

Definition 12 ► *At time t, the reliability is given by:*

$$\forall \tau \geqslant 0, R_t(\tau) = P(T_{N_t+1} - t > \tau) \tag{2.1}$$

The reliability is related to the *cumulative failure probability* $F(t)$ by:

$$R_t(\tau) = 1 - F_t(\tau) \tag{2.2}$$

Where $F_t(\tau)$ is the probability that the software fails by time t.

Definition 13 ►*At time t, the failure probability is given by:*

$$\forall \tau \geqslant 0, F_t(\tau) = P(T_{N_t+1} - t \leqslant \tau) \tag{2.3}$$

If $F_t(\tau)$ is differentiable, then the *failure density* $f(\tau)$, is the first derivative of $F_t(\tau)$.

Remark 5 *The reliability at time t is given by:*

$$R_t(\tau) = \exp\left(-\int_t^{t+\tau} \lambda(y)\, dy\right) \tag{2.4}$$

Where $\lambda(y)$ is the failure intensity.
The reliability function can also be rewritten as:

$$R_t(\tau) = \exp(-[m(t+\tau) - m(t)]) \tag{2.5}$$

2.4.2 Mean value function

The mean value function Mean value function $m(t)$ represents the expected number of software failures associated with each time point.

Definition 14 ► *The mean value function is given by:*

$$\forall t \geqslant 0; m(t) = aF(t) \tag{2.6}$$

Where a is the expected number of failures eventually observed.

2.4.3 Hazard rate

The hazard rate is the rate at which the individual faults manifest themselves as failures during testing.

Definition 15 ▶ *The hazard rate $Z(t)$ is the probability density of failure given that failure has not occurred up to the present:*

$$\forall t \geqslant 0; Z(t) = \frac{f(t)}{1 - F(t)}$$

The hazard rate can be rewritten by using *Eq.* (2.6) into the previous equation:

$$Z(t) = \frac{m'(t)}{a - m(t)} \tag{2.7}$$

Let $\lambda(t)$ be the derivative of $m(t)$ so:

$$Z(t) = \frac{\lambda(t)}{a - m(t)} \tag{2.8}$$

2.4.4 The residual function

The residual function represents the expected number of faults remaining in the software.

Definition 16 ▶ *The residual function $Q(t)$ is given by:*

$$\forall t \geqslant 0; Q(t) = a - m(t) \tag{2.9}$$

2.4.5 Failure intensity

The failure intensity function is the rate of change of the mean value function or the number of failures per unit time.

Definition 17 ▶ *The failure intensity is given by:*

$$\forall\, t \geqslant 0;\ \lambda(t) = \frac{\partial m(t)}{\partial t} \tag{2.10}$$

Which can be rewritten as:

$$\begin{aligned} \lambda(t) &= [a - m(t)] \frac{f(t)}{1 - F(t)} \\ &= Q(t) Z(t) \end{aligned} \tag{2.11}$$

2.4.6 The mean time to failure

The mean time to software failure is the expected waiting time to the next failure.

Definition 18 ▶ *At time t, the Mean Time To Software Failure is given by:*

$$MTTSF_t = E(T_{N_t+1} - t) \tag{2.12}$$

Remark 6 *If the MTTSF is finite then:*

$$MTTSF_{T_n} = \int_0^\infty R_t(\tau)d\tau \tag{2.13}$$

Basically, the failure intensity depends on a lot of random variables so it is extremely complicated which make hard the calculation of the reliability defined above. To overcome this problem, assumptions should be done to simplify the definition of λ_t and hence to obtain results easy to use mathematically.

The diversity of assumptions used to define λ_t will give different models and different strategies to assess the software reliability. The software reliability modeling is completely defined when a specified form of the failure intensity is proposed.

2.5 Conclusion

In this chapter we describe basic concepts related to software reliability. Needs to reliability to evaluate the system's ability to perform its specification. Software characteristics are revealed to show that hardware reliability have not to be used without taking these features into account.

Human errors are translated by software faults, these faults cause software failures when the faulty code is executed with a particular operational profile.

To measure software reliability quantitatively, some metrics are given to evaluate and predict software behavior. These metrics will be completely determined when software reliability models are applied, this will be the topic of the next chapter.

Chapter 3

Reliability Modelling

AIM

The aim of this chapter is the application of certain software reliability models to actual data. Using a statistical tools we can compare the fitted models to data and also between each others. Reliability metrics are measured to assess software reliability and to make decision in the future.

CONTENT

3.1 Statistical testing
3.2 Trend analysis
3.3 Reliability growth models
3.4 Estimation of models parameters
3.5 Application example
3.6 Goodness of fit
3.7 Conclusion

3.1 Statistical testing

The major aim of statistical testing process is to determine the reliability of the software, it involves four steps [15]:

- Determine the operational profile (software input),
- Apply these inputs to the software, recording the time between each observed failure X_i.
- Generate a data set, corresponding to the operational profile,
- Compute the software reliability after a statistically significant number of failures have been observed.

However statistical testing is usually combined with reliability growth models, so prediction and assessment of the software reliability can be made. Almost all these models assume an increasing in the reliability so, before their application, reliability growth and software improvement have to be checked. These improvements are available only with a constant operational profile.

To check reliability growth we apply a trend analysis such as *Laplace test.*

3.2 Trend analysis

Trend analysis can significantly help in choosing the appropriate model for a given data set. Taking into consideration the trend displayed by the data can lead to a good model choice. However using a model without taking into account the trend analysis can lead to unrealistic results, when the trend displayed is different than that assumed by the model.

The most commonly used trend analysis test is the Laplace test [2] where we choose the hypothesis H_0 *:"there is no reliability trend"* and the alternative hypothesis H_1*:"there is a reliability growth"* or H_2*:"there is a reliability decay"*.

Laplace test

Under H_0, the random variable

$$U = \frac{\sum_{i=1}^{n} T_i - n\frac{T}{2}}{\sqrt{\frac{nT^2}{12}}} \rightsquigarrow N(0,1) \tag{3.1}$$

The aim of Laplace test is to compute the observed value u and to make this comparison:

At a significant level α,

If $u < \lambda_\alpha$ then reliability increases

If $u > \mu_\alpha$ then reliability decreases

The value u gives a global indication of increasing (decreasing) reliability. However, if we need more precision about the reliability behavior we should apply a step-by-step Laplace test [3] which is given by:

$$l(t) = \frac{\frac{1}{i-1}\sum_{n=1}^{i-1}\sum_{j=1}^{n} x_j - \frac{1}{2}\sum_{j=1}^{i} x_j}{t\sqrt{\frac{1}{12(i-1)}}} \tag{3.2}$$

Where:

i is the $i\ th$ failure,

t is the time corresponding to the $i\ th$ failure,

x_i is the time between $(i-1)\ th$ and $i\ th$ failures,

n is the total number of failures.

If we plot $l(t)$ for each failure i, we can conclude that there is an increasing, decreasing or increasing/decreasing reliability and hence choose the appropriate model which represent the above behavior.

A decreasing failure intensity $\lambda(t)$ implies reliability growth and an increasing / decreasing $\lambda(t)$ implies an increasing / decreasing reliability.

3.3 Reliability growth models

A reliability growth model is a mathematical model of software reliability, which can be used to predict when a particular level of reliability is likely to be reached. The software reliability models - some of them described below - assess the reliability of software in the testing and operational phase, they try to answer to the question " **How reliable is the software?**"

3.3.1 Jelinski and Moranda model [1972]

The model of Jelinski and Moranda [2], known as the JM model is one of the earliest for assessing software reliability and has formed the basis for many models developed after.

Assumptions

JM1 There are N software faults at the start of testing,

JM2 Each fault is independent of others and is equally likely to cause a failure during testing,

JM3 A detected fault is removed with certainty in a negligible time and no new faults are introduced during the debugging process,

JM4 the times between failures X_i are independent random variables with:

$$X_i \rightsquigarrow \exp[\Phi(N - i + 1)] \tag{3.3}$$

Where Φ is proportionality constant.

Model

At any time, the failure intensity is assumed to be proportional to the number of residual faults

$$\forall t, \lambda_t = \Phi(N - N_t) \quad \text{Where} \quad \begin{matrix} N \in \aleph \\ \Phi \in \Re_+^* \end{matrix} \tag{3.4}$$

N_t is the number of detected fault at t.

Φ is the per fault failure rate. It represents at each correction a constant improvement carried out on λ_t. The failure intensity of the JM model is illustrated in Figure 3.1.

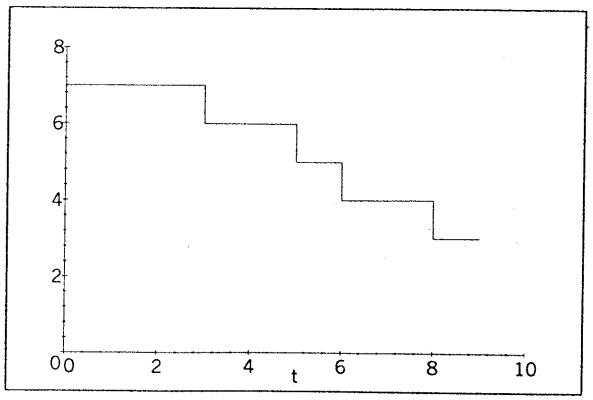


Figure 3.1: The failure intensity for the JM model [2, Page 8]

Criticism

The assumptions described above are criticized because:

- We cannot enumerate all faults in the software since we ignore their location.
- A fault never manifest it self, unless a certain input is introduced to the software.
- faults are equally likely to cause a failures.

3.3.2 Goel and Okumoto NHPP model [1979]

This model [5] belongs to the class of NonHomogenous Poisson Process models where the failure intensity is a continuous function and depends only on time.

Assumptions

GO1 The initial number of software faults is treated as random variable (in JM model it is an unknown, fixed constant),

GO2 The time between failures $(i-1)$ and i depends on the time to failure $(i-1)$ (Independent of each other in the JM model),

GO3 Faults are removed immediately from the software and no new faults are introduced.

Model

The failure intensity called also the error detection rate is given by [Figure 3.2]:

$$\forall t, \lambda_t = ab\exp(-bt) \quad \text{Where} \quad \begin{matrix} a \in \Re_+^* \\ b \in \Re_+ \end{matrix} \tag{3.5}$$

a is the expected number of failures to be eventually observed.
b is the fault detection rate per fault.

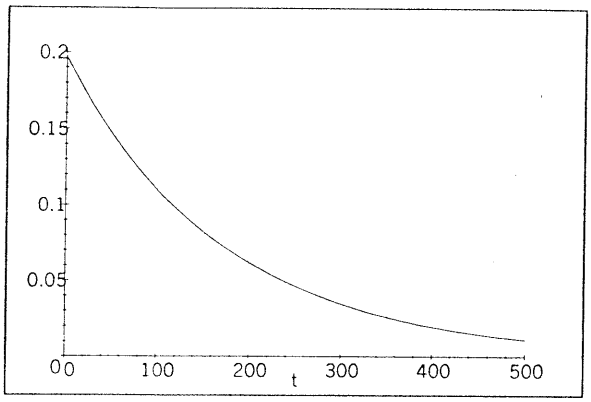


Figure 3.2: Failure intensity for the G-O model [4, Page 300]

3.3.3 S-shaped model [1983]

The model proposed by Goel and Okumoto is characterized by a mean value function which has exponential growth for the number of detected software faults. However, the observed growth curve is sometimes S-shaped according to Yamada et al [17]. They propose a model with a mean value function showing an S-shaped growth curve.

Assumptions

SS1 A software system is subject to failures at random times caused by faults present in the system.

SS2 The time between failures $(i-1)$ and i depends on the time to failure $(i-1)$.

SS3 Each time a failure occurs, the fault which cause it is immediately removed from the software and no new faults are introduced.

Model

At any time, the failure intensity for the S-shaped model is given by [Figure 3.3]:

$$\forall t, \lambda_t = ab^2 t \exp(-bt) \quad \text{Where} \quad \begin{matrix} a \in \Re_+^* \\ b \in \Re_+ \end{matrix} \tag{3.6}$$

a is the expected number of failures to be eventually observed.
b is the fault detection rate per fault.

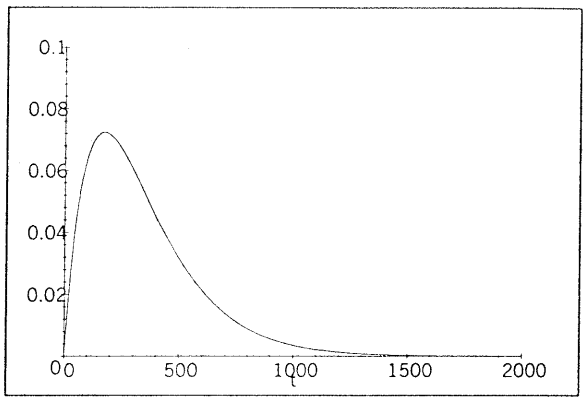


Figure 3.3: The S-shaped failure intensity

Remark 7 *The S-shaped failure intensity picks at point:*

$$\left(t = \frac{1}{b}, \lambda(t) = ab\exp^{-1}\right)$$

3.3.4 Log-logistic model

As the S-shaped model described above, the log-logistic growth reliability model [3] have an increasing / decreasing failure intensity or also an S-shaped mean value function.

Assumptions

LL1 The log-logistic model assume that software system is subject to failures at random times caused by faults present in the system.

LL2 The time between failures $(i-1)$ and i depends on the time to failure $(i-1)$.

LL3 Each time a failure occurs, the fault which cause it is immediately removed from the software and no new faults are introduced.

LL4 The rate at which faults manifest themselves exhibit an increasing / decreasing behavior.

Model

At any time, the log-logistic failure intensity is given by [Figure 3.4]:

$$\forall t, \lambda_t = a\frac{pk\,(pt)^{k-1}}{\left[1+(pt)^k\right]^2} \quad \text{Where} \quad a \in \Re_+^* \qquad (3.7)$$
$$p, k \in \Re_+$$

a is the expected number of failures to be eventually observed.
p is the fault detection rate.
k shifts the curve in time but does not affect its shape.

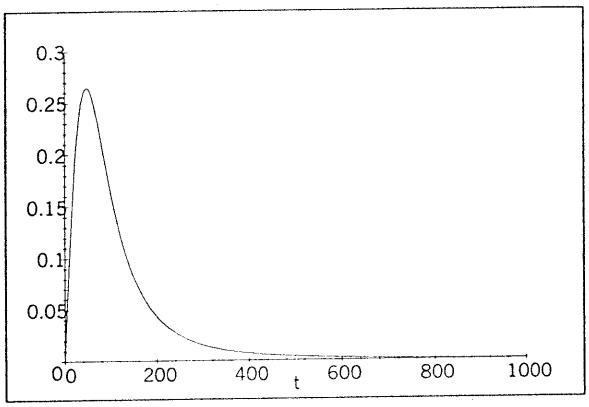


Figure 3.4: The log-logistic failure intensity

Remark 8 *The log-logistic failure intensity picks at point:*

$$\left(t = \frac{1}{p}\left[\frac{k-1}{1+k}\right]^{\frac{1}{k}}, \lambda(t) = \frac{ap\,(k-1)^{\frac{k-1}{k}}}{4k\,(k+1)^{\frac{3k-1}{k}}}\right)$$

3.4 Estimation of models parameters

The model parameters described above need to be estimated. These unknown parameters are N and Φ in the JM model, a and b in the Goel-Okumoto model and S-shaped model, and eventually a, p and k in the log-logistic model. The most commonly methods used to estimate these parameters are the maximum likelihood and the least squares.

3.4.1 The maximum likelihood method

To illustrate the maximum likelihood method to estimate model parameters [18], we use for example the case where the model belongs to NHPP class. We assume that the data set is given as the times between software failures X_i.

Let T_i be the cumulative time so it is given by:

$$T_i = \sum_{i=1}^{n} X_i \tag{3.8}$$

and β a vector for $(w+1)$ unknown parameters.

λ_t is the failure intensity and its derivative $m(t)$ is the mean value function of the NHPP model.

The likelihood function is the joint pdf of $T_1, T_2, ..., T_n$ and it is given by [17]:

$$L(\beta/T) = [\prod_{i=1}^{n} \lambda(T_i)] \exp(-m(T_n)) \tag{3.9}$$

Taking the logarithm of $L(\beta/T)$ yields to

$$l(\beta/T) = \sum_{i=1}^{n} \ln \lambda(T_i) - m(T_n) \tag{3.10}$$

$\hat{\beta}$ is then given by solving the likelihood equations:

$$\frac{\partial l(\beta/T)}{\partial \beta_i} = 0;\ i = 0, 1, 2, ..., w \tag{3.11}$$

However, estimators are not given directly, the likelihood equations have to be solved numerically by using methods like Newton-Raphson algorithm [10].

3.4.2 Newton-Raphson algorithm

Let $V(\beta)$ be a $(w+1) * 1$ column vector with elements given by:

$$V_i(\beta) = \frac{\partial \ln L(\beta/T)}{\partial \beta_i};\ i = 0, 1, 2, ..., w \tag{3.12}$$

and $W(\beta)$ be a $(w+1) * (w+1)$ matrix with elements:

$$W_i(\beta) = \frac{\partial^2 \ln L(\beta/T)}{\partial \beta_i \partial \beta_j};\ i, j = 0, 1, 2, ..., w \tag{3.13}$$

ALGORITHM *Newton_Raphson*
BEGIN
β_init $\leftarrow$ *Trial value*
$\hat{\beta} \leftarrow \beta\_init - INVERSE(W(\beta\_init)) * V(\beta\_init)$

WHILE ($|\hat{\beta} - \beta\_init| > \varepsilon$) LOOP
$\beta\_init \leftarrow \hat{\beta}$
$\hat{\beta} \leftarrow \beta\_init - INVERSE(W(\beta\_init)) * V(\beta\_init)$
END
END

As soon as $\hat{\beta}$ the estimated column vector of β, is calculated, the *fitted model* is given by substituting the parameters of $\hat{\beta}$ in the chosen model.

3.5 Application Example

3.5.1 Software failure data

The below data table is taken from the United States Navy Fleet Computer Programming Center and consists of faults detected in the development of the Navel Tactical Data System ($NTDS$) [14].

Table 3.1: NTDS Data

Failure rank y	Time between failures x_i (days)	Time to the $i\ th$ failure $t_k = \sum_{i=1}^{n} x_i$
1	9	9
2	12	21
3	11	32
4	4	36
5	7	43
6	2	45
7	5	50
8	8	58
9	5	63
10	7	70
11	1	71
12	6	77
13	1	78
14	9	87
15	4	91
16	1	92
17	3	95
18	3	98
19	6	104
20	1	105
21	11	116
22	33	149
23	7	156
24	91	247
25	2	249
26	1	250

3.5.2 Goel and Okumoto model

Laplace test

Using the *NTDS* data, the Laplace factor can be calculated as:

$$u = \frac{\sum_1^{26} t_i - 26\frac{250}{2}}{\sqrt{26\frac{250^2}{12}}} = -2.059$$

λ_α and μ_α are given from the table of the cumulative normal distribution with $\alpha = 3\%$ and are respectively: $\lambda_\alpha = -1.89$ and $\mu_\alpha = 1.89$.

$$u = -2.059 < \lambda_\alpha = 1.645$$

With a probability of error $\alpha = 3\%$ we conclude a reliability growth, so we can apply the above models to the NTDS data.

The fitted model

If we integrate *Eq.* (3.5) we get the mean value function of Goel Okumoto model:

$$m(t) = a(1 - \exp(-bt)) \tag{3.14}$$

To estimate the model's parameters a and b we substitute *Eq.*(3.5) and *Eq.*(3.14) to *Eq.*(3.10) [5]:

$l(a, b/t) = n \ln a + n \ln b - b \sum_{i=1}^{n} t_i - a(1 - e^{-bt_n})$

Taking the derivative of l respect to a and b yields to [5]

$$\begin{cases} \dfrac{\partial l}{\partial a} = \dfrac{n}{a} - \left(1 - e^{-bt_n}\right) \\ \dfrac{\partial l}{\partial b} = \dfrac{n}{b} - \sum_{i=1}^{n} t_i + at_n \, e^{-bt_n} \end{cases}$$

Maximizing the above equations give:

$$\begin{cases} \dfrac{n}{a} = 1 - e^{-bt_n} \\ \dfrac{n}{b} = \sum_{i=1}^{n} t_i - at_n \, e^{-bt_n} \end{cases}$$

Applying the Newton-Raphson algorithm to the above equations we get:

$$\begin{cases} \hat{a} = 34 \\ \hat{b} = 0.00579 \end{cases}$$

So the estimated mean value function and failure intensity are respectively [Figure 3.5 and Figure 3.6]:

$$\hat{m}(t) = 34(1 - \exp(-0.00579t))$$

and

$$\hat{\lambda}(t) = 0.19686 \exp(-0.00579t)$$

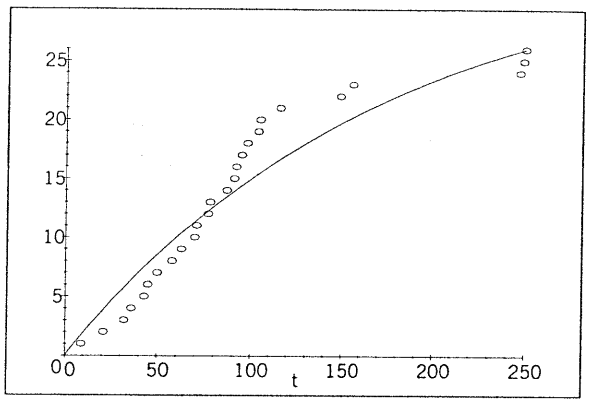


Figure 3.5: The actual and the fitted $m(t)$

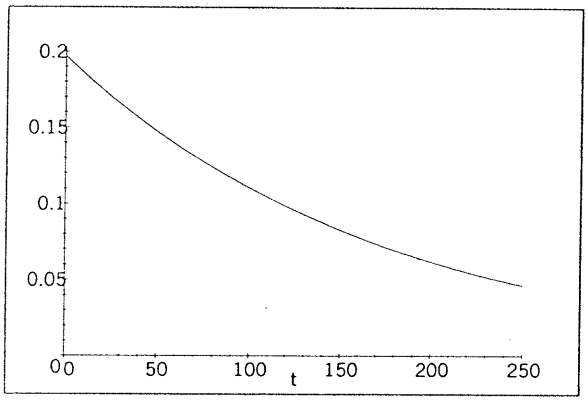


Figure 3.6: The fitted failure intensity

The residual function

At time t, the number of remaining faults in the software is given by substituting *Eq.* (3.14) to *Eq.* (2.9):

$$\forall\, t,\, Q(t) = a \exp(-bt) \tag{3.15}$$

The estimated expected number of residual faults is:

$$\hat{Q}(t) = 34 \exp(-0.00579t)$$

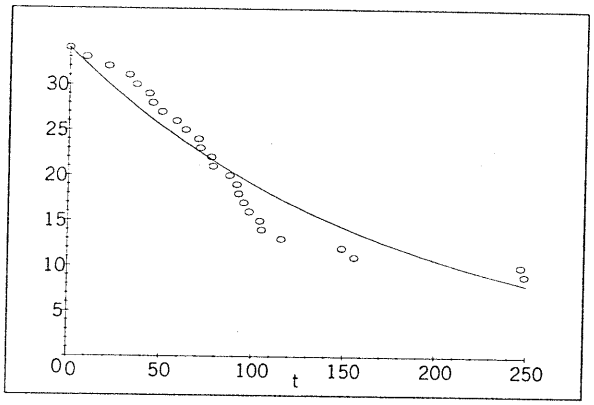


Figure 3.7: The actual and fitted residual function

If $n = 26$ faults have been found by time t then, $\hat{Q}(t) = a - n = 34 - 26 = 8$ faults, which is equal to the actual number of residual faults.

Reliability function

The software reliability for the Goel and Okumoto model is given by substituting *Eq.* (3.14) to *Eq.* (2.5) :

$$R(x/t) = \exp\left[-a\left\{\exp\left(-bt\right) - \exp\left(-b\left(t+x\right)\right)\right\}\right] \tag{3.16}$$

The estimated software reliability is then given by [Figure 3.8] :

$$\hat{R}(x/t_{26} = 250) = \exp[-34\{\exp(-1.4475) - \exp(-0.00579(250+x))\}]$$

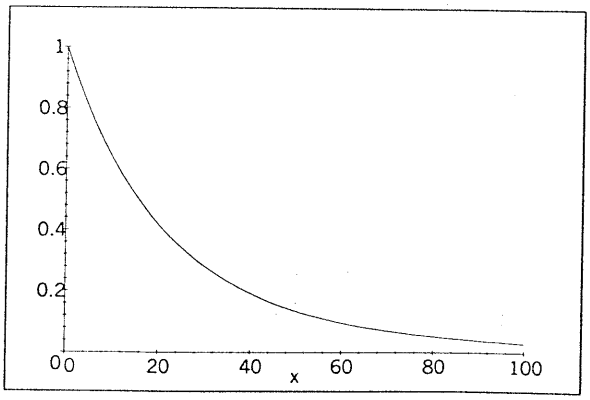


Figure 3.8: The fitted software reliability

After $t = 250$ days, the conditional reliability decreases because faults are not removed in totality from the software which cause software degradation and hence a reliability decay.

Hazard rate

The hazard rate is given by substituting $Eq.$ (3.5) and $Eq.$ (3.14) into $Eq.$ (2.8) so we get:

$$Z(t) = b \tag{3.17}$$

Which is constant. The estimated hazard rate for the Goel and Okumoto model is [Figure 3.9]:

$$\hat{Z}(t) = 0.00579$$

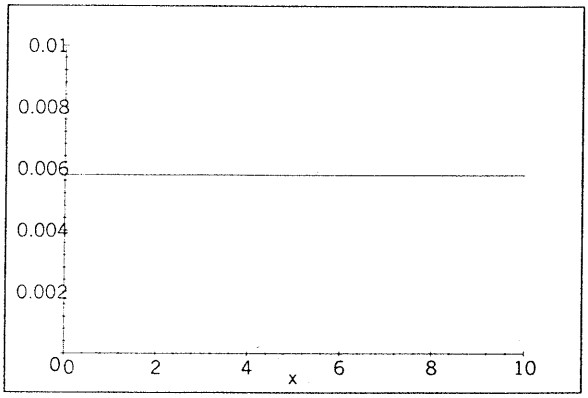


Figure 3.9: The hazard rate

3.5.3 S-shaped model

The fitted model

If we integrate $Eq.$(3.6) we get the mean value function of the S-shaped model, which is given by:

$$m(t) = a\left[1 - (1 + bt)\exp(-bt)\right] \tag{3.18}$$

To estimate the models's parameters a and b we substitute $Eq.$ (3.6) and $Eq.$(3.19) to $Eq.$(3.10) to get [17]:

$$l(a, b/t) = n\ln a + 2n\ln b + \sum_{i=1}^{n} \ln t_i - b\sum_{i=1}^{n} t_i$$

$$-a\left[1 - (1 + bt_n)\exp(-bt_n)\right]$$

Taking the derivative of l respect to a and b yields to [17]:

$$\begin{cases} \dfrac{\partial l}{\partial a} &= \dfrac{n}{a} - [1 - (1 + bt_n)\exp(-bt_n)] \\ \dfrac{\partial l}{\partial b} &= \dfrac{2n}{b} - \sum_{i=1}^{n} t_i - abt_n^2 \exp(-bt_n) \end{cases}$$

Maximizing the above equations give:

$$\begin{cases} \dfrac{n}{a} &= [1 - (1 + bt_n)\exp(-bt_n)] \\ \dfrac{2n}{b} &= -\sum_{i=1}^{n} t_i - abt_n^2 \exp(-bt_n) \end{cases}$$

Applying the Newton-Raphson algorithm to the above equations we get:

$$\begin{cases} \hat{a} &= 27.5 \\ \hat{b} &= 0.018579 \end{cases}$$

So the fitted mean value function and the failure intensity are respectively [Figure 3.10 and Figure 3.11]:

$$\hat{m}(t) = 27.5\,[1 - (1 + 0.018579t)\exp(-0.018579t)]$$

and

$$\hat{\lambda}(t) = 9.49\ 10^{-3} t \exp(-0.018579t)$$

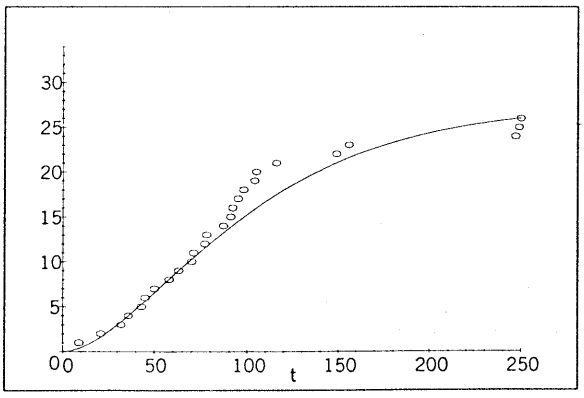


Figure 3.10: The actual and the fitted $m(t)$

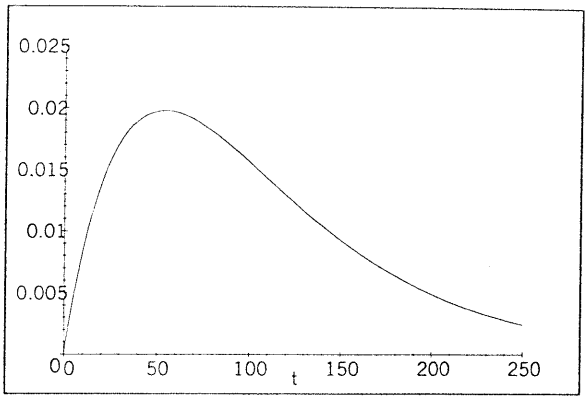


Figure 3.11: The fitted failure intensity

The residual function

At time t, the number of remaining faults in the software is given by substituting *Eq.* (3.19) to *Eq.* (2.9):

$$Q(t) = a(1 + bt)\exp(-bt) \tag{3.19}$$

The estimated residual function for the S-shaped model is given by:

$$\hat{Q}(t) = 27.5(1 + 0.018579t)\exp(-0.018579t)$$

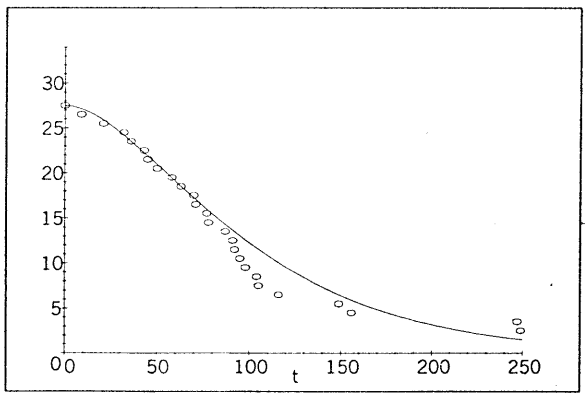


Figure 3.12: The actual and fitted residual function

For a total number of faults detected in software, $n = 26$, the estimated number of faults remaining in the software is given by $\hat{Q} = 27.5 - 26 = 1.5$ faults.

So we can conclude that the S-shaped model under estimate the expected number of residual faults.

Reliability function

The software reliability for the S-shaped model is given by substituting *Eq.* (3.19) to *Eq.* (2.5) :

$$R(x/t) = \exp\left[-a\left\{(bt+1)\exp\left(-bt\right) - (b(t+x)+1)\exp(-b(t+x))\right\}\right] \tag{3.20}$$

The estimated software reliability for the S-shaped model is given by [Figure 3.13]:

$$R(x/t_{26} = 250) = \exp\left\{-1.4907355 + (1.49207355 + 0.004910757x)\right.$$
$$\left.\exp\left(-0.018579x\right)\right\}$$

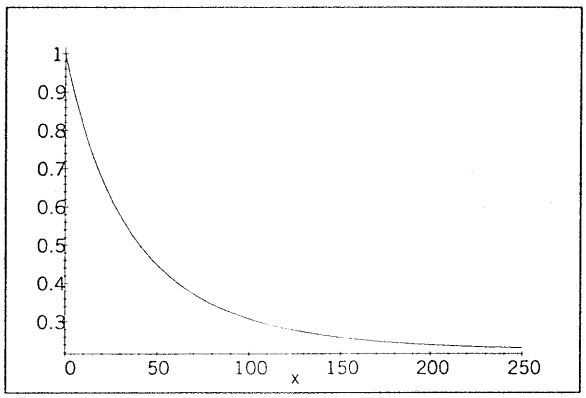


Figure 3.13 : Reliability function for the S-shaped model

Which shows degradation after $t = 250$, since the software still contains faults.

Hazard rate

The hazard rate is given by substituting *Eq.* (3.6) and *Eq.* (3.19) into *Eq.* (2.8) so we get:

$$Z(t) = \frac{b^2}{1+bt} \tag{3.21}$$

The estimated hazard rate for the S-shaped model is [Figure 3.14]:

$$\hat{Z}(t) = \frac{3.4519 \times 10^{-4}t}{1 + 1.8579 \times 10^{-2}t}$$

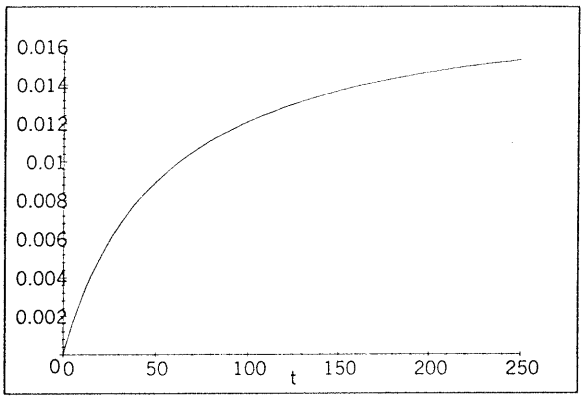


Figure 3.14: The hazard rate for the S-shaped model

3.5.4 Log-logistic model

The fitted model

If we integrate *Eq.*(3.7) we get the mean value function of the log-logistic model, which is given by:

$$m(t) = a\frac{(pt)^k}{1+(pt)^k} \tag{3.22}$$

To estimate the models's parameters a, p and k we substitute *Eq.*(3.7) and *Eq.* (3.24) to *Eq.* (3.10) to get [5]:

$$l(a,p,k/t) = n\ln a + nk\ln p + n\ln k - a\frac{(pt_n)^k}{1+(pt_n)^k} + (k-1)\sum_{i=1}^{n}\ln t_i$$
$$-2\sum_{i=1}^{n}\ln\left(1+(pt_i)^k\right)$$

Taking the derivative of l respect to a, p and k yields to [5]:

$$\begin{cases} \dfrac{\partial l}{\partial a} = \dfrac{n}{a} - \dfrac{(pt_n)^k}{1+(pt_n)^k} \\ \dfrac{\partial l}{\partial p} = \dfrac{n}{p} - \dfrac{2(1+(pt_n)^k)\sum_{i=1}^{n}\dfrac{(pt_i)^k}{1+(pt_i)^k}}{(pt_n)^k} \\ \dfrac{\partial l}{\partial k} = \dfrac{n}{k} - \dfrac{n\ln pt_n}{[1+(pt_n)^k]^2} + n\ln p + \sum_{i=1}^{n}\ln t_i - 2\sum_{i=1}^{n}\dfrac{(pt_i)^k\ln pt_i}{1+(pt_i)^k} \end{cases}$$

Maximizing the above equations give:

$$\begin{cases} \dfrac{n}{a} = \dfrac{(pt_n)^k}{1+(pt_n)^k} \\ \dfrac{n}{p} = \dfrac{2(1+(pt_n)^k)\sum_{i=1}^{n}\dfrac{(pt_i)^k}{1+(pt_i)^k}}{(pt_n)^k} \\ \dfrac{n}{k} = \dfrac{n\ln pt_n}{[1+(pt_n)^k]^2} - n\ln p - \sum_{i=1}^{n}\ln t_i + 2\sum_{i=1}^{n}\dfrac{(pt_i)^k \ln pt_i}{1+(pt_i)^k} \end{cases}$$

Applying the Newton-Raphson algorithm to the above equations we get:

$$\begin{cases} \hat{a} = 28.5 \\ \hat{p} = 0.012 \\ \hat{k} = 1.9935 \end{cases}$$

So the fitted mean value function and the failure intensity are respectively [Figure 3.15 and Figure 3.16]:

$$\hat{m}(t) = 28.5 * \frac{(0.012t)^{1.9935}}{1+(0.012t)^{1.9935}} \tag{3.23}$$

and

$$\hat{\lambda}(t) = 28.5 * \frac{2.9544 \times 10^{-4} t^{.9935}}{(1.0 + 1.482 \times 10^{-4} t^{1.9935})^2} \tag{3.24}$$

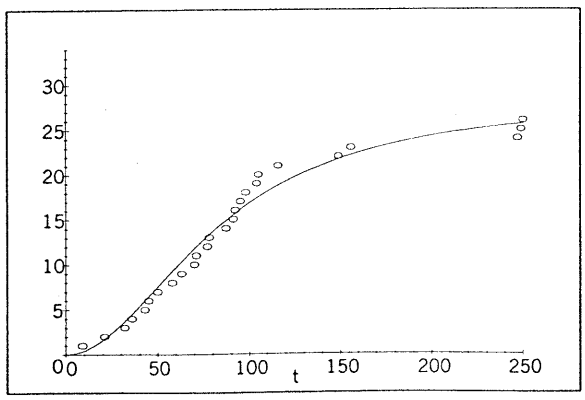


Figure 3.15: The actual and the fitted $m(t)$

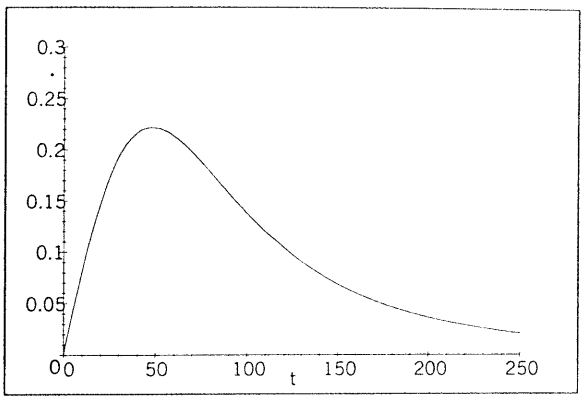


Figure 3.16: The fitted failure intensity

The residual function

The estimated residual function for the log-logistic model is given by:

$$Q(t) = \frac{a}{1 + (pt)^k} \tag{3.25}$$

The estimated expected number of residual faults is:

$$\hat{Q}(t) = \frac{28.5}{1 + (0.012t)^{1.9935}}$$

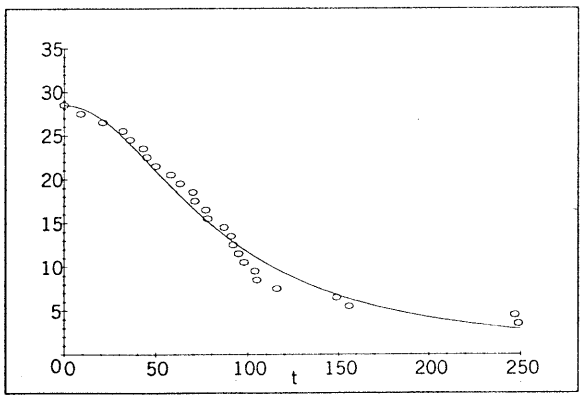


Figure 3.17: The residual function

For a total number of faults detected in software, $n = 26$, the estimated number of faults remaining in the software is given by $\hat{Q} = 28.5 - 26 = 2.5$ faults.

Reliability function

The software reliability for the log-logistic model is given by substituting *Eq.* (3.24) to *Eq.* (2.5) :

$$R(x/t) = \exp\left[-a\left\{\frac{(pt)^k}{1+(pt)^k} - \frac{p^k(t+x)^k}{1+p^k(t+x)^k}\right\}\right] \tag{3.26}$$

The estimated software reliability for the log-logistic model is given by [Figure 3.18]:

$$\hat{R}(x/t_{26}=250) = \exp\left\{-34\left(\frac{0.012^{1.9935}(250+x)^{1.9935}}{1+0.012^{1.9935}(250+x)^{1.9935}} - \frac{(0.012*250)^{1.9935}}{1+(0.012*250)^{1.9935}}\right)\right\}$$

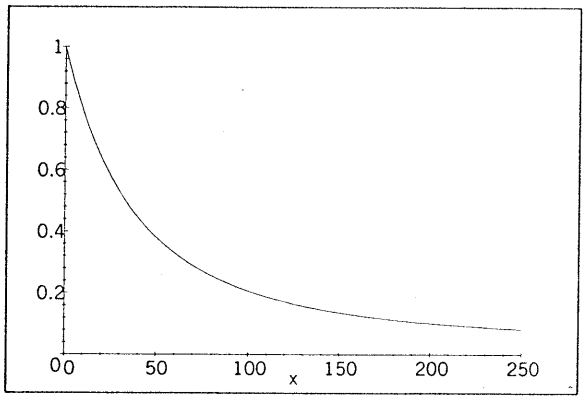


Figure 3.18: Log-logistic Reliability function

Which shows a software degradation after $t = 250$, since it still contains faults.

Hazard rate

The hazard rate is given by substituting *Eq.* (3.7) and *Eq.* (3.24) into *Eq.* (2.8) so we get:

$$Z(t) = \frac{pk(pt)^{k-1}}{1+(pt)^k} \tag{3.27}$$

The estimated hazard rate for the log-logistic model is [Figure 3.19]:

$$\hat{Z}(t) = \frac{0.012 * 1.9935\,(0.012 * t)^{1.9935-1}}{1 + (0.012 * t)^{1.9935}}$$

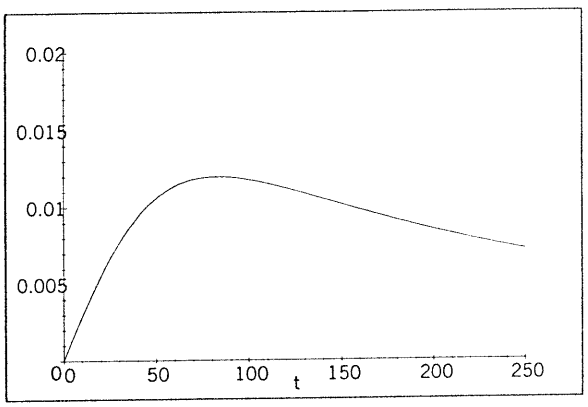


Figure 3.19: Log-logistic hazard rate

3.6 Goodness of fit

As criteria for model comparison, we choose the goodness of fit [18], which is the sum of squares of the difference between the actual cumulative number of faults y_i and the estimated number of faults $\hat{y}_i$ $(= \hat{m}_i(t))$ detected in the time interval $[0, t_i]$ where $i = 1, 2, .., n$.

$$GF = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{3.28}$$

The goodness of fit determines the capability of a model to reproduce the observed failure behavior of the software. The lower the error sum of squares, the better is the goodness of fit.

Table 3.2: Goodness of Fit with NTDS Data

	$\hat{y}_i$		
y_i	$G-O$	$S-shaped$	$Log-logistic$
1	1.726	0.3442	0.333
2	3.893	1.621	1.716
3	5.750	3.303	3.682
4	6.397	3.988	4.503
5	7.493	5.247	6.013
6	7.798	5.6161	6.454
7	8.546	6.548	7.562
8	9.698	8.051	9.315
9	10.392	8.984	10.377
10	11.329	10.287	11.798
11	11.460	10.448	11.994
12	12.230	11.514	13.129
13	12.356	11.688	13.312
14	13.455	13.209	14.861
15	13.925	13.856	15.497
16	14.041	14.015	15.651
17	14.385	14.483	16.101
18	14.722	14.941	16.533
19	15.381	15.822	17.346
20	15.488	15.964	17.476
21	16.630	17.445	18.785
22	19.651	20.995	21.689
23	20.221	21.591	22.153
24	25.864	25.598	25.569
25	25.958	25.985	25.611
26	26.005	26.008	25.632
$GF = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$	129.587	71.728	33.439

We can conclude that the log-logistic model fits better the $NTDS$ data than the Goel Okumoto model and the S-shaped model, since it has the lower error sum of squares.

3.7 Conclusion

In this chapter, we have provided a review and an evaluation of software reliability models. Trend analysis such as Laplace test is given to choose the appropriate model. An application of Goel and Okumoto, S-shaped,

and log-logistic models to $NTDS$ data is studied, where some measures (Reliability, MTTSF,...) of interest are computed. As criteria for model comparison, the goodness of fit is evaluated, indicating that the log-logistic model provides a good fit to the observed failure phenomenon than the others.

Chapter 4

Multi-log-logistic Software Reliability Model

■ AIM

The aim of this chapter is to develop a multi-log-logistic model which exhibit an iterative increasing / decreasing failure intensity instead of a monotonic decreasing (Goel-Okumoto model) or increasing / decreasing (log-logistic model) nature. The goodness of fit evaluated for multi-log-logistic and log-logistic models prove that multi-log-logistic fits better the failure process. A model classification scheme is then deduced.

■ CONTENT

4.1 Introduction
4.2 Trend analysis
4.3 Multi-log-logistic model
4.4 Application example
4.5 Model classification
4.6 Conclusion

4.1 Introduction

In the previous chapter, we have seen that the failure intensity $\lambda(t)$ can exhibit a decreasing or an increasing / decreasing behavior.

Models with a decreasing failure intensity such as Goel and Okumoto assume that a detected fault is immediately eliminated from the software, without introducing others, which cause an improvement of the software since test starting, and hence reliability growth. The monotonic failure intensity is due to the fact that faults have the same rate or probability to be detected (constant hazard rate). After test period reliability decreases because not all faults are removed from the software during testing.

On the other hand increasing / decreasing failure intensity models such as logistic [11], log-logistic and S-shaped assume that some faults are liable than others to cause failure. Critical and serious faults are detected before moderate and tolerable faults -since they have greater probability of appearance-, which cause an increasing failure intensity or equivalently software degradation and reliability decay. However, their correction will produce a greater decrease to failure intensity, therefore software improvement and reliability growth.

However, in practice the fault correction process introduces others, which is not taken into consideration in the above models.

At the beginning of test process, critical faults manifest themselves because they have a greater rate of appearance, their correction will improve software, but fault introduced will cause software degradation. Serious faults will then appear and cause software degradation, after correction, faults are introduced causing software degradation, and so on...

This process can be repeated N times, where N is the number of faults classes (see chapter1). This behavior is described by the below pseudocode:

Detection of first class faults (Critical fault for instance)
ITERATE
Step1: Software degradation
Step2: Fault correction
Step3: Software improvement
Step4: Next class of faults
End

Remark 9 *Faults are classified by severity.*

Remark 10 *The number of classes depend on the software specification.*

According to the above algorithm, the failure intensity can be seen as the log-logistic failure intensity followed by a log-logistic failure intensity. This failure intensity can be called a *multi-log-logistic* failure intensity.

To analyze the fault behavior during testing period we use a trend analysis with the set of data described in the previous chapter.

4.2 Trend Analysis

To obtain various measures of interest, software reliability studies are usually based on the application of reliability growth models. In order to determine which reliability growth model to use, trend tests are very useful. The most frequently used is the *Laplace test* described previously.

Laplace test shows an iterative increasing / decreasing trend so the failure intensity exhibit an iterative increasing /decreasing nature which cannot be capture by the existing software reliability models, thus we propose a multi-log-logistic model.

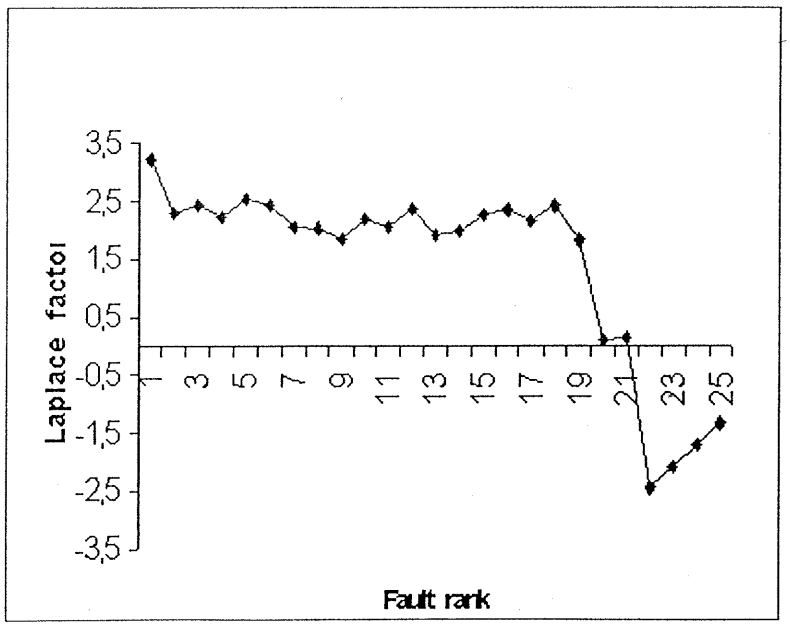


Figure 4.1 : Laplace test with NTDS data [5, Page 39]

4.3 Multi-log-logistic model

First, let us consider the case of a software with a mean value function composed of two phases, then we will extend this to an arbitrary number of phases.

4.3.1 Bi-log-logistic

Assume a software contains only faults belong to critical and serious class.

Let F_c be the set of faults belong to critical class.

Let F_s be the set of faults belong to serious class.

When test starts, assume that faults in F_c are fully detected with a simple log-logistic mean value function $m_c(t)$. Next faults in F_s manifest themselves with a simple log-logistic mean value function $m_s(t)$. So in the bi-log-logistic model the detection of faults (in F_c and F_s) is the sum of two mean value functions $m_c(t)$ and $m_s(t)$, each of which is a three parameters log-logistic. Hence the mean value function of software is:

$$m(t) = m_c(t) + m_s(t) \tag{4.1}$$

Where $m_c(t) = a_c \frac{(\lambda_c t)^{k_c}}{1 + (\lambda_c t)^{k_c}}$ and $m_s(t) = a_s \frac{(\lambda_s t)^{k_s}}{1 + (\lambda_s t)^{k_s}}$

Taxonomy of bi-log-logistic curves [11, Page 9-10]

Phases often overlap in time; depending on the order and magnitude of the overlap, the aggregate mean value function can take different classifications, which can be useful to describe software behavior. Figure 4.2 shows a taxonomy of bi-log-logistic process with their decomposition to illustrate the overlap of phases.

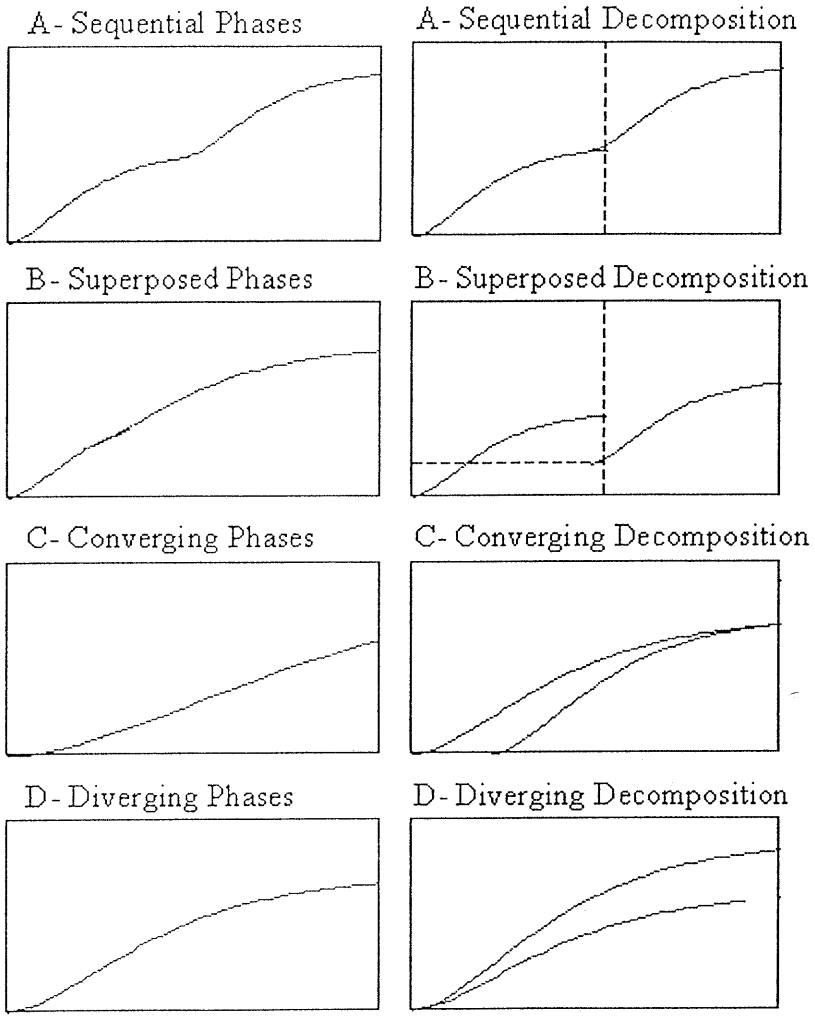


Figure 4.2: Bi-log-logistic Taxionomy [11, Page 10]

The above taxonomy can be fitted to software behavior with two classes of faults described previously F_c and F_s.

State A: Bi-log-logistic model with sequential phases:

The second mean value function $m_s(t)$ does not start growing until the first mean value function $m_c(t)$ nearly reached its saturation a_c. This state can be reached if we assume a software which pauses between $m_c(t)$ and $m_s(t)$. In other words all faults belong to critical class are fully detected, next we detect faults belong to serious class.

State B: Bi-log-logistic model with superposed phases:

The second mean value function $m_s(t)$ begins growing when the first mean value function $m_c(t)$ reached a certain level of saturation. When critical faults reach a certain level,serious faults begin growing (detected). This

state is not realistic because critical faults still have the greater probability to manifest themselves before serious faults.

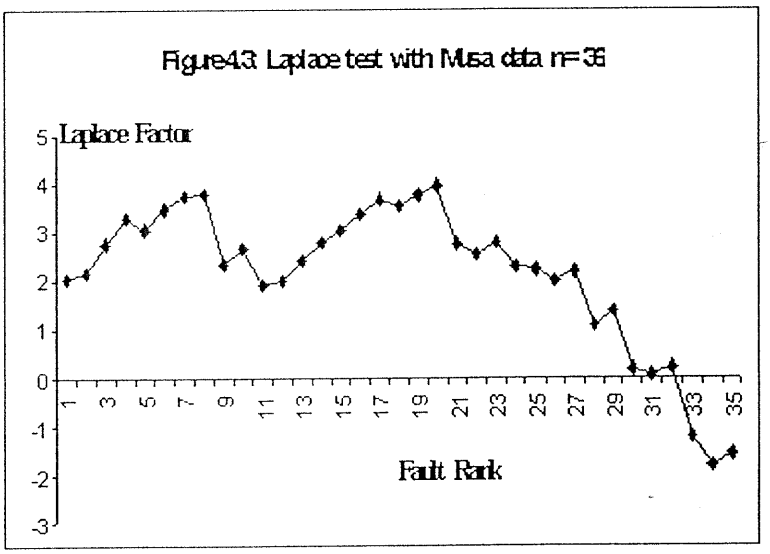
State C: Bi-log-logistic model with converging phases:

The first mean value function $m_c(t)$ is joined by a faster mean value function $m_s(t)$. $m_c(t)$ and $m_c(t)$ culminate at about the same time. Often a late phase $[m_s(t)]$ learns from the experiences of an early phase $[m_c(t)]$. The correction critical faults will give some skills and experiences to tester team which help them to detect and correct serious faults rapidly.

State D: Bi-log-logistic model with diverging phases:

All faults begin at the same time but detected with higher probability manifest themselves before faults with lower probability.

Example 19 *In figure 4.3, we illustrate a bi-log-logistic model with Musa* 14c *data for* $n = 36$.



4.3.2 Generalization of the log-logistic model [11, Pages 14,18]

If we assume that we have more than two fault classes, we generalize the bi-log-logistic model to multi-log-logistic model where the entire mean value function of the software is composed of n simple log-logistic mean value function:

$$m(t) = \sum_{i=1}^{n} m_i(t) \tag{4.2}$$

Where $m_i(t) = a_i \frac{(\lambda_i t)^{k_i}}{1+(\lambda_i t)^{k_i}}$

Let D be the two dimensional space in which our data set exits.

$D = \{(t_1, d_1), .., (t_i, d_i), .. (t_m, d_m)\}$

Where t_i represents testing time and d_i represents faults rank.

Our aim is to fit a multi-log-logistic model curve of N fault classes to D. We will require $3 * N$ parameters, represented as $3 * N$ matrix M, where the ith row describes the ith fault class.

$$M = \begin{bmatrix} a_1 & p_1 & k_1 \\ \cdots & \cdots & \cdots \\ a_i & p_i & k_i \\ \cdots & \cdots & \cdots \\ a_N & p_N & k_N \end{bmatrix}$$

Thus a multi-log-logistic model can be specified by a mean value function and failure intensity represented as:

$$m(t) = \sum_{j=1}^{N} a_j \frac{(p_j t)^{k_j}}{1+(p_j t)^{k_j}} \tag{4.3}$$

and

$$\lambda(t) = \sum_{j=1}^{N} a_j \frac{p_j k_j (p_j t)^{k_j - 1}}{\left[1+(p_j t)^{k_j}\right]^2} \tag{4.4}$$

As we see, it is very hard to estimate multi-log-logistic model with $3 * N$ parameters; to overcome this problem, we use *decomposition principal.*

4.3.3 Decomposition Principal

As a first decomposition we apply a log-logistic model to data and we evaluate the goodness of fit. If we need to improve the results gave by this model we propose a second decomposition to the set of data ($N = 2$) and we apply a bi-log-logistic model, then we evaluate the goodness of fit. If the sum of error square is lower than the log-logistic we can accept the bi-log-logistic as fitted model, if it is not the case we can propose a new decomposition ($N = 3$),

evaluate the goodness of fit and compare it to the log-logistic. This process can be repeated until we cannot decompose the data set or we are satisfied, i.e. we reach a lower goodness of fit factor.

The decomposition of failure phenomenon is based on the fault behavior generated by Laplace test. Since if we plot a step-by-step Laplace test we see that the fault behavior exhibit an iterative increasing / decreasing nature instead of monotonic decreasing or increasing / decreasing nature in the other models.

Therefore, the decomposition principal is based on the trend analysis given by Laplace test. We have to find the subset data D_j where we visualize a clear increasing / decreasing fault behavior in the Laplace test.

We note $D_j = \{(t_j, d_j) \text{ with } t_j \in x_j\}$, where each D_j is modelized by a log-logistic model and x_j is the time interval in which D_j is represented..

4.3.4 Parameters Estimation

¨For each decomposition j, we use the maximum likelihood function of the simple log-logistic model (described in the previous chapter), to estimate the matrix of parameters M, of the multi-log-logistic model.

$$\begin{aligned} l_j\left(a_j, p_j, k_j/t\right) = \; & n \ln a_j + nk \ln p_j + n \ln k_j - a_j \frac{(p_j t_n)^{k_j}}{1 + (p_j t_n)^{k_j}} \\ & + (k_j - 1) \sum_{i=1}^{n} \ln t_i - 2 \sum_{i=1}^{n} \ln \left(1 + (p_j t_i)^{k_j}\right) \end{aligned} \tag{4.5}$$

Taking the derivative of l respect to a_j, p_j and k_j yields to:

$$\begin{cases} \dfrac{\partial l}{\partial a_j} = \dfrac{n}{a_j} - \dfrac{(p_j t_n)^{k_j}}{1 + (p_j t_n)^{k_j}} = 0 \\ \dfrac{\partial l}{\partial p_j} = \dfrac{n}{p_j} - \dfrac{2(1 + (p_j t_n)^{k_j}) \sum_{i=1}^{n} \dfrac{(p_j t_i)^{k_j}}{1 + (p_j t_i)^{k_j}}}{(p_j t_n)^{k_j}} = 0 \\ \dfrac{\partial l}{\partial k} = \dfrac{n}{k_j} - \dfrac{n \ln p_j t_n}{[1 + (p_j t_n)^{k_j}]^2} + n \ln p_j + \sum_{i=1}^{n} \ln t_i - 2 \sum_{i=1}^{n} \dfrac{(p_j t_i)^{k_j} \ln p_j t_i}{1 + (p_j t_i)^{k_j}} = 0 \end{cases}$$

Solving the above equations give:

$$\begin{cases} \dfrac{n}{a_j} = \dfrac{(p_j t_n)^{k_j}}{1+(p_j t_n)^{k_j}} \\ \dfrac{n}{p_j} = \dfrac{2(1+(p_j t_n)^{k_j})\sum_{i=1}^{n} \dfrac{(p_j t_i)^{k_j}}{1+(p_j t_i)^{k_j}}}{(p_j t_n)^{k_j}} \\ \dfrac{n}{k_j} = \dfrac{n \ln p_j t_n}{[1+(p_j t_n)^{k_j}]^2} - n \ln p_j - \sum_{i=1}^{n} \ln t_i + 2\sum_{i=1}^{n} \dfrac{(p_j t_i)^{k_j} \ln p_j t_i}{1+(p_j t_i)^{k_j}} \end{cases}$$

Solving the above three non-liner equations simultaneously, we obtain the point estimates of parameters a_j, p_j and k_j.

Consequently the fitted mean value function and the failure intensity of the multi-log-logistic model are given by:

$$\hat{m}(t) = \sum_{j=1}^{N} \hat{a}_j \frac{(\hat{p}_j t)^{\hat{k}_j}}{1+(p_j t)^{k_j}} \tag{4.6}$$

and

$$\hat{\lambda}(t) = \sum_{j=1}^{N} \hat{a}_j \frac{\hat{p}_j \hat{k}_j (\hat{p}_j t)^{\hat{k}_j - 1}}{\left[1+(\hat{p}_j t)^{\hat{k}_j}\right]^2} \tag{4.7}$$

4.4 Application example

As an example, we apply the multi-log-logistic model to $NTDS$ data

Bi-log-logistic model

Apply Laplace test to data $NTDS$ [Figure 4.1]

Propose a data decomposition based on the increasing decreasing fault behavior given by the trend analysis

As a first attempt, we apply the bi-log-logistic model to modelize two sets of data

Step1 Decompose the $NTDS$ data into two sets

Set I		Set II		Set II (*Next*)	
Failure number y	TBF x_i (days)	Failure number y	TBF x_i (days)	Failure number y	TBF x_i (days)
1	9	9	5	18	3
2	12	10	7	19	6
3	11	11	1	20	1
4	4	12	6	21	11
5	7	13	1	22	33
6	2	14	9	23	7
7	5	15	4	24	91
8	8	16	1	25	2
9	5	17	3	26	1

Step2 Estimate the bi-log-logistic parameters, in other words estimate the log-logistic paramerters for each set of data

The estimated parameters are given by:

$$\hat{M} = \begin{bmatrix} 25 & 0.01193 & 1.96 \\ 28 & 0.01190 & 1.1 \end{bmatrix}$$

Step3 Deduce the fitted bi-log-logistic model from the above estimation

The fitted bi-log-logistic model is then given by:

$$\hat{m}(t) = 25\frac{(0.01193t)^{1.96}}{1+(0.01193t)^{1.96}} + 28\frac{(0.0119t)^{1.1}}{1+(0.0119t)^{1.1}}$$

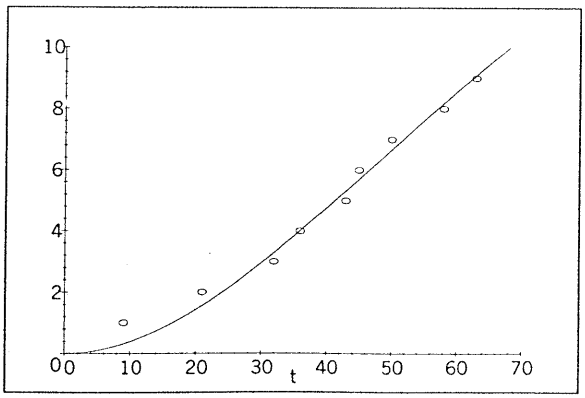


Figure 4.4: Fitted model (*Set* I)

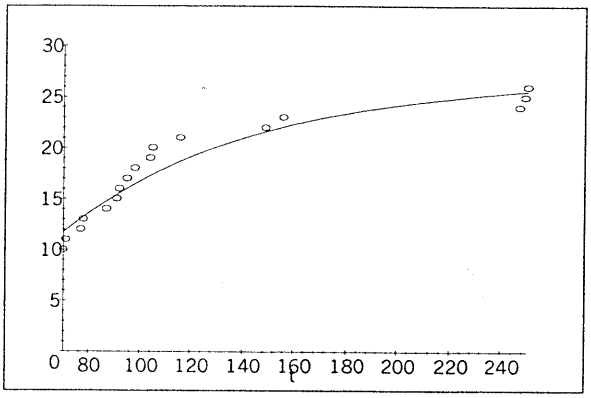


Figure 4.5: Fitted model (*Set* II)

Step4: Evaluate the goodness of fit for the bi-log-logistic model [Table 4.]

Compare the goodness of fit of bi-log-logistic model and log-logistic model and choose the lower

We conclude that the bi-log-logistic goodness of fit is lower than the goodness of fit of the log-logistic model, evaluated in the previous chapter. So the bi-log-logistic model fits better the $NTDS$ data than the log-logistic model. $GF_{bi-\log-\log istic} = 30.546 < GF_{\log-\log istic} = 33.439$

If we are not satisfied with the above decomposition or we found that bi-log-logistic model does not fit the data we propose another decomposition (three set of data); in other words we apply a multi-log-logistic model with $N = 3$ and redo the same steps

Multi-log-logistic model

Step1 Decompose the $NTDS$ data into three sets

Set I		Set II		Set III	
Failure number y	TBF x_i (days)	Failure number y	TBF x_i (days)	Failure number y	TBF x_i (days)
1	9	9	5	18	3
2	12	10	7	19	6
3	11	11	1	20	1
4	4	12	6	21	11
5	7	13	1	22	33
6	2	14	9	23	7
7	5	15	4	24	91
8	8	16	1	25	2
9	5	17	3	26	1

Step2 Estimate the multi-log-logistic parameters
The estimated parameters are given by:

$$\hat{M} = \begin{bmatrix} 25 & 0.0118 & 1.957 \\ 28 & 0.01209 & 2.83 \\ 29 & 0.0128 & 1.999 \end{bmatrix}$$

Step3 Deduce the fitted multi-log-logistic model
The fitted multi-log-logistic model is then given by:

$$\hat{m}(t) = 25\frac{(0.0118t)^{1.957}}{1+(0.0118t)^{1.957}} + 28\frac{(0.01209t)^{2.83}}{1+(0.01209t)^{2.83}} + 29\frac{(0.0128t)^{1.999}}{1+(0.0128t)^{1.999}}$$

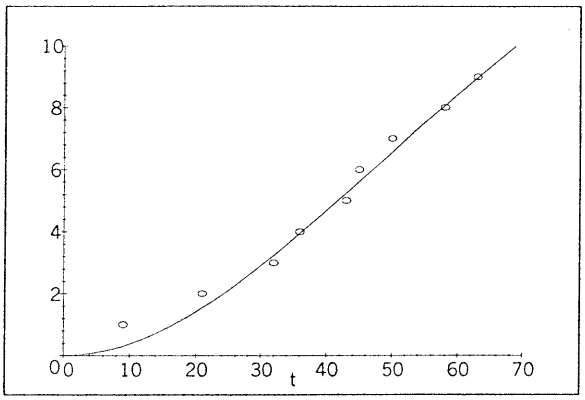


Figure 4.6: Fitted model (*Set* I)

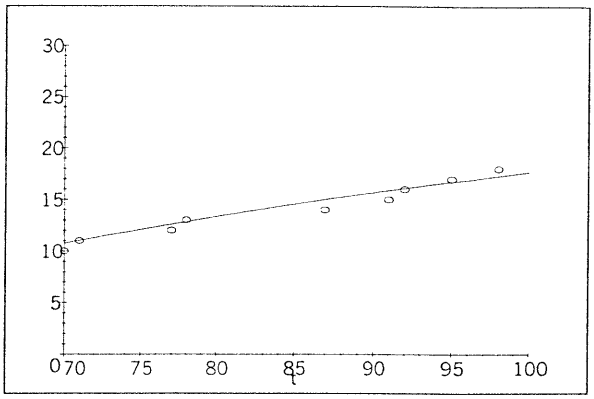


Figure 4.7: Fitted model (*Set* II)

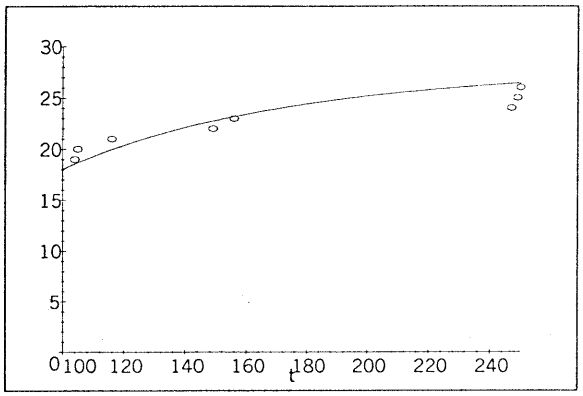


Figure 4.8: Fitted model (*Set* III)

Step4: Evaluate the goodness of fit for the multi-log-logistic model [Table 4.1]

The multi-log-logistic goodness of fit is lower than the bi-log-logistic. So we conclude that the multi-log-logistic fits better the above data than the bi-log-logistic.

Table 4.1: Goodness of Fit with NTDS Data

	$\hat{y}_i$	
y_i	$Muti - \log$	$Bi - \log$
1	0.481	0.474
2	0.221	0.198
3	0.056	0.083
4	0.003	0.000
5	0.057	0.102
6	0.147	0.089
7	0.190	0.115
8	0.004	0.031
9	0.001	0.008
10	0.569	3.620
11	0.000	1.211
12	0.345	1.555
13	0.025	0.186
14	0.999	0.997
15	0.775	0.412
16	0.009	0.041
17	0.085	0.557
18	0.073	1.710
19	0.215	2.195
20	1.784	5.524
21	1.106	4.077
22	0.553	0.003
23	0.033	0.335
24	5.569	3.759
25	1.955	0.965
26	0.174	0.000
$GF = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$	15.536	30.405

4.5 Model Classification Scheme

Several classification scheme have been proposed in the literature. Musa an Okumoto[1983] present a classification scheme which permits relationships to be derived for groups of models. Models are classified in terms of failure category (finite or infnite).

Gaudoin[1994] proposed another classification scheme based on failure intensity variables (time, number of failures,..).

In this topic we propose another model classification scheme based on the

failure intensity nature [Table 4.2]

Table 4.2: Model Classification Scheme

Failure intensity nature	Model
Monotonic Decreasing	Jelinski and Moranda [1972] Littlewood and Verrall [1973] De-eutrophication model of Moranca [1975] Goel and Okumoto [1979] Musa and Okumoto [1984]
Increasing/Decreasing	S-shaped [1983] Log-logistic [1998]
Iterative Increasing/Decreasing	**Multi-log-logistic**

4.6 Conclusion

In this chapter, we have proposed a multi-log-logistic model which exhibit an iterative increasing / decreasing failure intensity, since the existing software reliability models can capture a decreasing or an increasing / decreasing failure intensity. With the same data set ($NTDS$) the multi-log-logistic model provides a lower goodness of fit, in other words, fits better the failure process than the other models (Goel-Okumoto, S-shaped, and log-logistic). We have also presented another model classification scheme based on the failure intensity behavior (decreasing, increasing / decreasing, iterative increasing / decreasing)

Chapter 5

Conclusions

This dissertation deals with the development of multi-log-logistic software reliability model. This model is a generalization of log-logistic model, where the failure intensity exhibit an iterative increasing / decreasing nature instead of monotonic increasing / decreasing in other models. By the way, we offer a new model classification scheme based on failure intensity behavior.

Using the NTDS data, multi-log-logistic model provides a good fit of the observed failure phenomenon with comparison to log-logistic, S-shaped and Goel and Okumoto models.

It should be noted that the above analytical models are primarily useful in estimating and monitoring software reliability, viewed as a measure of software quality, and other performance measures described in previous chapters. The software reliability analysis should be used for software development management in other words making some decisions about the software system, such as release the software or continue testing? How much more testing to do? Etc...

Perspectives

Here we provide a list of open research problems that arise in software reliability..

P1 Applicability of reliability model to Software life cycle phases;

P2 One area of possible extension of software reliability models, relatively unexplored, is the unit test phase;

P3 Reliability belongs to the operational profile, so we should define a significant operational profile to reach a good level of reliability;

P4 Reliability models should introduce a cost factor;

P5 Evaluate system reliability without separation between hardware and software;

P6 For the same data set, combine different reliability models;

Therefore, there still several problems areas that need work.

Bibliography

[1] Beizer B. (1989). Software Testing Techniques. Van Norstrand Reinhold.

[2] Gaudoin O. (1994). L'Evaluation Statistique de la Fiabilité des Logiciels. Rapport Technique. Université Joseph Fourier. Grenoble.

[3] Ghokale S.S. and Trividi K.S. (1998). Log-Logistic Software Reliability Growth Model. In Proc. of the 3 rd Assurance Systems Engineering Symposium (HASE'98), IEEE Computer Society Press, pp 34-41.

[4] Goel A. L (1985). Software Reliability Models: Assumptions, Limitations, and Applicability. IEEE Transactions on Software Engineering, SE-11, pp 1411-1423.

[5] Goel A. L. and Okumoto K. (1979). Time Dependent Error Detection Rate Model for Software Reliability and Other Performance Measures. IEEE Transactions on Reliability, R-28(3), pp 206-211.

[6] Iannino A., Musa J. D., Okumoto K. and Littlewood B. (1984). Criteria for Software Reliability Model Comparisons. IEEE Transactions on Software Engineering, SE-10(6), pp 687-691.

[7] Littlewood B. (1989). Prediction Software Relibility. Phil. Trans. Royal Society London. A 327. pp 513-527.

[8] Musa J. D. (1975). A Theory of Software Reliability and its Applications. IEEE Transactions on Software Engineering, SE-1, pp 312-327.

[9] Musa J. D. (1992). The Operational Profile in Software Reliability Engineering: an Overview. Proc. 3 rd Int. Symposium on Software Reliability Engineering Research Triangle Park. pp 140-154.

[10] Musa J. D., Iannino A. and Okumoto K. (1987). Software Reliability: Measurement, Prediction, Application. Mc Graw-Hill. New York.

[11] Perrin S. M., (1994). Bi-Logistic Growth. Technological Forecasting and Social Change. Adress Web http://phe.rockefeller.edu/Bi-logistic

[12] Printz J. (1995). Que sais-je?: Le Génie Logiciel. Presses Universitaires de France.

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

[14] Singpurwalla N. D. and Wilson S. P. (1994). Software Reliability Modeling. International Statistical Review. 62, 3. pp 289-317.

[15] Sommerville I. (1989). Software Enginneering Addison-Wesly (Fourth Edition).

[16] Tobias P.A. and Trindade D.C. (1995). Applied Reliability. Van Nostrand Reinhold (Second Edition).

[17] Yamada S., Ohba M. and Osaki S. (1983). S-Shaped Reliability Growth Modeling for Software Error Detection. IEEE Transactions on Reliability. R-32(5), pp 475-478.

[18] Yamada S. and Osaki S. (1985). Software Reliability Growth Modeling: Models and Applications. IEEE Transactions on Software Engineering, SE-11(12), pp 1431-1437.