CODEN:LUTEDX(TETS-5315)/1-75/(1998)&local 2

Department of
Communication Systems
Lund Institute of Technology
Lund University

Software Engineering
Research Centre
The Royal Melbourne
Institute of Technology and
The University of Melbourne

How to measure reliability

in an Erlang System

a Master Thesis by

Hans Danielsson and Kent Olsson

February 26, 1998

Supervisor: Claes Wohlin, Lund
Fergus O'Brien, Melbourne

Abstract

Today, the quality of a software product has become more and more important when a customer choose supplier. This has led to desire to be able to write a contract where you can specify the quality and then at delivery show the customer that your product fulfil these requirements. This thesis shows that it is possible to specify the requirements for at least one quality aspect, the reliability, in the contract and then have full control over these through the whole development process and even during the operational phase.

We have defined reliability metrics and described aspects that one have to take in consideration when collecting these metrics. We have also developed design guidelines to follow, to be able to collect these metrics in an Erlang system.

To validate our method we have build a monitor and analysis program and used this with a web server, written in Erlang.

Preface

This Master Thesis is undertaken for the Department of Communication System at Lund Institute of Technology, Lund, Sweden, Erlang System, Stockholm, Sweden and Software Engineering Research Centre (SERC) in Melbourne, Australia. The thesis has been performed at SERC.

We would like to thank all the people at SERC, and especially Prof. Fergus O’Brien, for letting us conduct our Master Thesis at SERC. We really enjoyed performing our thesis at SERC and we are grateful for that we had the opportunity to stay in such a lovely place as Melbourne. This thesis would never been possible to carry through without all the helpful suggestions and ideas from the people at SERC.

We also like to thank Roy Bengtsson at Erlang System and Mike Williams at OTP for their allowance that helped us to come to Australia.

A special thank you, to our supervisor in Sweden, Claes Wohlin at Lunds Institute of Technology for all his ideas and views through the project and especially his support at the end.

Now you probably think that we will go on thanking people through the whole report, but we only have one thank left and that is the most important.

Thank you, Torbjörn Törnkvist, for all you support through this thesis. You have given us all support we could ever ask for about Erlang. You have also been very helpful for the other parts in this thesis, with all your ideas and the discussions we had. You have been an outstanding supervisor, but a poor squash- and table tennis player, once again THANKS.

Kent Olsson
Hans Danielsson

25 February, 1998
Melbourne

Table of contents

CHAPTER 1

Introduction

This chapter gives a brief, broad view of everything that is relevant to this thesis. We start with what software engineering is and why it is needed and then continue with monitoring. Finally, we describe Erlang/OTP, the computer environment used in this thesis.

1.1 Software Engineering

For about ten years ago software products were something the ordinary human never used. The technical machines around us mostly consisted and depended on hardware, but during the last decade software has become a big part of our everyday life. Everywhere there is more software and less hardware. Today, programs must be so easy to use that everyone can manage them, even if they do not have any training. The software also handles more and more critical and dangerous tasks, such as nuclear power plants, landing systems for aeroplanes and so on. This has resulted in higher requirements on the software, both when it comes to functionality and quality. It has also made software quality a key factor for the customer when he or she chooses supplier.

To be able to meet the new requirements software companies have to change their development organisation. They must find ways of controlling the whole development process. However one problem is that so far most people have seen software development as an art performed by “Hackers” and not as a well engineering discipline. To cope with this problem companies and universities have initiated a lot of research in this new field called “Software Engineering” has evolved. Software Engineering deals with all problems related to software development and has till now resulted in a number of different tools and methods. These methods intend to help the organisation, and in particular the manager, to keep track of the progress during a project. This way he can make sure that a product has a certain quality. Since the software industry is very changeable and there are seldom two projects with the same requirements, the methods must be under constant development.

1.1.1 Software quality

Software products are very hard to understand. The reason for this is that you can not see or touch software. The only thing you can do is to see their reaction on different inputs. This makes them very subjective. Software products are often very logically complex, which makes them very hard to test as well. This uncertainty of what a software product really is, and the problems involved with testing have led to numerous different definitions of software quality. Almost every organisation has its own way of defining quality based on what characteristics they think are important. However, there is some characteristics that most organisations have in common, some of these are:

- **Reliability** -How reliable the program is. How often it fails.
- **Performance** -Which performance the program has. How long time it takes before I get a response.
- **Maintainability** -How easy it is to maintain the program during operation.
- **Portability** -How easy it is to move the program to another environment.

None of these characteristics are more important than anyone else, it depends on the situation. If the software should control a nuclear power plant reliability is very important. You really do not want the program to fail. But, since the program is tailor made for this particular plant and thus should run in the same computer environment, you are not so interested in high portability. In another situation it could be the opposite, e.g. a word processor. You do not care if it fails sometimes. The most important aspect is that as many people as possible can use it in their environment

1.1.2 Software Reliability

According to [1] the definition of reliability is *"The probability of failure-free software operation for a specified period of time in a specified environment"*. This is widely used but it is not the only definition. Another commonly used way to express reliability is Mean Time To Failure (MTTF). Reliability is one characteristic of software quality that often is considered as important. Even if there is no danger involved when a program fail, the user will probably not tolerate this too happen to frequently.

When we have decided which definition to use there are still a number of aspects to consider. First of all, what is a failure? Is it when the program stops working or is it when it gives you an incorrect answer? One definition of failure, which is broadly used, is when the program behaviour departs from the requirements.

1.1.3 Software measurements

If you want to improve the quality of the products you develop you must first of all define what you mean by quality (see above). Then you have to find a way to measure the quality, otherwise you do not know if you have improved your product or not. If it is a hardware product you can easily pick a random sample and then run some tests on that sample. But how do you perform measurements on a software product which you can not see or touch? There is no short answer to this question but there are two methods you can use to collect measures from software:

- **Execution** -Execute the program and see how the program behaves for different sets of inputs.

- **Reviews** -During the whole development process you stop and inspect your result at certain points to make sure that it follows the requirements and meets your quality objectives as well.

These two methods does not exclude each other, on the contrary, they should be used together. The first method is good when the program is finished, but then it may be too late to do any changes. It can also be hard to analyse the internal behaviour of the program. The good thing with reviews is that you will early see your mistakes so that you can correct them before it is too late. You can also early predict if your program will meet the objectives when it is finished.

1.1.4 Monitoring

If you want to collect data from your program when it is executing, you do not want this to interfere with the normal behaviour of the program. If it should interfere, your data will be inaccurate and thus you will draw the wrong conclusions. A good way to avoid this is to monitor your program. This means that you only watch the execution of the program and passively collect data. You may have to do some changes in your program to be able to extract the data. If so, you must consider this very carefully and be sure it does not affect the behaviour of the program.

1.2 Computer environment

When you want to develop a program with very high quality it may not be enough to change your development process. You may also have to change your development environment. If you can not find anyone that suits you, you may be forced to take this one step further and develop your own environment.

1.2.1 The Erlang Engine

One of Ericsson’s major successes is the telephone switch AXE-10. It has been sold to a numerous of countries all over the world. The problem now is that it is getting old and needs to be upgraded to stay competitive. Of course, this has been done regularly ever since it was first introduced but now it is not enough to do small changes. This time it needs a major upgrade. It is the extreme growth in applications that causes the biggest problem and one way to solve this is to connect the AXE-10 to new architectures. One of these architectures uses the Erlang/OTP environment (see below) and is called The Erlang Engine. The Engine is a high performance multiprocessor approach to the implementation of intelligent network applications.

1.2.2 Erlang/OTP

For about ten years ago, the Swedish telephone company Ericsson was in the situation mentioned above. They wanted a computer environment that could help them to develop telecommunication applications with very high reliability. What they[1] did was to take a telecommunication application and implement several programs with the same functionality using different computer languages. Then they took the best parts from each program and put them together in a new language, called Erlang.

It was not enough to develop a new programming language. Ericsson also had to develop a whole new environment to support the development of Erlang products. They called this environment Open Telecom Platform (OTP). OTP is based on a high level language and contains not only Erlang but also a number of well tested modules which solves critical basic problems.

1. The actual development was done by CSLab (Ericsson Computer Science Laboratory)

CHAPTER 2

Purpose

This chapter describes the purpose with this thesis and how we are intending to set about it. The chapter also contains a short description on other related projects at SERC.

2.1 Software quality research at SERC

In chapter 1 we mentioned that the increasing requirements on software quality initiated a lot of research in a new field called Software Engineering. Software quality is also referred to as non-functional requirements, i.e. everything that does not relate to the actual functionality of the software. One research centre that takes this very seriously is the Software Engineering Research Centre (SERC) of the Royal Melbourne Institute of Technology, Melbourne (RMIT), Australia. At SERC they concentrate their software quality research on three of these non-functional requirements, reliability, performance and, maintainability.

A lot of research in software quality concentrates on the actual construction of the program, and the steps before and after are often forgotten. At SERC they wanted to control the quality through all steps involved in a software project starting with the contract and ending when the program is thrown away. To achieve this they have set out several projects each addressing quality from a different point of view.

Till now, there has been one project undertaken, Quality of Telecommunication Application Software by Anna-Karin Carlsson and Fredrik Gustavsson [2], that addressed the problems involved in identifying the requirements and specify them in the contract. The outcome of this project was a way to identify and describe a business as a number of different transactions, e.g. ordinary call, call forward, billing. Carlsson and Gustavsson also found a way to decide which level of quality you need for every type of transaction to make a good business, e.g. a reliability of 99,9% for call forward.

The other projects that has been undertaken have been concentrated on monitoring the quality in a software based system, both during development and operation [3]. Since SERC is closely connected to Ericsson, they were introduced to Erlang/OTP and they found that it already contained many of the desired features of a monitoring system. To be able to monitor the system continuously, without affecting the performance, a dual pentium based system, using a new Erlang compiler, is under construction by Geoff Wong. The idea is to execute the main application on one processor and the monitor program on the other. So far there has been projects undertaken to solve the problems with monitoring performance and maintainability. Wong’s system is intended to be included in the Erlang Engine mentioned in chapter 1.

2.2 Purpose of this thesis

This thesis is both a continuation on the project by Carlsson and Gustavsson and a part of the monitor project conducted by Geoff Wong. The whole thesis is concentrated on one quality characteristic, namely reliability. Figure 2.1 shows how the three different projects are related.

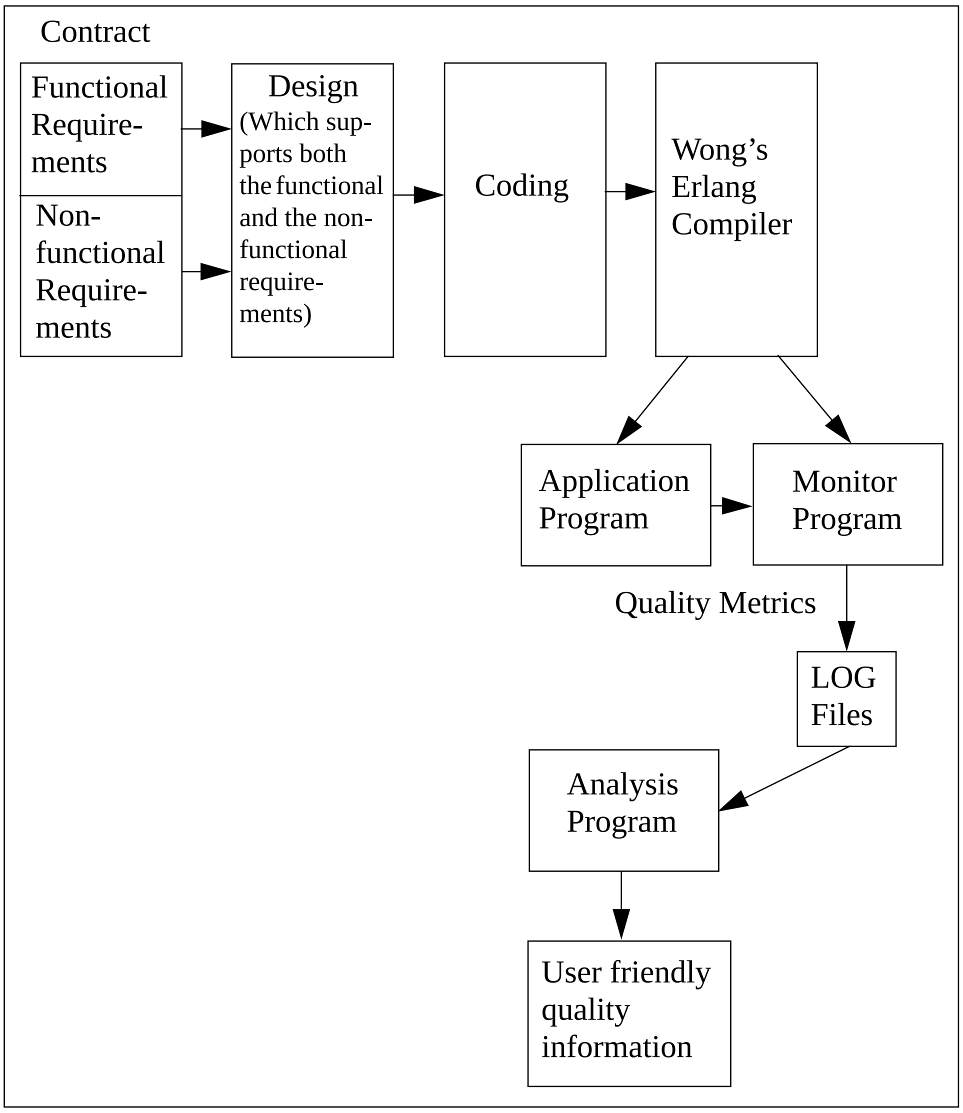

Contract
Functional Require-ments
Non-functional Require-ments
Design (Which sup-ports both the functional and the non-functional require-ments)
Coding
Wong's Erlang Compiler
Application Program
Monitor Program
Quality Metrics
LOG Files
Analysis Program
User friendly quality information

Figure 2.1: Project relations

The outcome from Carlsson and Gustavsson's method are the non-functional requirements in the contract. Our job is to first define reliability metrics, which is a part of the quality metrics, in a way that suites Erlang systems. Then we should find a way to connect these metrics to the non-functional requirements in the contract and also supports the use of Wong's Erlang compiler. Our method will be presented as a set of design guidelines. Since the monitor project is not finished we will also have to write a monitor program, in Erlang, to collect the data we need to calculate the reliability. We will also write a program to analyse the data we have collected.

2.3 Course of action

First we will learn Erlang so that we know the possibilities and limitations of an Erlang system [4], see chapter 5. We will also conduct a literature study to see what has already been done in the field of reliability and monitoring, see chapter 3 and 4. Then we will use our new knowledge to join these three parts together and identify different possible solutions, see chapter 6. When this is done we will implement the best solution in our monitor program, see chapter 7-9, and to prove that our solution works, we will finally use this program together with a suitable Erlang application, see chapter 10.

CHAPTER 3

Software Reliability Engineering

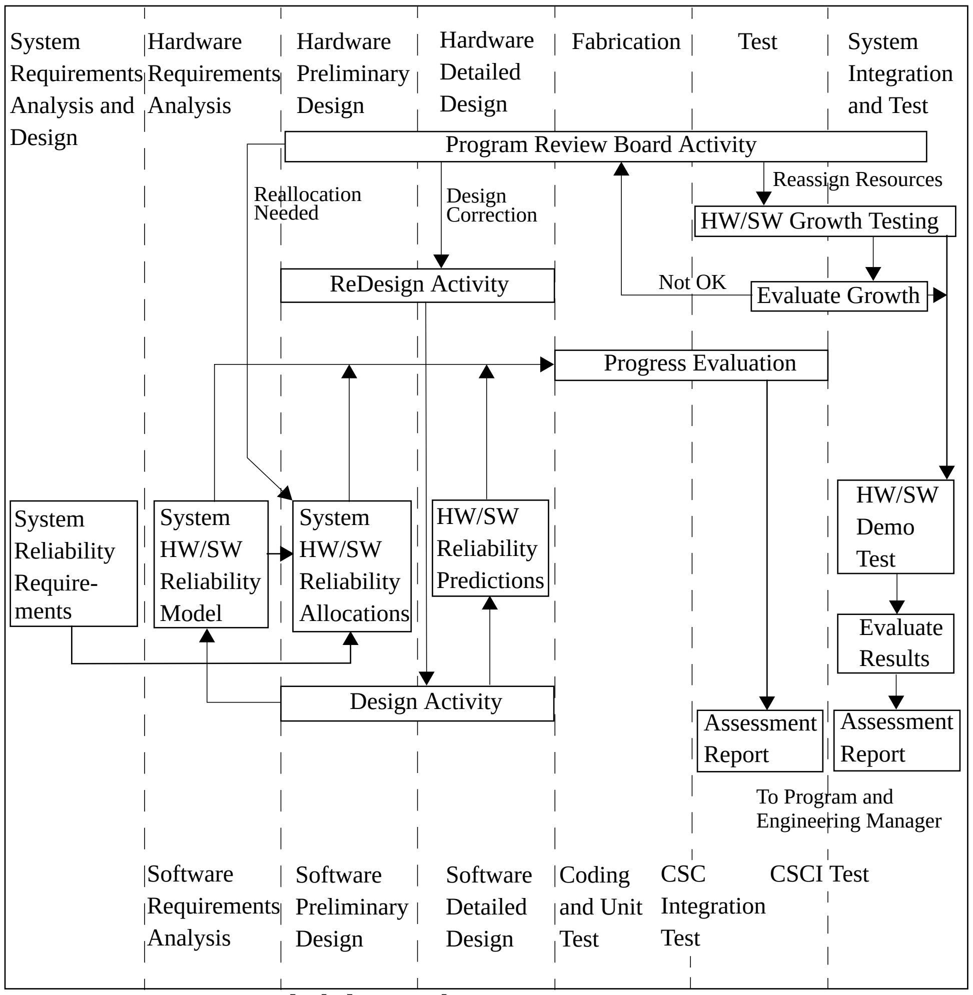
This chapter describes the software reliability theory we use in this project. We start with an overview of Software Reliability Engineering and gives some basic definitions. After this, we move deeper down in reliability growth and demonstration testing and also describe how you develop your operational profile. Then we describe in detail two ways of modelling reliability and ends with a quick view of modelling tools.

3.1 Overview

Software quality contains a lot of different aspects. One of the most important is Reliability. The good thing with reliability is that it is user-oriented instead of developer-oriented, e.g. fault counting. This means that it is derived from a user point of view which makes it more understandable for the customer. It also relates to operation rather than design which makes it more dynamic than static. Another advantage is that you can calculate reliability both for hardware and software which makes it possible to calculate the whole computer system's reliability.

Software Reliability Engineering (SRE) is a part of Software Engineering that, just as it sounds, concentrates on the quality characteristic Reliability. SRE involves all steps in the development process, from identifying reliability requirements to verifying that the requirements are fulfilled in the finished product. Figure 3.1 shows the different tasks, and their interrelationships, that has to be solved to be in full control of the development process [5]. Since telecommunication applications consist of both hardware (HW) and software (SW) components this must be considered when building up the control process. The Reliability control process consists of seven major components. These components interact with each other and most of them are under constant refinement during the project. All components consists of both a hardware part and a software part.

This thesis concentrates on the software part of two of the seven components, System Reliability Growth Testing and System Reliability Demonstration Testing. The other five will only be briefly described since these are not directly in the scope of this thesis. For a more thorough description see [5].



System Requirements Analysis and Design
Hardware Requirements Analysis
Hardware Preliminary Design
Hardware Detailed Design
Fabrication
Test
System Integration and Test
Program Review Board Activity
Reassign Resources
Reallocation Needed
Design Correction
HW/SW Growth Testing
ReDesign Activity
Not OK
Evaluate Growth
Progress Evaluation
System Reliability Require-ments
System HW/SW Reliability Model
System HW/SW Reliability Allocations
HW/SW Reliability Predictions
HW/SW Demo Test
Evaluate Results
Design Activity
Assessment Report
Assessment Report
To Program and Engineering Manager
Software Requirements Analysis
Software Preliminary Design
Software Detailed Design
Coding and Unit Test
CSC Integration Test
CSCI Test

Figure 3.1: System Reliability Tasks

3.1.1 System Reliability Requirements

The System Reliability Requirements should be elaborated together with the customer. A good way to do this is to use Carlsson's and Gustavsson's [2] method mentioned in chapter 2. This method gives a good assessment of the overall reliability but it does not give any separate estimate for hardware and software components which is needed. This has to be done with another method.

3.1.2 System Reliability Modelling

System reliability modelling provides a functional representation of the system under analysis. An accurate system model provides a mechanism for all reliability analysis performed. System reliability modelling for hardware and software systems is an evaluation of the dependency between system services and the various hardware components and their associated software. The system model is developed as an iterative process of decomposing the dependencies within the various system structural components.

The system is decomposed until it consists of a series of discrete hardware and hardware/software components, see example 3.1 taken from [5]. Detailed models of the individual hardware components that do not host software can then be developed in accordance with established hardware reliability techniques.

Example 3.1: Hardware/Software Decomposing
In figure 3.2 some of the components in a missile guidance system is illustrated. The Ineritał Measurement Unit (IMU) and Inerital Navigation System (INS) are hardware-only components, but the mission computer contains both software and hardware. The Operating System (OS) is operating continuously while the Application Software (AS) only operates when it is commanded to do. These time factors must be accounted for in the reliability analysis.

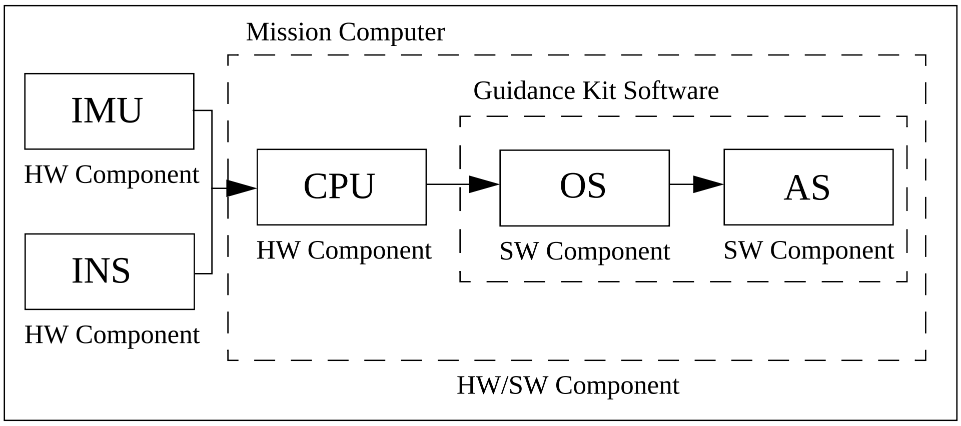
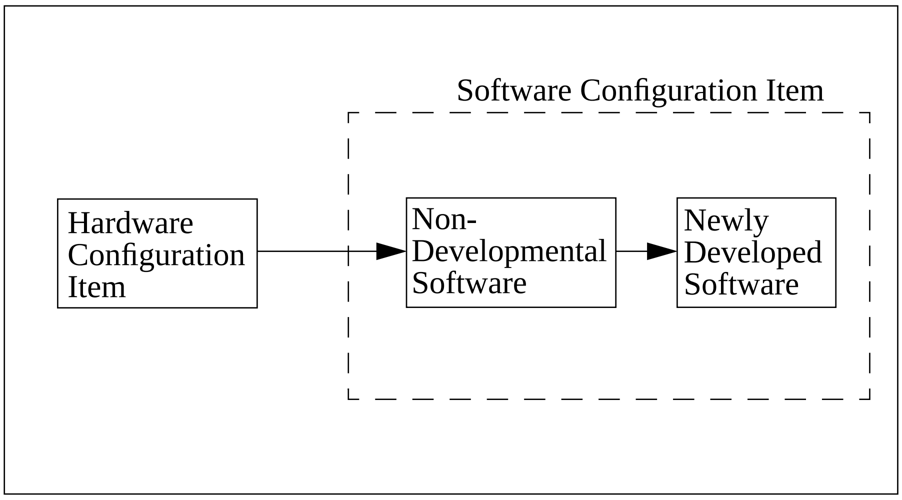


Figure 3.2: Block diagram for Missile Guidance System

The hardware/software components are decomposed as shown in figure 3.3. As you can see, the software part of the component is divided into two parts. Non-Developmental Software, which includes the operating system, and Newly Developed Software. Since the different components are independent the failure rates also are independent, and when failure rates are independent and exponential they can be directly added together.


Software Configuration Item
Hardware Configuration Item
Non-Developmental Software
Newly Developed Software
Figure 3.3: Reliability model for HW/SW components

For those cases where redundant equipment for hardware/software elements is supplied, the reliability model developed will depend on the exact method used to implement fault detection and recovery.

Even though the tasks involved in modelling software and hardware are similar there is one big difference. Software systems consist of Computer Software Configuration Items (CSCIs) which are generally independent programs associated with Hardware Configuration Items. The CSCI can be divided into Computer Software Units (CSUs), that perform a given software function, see figure 3.4. These CSUs can never be independent which means that they will not have independent failure rates.

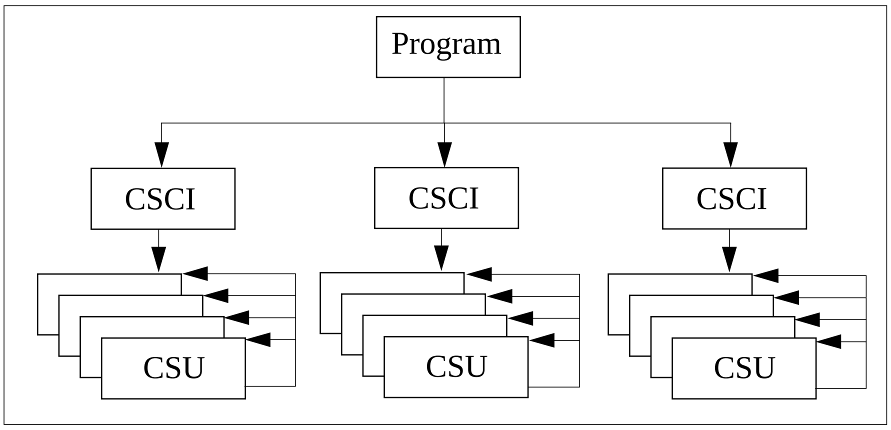


Figure 3.4: Dependency of software CSUs

3.1.3 System Reliability Allocation

During system reliability modelling you divided your system into different independent components. When this is done, you need to allocate every component a reliability goal so that the total system reliability meets the requirements. This allocation is an iterative process as knowledge of achievable failure rates is not always available.

3.1.4 System Reliability Prediction

System Reliability Predictions are used to assess overall design progress toward achieving the specified system reliability. Reliability predictions and estimates for the various system components are combined using a system model. The resultant reliability calculation is then compared against the specified system requirement to determine whether or not the current system design achieves the specified reliability.

3.1.5 System Reliability Growth Testing

System Reliability Growth consists of hardware and software reliability growth. By using the system reliability model you combine the two independent growth estimates into one total system reliability growth over time estimate.

Software Reliability Growth testing is conducted during the software system test phase, after the software has been fully integrated. The purpose with this testing is to assess the current reliability, identify and eliminate faults, and forecast future reliability. During growth testing you should execute the program in an environment that as much as possible resemble to the environment used during the operational phase. It is very important that the test cases are randomly selected in accordance with the software's operational profile. This way of testing is very efficient in finding the failures that affects the reliability most, i.e. those failures the user is most likely to encounter. When a failure is observed, the execution time is recorded and these failure times are then used as input to a software reliability growth model. The models normally assume that the faults causing the failures are immediately removed.

3.1.6 System Reliability Demonstration Testing

System Reliability Demonstration Testing is conducted in the end of system test and its purpose is to, with a certain statistical confidence, prove that the system meets the requirements.

3.1.7 System Failure Reporting and Corrective Action Systems

System Failure Reporting and Corrective Action Systems (FRACAS) describes how the different components in the reliability control process interacts. It should include, among other things, documented procedures for reporting failures, analysing failures to determine their root causes, and establishing effective corrective action to prevent future recurrence of the failure.

3.2 Basic Definitions

3.2.1 Failures and Faults

One of the first thing you have to do is to define what you mean by the term failure. Which failures to count and which not to. Usually a failure means that the program's behaviour departs from the requirements. In this thesis however, we define failures as failed transactions, i.e. non completed transaction.

A fault, on the other hand, is a defect in the program that under certain circumstances causes a failure. It is often referred to as a bug. One fault can cause more than one failure since different inputs makes the program behave differently.

It is very important to know the difference between failures and faults. These two terms often get mixed up and sometimes people even thinks that they mean the same thing.

3.2.2 Time

Reliability quantities are often related to time since this feels natural for most people. However, there are different types of time units.

Calendar time is what we usually refer to when we talk about time in our daily life. It is very useful to express reliability with respect to calendar time because this gives managers and developers, mostly in test groups, the chance to see when, which date, they will reach their objectives. But, as you will see later, most models use another kind of time, namely execution time to derive reliability. The reason for this is because these models are superior to those which use calendar time. To be able to express reliability with respect to calendar time, the models convert execution time to calendar time in a later state.

The second type of time is, as mentioned above, execution time. This is the time the program spends in the processor. It can be hard to derive this time since there can be many different programs running at the same time. The models we will use, and most others, to calculate the reliability is based on that failures occur randomly. Since it is only during the execution time that a failure can occur, it is very important to use this time in your calculations.

3.2.3 Input space and Operational profile

The input space is all the sets, states, of inputs that the program should be able to handle. If you then add the probability that this particular state should be chosen to every state, you have your operational profile. That is, you decide which probability a input state should be chosen with during test, see example 3.2. The different probabilities are determined so that the operational profile reflects how the program is executed during normal operation.

Example 3.2

Suppose we only have three different end user functions (transactions), ADD, DELETE, and UPDATE. The user has estimated that he will of all runs use ADD 23%, DELETE 12% and UPDATE 65% of the time. Therefore we will associate ADD with the real interval [0, 0.23], DELETE with [0.23, 0.35] and UPDATE with [0.35, 1]. This is our operational profile. When we then shall choose test cases we simply generate random numbers between [0, 1] for each test case.

3.2.4 Reliability

One way to define reliability is the probability of failure-free operation of a computer program for a specified time in a specified environment [6]. This means that if you have a reliability of 0.99 for 10 hours the program will, in average, during these 10 hours fail one time out of hundred.

Another way to express reliability is failure intensity, i.e. failures per time unit. The relationship between failure intensity and reliability depends, as you will see later in this chapter, on which model you use. One can not say that one is better than the other, it depends on what you will use it for. Reliability is better when you want to combine different component’s reliability to one system reliability because the calculations will be simpler. Failure intensity however, can be better suited in a program where the risk of failure at any point in time is of paramount concern, e.g. a nuclear power plant.

For hardware reliability Mean Time To Failure, MTTF, is often used. It is also used in software reliability but more and more seldom. The reason for not to use it, is that there are many cases in software reliability in which MTTF is undefined. Even if the relation between reliability and failure intensity depends on the model you use, they usually vary during the test phase, as faults are removed, as depicted in figure 3.5. As you can see, the failure intensity decreases and the reliability increases with time.

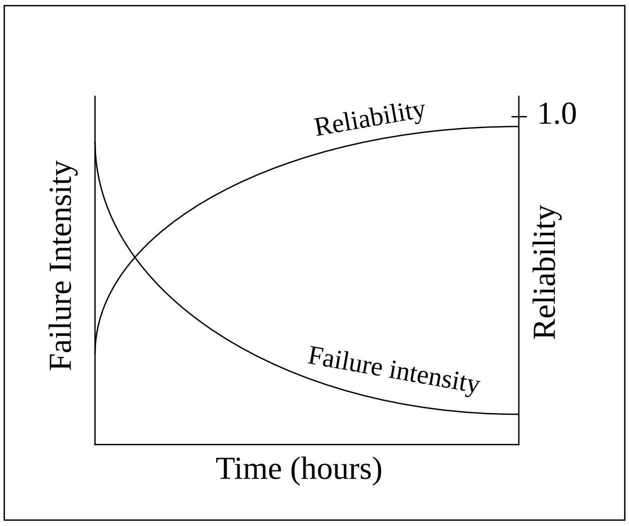


Figure 3.5: Reliability and Failure Intensity

3.3 Software Reliability Growth Testing

As mentioned in section 3.1.5 the purpose with reliability growth testing is to assess the current reliability, identify and eliminate faults, and forecast future reliability. You do this by executing the program, in an environment that as much as possible reflects the environment it will operate in after delivery, and record the times when failures occur. You remove the fault that caused the failure immediately and then continue to test the program. The test cases should be selected according to the operational profile as discussed later in this chapter.

3.3.1 Software Reliability Growth Models

To keep track of the progress you use a reliability growth model. The model takes the failure times as input and then calculates the current reliability and also projects the future reliability. This way the project manager knows how much more testing that needs to be done before they reach their objective.

When you shall model reliability, the first you have to consider are the factors that affect the reliability. These factors are [6] fault introduction, fault removal and the environment. Fault introduction mainly depends on the characteristics of the developed code, primary size, and the development characteristics, software engineering technologies, tools and the level of experience of personnel. Fault removal depends on time, operational profile and the quality of the repair activity. Finally the environment depends directly on the operational profile.

Some of these factors are probabilistic and therefore most models are based on random processes. The different models use different probability distribution of failure times or number of failures experienced and also assumes different time variations of the random process. All models have a set of parameters that have to be set for every project e.g. Initial failure intensity and Total number of failures. By doing this you can tailor the model to suit your specific project. The values of the parameters can be determined through either prediction or estimation. Before you start to execute your program you can use properties of the program to predict the parameters, or you can wait until you have executed your program and collected some failure data. You can then use these to estimate the parameters. In both cases, there will be some uncertainty and therefore you should always use confidence intervals. There is always a risk that the parameters will change during a project. If they do, you have to compensate for this otherwise your calculations will be inaccurate.

There are several important characteristics that a model must have. The model should:

- give good predictions of future failure behaviour
- compute useful quantities
- be simple
- be widely applicable
- be based on sound assumptions

There is a substantial amount of theoretical work involved in developing a new useful software reliability model. This effort generally requires several person years.

3.3.2 Model selection

There are several models available today and it can be very hard to know which one to use. If you want to you can choose more than one model. If you do this, you can compare the results from every model and maybe derive a better final result. The disadvantage is that the more models you use the higher the cost will be. In research projects it can be good to use several models, but in real projects the cost for using more than two will probably be too high.

Lakey and Neufelder have in [5] made a summarize of the most commonly used models in industry and also a chart that shows when to use which one. This is reconstructed in table 1 and figure 3.6A-B. For more information on these models see [6], [7], [8], [9].

Model Name	Function for failure intensity decrease	Data and/or estimation required	Limitations and constraints
General Exponential (General form of the Shooman, Jelsinski-Moranda, and Keene-Cole exponential models)	$K(E_0 - E_C(X))$	• Number of corrected faults (Ec) at some time x. • Estimate of E_0	• Software must be operational. • Assumes that no new faults are introduced in correction. • Assumes that number of residual faults decreases linearly over time
Musa Basic	$\lambda_0\left(1 - \frac{\mu}{\nu_0}\right)$	• Number of detected faults (μ) at some time x. • Estimate of initial failure intensity λ_0.	• Software must be operational. • Assumes that no new faults are introduced in correction. • Assumes that the number of residual faults decreases linearly over time
Musa Logarithmic	$\lambda_0 e^{-\phi\mu}$	• Number of detected faults (μ) at some time x. • Estimate of initial failure intensity λ_0. • Relative change of failure rate over time (ϕ).	• Software must be operational. • Assumes that no new faults are introduced in correction. • Assumes that the number of residual faults decreases exponential over time
Littlewood/Verrall	$\frac{\alpha}{(t + \psi(i))}$	• Estimate of α (Number of failures). • Estimate of ψ (Reliability growth). • Time between failures detected or the time of the failure occurrence	• Software must be operational. • Assumes uncertainty in correction process

Table 1: Reliability Growth Models

Model Name	**Function for failure intensity decrease**	**Data and/or estimation required**	**Limitations and constraints**
Schneidewind model	$\alpha e^{-\beta i}$	• Faults detected in equal interval i. • Estimation of α (failure rate at start of first interval). • Estimation of β (proportionality constant of failure rate over time).	• Software must be operational. • Assumes that no new faults are introduced in correction. • Rate of fault detection decreases exponentially over time.
Duane's model	$\frac{\lambda t^b}{t}$	• Time of each failure occurrence. • b estimated by (n=number of detected failures) $\frac{n}{\sum_{i=1}^{n} \ln (t_n + t_i)}$	• Software must be operational
Brook's and Motley's IBM model	Binomial model Expected number of failures= $\binom{R_i}{n_i} q_i^{n_i} (1-q_i)^{R_i - n_i}$ Poisson model Expected number of failures= $\frac{(R_i \phi_i)^{n_i} e^{-R_i \phi_i}}{n_i!}$	• Number of faults remaining at start of i:th test (R_i). • Test effort of each test (K_i). • Total number of faults found in each test (n_i). • Probability of fault detection in i:th test. • Probability of correcting faults without introducing new ones.	• Software developed incrementally. • Rate of fault detection assumed to be constant over time • Some software modules may have different test effort than others
Yamada, Ohba, and Osaki's S-shaped model	$ab_2 te^{-bt}$	• Time of each failure detection. • Simultaneous solving of a and b.	• Software must be operational. • Fault detection rate is S-shaped over time.
Weibull model	MTTF= $\frac{b}{a}\Gamma\left(\frac{1}{a}\right)$	• Total number of faults found during each testing interval. • The length of each testing interval. • Parameter estimation of a and b.	• Failure rate can be increasing, decreasing or constant

Table 1: Reliability Growth Models

Model Name	Function for failure intensity decrease	Data and/or estimation required	Limitations and constraints
Geometric model	$D\Phi^{t-1}$	• Either time between failure occurrences X_i or the time of the failure occurrence. • Estimation of constant D which decreases in geometric progression (0<ϕ<1) as failures are detected.	• Software must be operational. • Inherent number of faults are assumed to be infinite. • Faults are independent and unequal in probability of occurrence and severity.
Thompson and Chelson's Bayesian model	$\frac{(f_i + f_0 + 1)}{(T_i + T_0)}$	• Number of failures detected in each interval (f_i). • Length of testing time for each interval i (T_i).	• Software is corrected at the end of testing interval. • Software must be operational. • Software is relatively fault free.

Table 1: Reliability Growth Models

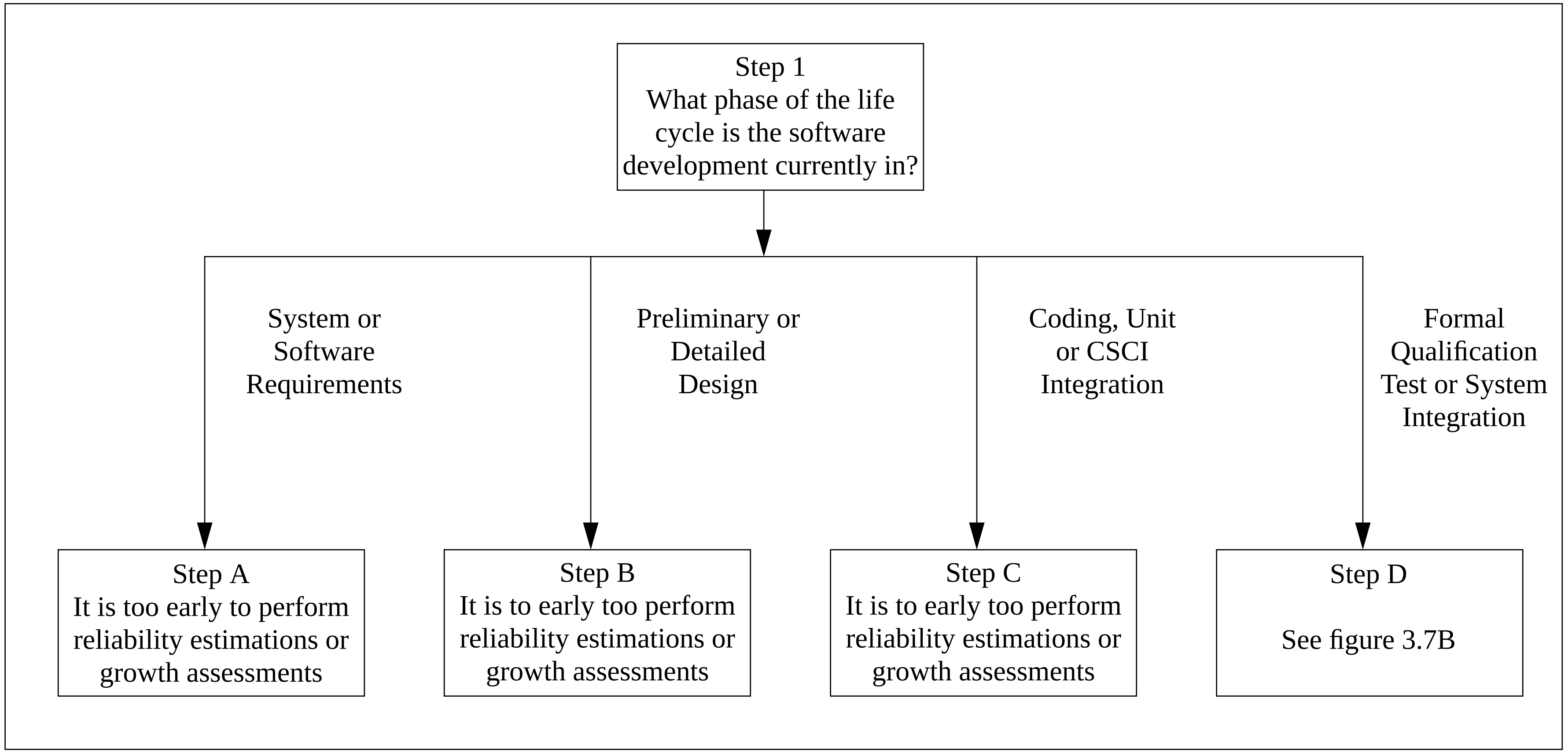

Step 1
What phase of the life cycle is the software development currently in?
System or Software Requirements
Preliminary or Detailed Design
Coding, Unit or CSCI Integration
Formal Qualification Test or System Integration
Step A
It is too early to perform reliability estimations or growth assessments
Step B
It is to early too perform reliability estimations or growth assessments
Step C
It is to early too perform reliability estimations or growth assessments
Step D
See figure 3.7B

Figure 3.6A: Reliability Growth Models

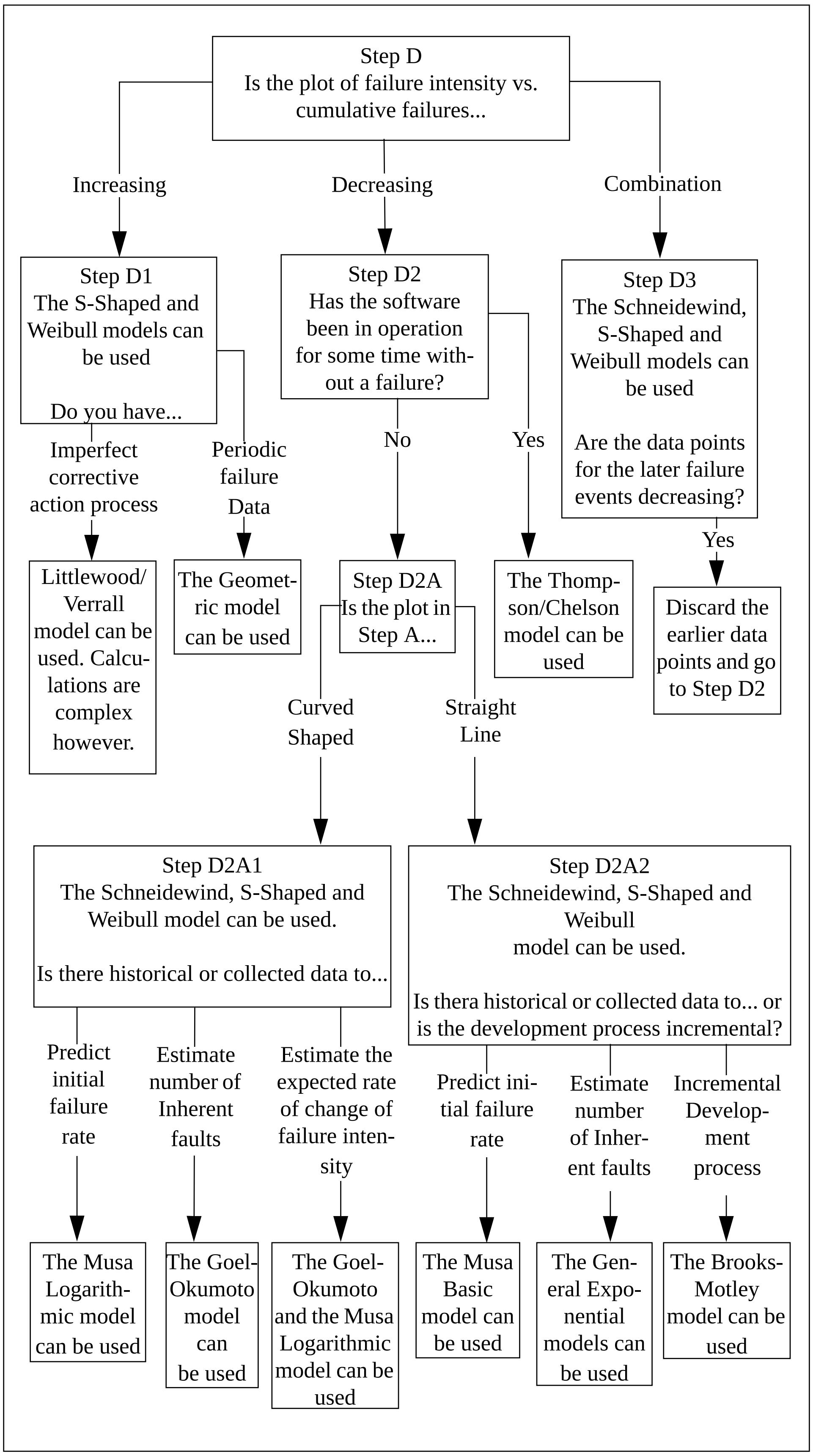

Step D
Is the plot of failure intensity vs. cumulative failures...
Increasing
Decreasing
Combination
Step D1
The S-Shaped and Weibull models can be used
Do you have...
Step D2
Has the software been in operation for some time without a failure?
Step D3
The Schneidewind, S-Shaped and Weibull models can be used
Are the data points for the later failure events decreasing?
Imperfect corrective action process
Periodic failure Data
No
Yes
Yes
Littlewood/ Verrall model can be used. Calculations are complex however.
The Geometric model can be used
Step D2A
Is the plot in Step A...
The Thompson/Chelson model can be used
Discard the earlier data points and go to Step D2
Curved Shaped
Straight Line
Step D2A1
The Schneidewind, S-Shaped and Weibull model can be used.
Is there historical or collected data to...
Step D2A2
The Schneidewind, S-Shaped and Weibull model can be used.
Is thera historical or collected data to... or is the development process incremental?
Predict initial failure rate
Estimate number of Inherent faults
Estimate the expected rate of change of failure intensity
Predict initial failure rate
Estimate number of Inherent faults
Incremental Development process
The Musa Logarithmic model can be used
The Goel-Okumoto model can be used
The Goel-Okumoto and the Musa Logarithmic model can be used
The Musa Basic model can be used
The General Exponential models can be used
The Brooks-Motley model can be used
Figure 3.6B: Reliability Growth Models

Figure 3.6A shows that it is only in Step D that you can perform reliability estimations and growth assessments. If you should be in one of the other steps there are ways to predict certain characteristics that can be used in the beginning of growth testing.

3.4 System Reliability Demonstration Testing

System Reliability Demonstration Testing is conducted in the end of system test and its purpose is to, with a certain statistical confidence, prove that the system meets the reliability requirements. This is sometimes referred to as *beta test* or *first office application.* During this testing it is even more important that you test the system in an environment that reflects field use. During demonstration testing, the software should be under configuration control (just as it would be between releases) which means that when a failure is observed, it will only be recorded and not corrected. You should also this time count multiple occurrences of the same failure since the demonstration test should represent a true operational environment.

The basic idea of demonstration testing is described in figure 3.7. In this example we have used sequential testing (see below). As you can see we continue to test the program until we have had four failures. Then we stop and accept the software.

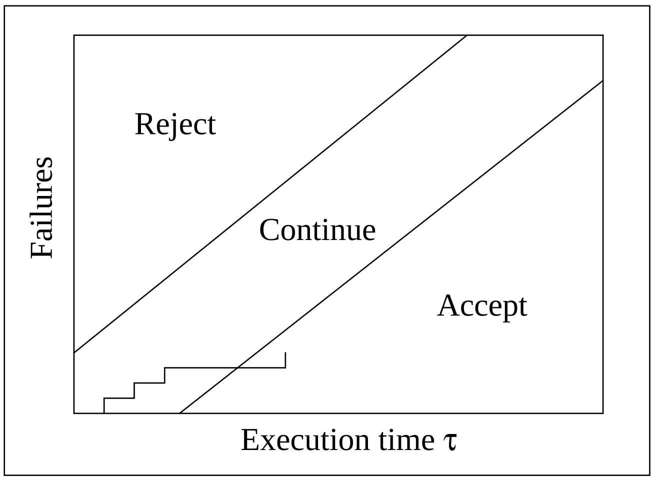


Figure 3.7: Reliability Demonstration Chart

There are three different types of test methods that can be used during demonstration testing. These and their advantages and disadvantages are summarized in table 2.

Test Type	Advantages	Disadvantages
Fixed Duration	Total test time is known in advance. An estimate can be made of true failure rate	Takes longer time than sequential test on average.
Sequential	Accepts very low and rejects very high failure rate quickly. Shorter test times on average than the other types.	Total test duration is undetermined. Maximum duration must be planned for.
Failure-Free Execution Interval	Will accept very quickly if true failure rate is much better than required	Can take a long time if true failure rate is close to that required

Table 2: Demonstration Test Types

As mentioned earlier in this subsection, the failures discovered will not be corrected, only recorded. Due to this the system can, under certain circumstances listed below, be modelled to have a constant failure rate and inter-failure times exponentially distributed. The following types of hardware are modelled with a constant failure rate:

- Components that are in their “useful life” period, i.e. after burn-in but before wearout.
- Assemblages of those parts, when in a series reliability configuration.
- Complex, maintained equipment that does not have redundancy.

When software runs concurrently and in series with such hardware, the overall failure intensity will be a constant that is the sum of the constant hardware failure intensity and the constant software failure intensity. If you then also assume that the times between failures are exponentially distributed the Reliability R and the failure intensity λ will be exponentially related as shown in equation 3.1 [6]. Note that the reliability is dependent not only on the failure intensity but also on the period of execution time τ, se figure 3.8.

$$Reliability \qquad R(\tau) = e^{(-\lambda\tau)} \qquad (3.1)$$

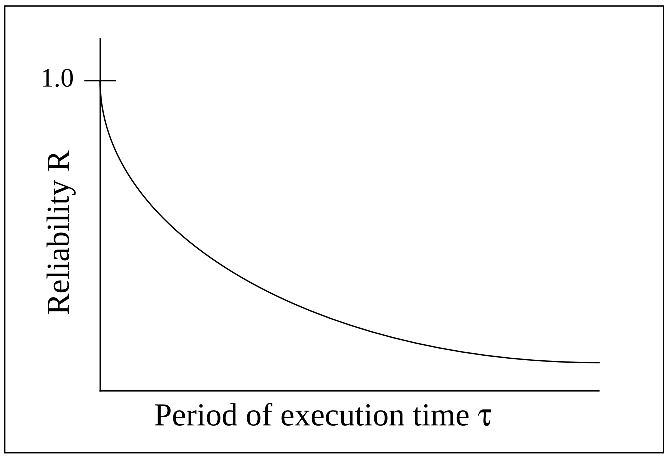


Figure 3.8: Reliability versus period of execution time

A lot of programs have several releases during their operational phase, which makes the reliability and failure intensity functions to step functions. For every new release there will be a new step. If the releases are frequent and the trend of failure intensity is decreasing, one can often approximate the step functions by one of the software reliability models we have presented. One can also apply the models directly to reported unique failures. Note that the model then will represent the failure intensity that will occur when the failure has been corrected. This approach is analogous to the one commonly taking place during system test of ignoring the failure correction delay, except that the delay is much longer here. If the failures are actually corrected in the field, then the operational phase should be handled just like the growth testing phase.

3.5 Operational Profile

The software's behaviour is significantly dependent on the environment in which it is executed and it consists of the hardware platform, the operating system, the workload and the operational profile. The operational profile is, as described earlier in this chapter, a set of input states and their probability of occurrence.

Software reliability testing is based on randomly selecting input states from an input space according to the operational profile. If the reliability estimate should be accurate, this selection must reflect the way the software will be operated in the field. Therefore it is very important to derive a good operational profile that really reflects the field environment.

According to [10] the process for developing the operational profile is as described in figure 3.9. You start from the top and work your way down to the bottom.

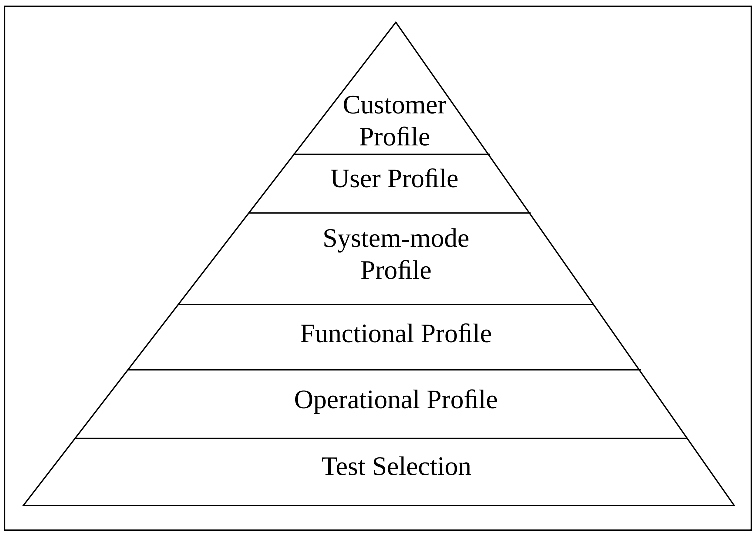


Figure 3.9: Operational Profile Development

3.5.1 Customer Profile

The customer is the individual, group or organisation that is purchasing the software system. If the system is meant to be sold to several customers there could be different types of customers, e.g. educational institutions, businesses, and individual home users who will use the system in different ways. All the users in a group uses the system in a similar way. The customer profile consists of a list of the customer types and their probability of occurrence, see table 3.

Customer Type	Occurrence Probability
Educational Institution	0.45
Business Organisation	0.35
Individual Home User	0.20

Table 3: Customer Profile

3.5.2 User Profile

A system's users may be different from the customers of a software product. A user is a person, group, or institution that operates, as opposed to acquires, the system. A user type is a set of users that operates the system similarly. The User Profile is the set of user types and their associated probabilities of using the system.

3.5.3 System-mode Profile

A system mode is a way that a system can operate. Most systems have more than one mode. For example, a car can be in normal mode or four wheel drive. It may also be in normal mode or cruise control. A system can switch modes sequential, or it can permit several modes to operate concurrently, sharing the same system resources. A System Mode Profile is the set of modes and their associated probabilities of occurrence.

3.5.4 Functional Profile

After a good system mode profile has been developed, the focus should turn to evaluation of each system mode for the functions, in this thesis called transactions, performed during that mode. Functions are tasks that a user can perform with the system. For example, the user of an e-mail system would want to: create messages, send messages, open messages etc. Functions are established during requirements based on what activities the customer wants the system to be able to perform.

3.5.5 Operational Profile

Figure 3.10 shows the elements involved in determining the operational profile from functions. A function may comprise several operations. In turn, operations are made up of many run types. Grouping run types into operations partitions the input space into domains. A domain can be partitioned into subdomains or run categories. To use the operational profile, first choose the domain that characterizes the operation, then the subdomain that characterizes the run category, and finally the input state that characterizes the run.

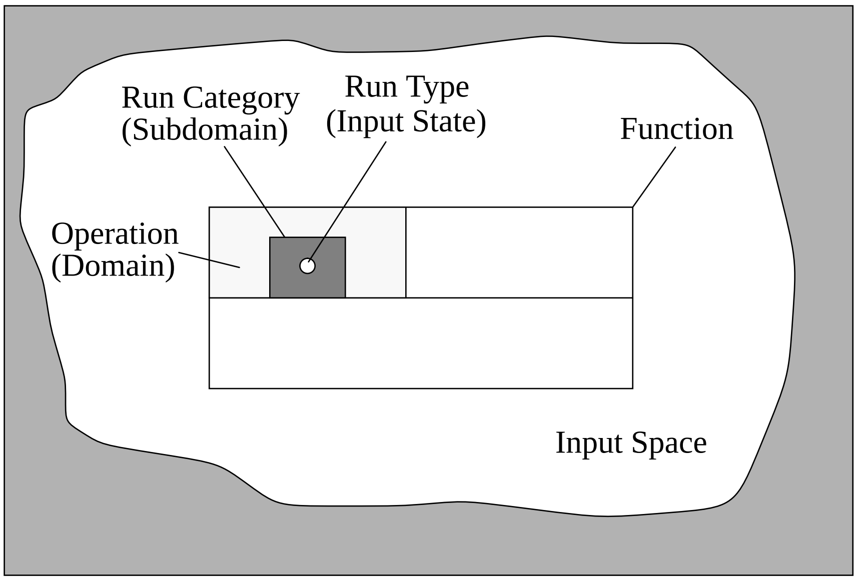


Figure 3.10: Operational Elements

3.5.6 Test Selection

Now, when you have derived your operational profile you simply generate a random number between zero to one and then choose that test case which is associated with that number. You do this until you have selected enough test cases to satisfactorily test your software system.

3.6 Two selected models

In this project we have chosen to more deeply describe two of the different models in table 1, *Musa Basi*c and *Musa Logarithmic* [6]. These will from now on be referred to as *Basic* and *Logarithmic*. The reasons we chose *Basic* are because it is simple, well established, widely applied to actual projects, its parameters has a clear physical interpretation and it usually predicts satisfactorily. *Logarithmic* is a newer model and has not been used as much as *Basic,* but it is almost as simple and gives good predictions. Both models also use execution time. It should be noted that none of these models should be used on very small programs, less than 5000 lines of code, since they may not experience sufficient failures to permit accurate estimations of Execution Time Component parameters and the various derived quantities.

Each model consists of two components, Execution Time Component and Calendar Time Component. In the first component, the models use, just as it says, execution time. After this, in the calendar time component, the models convert execution time to calendar time. The reason for this is because quantities expressed in calendar time are more meaningful to most engineers and managers. During this conversion, the models use the way in which human and computer resources are applied to the project. If you are not interested in calendar time, you do not need to use this component.

3.6.1 Execution Time Component

Both models assume that failures occur as a random process[1] but they have different failure intensity functions. The easiest way to illustrate the difference is to look at how the failure intensity varies with mean failures experienced[2]. *Basic* assumes that the decrement in failure intensity per experienced failure is constant, se figure 3.11 and equation 3.2. *Logarithmic* however, assumes that the failure intensity decreases exponentially, see figure 3.12 and equation 3.3. The quantity θ is called the failure intensity decay parameter and it represents the relative change of failure intensity per failure experienced. As you can see, *Basic* assumes that there is a total number of failures in the system, while *Logarithmic* assumes there is an infinite number of failures.

Both models also assume that after you have found a failure you will identify the fault, that caused the failure, and remove it immediately. However, in practise you probably do not stop testing for every failure. You wait until you have found a couple of failures and then removes all the faults at the same time. This delay in fault removal is called failure correction delay and as long as it is kept reasonably short, it will not affect the correctness of the models much. Note that sometimes you will fail to remove the fault and even inject new ones. This can make the failure intensity higher for a short period of time.

1. More specific, as a nonhomogeneous Poisson process
2. In this discussion, since failures occur randomly, we are referring to “average failures experienced”.

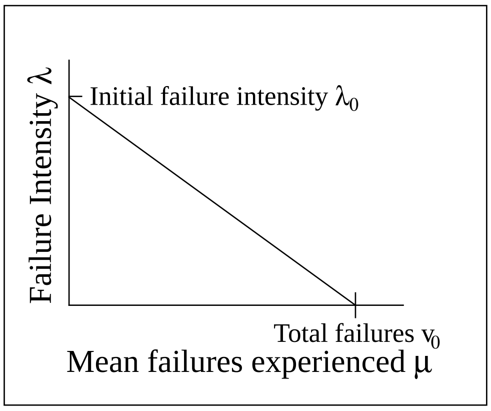


Figure 3.11: *Basic* Failure Intensity function

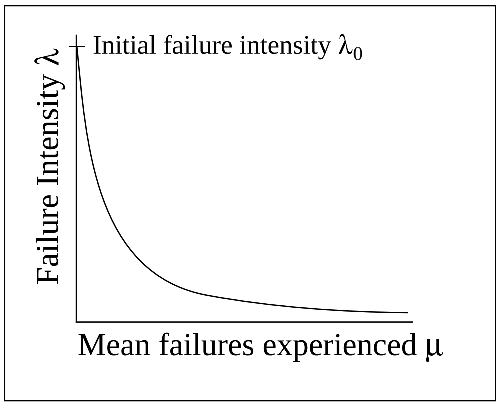


Figure 3.12: *Logarithmic* Failure Intensity function

Basic $$\lambda(\mu) = \lambda_0\left(1 - \frac{\mu}{v_0}\right) \qquad (3.2)$$

Logarithmic $$\lambda(\mu) = \lambda_0 e^{(-\theta\mu)} \qquad (3.3)$$

With straightforward [6] derivation we can obtain an interesting relationship, namely mean failures experienced as a function of execution time. This is shown for both models in figure 3.13 and also in equation 3.4 and 3.5.

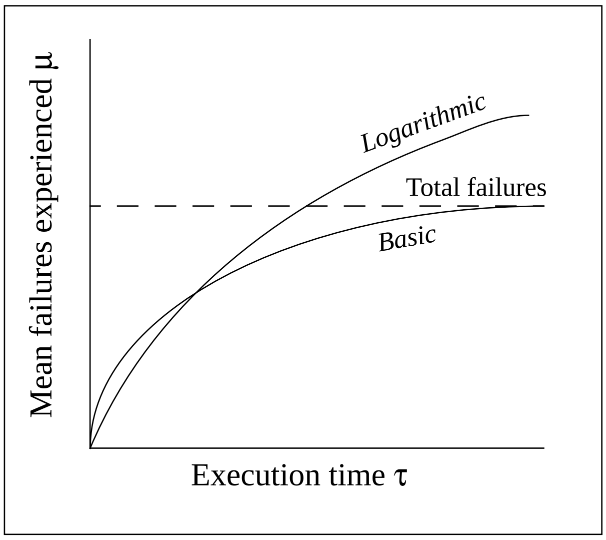


Figure 3.13: Mean failures experienced versus execution time

Basic $$\mu(\tau) = v_0\left(1 - e^{\left(-\frac{\lambda_0}{v_0}\tau\right)}\right) \qquad (3.4)$$

$$\textit{Logarithmic} \qquad \mu(\tau) = \frac{1}{\theta}\ln(\lambda_0\theta\tau + 1) \qquad (3.5)$$

Another relationship that is of real interest is the failure intensity as a function of execution time, see figure 3.14. This is, in fact, the fundamental expression of the failure intensity function. If you compare figure 3.14 with figure 3.11 and 3.12, you can see that figure 3.14 is stretched out to the right. This is because intervals between failures increase as failures are experienced.

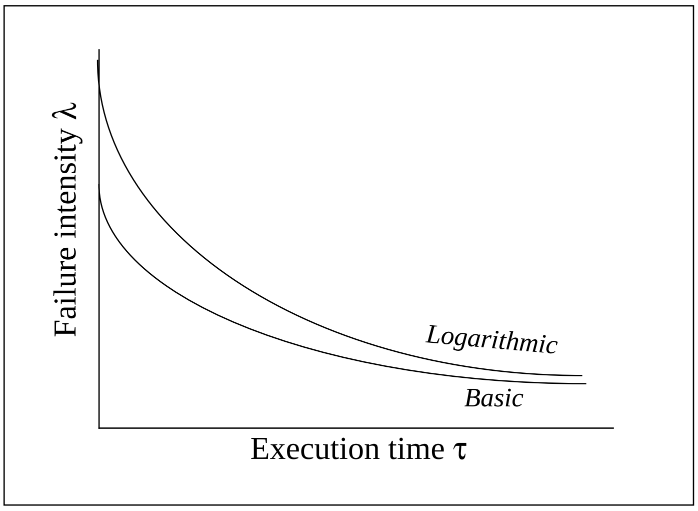


Figure 3.14: Failure intensity versus execution time

Equation 3.6 shows the aritmetic expression for *Basic* and equation 3.7 for *Logarithmic*. This relationship is useful for determining the present failure intensity at any given value of execution time. For the same set of data, the failure intensity drops more rapidly for *Logarithmic* than for *Basic* at first. Later, it drops more slowly. At large values of execution time, *Logarithmic* will have larger values of failure intensity than *Basic*.

$$\textit{Basic} \qquad \lambda(\tau) = \lambda_0 e^{\left(-\frac{\lambda_0}{v_0}\tau\right)} \qquad (3.6)$$

$$\textit{Logarithmic} \qquad \lambda(\tau) = \frac{\lambda_0}{\lambda_0\theta\tau + 1} \qquad (3.7)$$

Now, we are ready to derive some useful quantities which, among other things, will help us to estimate additional time spent in test to reach our failure intensity objective. Assume you have set a failure intensity objective and suppose some portion of failures has been removed. Then you can use your objective and the present failure intensity to determine the additional expected number of failures $\Delta\mu$ that has to be experienced to reach that objective, see figure 3.15. Equations describing the relationships can easily be derived using equation 3.2 and 3.3. The results are showed in equation 3.8 and 3.9.

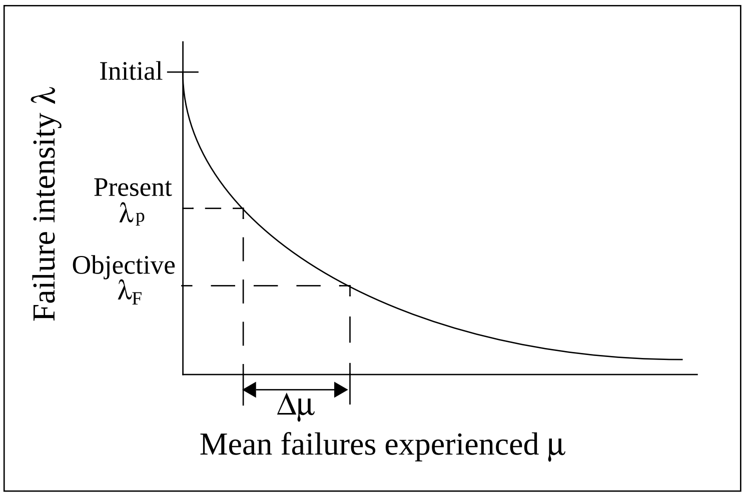


Figure 3.15: Additional failures to reach failure intensity objective

Basic $$\Delta\mu = \frac{v_0}{\lambda_0}(\lambda_P - \lambda_F) \qquad (3.8)$$

Logarithmic $$\Delta\mu = \frac{1}{\theta}\ln\left(\frac{\lambda_P}{\lambda_F}\right) \qquad (3.9)$$

In the same way, you can, by using equation 3.6 and 3.7, determine the additional execution time $\Delta\tau$ required to reach the failure intensity objective for either model. The resulting equations will then be described by 3.10 and 3.11. The only difference from figure 3.15 is that the horizontal-axis now shows execution time.

Basic $$\Delta\tau = \frac{v_0}{\lambda_0}\ln\left(\frac{\lambda_P}{\lambda_F}\right) \qquad (3.10)$$

Logarithmic $$\Delta\tau = \frac{1}{\theta}\left(\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right) \qquad (3.11)$$

The foregoing derived quantities are of interest. The additional expected number of failures required to reach the failure intensity objective gives some idea of the failure correction workload. The additional execution time required gives an indication of the remaining amount of test required. Both these quantities are also used in making estimates of the additional calendar time required to reach your failure intensity objective. How to do this, will be described in the next section about the Calendar Time Component.

3.6.2 Calendar Time Component

If you have calculated the additional execution time required to reach your objective, you, or at least your manager, probably would like to know how many days it will take. To do this you have to convert execution time to calendar time, which is exactly what the Calendar time Component do. During a project, the resources available will be more or less constant and established in its early stages. Increases are generally not feasible during the system test phase because of the long lead times required for training and computer procurement. There will always be one of these resources limiting the rate at which execution time can be spent per unit calendar time.

The limiting resource will change during the test phase. At the beginning, the failure intensity will be high and the time interval between failures will be short. Testing must be stopped from time to time to let the failure correction personnel keep up with the load. Thus, the limiting resource will here be the failure correction personnel. After a while the failure intensity has decreased and there is not so many failures to correct. At this point it is the failure identification personnel who is the bottleneck. Finally, when the failure intensity has decreased even more, the capacity of the computing facilities becomes the limiting resource. This resource then determines how much testing that will be accomplished.

All the different parameters used and their connections to the three different resources are listed in table 4 and explained in table 5. The resource utilization of failure identification personnel ρ_I is 1 because it is not restricted by any queuing constraints. As you can see in table 4, the failure correction is not dependent on execution time. This is quite obvious since the program is not running during changes. However, the failure correction time is dependent on mean failure experienced μ and this is in turn dependent on execution time. This makes the resource usage a function of execution time only. The parameter P_C is also a little special. It represents the available computer time in terms of prescribed work periods. For example, if available computer time per week is 80 hours and the prescribed work week is 40 hours, then $P_C = 2$. This unit was chosen so that computer resources would be stated in the same units as personnel.

Resources	Usage parameters		Planned parameters	
	requirements per:			
	CPU hour	Failure	Quantities available	Utilizations
Failure identification personnel	θ_I	μ_I	P_I	$\rho_I=1$
Failure correction personnel	$\theta_F=0$	μ_F	P_F	ρ_F
Computer time	θ_C	μ_C	P_C	ρ_C

Table 4: Calendar Time Component Resources and Parameters

Parameter	Explanation
θ_I	Average failure identification work expended per unit execution time (e.g.person hours per execution hours)
θ_C	Average computer time expended per unit execution time
μ_I	Average failure identification work required per failure (hours per failure)
μ_F	Average failure correction work required per failure (hours per failure)
μ_C	Average computer time required per failure (hours per failure)
P_I	Number of available failure identification personnel
P_F	Number of available failure correction personnel
P_C	Available computer time (measured in terms of prescribed work periods)
ρ_I	Failure identification personnel utilization factor during the failure-identification-personnel-limited period
ρ_F	Failure correction personnel utilization factor during the failure-correction-personnel-limited period
ρ_C	Computer utilization factor during the computer-limited period

Table 5: Parameter explanations

Equation 3.13 shows how the usage χ of a resource r (I, F or C) is computed. τ is still the execution time.

$$\text{Usage} \qquad \chi_r = \theta_r\tau + \mu_r\mu \qquad (3.13)$$

> ***Example 3.3:*** *Suppose a test team runs test cases for 10 CPU hours and during this time they identify 30 failures. The resource usage is 8 person hours per execution hour and 2 person hour per failure. The total failure identification effort required will then, by using equation 3.13, be*
>
> $$\chi_I = \theta_I\tau + \mu_I\mu = 8(10) + 2(30) = 140 \ \textit{person hours}$$
>
> *We can also calculate the effort per execution hour by dividing the result with the execution time (140/10 = 14 person hours / execution hours) or the effort per failure by dividing the result with the number of failures (140/30 = 4.67 person hours / failure).*

If we differentiate equation 3.13 with respect to execution time, we will get an expression, see equation 3.14 that gives us the change in resource usage per unit of execution time. Since the failure intensity decreases with testing, the effort used per hour of execution time also tends to decrease with testing, see figure 3.16.

$$\text{Usage per time unit} \qquad \frac{d\chi_r}{d\tau} = \theta_r + \mu_r\lambda \qquad (3.14)$$

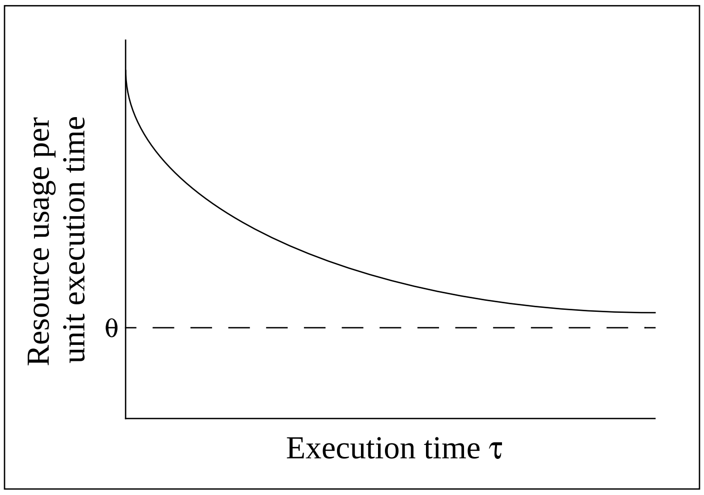


Figure 3.16: Variation of resource usage per unit execution time with execution time

We can also differentiate equation 3.13 with respect to failures. This will give us the change in resource usage per failure, see equation 3.15. Note that the execution time between failures tends to increase with testing. Thus, failure identification effort and computer time required for each failure tend to increase throughout the test period, see figure 3.17.

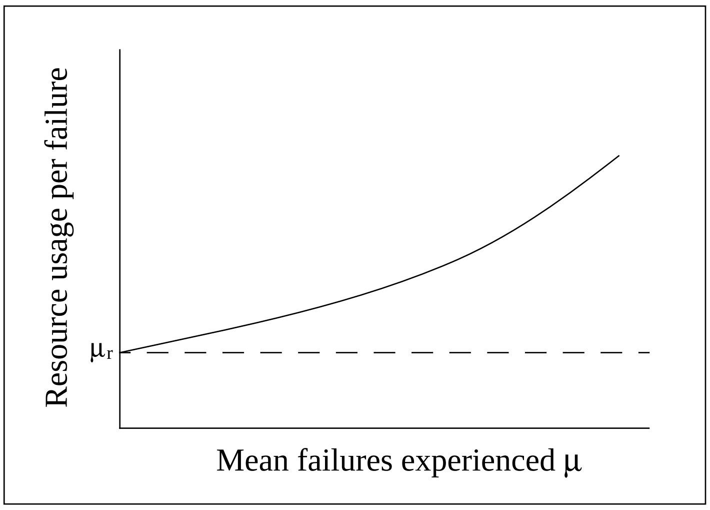


Figure 3.17: Variation of resource usage per failure with mean failures experienced

$$\textit{Usage per failure} \qquad \frac{d\chi_r}{d\mu} = \theta_r \frac{d\tau}{d\mu} + \mu_r \qquad (3.15)$$

As mentioned before, it is hard to increase the resources during the test phase, thus it can be assumed that resource quantities and utilizations are constant for the period over which the model is being applied. By dividing the resource usage per unit execution time, equation 3.14, with the resources available that can be utilized, we get the instantaneous ratio of calendar time to execution time, see equation 3.16. t is here the calendar time.

$$\textit{Time ratio} \qquad \frac{dt}{d\tau} = \frac{1}{\rho_r P_r} \frac{d\chi_r}{d\tau} = \frac{\theta_r + \mu_r \lambda}{\rho_r P_r} \qquad (3.16)$$

Example 2: *Consider example 1 and assume that it is the failure identification personnel (resource I) that is the limiting one. Recall that the 14 person hours of effort were used per hour of execution time, on the average. Suppose there are 2 members and they are fully utilized. Then the calendar time to execution time ratio will be:*

$$\frac{\Delta t}{\Delta \tau} = \frac{1}{\rho_I P_I} \frac{\Delta \chi_I}{\Delta \tau} = \frac{1}{1(2)}(14) = 7$$

*We can also compute the elapsed calendar time to 7*10=70 hour which is almost 2 working weeks.*

Since the limiting resource changes during the test phase, you have to calculate the calendar time to execution time ratio for all resources to see which one is the bottleneck, has the highest ratio. If you plot the instantaneous ratio of calendar time to execution time for all your resources you can see the rate at which calendar time is expended, see figure 3.18. You can also see where the changes between different limiting resources occur.

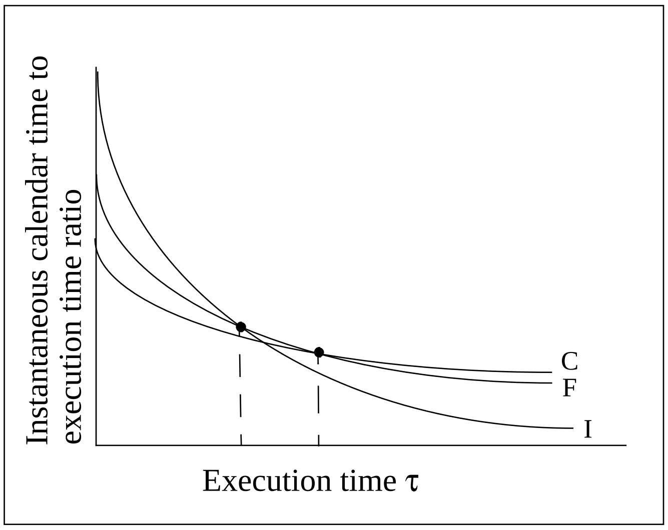


Figure 3.18: Calendar time to execution time ratio
for different limiting resources

3.6.3 Parameter Determination

To use these two models you have to determine a couple of parameters. This section briefly describes how this is done. For a more thorough interpretation see [6].

The different parameters are summarized in table 6 and 7. One thing you have to bear in mind is that the values of the parameters are estimations of their real value. Therefore you should always use confidence intervals. Table 8 shows which methods you can use to determine the different parameter values.

Parameter	***Basic***	***Logarithmic***
Initial failure intensity	λ_0	λ_0
Failure intensity change: Total failures Failure intensity decay parameter	 v_0 -	 - θ

Table 6: Execution Time Component parameters

Resources	**Usage parameters**		**Planned parameters**	
	requirements per:			
	CPU hour	**Failure**	**Quantities available**	**Utilizations**
Failure identification personnel	θ_I	μ_I	P_I	ρ_I
Failure correction personnel	0	μ_F	P_F	ρ_F
Computer time	θ_C	μ_C	P_C	ρ_C

Table 7: Calendar Time Component Resources and Parameters

Parameter group	**Method**				
	Pred-iction	**Est-imation**	**Ident-ification**	**Formula and/or Experience**	**Data**
Execution Time Component					
Basic	X	X			
Logarithmic		X			
Calendar Time Component (both models)					
Planned					
Resource quantity			X		
Resource utilization				X	
Resource usage					X

Table 8: Method of parameter determination

Execution Time Component
The Execution Time Component parameters for *Basic* can be determined both by prediction and estimation. Prediction is done before execution of the program and uses various characteristics of the program code. However, we will not use this since we do not have enough experience and no previous projects to rely on. Erlang is a quite new computer language and there has not been much research in this area.

When the program has been executed and failure data has been collected one can determine the Execution Time Component parameters for both *Basic* and *Logarithmic* using maximum likelihood estimates. It is most efficient to use a program to do the calculation. The process is illustrated schematically for *Basic* in figure 3.19.

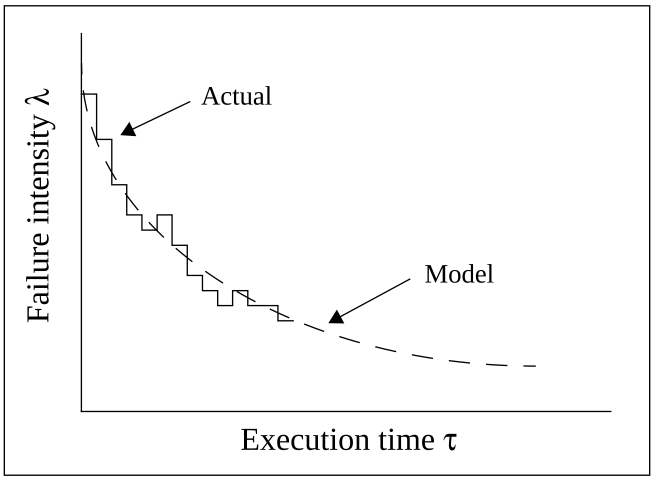


Figure 3.19: Conceptual view of parameter estimation

The solid line is the actual failure intensity. This is calculated as number of failures in a time interval divided by that time interval. *Basic* then provides the basis for plotting the dashed line of anticipated failure intensity. This line is determined by the two parameters λ_0, initial failure intensity, and V_0, total expected failures. The values of these should be chosen to maximize the likelihood of occurrence of the set of failures intensities that have been experienced. In an approximate sense, the curve is being fit to the data indicated. The process is exactly the same for *Logarithmic*. The only difference is that you have a different failure intensity function. This has also two parameters namely λ_0, initial failure intensity, and θ, failure intensity decay parameter.

Calendar Time Component

Table 8 shows that the values of the Calendar Time Component parameters is the same for both models. The resources you should include are those who can limit the amount of work done, e.g. test or failure correction. In some projects, the failure identification personnel and the failure correction personnel are the same persons. In this case you must merge the resources and the resource requirements.

The resource quantities available are quite easily determined by identification, i.e. ask the person responsible, usually the project manager. Note that available is not the same thing as used. Personnel are simply counted regardless of the length or time of their working hours.

The resource usage parameters tend to be dependent on the factors of programmer skill level, level of task difficulty and development environment. Their values are determined through experience and prior data.

3.6.4 Model choice

The choice of which of the two models to use in, any given application, depends on several factors, see table 9. Sometimes you may be able to use both of them and sometimes the choice is critical. One option may be to employ *Basic* for pretest studies and estimates and during periods of evolution. When integration is complete you should switch to *Logarithmic*.

Purpose of application or existence of condition	***Basic***	***Logarithmic***
Studies or predictions before execution	X	
Studying effects of software engineering technology (through study of faults)	X	
Program size changing continually and substantially	X	
Highly nonuniform operational profile		X
Early predictive validity important		X

Table 9: Choice between models

3.7 Software Reliability Tools

As described in this chapter there are a lot of work involved with software reliability engineering. If we only look at reliability growth testing there are still several steps that needs to be executed:

1. Collecting failure and test time information.
2. Calculating estimates of model parameters using the information.
3. Testing the fit of a model against the collected information.
4. Selecting a model to make predictions of remaining faults, time to test, or other items of interest.
5. Applying the model.

To simplify this work there have been several automatic tools developed. Lyu gives in [11] a summary of some of these tools, see table 10. Note that there is a constant flow of new tools being developed and it is a good idea to look around before choosing one.

Tool name	Models	Minimum Operating System	Current release/ Original release
Statistical modelling and estimation of reliability functions for software (SMERFS)	Littlewood/Verrall Musa Basic Musa/Okumoto Jelsinski-Moranda Geometric Execution Time NHPP Generalized Poisson NHPP Brooks/Motely Schneidewind S-Shaped	DEC VMS MS DOS 3.0 Cyber Operating System	Oct-93/ Oct-83
Software Reliability Modelling Programs (SRMP)	Musa/Okumoto Duane Jelsinski/Moranda (JM) Goel/Okumoto Bayesian JM Littlewood/Verrall Littlewood Keiller/Littlewood Littlewood NHPP	MS DOS 3.0	May-88/ May-88
GOEL	Goel/Okumoto	MS DOS 2.11	Nov-87/ Nov-87
ESTM	Goel/Okumoto with economic testing criteria	UNIX System	Jun-93/ 87
AT&T SRE Toolkit	Musa Basic Musa/Okumoto	UNIX System	May-91/ May-91
SoRel	Goel/Okumoto Littlewood/Verrall Hyperexponential Yamade S-Shaped	Macintosh	May-91/ May-91

Table 10: Software Reliability Tools

Tool name	Models	Minimum Operating System	Current release/ Original release
CASRE	Littlewood/Verrall Musa Basic Musa/Okumoto Jelsinski-Moranda Geometric Execution Time NHPP Generalized Poisson NHPP Brooks/Motely Schneidewind S-Shaped	MS DOS 5.0 or higher with Windows 3.1 Windows NT Windows 95	95/ 94

Table 10: Software Reliability Tools

CHAPTER 4

Monitoring

In this chapter we present monitoring theory used in this thesis and discuss what have been done by others. We will also describe a project in the field of monitoring that SERC has undertaken which is related to our work.

4.1 Overview

To be able to develop and deliver a software product with good quality, you have to understand the program and know how it behaves in its real environment. To make this easier, companies have started to develop program understanding tools. These helps the developers and the software quality assurance group to understand the program in full. These tools can be separated into two groups, *Static analysis tools* and *Dynamic analysis tools* [12]. With *Static analysis tools* you examine the program text and provide data for the program that is true for all kind of inputs to the program. The group we concentrate on is *Dynamic analysis tools*, also called *program execution monitors*. This means a program that monitors the execution of another program. With monitoring means that you collect information that helps you to supervise the program during execution. You can collect different information depending on what you are interested in.

Still there are a lot of companies, that are not performing any monitoring on their programs. Actually, we found out that there is no real good common way of program execution monitoring and there is not much information in this area. We also found that the companies, that are performing monitoring are doing this by collecting failures and special events, whatever they want to monitor, manually. So, our conclusion is that there are still a lot of work that has to be done in the area of monitoring. We will now describe the most common theories behind a program execution monitor.

4.1.1 Program execution monitor

The first thing you have to decide before performing an execution monitor is if you like to present the information to the user run-time or post-mortem. With run-time you present the information as the program executes and with post-mortem you present the information after the execution completes. Except from this there are three different main aspects you can characterize existing system in. These are:

Information sources and access methods

The most critical things when designing a monitor are how to obtain information from, and how to access, the target program. It is very important to solve these problem in an efficient way to reach a good quality in your monitoring results. The four most common methods to do this are, *run-time instrumentation* [13], *manual instrumentation* [14], *interpreter instrumentation* [15] and *instrumenting compilers* [16].

Run-time instrumentation means that you make the adjustments in the target program code prior to or during execution. These adjustments are often that you replace an instruction with a jump or operating system trap. Instead of executing this instruction you jump to some code that send information to the monitor. You make this modifications in a few selective areas and let the rest of the program execute as normal.

In *manual instrumentation* you implement code manually into the program being monitored. This method demands a lot of work for each program that should be monitored. *Manual instrumentation* are mostly used in system for algorithm animation and as a debugging tool when other debugging tools are ineffective.

When using *interpreter instrumentation* you do not make any modification in the program being monitored. Instead you make some changes in the language interpreter and you get the information from all programs executed by the interpreter. *Interpreter instrumentation* is common for high-level languages and varies widely in the range of features that are instrumented and the nature of the algorithm to be animated.

The last one, *instrumenting compilers,* modifies the code in the preprocessors and in the code generators as they produce output. With this method you can instrument any program written in the language that the compiler recognize.

Execution Models

There are a lot of different models for the relationship between the monitor and the program being monitored. However, it is possible to distinguish three main models: the *one-process model* [14,17], *the two-process model* [13,18] and the *thread model* [19].

In the *one-process model,* the monitor runs in the same process as the target program. The advantages of the *one-process model* are, simple, highest-performance arrangement and that the monitor has convenient access to the program being monitored. The disadvantages are, code intrusive and that a failure in the target program or in the monitor code affect each other in a critical way.

In the *two-process model,* the monitor runs in a separate process. The advantages are that it reduces the problem with that failures in the monitor code or the target program affects each other. The disadvantages are, that it becomes much more complicated to access the program being monitored. This can lead to performance problems.

The thread model works as if the monitor is a separate thread in a shared address space occupied by the program. This model has some of the advantages and disadvantages from both the one-process model and the two-process model. E.g. the risk that an error in either the monitor code or the target program will affect each other.

User-interaction facilities
How should you present the information for the user? You can present it as text or with graphics or maybe both. It should be up to the user to choose how he wants it. Another thing about how you present the data is if you should update the data continuously during execution, or provide the data during breaks in the execution. The third thing is how much you can control from your monitor interface. It could be everything between, start and stop the execution to modifying the target program.

Which methods you choose from above depends on several aspects and it is hard to recommend one or another.

4.2 Monitoring non-functional requirements

At SERC they have undertaken a project about how to perform continuous monitoring of non-functional requirements, during the lifetime of a software system. Non-functional requirement means e.g. reliability, maintainability and performance. To do this they started to develop a dual pentium based system using a new Erlang compiler. Parallel to this project there has been a few Master thesis projects running to see how to handle the non-functional requirements in an Erlang system. The outcome from this project is that you by specifying certain characteristics into the compiler makes it possible to monitoring the non-functional requirements. Then the monitoring process will be autobuilt by the system. Thus, this system will use the methods instrumenting compiler and two-process model described above. As a matter of fact it takes the model instrumenting compiler one step further when it autobuild the monitoring process. To avoid the disadvantages, that the target program becomes more complicated to access, when using the two-process model, SERC use Erlang, supporting distributed systems and messages sending between processes. Since they also use a dual pentium system, where the monitor executes on one processor and the target program on the other, they will avoid performance problem.

The essential components of the monitoring infrastructure are shown in Figure 4.1. In the upper part of the picture you have the setup environment. There are two different branches into this environment. The input to the left branch is your target program, the program you like to execute, and the input to the right branch are some parameters that are required for being able to do the monitoring. This environment compiles the target program and autobuild the monitoring process. The two programs execute on two different processors and communicate with message sending.

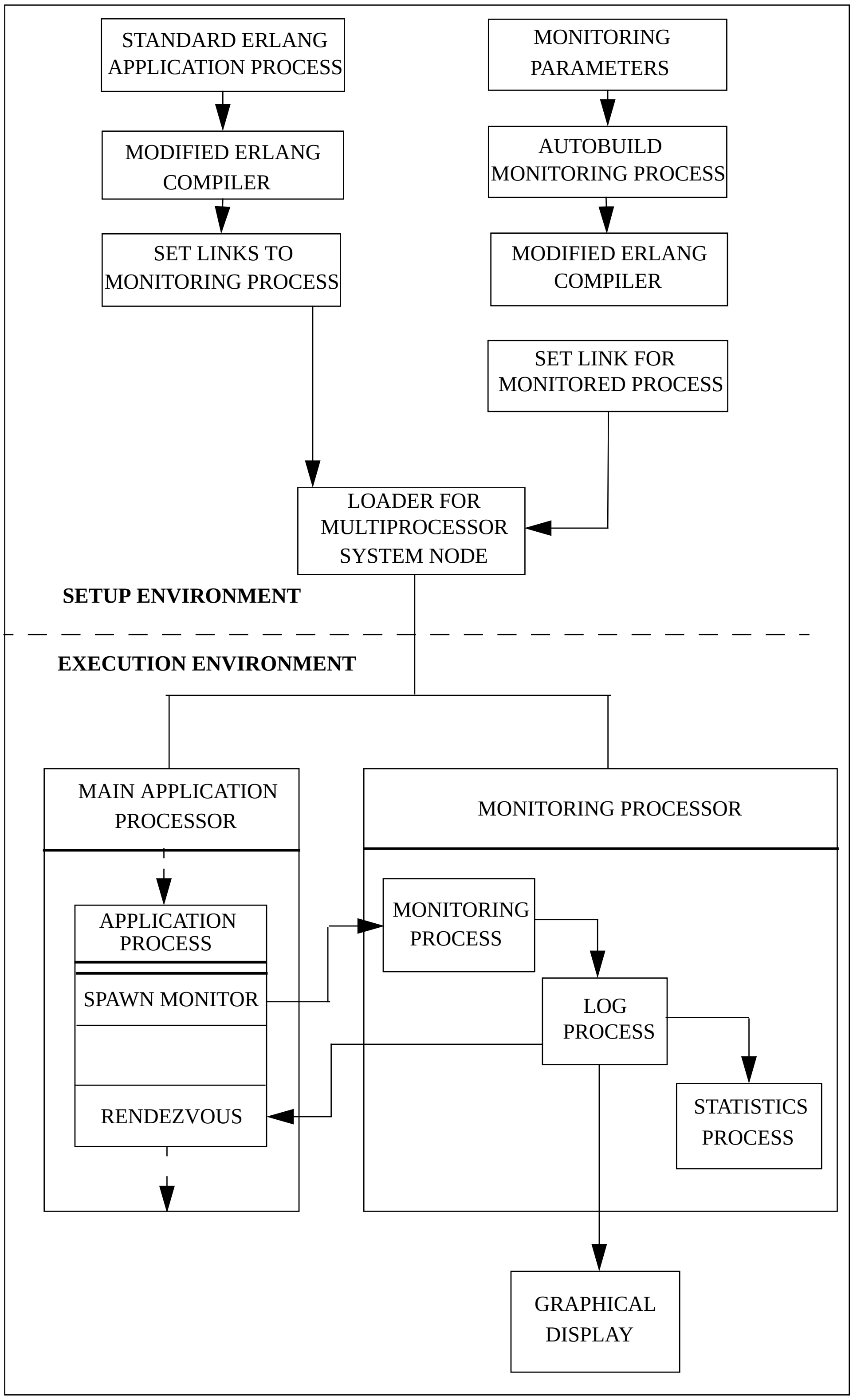


Figure 4.1: The essential components of monitoring infrastructure

CHAPTER 5

Computer environment

5.1 Erlang System/OTP

The computer language Erlang has, as mentioned in chapter 1, been developed by CSLab (Ericsson Computer Science Laboratory) in about the last ten years. The idea was to develop a language especially for telecommunication systems. Erlang's roots come from Prolog, Strand, Parlog and Eri-Pascal and it is a functional language which supports parallelism. A functional language is a language in which programs are constructed from a large number of functions [20]. The key features of Erlang are:

- **Concurrency -**Erlang has a process-based model of concurrency with asynchronous message passing. This makes it possible for the user to have full control over the sequential flow of execution and at the same time separate different tasks in different independent flows. The processes require little memory and creating, deleting and message passing require little computational resources.

- **Real-time** -Erlang is intended for programming soft real-time[1] systems where response times in the order of milliseconds are required.

- **Continuous operation** -Primitives for code replacement in runtime. This means that you can upgrade your system with a new version without shutting it down.

- **Robustness** -Mechanisms to detect run-time errors.

- **Memory management -**Erlang is a symbolic programming language with a real-time garbage collector.

- **Distribution** -There is no shared memory. All interaction between processes is by asynchronous message passing, which is transparent in respect to where in a computer network the processes are located.

- **Integration -**Erlang can easily call or make use of programs written in other languages.

- **Portability** -Current support exists for several platforms.

1. Soft Real-time is when you can “stretch” time, e.g. it does not matter if something takes 1,2 or 5 milliseconds.

To support the development of Erlang systems, CSLab has also developed a Open telecom Platform called Erlang System/OTP. The platform which is based on Erlang, contains a number of well tested modules which solves critical basic problems. The environment supports short time to market development, openness to sourced hardware/software components, platform independence and low costs for development and operations. Erlang System/OTP version 4.5.3 is available on a number of different platforms. These are SunOS 4, Solaris 2 (Sparc and intel/x86), Linux, Windows NT (4.*) and Windows 95.

Erlang System/OTP runs on standard computer hardware and operating systems. Figure 5:1 shows the structure of Erlang system/OTP and its key features are:

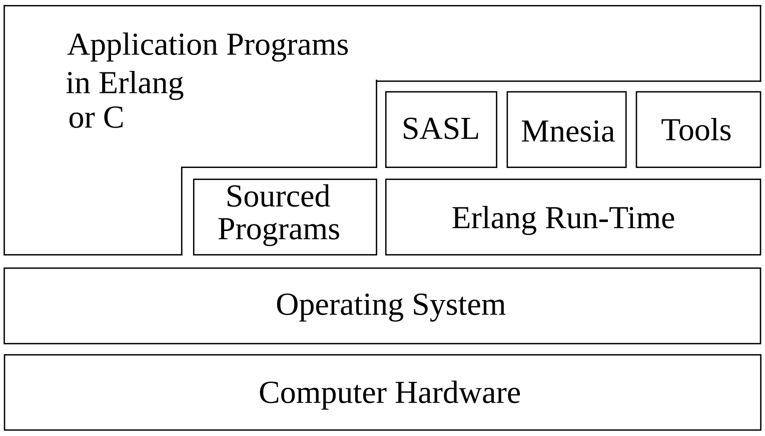


Figure 5.1: Open Telecom Platform

- **Erlang Run-Time** -This is the Erlang emulator that is running your Erlang code. It is written in C and contains commands and the Erl Interface C Library Functions [21]. The Run-Time system is a virtual machine, which handles memory management, light weight processes and real time behaviour.

- **SASL** -The SASL is the System Architecture Support Library which handles Start up scripts, Error handling, debugging and high level software upgrades, in runtime without shutdown. SASL also contains an application concept. This helps the developer to decompose the system into different applications and then program these with the help of a number of behaviours. The behaviours are design patterns such as application, supervisor and server.

- **Mnesia** -This is a real time, fault tolerant, distributed database. Complicated queries are supported by an optimising query compiler. Views are available as rules in the Mnemosyne query language [22].

- **Tools** -There are a number of different tools in OTP, e.g. development tools (compiler, debugger), performance tools (process monitors, coverage testers), jive (Erlang Java interconnection and inets (WWW server tools) [21].

- **Sourced Programs** -The OTP defines how sourced programs, which include protocol stacks and management applications, may be incorporated into a system and made to interwork with programs that were developed by application programmers.

- **Application Programs** -These programs are designed by the user and can be written in Erlang or C**.**

Erlang and Erlang system/OTP are constantly developed to include new functions and to support new platforms. There will also be changes in the already existing modules.

5.1.1 Program structure

Erlang uses, as mentioned earlier, a process based model of concurrency. This means that you can separate different tasks into different parallel processes. Figure 5.2 shows an example on how a program can be structured. The structure is dynamic, i.e. you can create new processes or terminate old ones as you like during execution. The lines in the figure shows which process that has created which (top-down). As default there is no connection between the processes once they have been created but if you like you can link them together. This way you will have full control over which processes you have in your system. In this example process A,C,E, and F are linked together. The links works both ways so it does not matter if process C has set up the link or if it is process E or F.

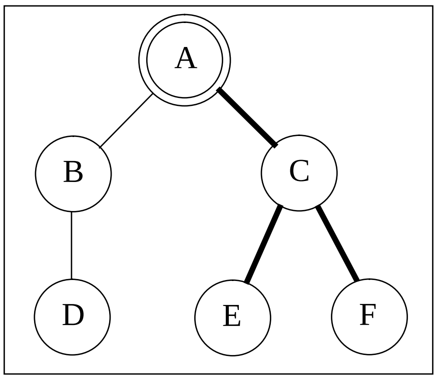


Figure 5.2: Process tree

The only way to communicate between processes is to send messages. This message passing is asynchronous which means that there is no synchronisation between sending and receiving messages. A process can send a message and then continue with something else regardless what happens with the message. The sent message will end up in the receiving process' mail box and will stay there until the process asks for it. One disadvantage with this is that you can not be sure that messages sent from different processes will arrive in the same order as they were sent. The only way to control this is to set a timestamp in the messages and then sort them after this.

5.1.2 Failure behaviour

When a failure occurs in a process it will terminate. At the same time it will also send a exit message, containing the reason for its termination, to all processes it is linked to. This will cause these processes to terminate with the same reason, and they will in turn send a exit message to the processes they are linked to. There is a way to stop this chain by trapping the exit messages (see process A). Then the process can take desired actions and if it wants to it can terminate in a latter stage. If we suppose that process E terminates it will send an exit message to process C. C will then terminate and send exit messages to process A and F. F will terminate immediately but A will trap the exit message and take action depending on with which reason process E terminated.

A big advantage with Erlang, compared to many other languages, is the failure handling. When a failure occurs in an Erlang program you will immediately get a notice about this. In other languages, for example C, you may not see the failure for after a few days, e.g. if a pointer in C points at the wrong location in the memory, or if you write over data in the memory that is needed later. These things cannot happen in an Erlang system, a failure can not hide.

5.2 The Erlang Engine

The Erlang Engine is a multiprocessor system within the context of AXE-10 and Erlang System/OTP [23]. The Engine is meant to be used for implementation of Intelligent Network (IN) applications and takes full advantage of the specific characteristics of an Erlang environment. As shown in figure 5.3 the Erlang Engine is a logically component of OTP but is physically a distinct Erlang node. The reason for separating the Erlang Engine is that the Erlang environment is based on the functional language Erlang which is significantly different from the procedural approach e.g. C/UNIX. This way each environment can be optimized based on its own characteristics.

The key features in figure 5.3 are:

- **High Speed Interface** -To be able to handle the growing requirements for IN applications the Engine must communicate with the AXE-10 at the maximum possible rate.

- **I/O** -This is a standard interface of OTP. The idea is to have a local and tightly coupled UNIX system so that a high speed link and minimum protocol may be utilised.

- **Kernel** -This is the operating system, which is to be chosen so that it fits the Erlang environment best way possible. One candidate is QNX.

- **Ericsson Core Software** -Surveys show that Ericsson will not be able to write all the applications by themselves. In fact, they will just be able to write 10% of what is needed. To handle this Ericsson will just write the critical applications, that among other things have to be free from side effects.

- **Outsourced Software** -This is the rest of the applications that a third party company will write. These applications should essentially be stateless which permits both a clear method for specifying the needed application components and generating a profitable contractual relationship.

There are several problems in the Erlang Engine concept that have to be solved. Today, there are a number of different projects running, which deals with different aspects of these problems. The Engine is constantly evolving and will hopefully help the AXE-10 to live for the next 10-15 years.

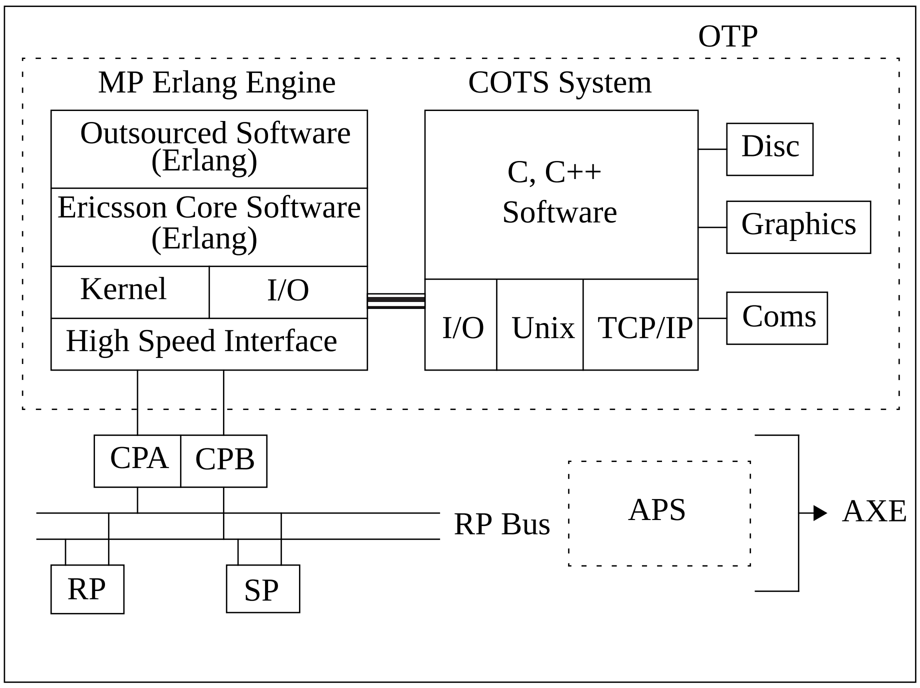


Figure 5.3: The Erlang Engine

CHAPTER 6

Possible Solutions

6.1 Overview

In this chapter we discuss a few different approaches to the problems in our thesis. There are four main parts in our thesis, where we came up with different solutions. We describe all these parts separately with their alternative solutions. In the end we specify how we decided to solve the problems and why we chose these solutions. The four theory parts that we talk about are:

- Which reliability measures shall we use?
- Shall the implementation of the monitoring code be automatic or semi-automatic?
- How can we divide the target program into smaller parts, which we shall measure the reliability on?
- When shall we monitor and for how long time?

Before we start to describe these parts we will give you a short overview of our solution, just to make the rest of this chapter easier to understand. For more detailed information on our solution, see chapter 7-9. In the solution we look at the reliability of the target program by finding out the reliability for small pieces of the program. It is important that these pieces can be specified in the contract. Our monitor program consists of three Erlang-processes that receive messages from the target program. These messages are sent by code implemented by us. The monitor program receives the messages and then takes the correct action. If you want to monitor a program with our method, you first start our monitor program. Into this program you send an infile where you have specified information about the target program. Then our program makes some changes in the code and then compiles the target program. After this, you start the target program and it will be monitored.

6.2 Different reliability solutions

When it comes to reliability one of the biggest problem is to develop a good understanding of the field. It is a very fuzzy field and almost everyone has his or her own opinion. When we started to read about reliability we thought that the problem was to find good definitions, it is not. The problem is how to conduct the measurements. You have to be very careful how you do this, otherwise your measures will not be usable.

In chapter 3 we have tried to describe the field without getting too deep down in small details. We have defined reliability in three different ways (probability, MTTF and failure intensity), which we believe will cover most situations and we have also given examples of several different models to use. As described in the chapter is it impossible to say which one to use, it depends on the situation.

Chapter 3 also describes how to choose test cases. This part is as important as the rest of the reliability theory. If you do not follow these guidelines, the theory is not applicable.

6.3 Automatic or semi-automatic

6.3.1 Automatic

With automatic implementation you should be able to execute the monitor program on every other program without doing any changes in the target program. All you need to do is specify some detailed input, about the target program, into the monitor program. By doing this you avoid manual instrumentation in the program and it is easy to turn off and on the monitoring. This method is good since there is always a risk to make mistakes when you implement new code manually in a program. When you want to monitor your program you monitor-compile the program, otherwise you perform a normal compilation. When you perform the monitor-compiling you create code in specified places in the target program. This code handles the sending of messages to the monitor process.

To perform the monitoring the automatic way would be the preferred way, but there are some major problems. To make this work, at least according to what we have come up with, you have to give the designer of the target program very specified rules on how to make the design and you also have to create a very detailed infile to the monitor program. The monitor program would also become very complex and large. This lead to that the automatic idea has lost its advantage of being easy to use on all programs, instead it creates the opposite affect. After we had drawn this conclusions we decided to try semi-automatic instead.

6.3.2 Semi-automatic

Semi-automatic is a solution between manual and automatic. Some of the code, that handles the message sending, will be implement in the monitor-compiling but some has to be manually implemented. The advantages of this method are, that you can use it on all programs independent of the design, and a big part of the implementation can be done automatic so it is not that labour intensive. The worst disadvantages are that you have to consider where this code should be implemented already in the design and there can be a problem to turn on and off the messages sent to the monitor. But still we think this is the best solution that we could come up with. In the end product from SERC all code will be automatic instrumented by the Erlang-compiler that we mentioned in chapter 2 and 4. How to do that is not in the scope of this thesis.

6.4 How to divide the target program

As we mentioned before we like to divide the target program into smaller subsystems and calculate the reliability for these subsystems. Then calculate the reliability for the whole system depending on the reliability of the subsystems. The problem was how to define a subsystem and how to calculate the total system's reliability, since you have to consider how important the different subsystems are. It is also important that you can specify the subsystems, and their influence on the reliability in the contract with the customer. We have been working with two different solutions.

6.4.1 Process- and function-subsystem

As we described in chapter 5 an Erlang program is built of processes. Our first theory was to divide the system into subsystems depending on the processes. Our idea was based on the theory that every process works with one application, e.g. a CGI-script, at the time and that you could match every process with an application. The user would have to specify, in a list, how important every process was. We would give them a couple of classes they could choose between. This list should be the base when we decided how important every failure was. We were also thinking about measuring the time spent in every process and all reductions made by functions in a specified process. A reduction can be described as a function call, and is a good way to measure the activity in a function. This should also help us to decide how important the process is. With this theory we could decide in which process the failure occurs and how important the process was, and thereby how imported the failure was.

Unfortunately, this theory failed due to a couple of reasons. The most important was that we could not decide how important the failure was for the end user only knowing the importance of the process. This is the case since a process can work with different applications and we could not restrict the design of a program to only one application in every process. A few other disadvantages with this solution are that it is hard to connect these subsystem to the contract, the monitor program would be very complex and the data sent between the target program and the monitor would be huge. Another problem with this solution is to calculate the total reliability of the system. The reason for this is that the different processes are not independent and their interrelations can be very complex. This makes the reliability calculations almost impossible. The only real advantage with this method is that the monitor program becomes a better debugger when we can decide more exactly where the failure occurs. But that is not supposed to be the outcome from this thesis.

6.4.2 Transaction subsystem

We decided to divide the target program into transactions instead. With a transaction we mean a user function going through the system. A few examples of a transaction in a webserver could be, loading a page or executing a CGI-script. By dividing the system into transactions we have obtained a lot of advantages compared to the other solution. Some of these are:

- Easy to decide how important the failure is for the end user. We know what he tried to do and what went wrong.

- It is easy to calculate the reliability for every transaction. You just fire these transactions through the system and see what will come out.

- It is easy to specify the different transactions in the contract and they are easy for the customer to understand.

- It is easy to calculate the reliability of the whole system. If you can get data about the user characterization it is easy to calculate how the total reliability depends on the subsystem's reliability.

The only disadvantage is that it will not be so easy to see where in the code the failure occurs, but we will use checkpoints to give you a possibility to roughly find out where it happened. We describe more about this in the next chapter.

6.5 When and for how long shall we monitor the program

In the next area we will discuss a couple of different solutions to monitor the target program. We will investigate three main solutions.

6.5.1 Only monitor during the test phase

One method could be to just perform the monitoring during the test phase. This was however never an alternative for us in this thesis. If we only do it then, we loose the information we are after. What we want to know are how the program behaves in its real environment and how it changes with time. You usually say that a software product does not grow old, but there could be some exceptions to this rule, e.g. if a failure occurs in a program and the program does not terminate, there can be strange things going on that you do not know about. In an Erlang program this could result in that a lot of processes are still alive even though they do not execute any more. This can for example lead to memory problems. There are a lot of other things that can happen in a program's real environment that we never can predict in a test-environment.

Another thing that is important for our solution is that we can see if an upgrade of the program will make it better or worse in the real environment. So, we definitely have to perform the monitoring in the real environment. There is however one advantage with only performing it in the test phase and that is that you gain better performance in the real environment. But since the end product from the monitoring projects at SERC, will execute on two processors, as described in chapter 4, this is not a problem for us.

6.5.2 Monitoring a certain time

One solution would be to monitor the program in the real environment for a certain time. You start to monitor the program and when the program has been executing for X hours without any failure we can stop the monitoring.

The disadvantages with this method, are:

- How long shall you monitor? What happens if you stop the monitor and then a failure occurs that terminates the system. Then you will not find out about this failure and what have gone wrong until it is to late. So, there is always a risk to trust the program and stop the monitoring.

- Another disadvantage is that you have to disconnect and connect the monitor between the monitoring. It is always risky to make changes in the code. And again, the performance gain is the only real advantage with this method. Like we said before we do not have to take this in consideration.

6.5.3 Monitoring all the time

Because of the disadvantages described about the other methods we choose to perform the monitoring all the time. By doing this we gain the following things:

- We have control of the program all the time. If something happens we will notice and can take care of the problem immediately. This can save a lot of money and time.
- We can also see what is happening to the program during a longer time in operation.
- We do not have to connect and disconnect the monitoring program. If we for example upgrade the program we can just keep on monitoring and see if the upgrade makes the program better or worse.

This method gives you a more stable and safer monitor.

CHAPTER 7

Our solution

In this chapter we will give a more detailed description of the solution we have come up with. The problems we should solve could be defined in two sentences.

- Define metrics which connects the customers's and the developers's view of the system's reliability.

- Find a way to monitor these metrics in an Erlang system during execution.

7.1 Metrics for calculating the reliability

The first part, mentioned above, is the more theoretical part of our solution. We have not only come up with a definition on reliability metrics, but also described important issues to think of when collecting these metrics. Further we have described different ways of analysing the metrics, both when it comes to calculate the current reliability and predicting the future reliability. The reason we have done this is because to be able to have full control over the reliability, which SERC wants to, you must consider all these aspects.

7.1.1 Definition

The definition we have used in our solution to calculate the reliability is "*The probability for failure free operation during a specified time in a specified environment*" and it has been calculated, by assuming constant failure intensity and exponentially distributed times between failures, according to equation 3.1. We have also chosen to express the reliability as Mean Time To Failure (MTTF) for those who prefer this measure.

7.1.2 System components

In chapter 3, we described how to break down the system in independent components. The reason for this is to be able to calculate the reliability for every component and then combine these to one system reliability. In our solution we have chosen to both divide the system in independent components but also to describe the system as a set of transactions. Transactions are more a way to look at the system from the user's point of view. The user is not interested in the different components of the system, he or she is more interested in what different types of functions the system performs and of course, if the system succeeds to perform these functions when required.

How do we then define a transaction? Examples of transactions could be a call forward or a ordinary call in a telephone switch, or a CGI-script or loading a normal page in a webserver. Sometimes you have to divide the transactions in the contract into sub transactions, to be able to follow them through the system. There are some specified guidelines to follow when you define your transactions. These are:

- It must be possible to connect the transaction to the contract with the customer.
- It must be possible to specify where the transaction starts and stops.
- If an Erlang process terminates, all transactions it works with fail.
- The same transaction can exist in many Erlang processes at the same time.
- The transaction should be a basic function in the system, so every system only consists of a small number of transactions.
- The different transactions should be independent.

When all the transaction types are defined you will be able to measure the reliability for every transaction type and also the whole system's reliability. If you find out that the reliability are too low, you can check which transaction that needs improvements.

To further simplify the search for finding the weak part in the program we also divide every transaction into smaller sequential parts. We do this by using checkpoints in every transaction. Checkpoints can be described as defined spots along the transaction's path through the system, see fig 7.1. We will be notified every time a checkpoint is passed by a transaction. This way we can decide between which checkpoints an eventual failure occurs and where the improvements have to be made.

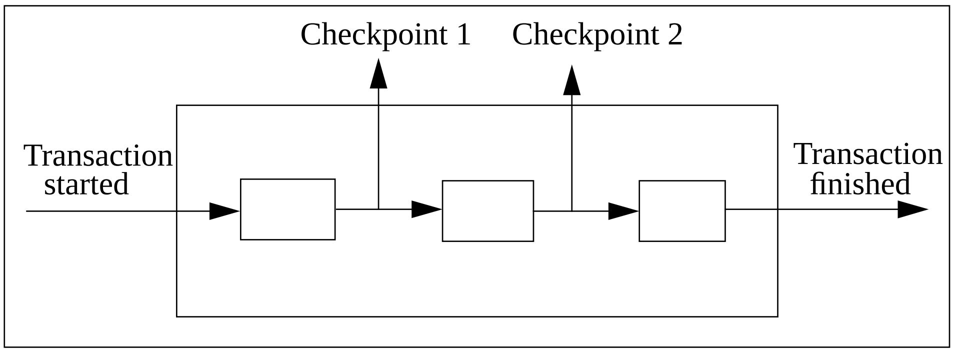


Figure 7.1: A transaction with its checkpoints.

There is a problem that may occur when you define your transactions. In some applications you cannot distinguish a special type of transaction in the beginning. All the transaction types can execute the same code in the beginning, so you cannot decide which transaction type it is. If a failure would occur during this common phase, how should we know which type of transaction the system tried to execute. The solution we come up with is that you should count all the failures for this common phase separately and then split them on the different transaction types. When you do this you have to consider how many transactions you have executed of every type, e.g. if 15% of all transactions was of type *vanilla*, 15% of the failures in the common phase should be added to this transaction type's failures and so on.

7.1.3 Analysing metrics

When the metrics have been collected it is time to analyse them, i.e. calculate the reliability. If you want to you can calculate the current reliability and then be satisfied with this, but then you will not be in control of the reliability. Instead you should use growth testing and the models described in chapter 3.3 to see how the reliability is evolving and also be able to see when you will reach your objectives. This way you will be in control over the reliability and know when to stop testing. If you have data from previous projects, you can even already in the end of the design phase predict your final reliability.

It is impossible to say which of the models described in chapter 3 you should use. As described, it depends on the situation. In the beginning it could be wise to use more than one model and then combine the results. This way you can get a more accurate result. When you get more experience you will know which model to use in the different situations. Note, that the models are based on statistical assumptions so you should always use confidence intervals.

7.1.4 Other aspects

Another really important aspect to think of is how to choose test cases. This should be done randomly with the probability according to the operational profile, described in chapter 3. If you do not do this, the reliability theory is not applicable. The reason for this is that all the models are, as mentioned before, built on statistical assumptions and since software failure occurrences are deterministic in themselves you have to introduce a randomness.

In chapter 3 we have given more aspects to think of when dealing with reliability but, these are the most important ones.

7.2 How do we monitor these metrics through the Erlang system

We will now describe the procedure of how we trace the transactions and the other information from the target program. The target program is the program that we like to monitor and calculate the reliability for.

7.2.1 Information sent about the transactions

We get the information from the target program by sending messages from the target program to the monitor at specified moments. To get the information about the transactions we send messages when the transaction starts, passes a checkpoint and when it is finished, see fig 7.2. When the transaction starts we create a Transaction Identifier, Tid, that are specific for this transaction. We use the Tid to identify every transaction through the target program and in the monitor. Then we send the message to the monitor including the start time for the transaction, the Tid and the Process Identifier, Pid, the transaction started in.

Then when the transaction arrives to a checkpoint we send the next message, including the time the transaction arrived to this checkpoint, the Tid and the Checkpoint Identifier, Cid. The reason for using checkpoint identifiers is that transactions, of the same type, could go different paths through the system depending on the input and the program environment. By using Cids we know which checkpoints we passed and which path the transaction took[1]. This message will of course be sent for every checkpoint that the transaction pass. Then when the transaction is finished we send a new message including the stop time, the Tid and the Pid we finished in. All information belonging to a specified Tid will be saved together by the monitor in a file.

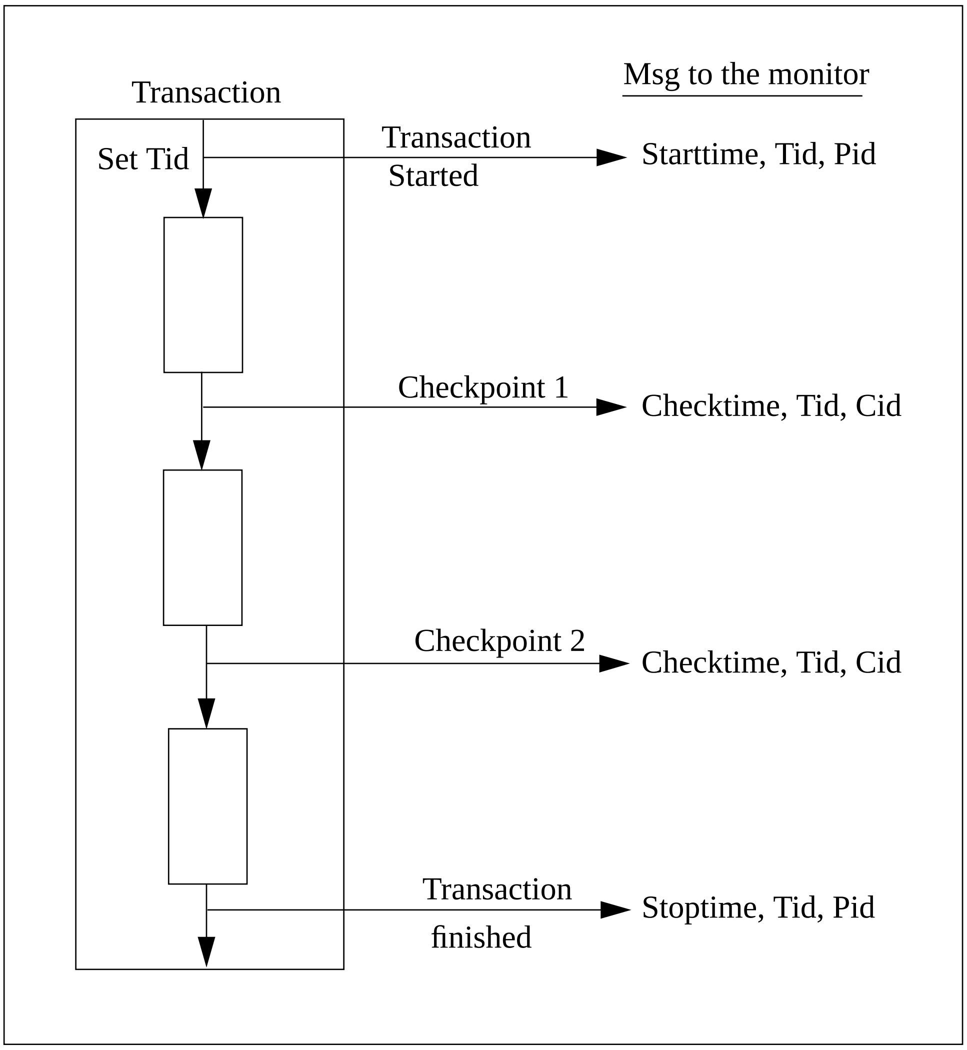


Figure 7.2: Transaction messages that sends to the monitor

7.2.2 How do we connect a failure to a transaction

Erlang is a good language if you like to trap failures. The reason for this is that you immediately notice all failures that occur, as described in chapter 5.1.2. So, when a failure occurs we immediately get a message from the target program. This message includes the Pid the failure occurred in, but we are interested to know which transaction it was that failed. How do we connect a failure to a specific transaction?

1. Cids have not been implemented in this version of the program.

To solve this problem we need to have full control over the transactions in the system. We need to know which transaction, or transactions, every Pid are working with. To do this we once again use message sending. Every time a process starts working with a transaction it sends a message to the monitor and every time a process has finished to work with a transaction it sends a message, see Fig 7.3.

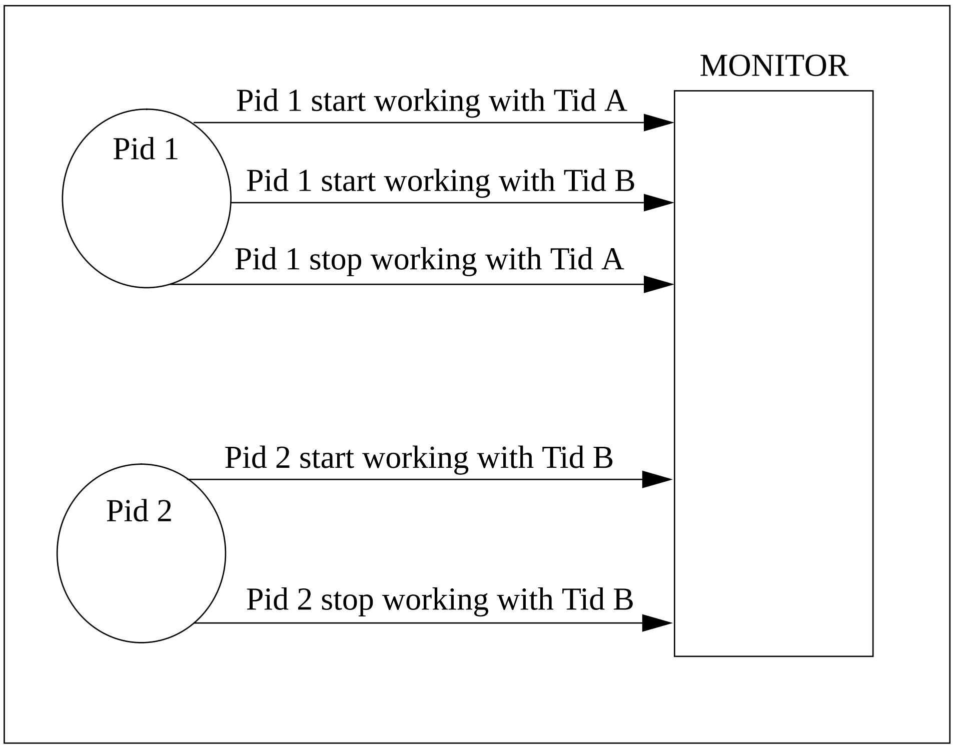

MONITOR
Pid 1
Pid 1 start working with Tid A
Pid 1 start working with Tid B
Pid 1 stop working with Tid A
Pid 2
Pid 2 start working with Tid B
Pid 2 stop working with Tid B

Figure 7.3: Messages that help you connect a failure to a transaction.

We then save this information in a list containing all information about which transactions that every process works with. So, when a failure occurs we get information about which Pid that terminated, then we check the list to see which transactions this Pid was working with. We then see these transactions as failed transactions and save them in a file. Then we take these transactions out from all the other Pids in the list. This way we know exactly which transaction, or transactions, the failure affected.

7.2.3 Future failure

Except from ordinary failures we also measure something we call future failure. With this we mean failures that do not cause a transaction to fail today, but indicates that something is wrong, which may result in a failed transaction in the future. There are two types of future failures, catched failures and message to a non existing process.

A catched failure occurs when an ordinary failure occurs in the monitored program, but the monitored program catches it by using the Erlang command catch[1]. It is not enough to only catch the failure, the program must also be able to handle this particular failure, otherwise a new failure will occur.

The second type of future failure, message to a non existing process, occurs when a process sends a message to another process which does not exist. This can for example depend on a fault in the sending command or if the receiving process has terminated abnormally, due to a failure.

These two values can help you to see, at an early stage, that something has gone wrong and prevent a future system breakdown. They could also be useful when you make an upgrade of a program, if one of these two rises after the upgrade, maybe the upgrade was not that good.

7.2.4 Information sent connected to failure and future failure

The information belonging to failure and further failure are, as mentioned above, sent by messages from the target program. When a failure, that is not within a catch-clause, occurs the process it occurred in will terminate. The target program will then send a message to our monitor process including the Pid, reason and time.

When a *catch-error* occurs the target program sends the Pid it occurred in, the expression, Reason and the time when it occurred. A catch-clause often look like this:

```
catch case function_A([Argument_list]) of
        ........
        ........
end
```

where function_A([Argument_list]) is the expression.

The reason is an Erlang-term describing the failure that occurred.

When a message is sent to a process that does not exist, the monitor receives a message including the message that was sent, the Pid it was sent from, the Pid it was sent to and the time when it was sent.

7.2.5 How do we get the target program to send the messages

To send these messages from the target program to the monitor we have to instrument code at some specified places in the target program. As we described in the previous chapter we do this instrumentation semi-automatic, i.e. both manual and automatic. All messages that help us to control the transactions have to be made manually and all messages containing information about failure and future failure are instrumented automatically. We will describe these two groups separately and start with the transaction messages.

1. Catch and throw are the Erlang exception handling primitives

Transaction messages. With transaction messages we mean, start messages, checkpoint messages, stop messages and the messages that help us to control which transactions a Pid works with. As we said above, the code that sends these messages has to be manually instrumented. You should decide where this code should be placed already in the design and this code should be written at the same time as the rest of the code.

Another thing that also has to be considered, already in the design, is how to send the Tid in all messages. A transaction could start in one process, having its checkpoints in another process and end in a third process. You then have to send the Tid as a parameter through the system together with the transaction, so we can include it in all the messages.

If the system is designed to have transaction start, stop and all checkpoints in the same process there is a short cut to this problem. You can then use the Erlang commands *get* and *put*. *Put* takes a value, the Tid, and put it in the process dictionary together with a key *key* and *get* gets the value, the Tid, with key *key.* If you can do it this way, you do not have to send it trough the whole system.

Failure and future failure messages. These messages are the catch-error-, message to a non existing process - and exit messages. The code that sends these messages are automatically instrumented. To get the messages belonging to exit and messages to a non existing process we use an Erlang command called *trace*. When you set the trace-flag = true you make it possible to trace the processes in this file. You can start this tracing from another process, the monitor. The process that is being traced will then send a message, to the process that started the trace, every time the process exits or sends a message to a non existing process. You can also set a flag that makes it possible to trace all the processes that the traced process spawns. So, by starting the trace in the right place you can easily trace a whole system. It is very important that you really think through where you should start to trace.

We cannot get the messages, that contain information about the catch-error, by using trace. Instead we wrote a program that look through the executable source code of the target program and swaps all catch commands to a function that we called hks:catch (Hans's and Kent's catch). We also use this program to instrument where we shall start the trace, more about this soon. The hks:catch works exactly as a normal catch except from that it also sends a message to the monitor server.

7.3 Overview of our practical solution and guidelines for usage

Here we present an overview of our practical solution, how it works in a higher level and guidelines for usage.

7.3.1 Overview of the practical solution

When you perform the monitoring, the system consists of the three bigger parts, the target program, the monitor server and the analysis program, see Fig 7.4. The target program and the monitor program are executed at the same node while the analysis program is executed at another node. By doing it this way you can sit at the office and make the analysis on the target program, even though this executes at some other place in its real environment. It is no problem to implement the solution this way, since Erlang supports distributed programming and message sending between different nodes.

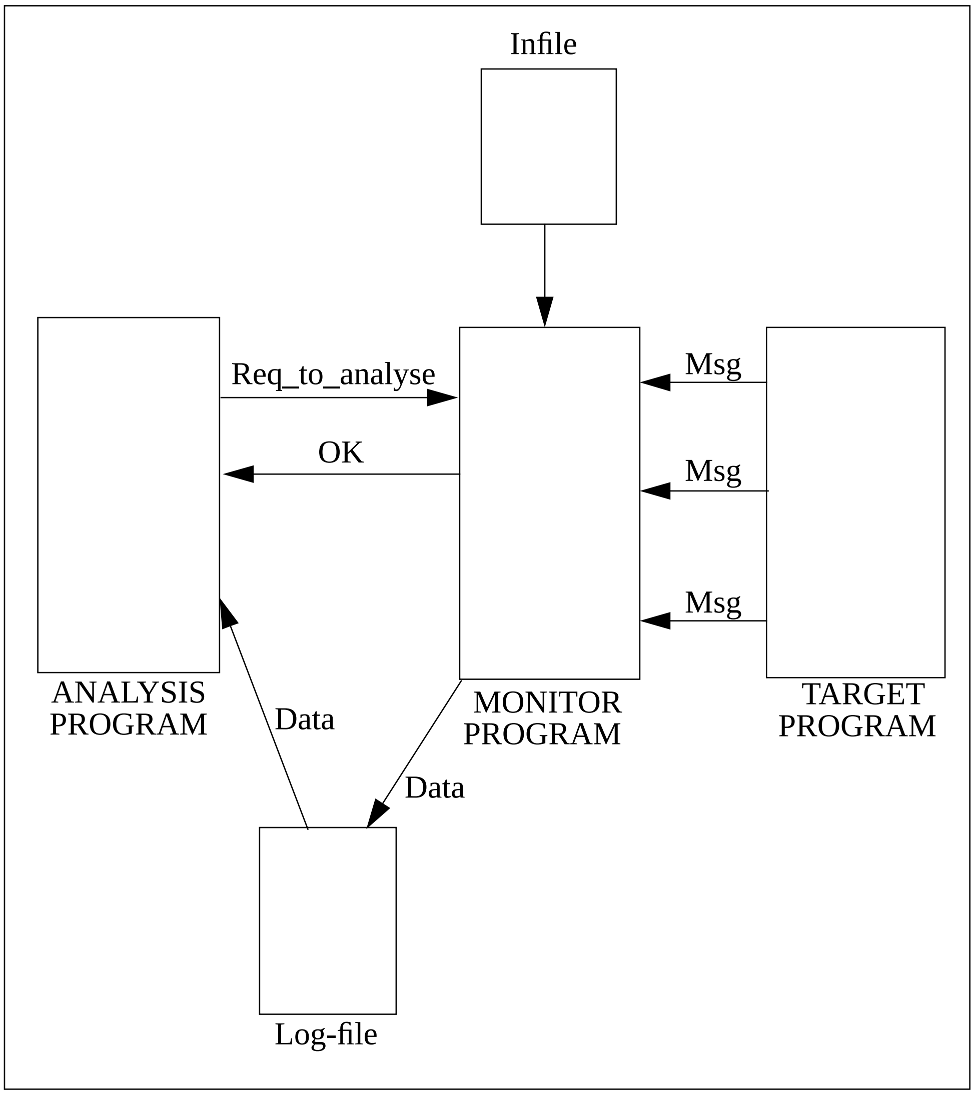


Figure 7.4: An overview of how the programs works.

The monitor program and the analysis program are more described in the next two chapter,

The first thing you have to do, to start up the system, is to write an infile. This infile contains the following information: the names of all modules in the target program, a start function including the name of the module and how many arguments there are in this function. For example an infile could look like this:

```
{files, [httpd, httpd_conf, httpd_example, httpd_listener
.........................................]}.
{startfkn,[{httpd_listener,connection,4}]}.
```

Then you start the monitor program with the infile as an argument. The monitor program then works like a preprocessor and swap all the catch-clauses, as we described above, in all the files specified in the infile. The start function defines where to start trace the target program, also described above. Our monitor program then also instruments code, at this location, that starts the trace of this process. We will then also trace all processes this process spawns. Then we compile all the specified files with the trace flag set to true. Now the system is ready to start up. After you started it you can perform an analysis without turning off the monitor, more about this in chapter 9.

7.3.2 Example of the implemented code

We will now give an example of how the target program's source code look like before and after we have done our implementation. Assume that we have the code below in the target program from the beginning. Note that this is only an example, written in pseudo code, for a webserver loading a page. Of course this code is simplified but it does not matter since we only use it for this example. For those that do not know Erlang there might be a problem to understand parts of this example, but hopeful the fundamental idea will be understood.

```
request_from_a_client() ->
        receive_the_request_from_the_client(),
        create_a_new_listener_for_next_client().

create_a_new_listener_for_next_client() ->
        case catch create_the_listener() of
                {'ERROR', Reason} ->
                        make_some_arrangement()
                OK ->
                        make_a_connection_to_the_client()
        end.

make_a_connection_to_the_client() ->
        make_the_connection(),
        deliver_the_page().

deliver_the_page() ->
        deliver_the_page_to_the_client().
```

To load a homepage is defined as a transaction in this system. We have also defined that this transaction starts when we get a request from the client. We also imagine two checkpoints for this transaction. Checkpoint1 is when the new listener is created and checkpoint2 when the connection with the client is set. Then we define the transaction as finished when we delivered the page. After we inserted the first part of the code that should be implemented manually the program look like this instead. We write these changes in Erlang code.

```
request_from_a_client() ->
    receive_the_request_from_the_client(),
    Tid = make_ref(),                                (creates a Tid)
    monitor_server ! {Tid,now(),self(),start_trans},   (sends a Msg)
    create_a_new_listener_for_next_client(Tid).

create_a_new_listener_for_next_client(Tid) ->
    case catch create_the_listener() of
        {'ERROR', Reason} ->
            make_some_arrangement()
        OK ->
            Cid = checkpoint1,
            monitor_server ! {Tid, now(), Cid,checkpoint},
            make_a_connection_to_the_client(Tid)
    end.

make_a_connection_to_the_client(Tid) ->
    make_the_connection(),
    Cid = checkpoint2,
    monitor_server ! {Tid, now(), Cid,checkpoint},
    deliver_the_page(Tid).

deliver_the_page(Tid) ->
    deliver_the_page_to_the_client(),
    monitor_server ! {Tid,now(),self(),stop_trans}.
```

If we now assume that all of the following functions are working in different processes:

```
    request_from_a_client(),
    create_a_new_listener_for_next_client(Tid),
    make_a_connection_to_the_client(Tid),
    deliver_the_page(Tid)
```

This is not very reasonable, but let us assume this, just for the example. Then we also should implement the code that sends messages when a process starts or stops working with a transaction. To settle potential misunderstandings we should also mention that, when you send the message that a transaction started you do not have to send a message for a new Tid in that Pid. Our program handles that automatically. The code will then look like this.

```
request_from_a_client() ->
        receive_the_request_from_the_client(),
        Tid = make_ref(), (creates a Tid)
        monitor_server ! {Tid,now(),self(),start_trans},
        **monitor_server ! {Pid,Tid,del_tid,now()},**
        create_a_new_listener_for_next_client(Tid).



create_a_new_listener_for_next_client(Tid) ->
        **monitor_server ! {Pid,Tid,new_tid,now()},**
        case catch create_the_listener() of
                {'ERROR', Reason} ->
                        **monitor_server ! {Pid,Tid,del_tid,now()},**
                        make_some_arrangement()
                OK ->
                        Cid = checkpoint1,
                        monitor_server ! {Tid, now(), Cid,checkpoint},
                        **monitor_server ! {Pid,Tid,del_tid,now()},**
                        make_a_connection_to_the_client(Tid)
        end.

make_a_connection_to_the_client(Tid) ->
        **monitor_server ! {Pid,Tid,new_tid,now()},**
        make_the_connection(),
        Cid = checkpoint2,
        monitor_server ! {Tid, now(), Cid,checkpoint},
        **monitor_server ! {Pid,Tid,del_tid,now()},**
        deliver_the_page(Tid).

deliver_the_page(Tid) ->
        **monitor_server ! {Pid,Tid,new_tid,now()},**
        deliver_the_page_to_the_client(),
        monitor_server ! {Tid,now(),self(),stop_trans},
        **monitor_server ! {Pid,Tid,del_tid,now()}.**
```

Now all the manual implementations are made, and it is time to show what those that are automatically implemented look like. Let us say that the start function in the infile was request_from_a_client() then the code will look this.

```
request_from_a_client() ->
        **mon:event_start_trace(self()),**
        receive_the_request_from_the_client(),
        Tid = make_ref(), (creates a Tid)
        monitor_server ! {Tid,now(),self(),start_trans},
        monitor_server ! {Pid,Tid,del_tid,now()},
        create_a_new_listener_for_next_client(Tid).
create_a_new_listener_for_next_client(Tid) ->
    monitor_server ! {Pid,Tid,new_tid,now()},
    case hks:catch create_the_listener() of
        {'ERROR', Reason} ->
            monitor_server ! {Pid,Tid,del_tid,now()},
            make_some_arrangement()
        OK ->
            Cid = checkpoint1,
            monitor_server ! {Tid, now(), Cid,checkpoint},
            monitor_server ! {Pid,Tid,del_tid,now()},
            make_a_connection_to_the_client(Tid)
    end.

make_a_connection_to_the_client(Tid) ->
    monitor_server ! {Pid,Tid,new_tid,now()},
    make_the_connection(),
    Cid = checkpoint2,
    monitor_server ! {Tid, now(), Cid,checkpoint},
    monitor_server ! {Pid,Tid,del_tid,now()},
    deliver_the_page(Tid).

deliver_the_page(Tid) ->
    monitor_server ! {Pid,Tid,new_tid,now()},
    deliver_the_page_to_the_client(),
    monitor_server ! {Tid,now(),self(),stop_trans},
    monitor_server ! {Pid,Tid,del_tid,now()}.
```

Now all the code that is needed are implemented.

7.3.3 Design guidelines

We present these guidelines to get it as simple and clear as possible.

DESIGN GUIDELINES
Define transactions that you can write in the software contract
Divide these transactions, if necessary, into new transactions you can trace through the system
Define the following locations, for every transaction, in the target program. *Where do the transaction start? *Where are the checkpoints we are interested in? *Where do the transaction stop?
Define, for every process, where it starts working with a new transaction and where it stop working with a transaction.
Instrument code that sends messages in all the locations you defined above.

CHAPTER 8

The monitor program

This chapter gives an overview on the monitor program and mention all special implementations.

8.1 Main components

The monitor program consists of four different modules, see appendix A, where every module is one process. The modules are *mon.erl, mon_collector, mon_log* and *mon_msg*. *mon_msg* is only used to control the message sending between the monitor program and the analysis program, more about this in the next chapter. Except from modules with executable code, the monitor program uses two configurations-files, *mon.hrl* and *mon_config.hrl*.

8.1.1 Mon

The Mon module is the start up module. It starts reading the infile and performs some action with this data. Then this process checks if there are any processes registered as monitor_server or monitor_log, if not it spawns the monitor_server process and the monitor_log process and register them with these names. Then this process starts to instrument the executable source code in the target program.

8.1.2 Mon_collector

Mon_collector is the main part of the monitor program. The process in which it is executed is registered as monitor_server. This process is receiving all the messages from the target program and take some action depending on the message. The process also handles the Tid-list, which contains all the information about the Tids in the target program, and the Pid-list which contains information about which transactions every Pid is working with. In this process we also measure the execution time for the target program, more about this in the next chapter, in this process. When this process has received all the information about a transaction, from start to stop, or failure, it sends a message to the monitor_log process. The monitor_log process then saves this information on disk. Then this process deletes these transactions in the Tid-list and in the Pid-list. When a failure occurs the process looks in the Pid-list and connect the failure to the right transactions. Then it deletes these transactions in the Pid- and Tid-list and sends a new message to the monitor_log. The monitor_log then saves this information on disk.

One special problem we had to think about in this program was, whether we can be sure of that the messages connected to the same transaction arrive to the monitor_server in the correct order. Because of this we have to look through the mail box after other messages belonging to a Tid when we receive a message for this Tid. This process is also, of course, the process that traces the target program, otherwise the messages would not be sent to this process.

8.1.3 Mon_log

Mon_log is executed in the process monitor_log. It handles all the communication with the log-files, it is registered as monitor_log. When something is to be saved to the files, this process receives a message from the monitor_server containing the information to be saved. Every transaction type has its own log-file. In these files we save both failed and succeeded transactions.

This process also opens and closes the log-files. When we open a file we get a handle to this file. After we open, we have to use this handle every time we like to communicate with the file. We save these handles in a list consisting of tuples. Every tuple consists of the handle and the name of the transactions type in this file. When we like to save something in a transaction type’s file, we look in the list after the name of this type and then get the handle to the file. The names of the transaction types that are used as a key to get the handle are defined in the mon_config.hrl file. These names and the names of the transaction types, sent from the target program, have to be the same.

Except from the files for the transactions, the monitor_log process also handles the files for execution time, catch-error and msg-to-non-existing-process.

CHAPTER 9

The Analysis Program

This chapter describes the analysis program. The program is not described in detail, but all special implementations are discussed.

9.1 Main components

The analysis program and the monitor program are, as described in previous chapter, two separate components. They are intended to run on separate nodes and communicate via an interface defined by the *mon_msg.erl* module. All other code needed for the analysis is assembled in the *analyse.erl* module, and uses the same configuration files as the monitor program, i.e. *mon_config.hrl* and *mon.hrl*. The analysis program is divided into three successive parts, transaction and failure analysis, reliability analysis and, future failure analysis. All Erlang code is listed in appendix A.

9.2 Transaction and failure analysis

9.2.1 Concurrent file handling

The first thing that happens in the transaction analysis is a control to see if the monitor server is running. If so, the analysis program asks for permission to start analysing. The reason for this, is because of problems with concurrent use of the log files. The monitor server must first close them before another program can open them. After the analysis program has opened the files, the monitor program can reopen them without problem, since that program created the files. When the analysis program has opened the files, it will not be able to read anything the monitor program has written in the files after this point. To do so a new analysis must be performed at a later time.

9.2.2 Execution time

As mentioned in chapter 3, it is very important to only count the time when the program is executed in the processor, since it is only then a failure can occur. We also only want to count the time when, at least, one transaction is being processed by the program. If the program is executing in spite the fact that the program does not process any transaction, a transaction type may have been forgotten in the design phase, or maybe the program performs some kind of action that does not classify as a transaction, e.g. upgrading source code. We do not separate different types of transactions when it comes to execution time. If you are interested in only one type of transaction, you have to perform tests which only use that type.

Since Erlang programs often are intended to run in a time sharing environment it is not obvious how to calculate the execution time. The foundation in our calculations is the use of the Erlang command *erlang:statistics(runtime)*. This command returns the time, in milli seconds, the program has been executed since this command last was called. If you at the same time measures the calendar time elapsed, between two calls, you can calculate the percentage part the program has been executed. This can then be used to transform calendar time to execution time. The reason for not simply using the answer from the Erlang call is because this answer is for the whole Erlang system. If there would be several transactions processed at the same time, in different processes, they would interfere with each other.

The problem is, due to the risk of being out scheduled, to exactly measure the calendar time at the same time as executing the runtime call. We have solved this problem by measuring the calendar time precisely before and after we execute the runtime command. Then we assume that the call was executed in the middle of these times. This does not solve the problem completely, but it gives a reasonable good estimation. In the future this will not be any problem, because then the monitor code will be autobuilt in Geoff Wong's system which supports better time measurements.

We execute the measurements mentioned above every time the system is empty and a new transaction comes in, when the last transaction leaves the system and when a failure occurs. If the system continuously process transactions and no failure occurs we will, with constant intervals, perform the measurements to avoid problems with load fluctuations in the time sharing system. If the time interval between two measurements is too long, the percentage part that the program executes, can vary heavily.

9.2.3 Data extraction

The analysis program reads all the log files created by the monitor program and extracts the desirable data. Some of the data are only stored in new log files, see appendix D, while others are also used as input to the reliability analysis, e.g. time to failure (in execution time) and total execution time. There are also plot files generated to make it possible to plot various data with the UNIX plot command gnuplot, see appendix C.

9.2.4 Pseudo failure

When the test run have been completed and the last test case does not fail one must make some accommodation for all the successful test executions since the last failure. According to [24] a good way to do this is to assume that the program will be failure free for at least as long as it has been since the last failure. This means that if the program has been executed 100 seconds since the last failure, we assume that the next failure will occur in additionally 100 seconds. The pseudo failure is automatically added to the real failures. Due to this, the total execution time for the various types of transactions can differ since we also have to add execution time as well, i.e. 100 seconds in the example above.

9.3 Reliability analysis

9.3.1 Metrics

The reliability analysis part uses the data produced by the transaction and failure analysis part, to produce various metrics which are of interests when studying the reliability, e.g. number of succeeded transactions, mean time to failure (MTTF), number of failures and, the probability of failure free execution. This is done for every transaction and also for all transactions together. Further it calculates the distribution between the different types of transactions. When we calculate the probability for failure free execution, here called reliability, we assume that the failure intensity, or MTTF, is constant. Due to this, we can calculate the reliability according to equation 3.1. If the failure intensity is not constant, you should not rely on this measure and instead use one of the reliability growth models, mentioned in section 3.3, to calculate the reliability. As input to these models you use the failure times calculated in the transaction and failure analysis part.

9.3.2 Multiple failures

During reliability growth testing it is assumed that all faults, that causes failures, are removed immediately when discovered, otherwise they will cause multiple failures. If this is not the case and multiple failures have occurred, the reliability growth models will not give an accurate result unless you remove the multiple, all except the first, failure times from the failure time file. This program cannot recognize multiple failures. However, during the demonstration test phase all failures should be counted since this testing reflects the operational phase where faults are not corrected.

9.4 Future failure analysis

The last part of this program is the future failure analysis part. In this part we analyse failures that do not cause a transaction to fail today, but indicates that something is wrong which may result in a failed transaction in the future, i.e. catched failures and messages to non-existing processes. All the information about future failures are stored in the two log files, Catched_error.log and Msg_non_ex_proc.log.

9.5 Outputs

The program produces both output to the screen and to disk. Some of the disk files are constructed in a way, that makes it possible to plot the data with the UNIX based plot program, gnuplot. In appendix D examples are given to show different types of outputs generated by the analysis program. The information shown on the screen are:

- Number of succeeded transactions, for every transaction type and total.
- Mean time of succeeded transactions, for every transaction type.
- Distribution between different types of transactions.

- Number of failures, for every transaction type and total.
- Mean Time To Failure (MTTF), for every transaction type and total.
- Reliability for a specified time with current MTTF, for every transaction type and total.

The different plots available are:

- Transaction times for succeeded transactions, for every transaction type.
- Time to failure, for every transaction type and total.
- Reliability versus execution time, for every transaction type and total.

All the plots are also saved as postscript files to support easy printing.

The data stored on disk are:

- Time to failure, for every transaction type and total.
- Failure times, for every transaction type and total.
- Transaction times, for every transaction type.
- Catched errors, for all transactions
- Message to non-existing process, for all transactions.

CHAPTER 10

Practical Application

This chapter first describes our test system. Then we explain the tests we conducted and also the results from these tests.

10.1 The Test Environment

To be able to test our solution, we needed an application. Our basic requirements on the application are that it should be similar to a telecommunication system and of course, written in Erlang. After discussions with the experienced researches here at SERC we decided to use a newly developed web server from CSLab/OTP. The similarity between a web server and a telecommunication system is that it should be up and running all the time and it should be possible to upgrade the system without shutting it down. Since the web server is fairly new it is also very likely that it still contains faults (which we found to be the case). The good thing with this is that we can not be sure that the code only contains our own simulated faults, which makes the tests more realistic.

10.1.1 System components

Figure 10.1 shows the different components of our test system and as you can see, it consists of both hardware and software components. Example 10.1 describes a typical scenario.

Example 10.1
The first thing that happens is that the user, called Tobbe, at computer 1 wants to look at a web page, e.g. The Erlang Association. He tells his browser, e.g. Netscape or Microsoft Explorer, to collect the page by writing (or click on) the address and name (URL) of the page. The browser then sends a request to the web server located in computer 2. The web server picks up the request and loads the right page from disk and then sends it back to the browser, which in turn shows it for Tobbe.

In this system, Tobbe is the end user, but since we are only interested in the web server, we only look at that part. If a failure, that affects the web server, would occur in any other part of the system we just ignore this when it comes to our reliability calculations.

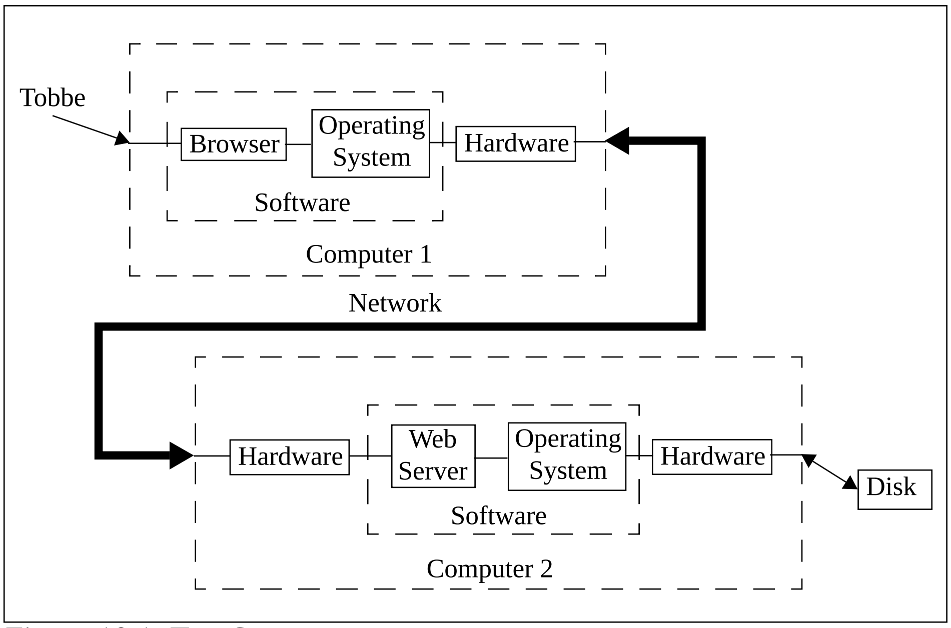


Figure 10.1: Test System

All testing is done with UNIX as operating system. This is a time sharing environment which, from a user point of view, seems to execute several different processes, programs, at the same time. In reality there is only one process at a time that is being executed in the processor, the others have to wait for their turn. All UNIX processes have to share the time in the processor. Note that we are talking about UNIX processes and not Erlang processes. The whole web server, which consists of several Erlang processes, is one UNIX process. Due to this it is very important that we only count the time when the web server is executed in the processor since it is only then a failure can occur. The computers are connected in a local area network.

10.1.2 Transaction types

The purpose of this thesis is to measure the reliability for different types of transactions. Therefore we must identify the different transactions the web server should handle. Normally, this is done in the requirements phase. Since the intention with this testing is to validate our solution and not to fully test the web server, we only use some of all the different types of transactions possible. The ones we use are:

- **Vanilla** -This is the transaction performed when it is an ordinary page.
- **Mod_CGI** -This is the action performed when a page contains a execution command to a program. Instead of returning a document the web server executes the program and returns the output. These types of programs are called Common Gateway Interface (CGI).
- **Mod_ESI** -This is the same case as for Mod_CGI but instead of executing CGI programs an Erlang program is executed. These types of programs are called Erlang Scripting Interface (ESI).

There are several more types of transactions that the web server can perform, but there is no special reason that we chose these particular ones.

10.1.3 Test cases

Before we start to test the web server we must create a set of web pages that uses one or more of the different transactions. This part has been done by manually writing source code (see Appendix C) for different web pages. As mentioned before this testing is only intended to validate our solution and not to fully test the web server, and therefore we have only created a couple of pages for each transaction type.

10.1.4 Test case generator

In chapter 3 we discussed the aspects involved when choosing test cases and this is valid for this situation too. To our luck, Geoff Wong, has already developed a test generator for a web server, which takes all these aspects into consideration. The generator randomly chooses which page to request and also how often.

To use the generator you first create different scenarios. A scenario is a sequence (one or more) of web pages that the generator should request from the web server. If you create more than one scenario the generator will randomly choose from these.

There are five different parameters you can use to tailor the generator to suit your case. The parameters are listed and described in table 11.

Parameter	Description
Number of Sessions	A session is the same as a user. By using more than one session you can simulate the realistic case when several different users try to collect pages from the web server.
Random number Seed	To generate random number the generator needs a start seed. By using the same value you can recreate the exact same test sequence several times.
Sequence probability	This is the probability that the sequence given in a scenario is followed. If you use zero the sequence picked will be totally random.
Sleep time	This is the average time a session waits until it makes a new request.
Run time	Test time in seconds.

Table 11: Test case generator parameters

10.1.5 Failure simulation

Even if there is a high probability that the web server contains faults we cannot be sure that we will find them. Especially not since we do not conduct a thorough and exhaustive test. Because of this we must simulate failures and then validate if we have found these. To make the test more realistic we want these simulated failures to occur randomly.

The way we have solved this is by manually inserting a new piece of code (see Appendix A) in the web server. This code will with a certain probability terminate the process, in which it is executed, with the reason simulation. This will then be traced by our monitor program. There is a good chance that our simulated failure will cause other failures in the system but it is only the first one, and time of occurrence, that we want to log.

10.2 Test Results

10.2.1 Verification of our solution

After conducting several different test runs we can, with very high confidence, claim that our solution works. We have with manual calculations verified all our results. Since the web server contained faults, as we suspected, we could also verify that our solution could handle failure propagation, only record the first failure, catched errors and messages to non-existing processes. We have also been able to create the scenario when the web server catches a simulated error but generated a new failure in a latter state. Our program successfully recorded the new failure. We have used different load, requests per second, and different combinations of transaction types, but all the tests shows the same thing, our solution works to the full. In appendix D we show some of the test results. The figures in themselves are not important since we did not intend to test the web server, only verify our solution.

10.2.2 Web server failures

As we suspected the web server contained faults. Some of the recorded failures have been sent to CSLab/OTP with the hope that they will be corrected in the next version. The failures that occurred, and reason why, are as follows:

Too high load -When the load is too high, the web server starts catching errors and sending messages to non-existing processes. Eventually the whole web server breaks down. The reason for this is that the socket handler, the part which connects the program to the physical port in the computer, in the web server cannot manage to serve all requests when they come to fast. This problem did not occur when the web server was used in a Free BSD UNIX environment instead of a SunOS UNIX environment.

Start up failures -When the web server is started, before you start sending requests, it catches a huge amount of errors. The reason for this does not have to be because of a fault in the program. Sometimes a designer chooses to use the catch command on purpose and this may be the case here.

Break down -If a transaction fails, between that the web server get a request from the client and that the webserver started a new socket listener, the whole web server breaks down. The reason for this is unknown to us.

10.2.3 Comments to the graphs

In appendix D we present figures and graphs from the tests on the webserver. There have been no time to make a complete analysis of the results, but we comment the graphs and also provide a possible reason for some of the behaviours.

The first three graphs are from the same test. To avoid the problems with high load, mentioned above, we used a relatively low load.

The first graph shows the transaction times for vanilla transactions. There is one really high spike around transaction 38. We have no idea what happened here. You can also see a trend that the transaction times rise the longer we execute. The reason for this, is that a new transaction starts before the old one is finished. This lead to that the number of transactions executed at the same time increases. Thus more processes, transactions, have to share the time in the processor.

The next graph shows the transaction times for the MOD_ESI transaction. You can also in this graph see a trend of increasing transaction time the longer we execute. We think that the reason for this is the same as for the vanilla transactions.

The third graph shows the transactions times for the MOD_CGI transaction. In this graphs something very strange happens. While executing the first 80 transactions there are no problem. Then all of a sudden, the transaction times rise real high for a few transactions. After this the webserver terminates. Precisely before the web server terminates we receive a higher amount of messages to non existing processes and catched errors. We do not know what happened in this test case. We tried to reproduce it, but we did not succeed. Since we used a low load in this test, we do not think that this termination depends on the same problem as the load termination.

The next three graphs are from another test. This time we have conducted the test over a longer period of time, i.e. executed more transactions.

The fourth graph is another one of transaction times, but now for a MOD_ESI transaction. There is not much to say about this graph. There are some spikes, but overall it looks okay.

The fifth graphs shows times to failure, or time between failure, for the transaction type MOD_ESI. The first failure occurs after about 330 seconds, then the next failure occurs after additionally 420 seconds, and so on. Those values that are real low, about 10 seconds, are really bad. This means that an failure occurred only 10 seconds after another failure. Note, that we simulate failures during these tests, which means that this does not say anything about the web server's quality. This only shows that our method works.

The last graph shows the reliability versus execution time for the web server when it is executing Mod_ESI transactions. The reliability in this graph are defined as *"The probability for failure free operation during a specified time in a specified environment"*. The equation we used is:

$$Reliability \qquad R(\tau) = e^{(-\lambda\tau)} \qquad (3.1)$$

The reason that the reliability is so poor in this graph, is that we only tested a short time and simulated several failures.

CHAPTER 11

Retrospect and Future work

11.1 Retrospect

To understand the purpose with this thesis one has to look at some of the other projects SERC has undertaken. SERC's goal is to develop a method where you can specify the non-functional requirements in the contract and then track, and be in full control of, them through the development process and even during the operational phase. The first part of this, the contract, has been solved by Anna-Karin Carlsson and Fredrik Gustavsson [2]. They developed a method to break down a customer's business into a set of transaction types and also define which quality needed for every type of transaction.

The main tasks of our thesis was to "Define metrics which connects the customer's and the developer's view of the system's reliability" and "Find a way to monitor these metrics in an Erlang system during execution", which is the next step in this chain. Our thesis in turn is a part of a monitor project conducted by Geoff Wong. His aim is to automatically create all monitor code in a special Erlang compiler. This compiler also divides the code in two different parts, so that the functional code is executed on one processor and the monitor code on another processor.

11.1.1 Reliability

When we started this part we thought that the problem was to define reliability and which metrics you need to calculate the reliability. We soon realized that this is not the case. The problem is how to collect the metrics.

First of all you need to divide your system into independent components. You do this so that you can calculate the reliability for every part and then combine these to one system reliability. We have also chosen to look at the system from the user's point of view by dividing it, in our case only the erlang part, into transactions according to Carlsson's and Gustavsson's method. You can then calculate the reliability for every type of transaction and directly relate this back to the contract.

Another problem is how to test the system. Test cases should be chosen randomly according to the probability that they would be chosen during operation. By choosing test cases randomly you introduce statistical independence for failure occurrences. This is very important, otherwise the different statistical reliability models we described will not be applicable. You also need to look for multiple failures so that you only count these ones. Next problem during test is to choose the right model to use. This depends on, among other things, the times between failures. Is the failure intensity for example decreasing or increasing?

The problems mentioned above are only a few of all those that exists. In chapter 3 we have described more characteristics that have to be considered, but we have not been able cover them all. Reliability is a quite new engineering discipline and much work is done to develop this further.

11.1.2 Monitoring

Our first approach how to handle the monitoring was to do it totally automatic since, this is the way it is supposed to be in Geoff Wong’s system. Unfortunately we could not find a way to connect a failure to a specific transaction this way. We needed some way to track a transaction’s path through the whole Erlang system. We did this by manually instrumenting code in our test application, the web server. Since we did this after the application was finished it took quite a time to find the right places. If you have this in mind already when you make your design, this will only take a fraction of time compared with making an ordinary design.

11.1.3 Test application

We chose to use a web server written in Erlang as test application. This turned out to be a good choice since the version we got was an early one. The good thing with this was that it contained bugs. For some of these bugs we have also found out where the problem is, which has been sent to the current developer, OTP.

11.2 Future work

The most important thing, when it comes to future work, is to integrate our method with Geoff Wong’s system, since this is the final aim with this project. Our monitor program is not intended to be used in the future, only the ideas. A first step to this could be to make small changes in our program so that you will have one main monitor process and then one monitor process for every application process. This is the way it is intended to work in Wong’s system.

References

[1] ANSI/IEEE, “Standard Glossary of Software Engineering terminology”, STD-729-1991, ANSI/IEEE, 1991

[2] A.-K. Carlsson, F. Gustavsson, “Quality of Telecommunication Application Software, SERC, 1997

[3] G. Wong, “Continuous Systems Monitoring”, Department of Computer Science, RMIT University, Melbourne, Australia, 1996

[4] J. Armstrong, R. Virding, C. Wikström, M. Williams, “Concurrent Programming in Erlang”, Second Edition, Prentice Hall, 1996

[5] P. B. Lakey, A. M. Neufelder, “System and Software Reliability Assurance Notebook”, Softrel, 1997

[6] J. D. Musa, A. Iannino, K Okumoto, “Software Reliability: Measurement, Prediction, Application”, McGraw-Hill publishing Company, 1990

[7] Farr, Dr. William, “A Survey of Software Reliability Modelling and Estimation”, Naval Surface Weapons Center, Dahlgren, 1983

[8] Dr. S Keene, G. F. Cole, “Reliability Growth of Fielded Software”, Reliability Review, Vol 14, 1994

[9] American Institute of Aeronautics and Astronautics, “Recommended Practice for Software Reliability”, ANSI/AIAA R-01301992, 1993

[10] J. D. Musa, “Operational Profiles in Software Reliability Engineering”, IEEE Software Magazine, 1993

[11] M. R. Lyu, “Handbook of Software Reliability Engineering”, McGraw-Hill, 1996

[12] L. J. Clinton, “A Framework for Monitoring Program Execution”, Department of Computer Science, The University of Arizona, Arizona, 1988

[13] M. A. Linton, “The Evolution of Dbx”, Proceedings of the Summer 1990 USENIX Conference, pages 211-220, June 1990

[14] M.H. Brown, R. Sedgewick, “A System for Algorithm Animation”, Computer Graphics, 18(3), pages 177-186, July 1984

[15] H. D. Bocker, G Fischer, H/ Nieper, “The Enhancement of Understanding through Visual Representations”, CHI ‘86 Proceedings, pages 44-50, June 1986

[16] R. R. Henry, K. Whaley, B. Forstall, “The University of Washington Illustrating Compiler”, Proc. ACM SIGPLAN ‘90, pages 223-233, White Plains, NY, June 1990

[17] R London, R Duisberg, “Animating Programs Using Smalltalk”, IEEE Computer, pages 61-71, Aug. 1985

[18] R Sosic, “Dynascope: A Tool for Program Directing”, Proceedings of the ACM/SIGPLAN ‘92 Conference on Programming Language Design and Implementation, volume 27, pages 12-21, San Francisco, California, June 1992

[19] Z. Aral, I Gertner, “High-level Debugging in Parasight”, Proceedings of the ACM/ SIGPLAN PPEALS 1988, pages 21-30, September 1988

[20] P. Henderson, “Functional Programming Application and Implementation”, Prentice Hall, ISBN:0133315797, 1980,

[21] A. Fedoriw (ed.), H. Nilsson (ed.), “Erlang System/OTP 4.5 Development Environment Reference manual”, 1997

[22] H. Mattson, H. Nilsson, C. Wikström, “MNESIA Reference Manual”, Rev. 1.2.1 1997

[23] SERC-group, “The Erlang Engine Manual”, version 1 January 1997

[24] M. Dyer, “The Cleanroom Approach to Quality Software Development”, John Wiley & Sons, Inc, 1992

Appendix A: Erlang source code

In this appendix we lists all the source code written in Erlang. The different modules are listed after which program they belongs to.

- **Configuration modules (used by both programs)**
  mon_config.hrl
  mon.hrl

- **Monitor program modules**
  mon.erl
  mon_log.erl
  mon_collector.erl
  instrument.erl
  tmon_cover.erl

- **Analysis program module**
  analyse.erl

- **Failure simulation module (inserted in the web server)**
  mon_failure.erl

- **Communication module (between the two programs)**
  mon_msg.erl

- **Instrumented code**

Appendix B: Gnuplot source code

In this appendix we lists all the source code written to create the different plots with gnuplot. The different source files are listed after which plots they create:

- **Transaction times and times to failure (for one type of transaction)**
  mon_Vanilla.plot
  mon_Mod_CGI.plot
  mon_Mod_ESI.plot

- **Reliability versus execution time (for one type of transaction)**
  mon_rel_Vanilla.plot
  mon_rel_Mod_CGI.plot
  mon_rel_Mod_ESI.plot

- **Time to failure and reliability versus execution time (for all transactions)**
  mon_rel_all.plot

Appendix C: Test source code

In this appendix we lists the source code of the web pages we used during validation of our method. The files are listed according to which transaction type they belong to. The files are:

- **Vanilla pages**
  hemsida.jpg
  index.html
  page1.html
  star.gif

- **Mod_CGI script**
  cgi-bin/test
  test.tif
  tmp.dvi

- **Mod_ESI script**
  kent_hans
  yahoo

Appendix D: Test results

In this appendix we show some of the results from the testing. The different outputs are listed depending on type:

- **Screen data**
  Screen sample

- **Plots**
  Transaction times for Vanilla transactions
  Transaction times for Mod_ESI transactions
  Transaction times for Mod_CGI transactions
  Transaction times for Mod_ESI transactions
  Time to failure for Mod_ESI transactions
  Reliability versus execution time for Mod_ESI transactions

- **Files**
  Transaction times for all transaction types
  Transaction errors for all transaction types
  Failure times for all transaction types
  Time to failure for all transaction types
  Messages to non-existing processes
  Catched errors