Rijksuniversiteit Groningen **Faculteit der Wiskunde en Natuurwetenschappen**

**Vakgroep Informatica**



# On the quality of embedded systems

Peter Smeenk

begeleider: Prof.dr.ir. L. Spaanenburg

augustus 1996



RuG

# Abstract

Risk Management is the key to quality production. For every new technology cycle new risks must be identified. Embedded Systems are mixtures from hardware and software parts and can therefore be expected to blend the risks from these the contributing technologies. Here we probe the ground. Notions and definitions for software / hardware reliability models are reviewed and after a small–scale experiment it is concluded where advances are needed to establish an effective way of creating quality Embedded Systems.



# Samenvatting

Beheersbare kwaliteit is belangrijk voor een productie proces. Voor een nieuwe technologie moeten opnieuw kwaliteitsnormen ontwikkeld worden. Embedded Systems is een nieuwe technologie waarvan kwaliteitsbegrippen vanuit de hardware– en software technologie over te nemen en aan te vullen zijn. We geven hiervan een eerste proeve. Na een overzicht van termen en definities worden de relaties naar hardware en software beschreven onder referentie naar betrouwbaarheidsmodellen. Een klein experiment voert tenslotte naar de presentatie van onderwerpen waarin vooruitgang noodzakelijk is om effectief Embedded Systems van bewezen kwaliteit te kunnen ontwikkelen.

# Preface

The life–time of a technology can be divided into several phases. This division is independent of the nature of the technology on hand and displays a moving attention within an ordered choice of scientific interests. At the start of a technology is the **innovation**: the discovery of a new technological fundament like the MOS–transistor or the object–oriented programming style. Ensuing the technological fundament is brought into usage, such that its basic features become visible. Often this **maturization** phase is characteristic for the pace of development: it swiftly moves on, it stalls or it even dies out. Predominant is the potential acceptance in relation to economic or technical factors. For the MOS–transistor the economical acceptance was created by the advent of the planar fabrication process; for object–oriented programming the growth of the individual, on–site computing power and storage capacity created a break–through.

At the end of the second phase the technology has matured to a sound professional expertise. This in turn blocks a further acceptance, unless the science is moved from art to craft: the **dissemination** over a large, non–specialist community. The central theme is quality of production as shows from the coming into existence of a large number of interesting support tools. The design of microelectronic circuits is strengthened from a large number of computer–aids. Once the technology has moved to a widely practiced production method, newer technologies can emanate: the **off–spring**. In the realm of computing science, the outgrowth towards standard software packages and standardized computers allows for System Sciences.

At this moment in time, software programming has by large become a **production process**. Herein, reliability, quality and risk are the key words that emphasize the need to produce a product that is functional within strict margins. Unfortunately, an uniform scientific field to support this research seems lacking: the key words reflect different views from a differing theoretical upbringing. Quality aims for an optimal functionality of which a reliable production or a minimized risk from malfunctioning are just aspects.

Despite all this, the difficulty to diagnose malfunctioning, locate the faults and introduce repairs is growing with the complexity of the task and with diminished access. Especially this last aspect urges for solutions in the area of **Embedded Systems**: relatively complex products with software buried deep within hardware parts. Consequently we have to cover both hardware and software while discussing reliability.

For reason of the above mentioned pluriformity we endeavor here firstly to bring related aspects together. In other words, concepts and definitions are brought into perspective. Then we set up a small experiment to support our argument: no easy overall solution seems to be in stock, but the future may spur a collection of techniques for tackling the various **reliability aspects** of Embedded Systems.

P. Smeenk
Groningen, 30.8.96

# Contents

# 1. Introduction

The following definitions have been taken from [24] for a correct understanding of this report:

- An **error** is a discrepancy between a computed, observed or measured value and the true, specified or theoretically correct value. Errors are concept–oriented [35].
- A **fault** is a specific manifestion of an error. Faults are developer–oriented [35].
- A **failure** may be the cause of several faults. A failure is the term that refers to what happens when one or more faults get triggered to cause the program to operate in another way as intended. A failure is either an inconsistency between specification and implementation or an inconsistency between implementation and user's expectation. Some also specify failure as a fault–effect. Failures are customer–oriented [35].

## 1.1. Quality

Systems are the overall indication for the assembly of collaborative parts from a variety of technological origins to a single complex part. Systems are omni–present: as there are ecological, financial and biological systems. We like to focus here on electronic systems and especially on methods to derive and evaluate their quality.

Electronic systems are built from software and hardware. Though eventually the hardware is used to perform the desired functionality, software will mostly be the enabling factor. It directly describes the desired functionality and is therefore a suitable view on the system quality. But with the constantly moving boundaries between software and hardware, it is increasingly important to take an hardware side into account. Therefore we will attempt to unify the view on software quality with that of hardware.

Software quality has many aspects. According to [28] quality of software involves capability, usability, performance, reliability, installability, maintainability, documentation and stability. Manufacturers have to deal with 4 important aspects when producing software: quality, cost, time to market and maintenance. And they have to find a balance between these aspects with the knowledge that maintenance costs are more than 50% of the total product cost.

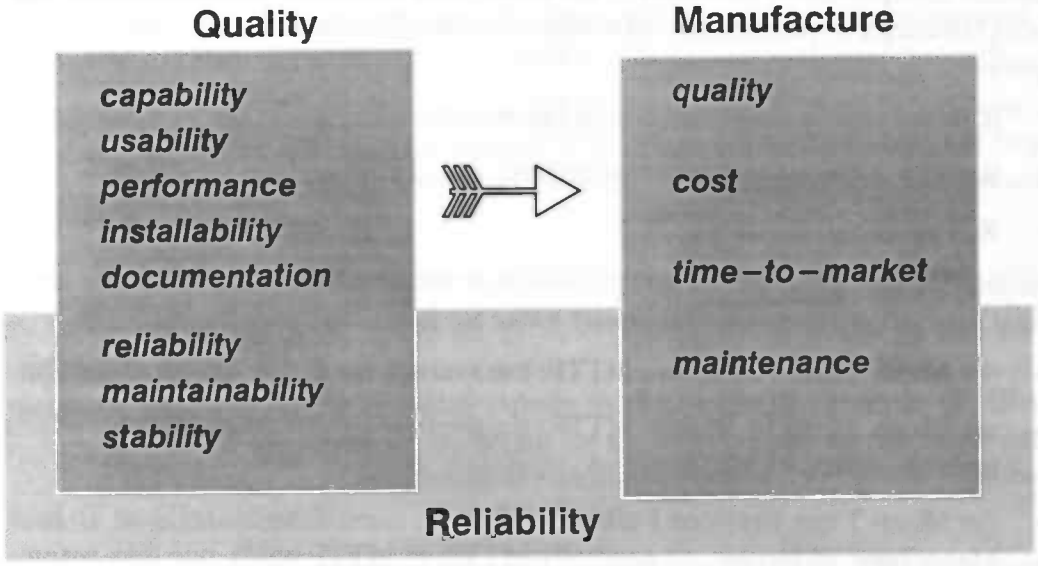



*Fig. 1: Different aspects of quality.*

Fundamentally we can discern two lines of thought in these lists. The first one has to do with the product specification. The quality of the product relates then to the way it will be perceived by

the potential customer. If it is viewed as a better product than others, it has apparently a higher quality. The second line of thought has to do with the development process: how does it conform to the specification in terms of development progress, maturization and final stability. Some confusion is caused by the fact that this can all be called reliability, despite the fact that it also contains elements of maintainability and robustness. Eventually this will all add to the cost of design and manufacture and we will therefore use the different meanings of reliability irrespectively.

> In RADC–TR–85–37 ("Impact of hardware/software on system reliability", January 85) is written: *The reliability of hardware components in Air Force computer systems has improved to a point where software reliability is becoming the major factor in determining the overall system reliability.*
>
> In 'Fatal Defect' from Ivar Peterson and Peter Neumann is written that in 1981, the launch of the Space Shuttle Columbia was postponed, because all 5 board– computers didn't act as was expected. The programs on the board–computer were so designed that if one programs gave a failure, one of the others would take over the job. Well, they didn't.
>
> In the video of IEEE about developing reliable software in the shortest time cycle Keene states that the software of the space shuttle at delivery had a reliability of 1 error for each 10000 lines of code, which is about 30 times better as normal code at delivery.
>
> With the first flight of the F16 airplane from the northern to the southern hemisphere, the plane flipped. It started flying upside down. The navigational system couldn't handle the change of coordinates from the northern hemisphere to the southern.

## 1.2. Reliability

Software cq. hardware **reliability** is the probability that faults in software cq. hardware will not cause the failure of a system. It is a function of the inputs to the system, the connectivity and type of the components in the system and the existence of faults in the software cq. hardware. The system inputs determine whether existing faults in the program will manifest themselves as failures. By this definition, reliability can be measured as the number of faults per thousand lines of code (kLoC) and indicates the maturity for a system under development.

In another interpretation, reliability is the probability that the software cq. hardware will work without failure for a specified period of time. A **reliability function** R(t) has to meet the following properties [29]]:

- $R(0) = 1$, the system is certain to begin without any cause for complaint.
- $R(\infty) = 0$, the system is certain to have failed ultimately at time $t=\infty$
- $R(t) >= R(t+i)$, $i > 0$.

By measuring the rate of arrival for the failures, a value for the system reliability can be derived. Typical ways to render such a reliability value for a system under test are:

- the **Mean Time To Failure** MTTF: the average time till a failure will occur.
- the **Mean Time To Repair** MTTR: the average time it takes to repair a failure, once it has manifested itself.
- the **Mean Time Between Failure** MTBF: the sum of the Mean Time To Failure and the Mean Time To Repair. As generally MTTR $\ll$ MTTF holds, one may assume MTBF $\approx$ MTTF.

Once a system is developed and tested, it is still not fault–free. Some failures may show up after years of application. During (pilot) test one may quote this reliability by:

- **System availability** SA: the percentage of the time, that the system is available, thus: SA = MTTF/(MTTF + MTTR). For a high SA you need: MTTR $\ll$ MTTF.

Later on, when the system is in the hands of the customers, some of the failures may easily be shaped as **Customer Change Requests** (CCR): an indication of the discrepancy between the expectation and the observation of the customer with respect to the system functionality, which in our terminology is a clear failure.

## 1.3. Risk

Safety, hazard and reliability refer to the same kind of studies, in which the equipment failure or equipment operability is essential. If the study is extended to include also the consequences of the failure, then you'll have a risk analysis study [21]. Most of the risk studies are done for the purpose of satisfying the public or government, and not for the purpose of reducing risk. A risk study consists of 3 phases: (a) risk estimation, (b) risk evaluation, and (c) risk management.

**Risk estimation.** The objective of this phase is to define the system and to identify in broad terms the potential failure. Once the risk has been identified in its physical, psychological or social settings, a quantification w.r.t. planned operations and unplanned events is performed. Overall one can discern 3 steps:

1. Identify the possible hazards. If a ranking of the hazards is being used, then a preliminary hazards analysis (PHA) is being used. A common class ranking is: Negligible (I), Marginal (II), Critical (III), and Catastrophic (IV). The next thing to do in this phase is to decide on accident prevention measures, if any, to possibly eliminate class IV and if possible class III and II hazards.
2. Identify the parts of the system which give rise to the hazard. So the system will be divided into subsystems.
3. Bound the study.

**Risk evaluation.** The objective of this phase is to identify accident sequences, which results into the classified failures. A first evaluation will be in terms of public references (such as "revealed" or "expressed"); ensuing a formal analysis will be attempted to reveal necessary decision, potential cost benefits and eventual utility. There exist 4 common used analytical methods:

1. **event tree** analysis; Event tree analysis (ETA) is a top–down method and depends on inductive strategies. You start the method at the place where the hazard has begun. Then you add every potential new hazard that this hazard could transform in with it's possibility to this hazard.
2. **fault tree** analysis; Fault tree analysis (FTA) works the same way, but then you start with an failure, and try to detect how this could be produced. This method is bottom–up and thus deductive.
3. **failure modes and effects analysis**; Failure modes and effects analysis (FMEA) is an inductive method. It systematically details on a component–by–component basis all possible fault modes and identifies their resulting effects on the system. This method is more detailed as event tree analysis.
4. **criticality analysis**; Criticality analysis is an extended FMEA in which for every component a criticality number is assigned with the formula:

   $C_r = \beta * \alpha * K_e * K_a * \lambda_g * t * 10^6$, where

   $\lambda_g$ = generic failure frequency of the component in failures per hour or cycle.

$t$ = operation time.
$K_a$ = operational factor, which adjusts $\lambda_g$ from test to practice.
$K_e$ = environmental factor, which adjusts $\lambda_g$ for the not so clean environment in practice.
$\alpha$ = failure mode ratio of critical failure mode.
$\beta$ = conditional probability that the failure will occur after the faults are triggered.
$10^6$ = factor transform from losses per trial to losses per million trials.
For every hazard a summation for every $C_r$ can be made. This assumes independence, which is not guaranteed.

| In short | Plus | Minus |
|---|---|---|
| PHA: | always needed | |
| FMEA: | easy<br>standard | non–dangerous failures<br>time consuming<br>human influence neglected |
| CA: | easy<br>standard | human influence neglected<br>system interaction not accounted for |
| FTA: | standard<br>failure relationships<br>fault oriented | explode till large trees<br>complex |
| ETA: | effect sequences<br>alternatives | parallel sequences<br>not detailed |

**Risk management.** The objective of this phase is the conclusion phase of the study. It is normally a comparison between cost and chance that something will/will not happen. The basis may be provided by juristic, political, historical or eco–sociological considerations. The result is a Risk Management strategy, which in turn can be coupled back to the previous phase. Some specific analysis tools for the development of risk management are:

1. **Hazards and operability studies** (HAOS); This method is an extended FMEA technique, in which operability factors are included in the study.
2. **Cause–consequence analysis** (CCA); This method starts with a choice of a critical event. Then the following questions has to be answered about this event:
   – What conditions are needed to have this event lead till further events?
   – What other components does the event affect.
   This method uses both inductive and deductive strategies.

| In short | Plus | Minus |
|---|---|---|
| HAOS: | large chemical plants | not standardized<br>not described in literature |
| CCA: | flexible<br>sequential | explode easily<br>complex |

# 2. On the Quality of Systems

Systems can be created in any technology: such as pneumatic, mechanic, hydrologic and (of course) electronic. In the realm of electronic products, a coarse division has been in machines (hardware) and programs (software on machines). Assuming a general–purpose machine, a large set of software programs can be set to work without paying attention to the underlying hardware platform. The simple assumption can be made that together with the Operating System an interface is created that allows to move large junks of software from one platform to the other without any changes (portability). On the other side of the spectrum are the machines themselves with (if needed) a small dedicated program, that does not need to be ported.

Because of the elaborate way in which hardware is designed, a large number of support tools have been developed, such as simulators, physical design engines, emulators and so on. This has boost up the efficiency of hardware design to a level wherein millions of transistors are handled for the same costs as single transistors a couple of decades ago. At the same time, the efficiency of creating software has hardly made any progress compared to hardware. So, the design bottleneck has moved from hardware to software. This is reflected in the popular saying for software: the unmovable object.

As the market requires a steady stream of products for a constantly decreasing price, the required time–to–market has dropped sharply. Where, in days past, a new hardware part was first prototyped in lump elements, initially used in mask–programmable logic and finally cast in the target technology when the market share has proven itself, one currently must deliver directly a fully functional part in the target technology. Boosted from a high–performance microelectronic fabrication technology, that can only be profitable when used in large quantities, high–performance software programmable parts have come into existence that allow for a one–time right product. In other words, new market segments have opened that require a mature software technology.

When we talk about systems, we will therefore mean some mixture of hard– and software parts, that must be designed simultaneously. Though we welcome re–usage of both hard– and software, we will not assume that such a re–usage has led to a standardization of either hardware and/or software. Though the design of software on a standard machine or the design of hardware to be used from standard software is by no means a solved problem, it is simply not the topic here. Further we assume that the binding of hard– and software is so intimate, that the design and assembly of the parts does not automatically bring the full product. In other words: we have to consider both and in combination.

## 2.1. Relation between software and hardware

The intimate interplay between software and hardware is especially apparent in the maintenance phase. Here, the product is designed and probably already manufactured and on the market, but next versions are required for a number of reasons. In [28] such reasons are named:

- **perfective changes** [55%]: –> functional specification changes in the software
- **adaptive changes** [25%]: –> tuning the software with hardware
- **corrective changes** [20%]: –> correct latent faults in the software or hardware

In Fig. 2 these maintenance steps are put into perspective. Here, we adhere already to the paradigm of **Collaborative Design**, where the initial specification is developed to a software program, from which parts will be realized by dedicated hardware modules and parts by programmed hardware modules. As the basic goal is an effective mapping of software on hardware, their rela-

tion will always play a role. This relation will gain importance when the involvement of hardware will be more unique. As a side–remark: It is important not to introduce new faults during the maintenance phase because these faults will reduce the reliability of the software.

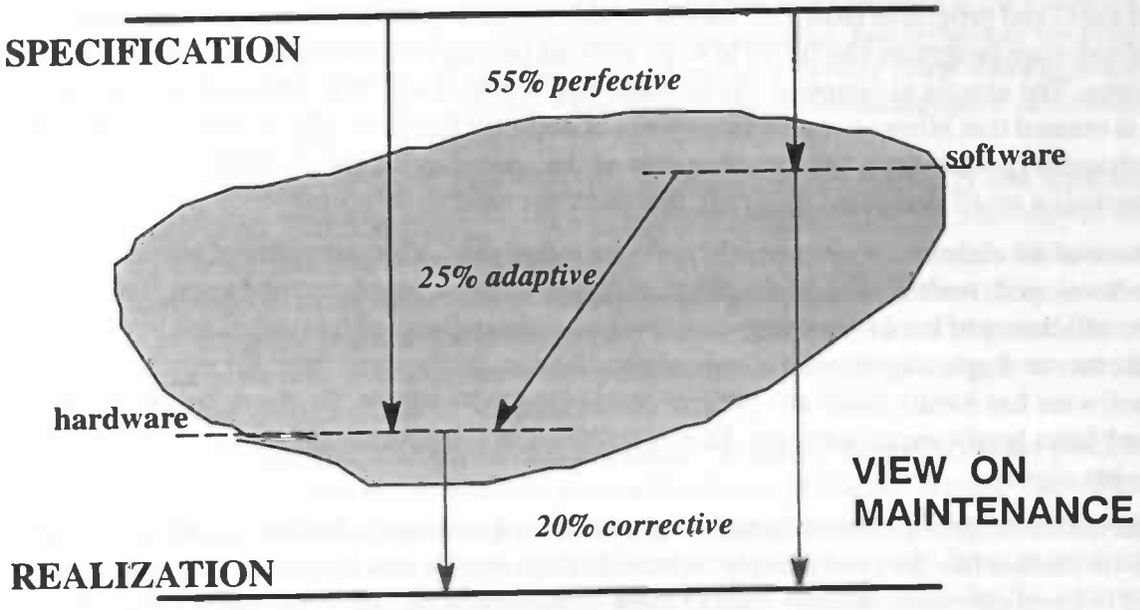



*Fig. 2: The software/hardware interplay.*

A study by the US Government Accounting Office (GAO) was released showing that a majority of the government software projects failed to deliver useful software. But most people didn't know that the GAO selected projects that were known to be in trouble. This study only proved that projects in trouble almost never recover [22].

In 1991 the Queens Award for the application of formal methods (in this case Z) was given to a release of the CICS system software. What had actually been done was a rewrite of the known failure prone modules. This gave a significant reduction of the errors in the program. Selecting and rewriting of the modules was the probable reason for success, and not using formal methods [22].

### 2.1.1. The many faces

The path from concept to reality will be travelled in a world with many different characteristics. Conceptually we discern here abstractions and views. An **abstraction** is a description of a design part in terms that provide a condensed meaning with respect to underlying abstractions. As such a synchronous model implies the notion of time, which in the underlying world will be modelled by means of a clock. At a further lower world, time will be carried by delays. Various researchers have attempted to standardize the choice of abstractions; sofar this has only been successful where the fabrication technology has dictated their usage. As a consequence, a variation in technology may lead to a different choice in abstractions.

Abstractions have different aspects in the various views on a design. A **view** is a filter on the characteristics carried in an abstraction. One usually differentiates between the functional, the structural and the geometrical view. In the functional view the software description is collected, while in the geometrical view the hardware description can be found. The structural description can be seen as an intermediate representation to aid the transformation between the (functional) specification and the (hardware) realization.

The various development paths from specification to realization can now be depicted as steps between abstractions and views in the so–called **Gajski–chart**. Various design methods result in different routes but still nothing is said about the notation by which the design is carried. Each design will feature a specific usage of alphanumeric and graphical notations. In practice, software will mostly be described alphanumerically while hardware is mostly using graphical means. Lately, however, this has been changing with the advent of Case–tools for software and HDLs for hardware.

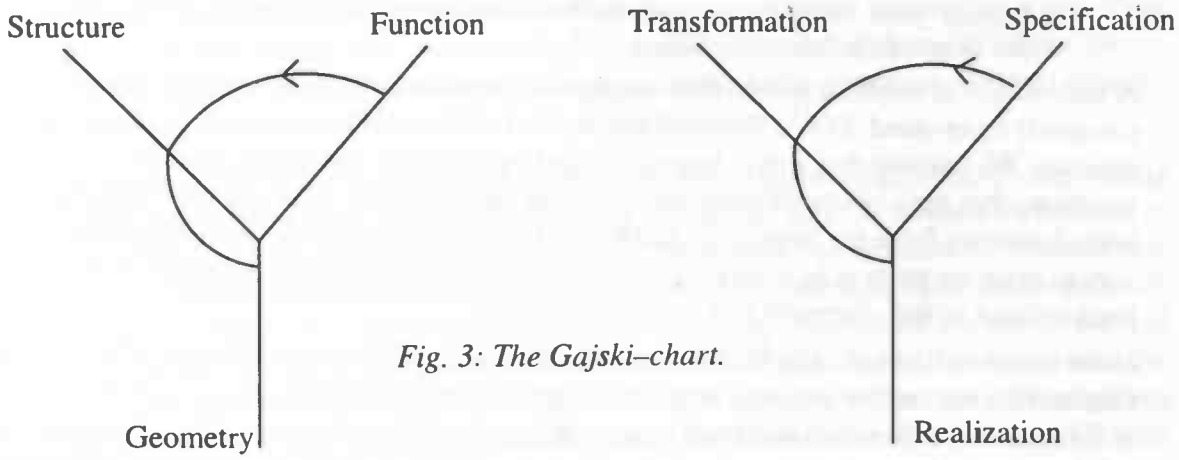



*Fig. 3: The Gajski–chart.*

The final product of hardware and software is different. Hardware is a collection of 3–dimensional units put together. Software is a collection of logics written in words and being compiled to an executable. This already gives a hint on the difference between hardware and software engineering. Since the 70's hardware–engineering has studied the reliability of hardware; hence they know a lot more about the creation of reliable hardware. As shown in Fig. 3 hardware programmers will start by making programs of what the hardware–unit should do in the form of functions. They use the divide and conquer method to solve problems. So these programs consists of a lot of procedures. Then they construct a hardware–unit from these functions. These units can easily be checked for flaws using all kinds of tests. A really useful test is $I_{ddq}$; this can be seen as a sort of profiling. Also they have many models for the development of reliable hardware. Software starts with a specification which may not yet be complete. After that, a program is created through development technologies (Fig. 3).

## 2.1.2. Modularization

Both the hardware module and the software procedure are characterized by a division between the body and the interface. In the function body all internal operations are provided and ideally these internal operations have nothing to do with the outside world and vice versa. On the interface all external operations are provided and ideally these external operations appear only on function interface. In order to achieve this separation of concerns, we will attempt to explain what is nowadays called **iconisation** in software engineering and **modularization** in hardware development.

I'll explain it with an example. In the early days of making a watch, the designers needed more than 1000 parts to create a watch. But nowadays, they only use 38 parts for a Swatch–watch. So putting these parts together is nowadays much easier as in the early days. But the number of parts had also an effect on the reliability. People make less mistakes putting 38 parts together as for 1000 parts. And thus the reliability of the modern watches is much higher as in the early days. So there is a reduction in design complexity and in assemble complexity of a watch. The watch has a higher manufacturability. Another example can be found in the car building. A car builder

doesn't make exhausts themselves anymore, but orders them at a company specialized in exhaust manufacturing. They only get the finished product and attaches it to the car. If an error occurs in the exhaust, it is a problem for the company that got specialized in the exhausts and not for the car–builder.

> The examples of modularisation and manufacturability has also happened in the hardware. This doesn't mean hardware–programmers can't make mistakes. Just remember the problems Intel had when the users discovered that the Pentium processor could make mistakes in floating–point calculations. And only 5 bits were wrong in the floating–point calculation tables to speed up these calculations [11].

A side–effect of modularization that has rapidly becoming the main virtue is that iconized parts can easily be re–used. This is standard practice in hardware engineering, but a novelty in software practice. We assume that this is largely caused by the main notational scheme. In hardware description, that may cover all three views on the product–part, the need for a clear interface has been dominant from the beginning and has led to various solutions. In the physical domain, the way to separate parts is by checking for overlap between the bounding boxes. Later on, this has been refined to the abutment box at no great effort.

In the structural domain, the function can be enriched by buffers and registers at the module interface and the interaction between modules is standardized over protocols. A well–known example is the standard cell, which has been in popular use for schematic entry. There have been no clear solutions in the behavioral domain, but as the behavior was usually implied from its entered structure, this was no real problem. This situation is different for software as under the conventional header/body convention, locality of specification can not easily be guaranteed. This has spurred the advent of object–orientation. The critique we want to raise here is that the classical separation between hardware and software may lead to situations where in both worlds the modularization problem is solved separately; thus requiring double the effort.

Sofar we have seen that the inherently 2–dimensional composition of hardware provides easy means to support iconisation. Sofar as software is restricted to this division of labor by allocating each software entity in a hardware entity, there is no need for iconisation at the specification level except for the support of the compilation process. Such programmable hardware parts can range from Programmed Logic Arrays to microprocessor core modules. Though this restriction is acceptable for silicon compilers, the general software problem will not really profit. In this case, an overall iconisation concept is required that is usable for software and hardware. We want to raise here an additional point: for the development of a quality product, it will also be necessary to iconize each function with respect to the effect that faults may have.

### 2.1.3. Towards CoDesign

We have regularly encountered a likeness between software and hardware. This does not mean that the problems and solutions are the same. Even if they were, then different names are being used. But even if the same names would be used, there appears to be a range of small differences that must be taken into account to determine the most efficient repairs. On the other hand, with so much being alike they can not be handled in separation. Replacing the hardware may urge to change the operating system and so on. There will always be differences, but it will pay to come up with a methodology that is aimed to restrict their spread through the re–engineering effort. This area of research is called CoDesign, short for Collaborative Design. And though it is originally advertised as a way to limit the re–design cycle, we see it rather as the discipline in which system quality is best researched.

From Fig. 3 we stipulate that the specification phase is deemed to be software–oriented. Such a specification should be platform–independent and therefore portable over many re–designs. Re-

use, a major factor in the reliability of hardware–engineering, is in the software–engineering still very rare. Nowadays there is a bit of reuse in the object–oriented approaches. But still the connections between the components of the reusable software have to be laid manually, and can be difficult as well, as I have experienced. The complexity issue hasn't been reduced yet. The more complexity in the software, the higher the variation in the software, and thus less chance on reuse. Except on functional abstraction, as an example the user– interface, where it is possible to have reuse in the software. Also the check on correctness of your program can only be done on the syntax and not on the semantics. So it is really difficult with software to get 'error–free' code. Even if they tell you software is bug–free, there is a high probability that it still contains hidden errors. Also the environment in which the software is being executed (the operating system) changes often and this can result in unexpected behavior of the software [25].

Testing has been a major issue in both software and hardware engineering. In hardware, testing has grown into a discipline of its own right, featuring a range of specific circuits techniques to improve testability and a range of CAD–tools to create the stimuli that are most efficient in performing the test. Design problems are assumed to be removed by extensive simulations and only single fabrication problems may have occurred. From this assumption, stimuli are found that will indicate the existence of exactly that problem. In software, testing has led to little more than the existence of debuggers. It is entirely up to the user what to investigate and no prior assumptions on the problems are made. Software testing techniques are therefore targeted on the identification of the unknown problem.

The problem can originate in the specification itself. Then it is called an **error**. The problem can also result from the specific coding; in this case it is called a **fault**. But irrespective of the origin, the problem will always have to show up during the execution of the program: the **failure**. These three notions are related. An error may be correctly coded but still be a fault. Vice versa, however, a coding fault will not necessarily be an error. As these problems have different origins and different effects, they often have to be handled differently. This has always to do with the potential proliferation of new problems introduced by the repair.

We will use the words error, fault and failure in direct connection with the system under design. A failure will always be caused by an error of a fault, but it may not be our error or fault. This implies that we may not be able to repair the failure. Then a **work–around** is necessary, but this will obviously introduce an error: i.e. when the specification makes false assumptions on the operation of the underlying software and hardware, we have to change the specification although it is essentially correct. Of course, this is extremely hazardous, as when the underlying software and/or hardware has been corrected, we have to de–falsify our specification (?!).

## 2.2. Faults do occur

People have tried to classify hardware and especially the software faults so that you get a global idea of what types of faults are in the system. Software faults are those faults that result from errors in system analysis or programming errors. Hardware faults are those faults that result from the malfunction of the system.

Although at first sight this is an easy and unambiguous distinction, in practice there are several border cases, which are increasingly difficult to classify. This will not improve with the growing integration of software and hardware parts in the area of Embedded Systems. We will follow the classification proposed in [29] as it is by large generative and does not use the hardware / software distinction: faults have an origin, can be observed, impact the system and are characteristic for a specific development phase.

### 2.2.1. Symptom and cause

This classification is often used because of it's practical and easy use. Often is the input also considered, and the system reaction on it. The following table will show a possible classification.

| Input→<br>↓ System reaction | Outside the domain | Inside the domain |
|---|---|---|
| input rejected | correct | normal fault |
| wrong results | normal fault | serious fault |
| system breakdown | serious fault | very serious fault |

Every fault in a computer system may be traced back to one of the three following:

1. Erroneous system design in hardware or software.
2. Degradation of the hardware due to ageing or due to the environment and/or degradation of hardware/software due to maintenance.
3. Erroneous input data.

A design fault occurs when, despite correct operation of all components and correct input data, the results of a computation are wrong. A degradation fault occurs when a system component, due to ageing or environmental influences, does not meet the pre–determined specification. This can not only occur in hardware, because the current hype called "legacy" shows that software is also not without. An input fault occurs when the actual input data is wrong, or from incorrect operation of the system. If a database is part of the system, then hazards from within the database are a software fault; otherwise they are input faults. The following table details these three types of fault.

| | Hardware faults | Software faults | Input faults |
|---|---|---|---|
| Fault cause | Age / Environment | Design complexity | Human mistake |
| Theoretical | NO | YES | NO |
| Practical | NO | NO | NO |

### 2.2.2. Control and Observation

Often the effects of a fault can only be detected by a change in the input– or output–behavior. They also use the term that an internal fault leads to an external fault. Only the external faults can be observed. If a system contains redundancy, which means that a system contains more resources than absolutely necessary for fulfilling its task, an internal fault doesn't have to lead to an external fault. The division of internal and external faults is heavily dependent on the chosen fault detection interface.

As already follows from the above discussion, it is important that the system is brought into a state in which a fault can be observed. Not just any input will do; one has first to sensitize the system for the potential fault and then to apply just that input that under these circumstances will produce an external event if and only if the considered fault is present. As a consequence, it is always required that the state of the system is known. If this is not the case, then a specific sequence of inputs is necessary to force the system into a known state. The most well–known type of such a homing sequence is the simple and straightforward hard reset.

To efficiently control and observe, fault assumptions are necessary. The well–known single–fault assumption works only for almost fault–free systems; if a large number of faults is still present, other models are required, such as a specific fault distribution.

### 2.2.3. Impact

The function specification of a system can be divided into primary and secondary functions. Primary or **core functions** are essential to the execution of the program as they produce results which are later on used in the execution. If a fault occurs in such functions, this will have fatal results for the system. Secondary or **support functions** are auxiliary to the program execution. If they fail, the results may be erroneous but not catastrophic. Sometimes the system may even recover from the failure.

A **permanent fault** occurs on a particular moment in time, and remains uninterrupted and repeatably in the system. A **transient fault** is a permanent fault that changes the system characteristics for only a relatively short period of time. Transient faults in the hardware are very difficult to distinguish from transient faults in the software.

### 2.2.4. During development

The development and use of a software system proceeds by a number of steps which lead to a further development stage. The faults that occur can be classified according to the development stage in which they have been made.

1. System analysis errors; such as (a) wrong problem definition, (b) wrong solution to the problem, or (c) inaccurate system definition.
2. Programming errors; such as (a) wrong translation from correct system definition to the program definition, (b) syntax errors, (c) compiler errors, and (d) hardware errors.
3. Execution errors; such as input errors.

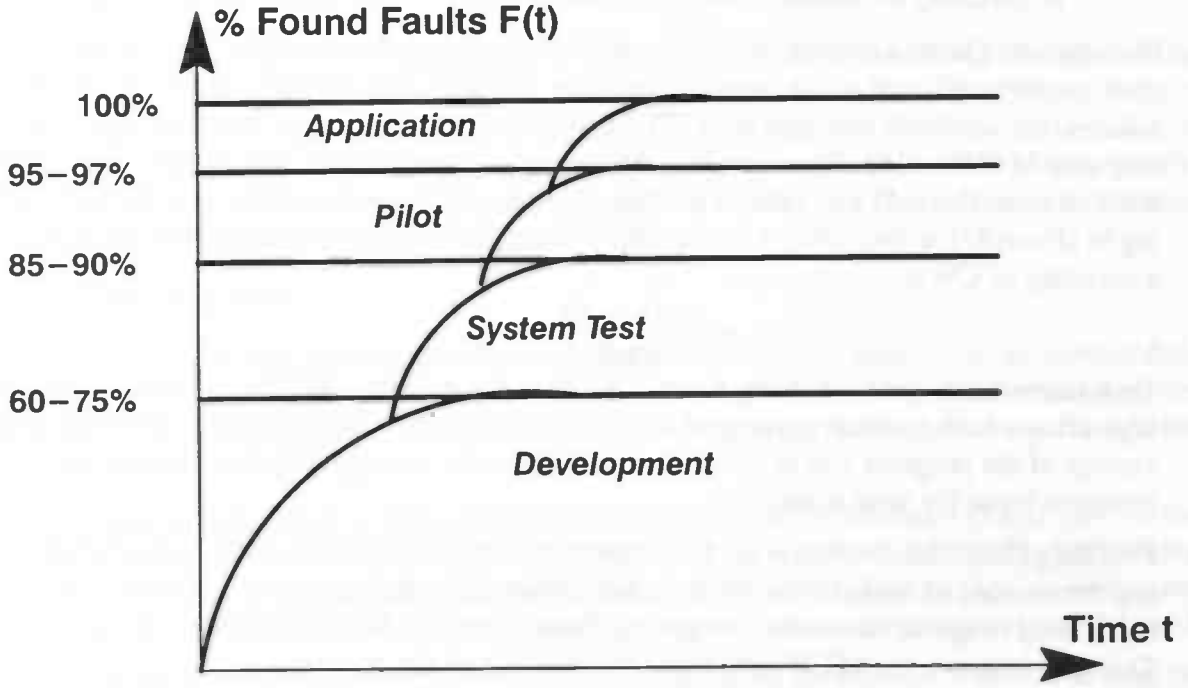



*Fig. 4: Fault recovery by development phases.*

## 2.3. Designing with faults

All the following models should help reducing the number of faults in the software. These models can be grouped into three categories: Avoidance, Detection and Tolerance. In [39] also Correction is named. To us, Correction is implied in either Detection or Tolerance.

### 2.3.1. Avoidance

Fault avoidance techniques are those design principles and practices, whose objectives are to prevent faults from ever coming into existence within the software product. Most of the techniques focus on the designing process. These techniques fall into the following categories:

1. Methods of managing and minimizing complexity.
2. Methods of achieving greater precision during the different stages within the design process. This includes the detection and removing of translation errors.
3. Methods of improving the communication of information.

**Case–tools:** Computer Aided Software Engineering (CASE) tools are systems designed to aid in the design of software. This only works for the less difficult programs that have to be designed. It helps also to show the structure of a program: which function is connected with other functions in the program.

**Design structure:** Using a top–down design, with structured code, design code inspections by other parties and incremental releases.

### 2.3.2. Detection

Provide the software with means to detect it's own faults. Most fault–detection techniques involve detecting failures as soon as possible after they arise. The benefits of early detection are:

1. fault–effect can be minimized.
2. researching the failure cause will become easier.

**Simulation:** Create a simulation version of your program and start testing. This model has some disadvantages. First of all, it costs a lot of time. MTTF is the average number of uses between failures. Reliability R is related to MTTF by: MTTF = 1/(1–R). If time is interpreted in any other way it is: MTTF = U/(1–R), where U is the average number of time units per use. This gives that Reliability is: R = (MTTF–1)/MTTF. Or on another time basis: R = (MTTF–U)/MTTF. According to [Poore93], a measure for the number of samples S needed to get a reliability score R with a certainty of C% is:

$$S = \frac{\log(1 - C)}{\log R}$$

This means that to get a reliability score of 0.999 with a certainty of 99%, you need 4603 samples. If you find a fault, you have to correct it and start over again. Also the creation of a representative version of the program you're simulating isn't that easy as suggested. Further you need a representative input for your simulation.

**Profiling [35]:** This method is used to improve reliability by making a listing of all the functions and the number of times these functions have been called during a period of time. This listing is being used to spend most effort on getting those functions fault–free that are being used most. This will improve the MTTF, and thus the reliability of the program.

**Error isolation:** This third concept is to isolate faults to a minimal part of the software system, so that if a fault is detected, the total system doesn't become inoperable. Either the isolated func-

tions become inoperable, or the particular users can no longer continue to function. As an example: Telephone switching systems try to recover from failures by terminating phone connections rather than risking a total system failure.

**Fallback:** This concept is concerned with trying to shutdown the system gracefully after a fault has been detected. This looks like creating an UNIX–core file and thereafter checking it with gdb. This can only be done, if in advance the code has been compiled for gdb.

### 2.3.3. Tolerance

These techniques are concerned with keeping the software functioning in the presence of faults. They go a step further as fault detection. Either the fault itself, or the effects of the fault are corrected by the software. The strategies fall into 2 categories:

**Dynamic redundancy:** The first idea was obtained from the hardware where an identical backup component is applied when the used component was detected faulty. This however didn't work in the software, because the software component will show the same fault, unless it has been coded completely different. Also the detection of a faulty component isn't as easy as suggested here. The remaining ideas are attempts to repair the damage, but then you have to know how the damage could look like, and this prediction is difficult.

Another concept is known as **voting**. Data is processed in parallel by multiple identical devices and the output is compared. If a majority produce the same result, then that result is being used. This model has the same drawback as the idea with the backup components.

**Prospective redundancy:** Different groups of programmers write the same procedure or function for the system. After the groups have written it, the code is compared, and eventually faults and errors are more likely to be found. This still gives problems with connecting all the procedures and functions into one program. Another drawback is that all programmers need to develop the same style of programming.

## 2.4. Coping with failures

The factual way to establish the presence of a fault is by noticing a failure. As the functioning of the system is basically determined by the software personalization, the modelling of failures is a software affair. However, through this software layer the effect of hardware faults can also become noticeable and have to be coped with. For the overall system we have to deal with specification, observation and recovery. Finally we propose in summary how the respective techniques for quality assessment fit together.

### 2.4.1. Specification

Failure models should be models that show the needs of the users. It reduces the number of failures because the users gets the results they expect. Also it gives the users an idea that they are being involved in the creation of the software and thus improving the trust the users have in the software and the builder [27].

**Non–structured Interview:** A non–structured interview should be used as a global view on the domain on which the user wants its software. It provides for an initial impression, but nothing more. Recording the interview is the best way to save the information. This method is mostly used as a start to get a first impression on the subject. After that a more structured way should be used.

**Structured Interview:** This model gives better information on the domain. It should be used on at least those pieces of the domain that aren't completely clear and need more structure, but will never replace the ideal of a formal specification.

**Prototyping:** Create as soon as possible a working version of your software to recognize the expectations of the user. This model aims to reduce the number of inconsistencies between the users and implementors idea of what should be constructed, and thus reduces also the number of failures. And with less failures, less faults. Note that prototyping gives no assurance and its value is fully dependent of its usage.

Thus the importance of the specification lies in the mutual agreement between supplier and customer about the desired functionality. Though a complete specification is the final target, this is often not the case in practice as much of the functionality will only become desired during the development. When the specification has finally become mature, we have the opportunity to formally proof an implementation to be correct. When the specification is still under construction, this proof technique is still feasible but not conclusive. The approach taken in proving a specification is to construct a finite sequence of logical statements from the input specification to the output specification. Each of the logical statements is an axiom or state derived from earlier statements by the application of one or more inference rules. More can be found in the literature written by Floyd, Hoare, Dijkstra and Reynolds [43]. Recently they are trying to create program–provers, but they aren't perfect yet, and use a lot of resources. Gerhart and Yelowitz have shown that several programs which were proved to be correct still had some faults. However the faults were due to failures in defining what to prove and not in the mechanics of the proof itself [13]. Therefore we stipulate here, that

> formal proof techniques are of immediate advantage when the specification is complete and the implementation has little errors left.

### 2.4.2. Observation

Another way to cope with failures is by testing. Program testing is the symbolic or physical execution of a set of test cases with the intent of exposing embedded faults in the program. Like program proving, testing is an imperfect tool for assuring program correctness, because a given strategy might be good for exposing certain kinds of faults, but not for all kinds of faults in a program. Each method has its advantages and limitations and should not be viewed as competing tools, but as complementing tools. To handle failures their recording should contain enough historical information that an analysis can be performed to establish the related faults and/or errors.

**File recovery:** Recovery programs can reconstruct databases in the event of a fault, if a transition journal is being kept, and also an old backup before the transitions is saved.

**Checkpoint/restart:** This strategy makes a backup of the system every x minutes/hours. This is an extremely appreciated feature in parts of the world, where the electricity tends to come and go unnoticed.

**Power–failure warning:** Some systems can detect a power failure and provide an interrupt to the software of a pending power failure. This gives the software time to make a backup or move the files to secondary storage.

**Error recording:** All hardware failures detected should be reported on an external file.

It is essential here that reliability engineering, risk assessment and quality assurance apply. We have failures recorded and if the amount of failures is large enough, we can predict by statistical means how much failures are still remaining. On the other hand, when there are just few failures but the code is large we can predict the vulnerability of code segments to hidden faults. We therefore state that:

> risk assessment and quality assurance are characteristics of industrial production.

### 2.4.3. Recovery

To handle failures, they should be recorded. However, as failures can be of an unknown origin, it will also be of interest to have measures to bring the system in a known state so that any recording can be analyzed without further assumptions.

**Operation retry:** A large number of hardware failures are just temporary, so it is always wise to retry the failing operation several times.

**Memory refresh:** If a detected hardware failure causes an incorrect modification of part of main storage, try to reload that particular area of storage. This method assumes that the data in the storage is static.

**Dynamic reconfiguration:** If a hardware subsystem fails, the system can be kept operational by removing the failing unit from the pool of system resources.

The essence of failure recovery is that either a known state can always be reached or (better) the impact of a fault can be limited. The former "homing" technique is widely applied and is implicitly connected to a full breakdown of the system. It will loose all unsaved information. The latter technique is based on encapsulation of the procedure in the sense used for object–oriented programming. In the extreme, it will also have to support "Soft Programming" whereby examples teach the intelligent software part how the failure can be avoided next time:

> intelligence aids coherence to support the continuous adaptation to the environment.

### 2.4.4. In summary

In this chapter we have taken a look at the quality of Embedded (or at least hardware / software) Systems. We have seen a remarkable alikeness, though usually different words are used for basically the same aspects. The similarity extends from the design notions to the way faults are treated. This has brought us to a short review of how faults are handled during the design phase and finally produced a view on how failures can be treated.

In the end, we stipulate that quality assurance should be based on a future collaborative usage of formal proof techniques, rigid encapsulation / adaptation and lastly risk management. In the following chapters, we will focus more on the reliability aspects.

# 3. Component reliability models

The reliability of a system depends on the reliability of its parts. This is often reflected in the saying that "a system is as strong as its weakest part". In general, reliability involves a dilemma because we need it when the system does not fail, but can only quantify it for failing systems. To break this dilemma we might take refuge to simulation techniques. Based on reliability data on the system parts or by judiciously creating faults, it is possible to quantify failures.

The question remains on the reality level of such experiments. It seems mandatory to start from facts and such facts can only emanate from real–life experiments. In the following we will largely discuss from a hardware background. In this area, a range of fault models are listed, that provide some basic knowledge on how faults can enter the system. As for a mature system, faults are rare to occur, a mechanism is required to accelerate the rate of occurrence. These comprise the world of "lifetime" experiments, where components are stressed to bring out the faults at a fast rate. Finally we will list some techniques to map the system failures to compute some reliability measures.

## 3.1. The origin of faults

One reason for a fault can be the occurrence of an error. But even when seemingly no error can be present after extensive proof and/or simulation, faults can pop up during the travel from specification to fabrication. Finally we may expect a fault–free realization, but again faults can come in from the blue during operation. Such may be **handling errors** or simply **latent errors**: faults of a mechanical/physical background that are potentially there and may come apparent during the life–time of a component. A proper design aims to reduce the probability of such errors: this marks the difference between a one–time device and a manufacturable one.

In the following we give a long, non–exhaustive list of probable causes for such life–time annoyance. In line with [9] we will look into effects that are related to environmental temperature (ET), dissipated power (DP), electric field (EF) and current density (CD), but otherwise group the error mechanisms together in cross–talk, hot–spot and wear–out effects.

### 3.1.1. Cross–talk

The physical placement of components may result in the introduction of unwanted additional components. Though a large number of these components come only into life outside the operational range of the wanted components, not all can be fully disabled. One important example is cross–talk. In its simplest format cross–talk occurs between two crossing connections. From the planar technology a crossing will always introduce a capacitive coupling, of which the effect may remain unnoticed by selecting a proper impedance level for the driving gate of the receiving connection. In a general sense, there are more situations that may lead to spontaneous coupling.

**Between components** a coupling may result from temperature or electro/magnetic effects. The antenna is an example of how electro/magnetic fields can be applied beneficially. Overall EMC (Electro Magnetic Coupling) will present so much problems that EMC–shielding has become a major technique. On–chip local temperature couplings can have the same impact. EM–faults are often hard to predict and require dedicated CAD–software.

**Over connections**, coupling may result from an incapacity to handle spuriously large signals. Too small lines may lead to current hogging that lifts the reference voltage levels such that stored values can disappear. This may especially be a worry in memory parts where the urge of miniaturization needs to be balanced against reliability.

Overall, the cross–talk phenomenon can be interpreted as a side–effect. Two seemingly separate values are temporarily connected. This has a clear interpretation in software where the side–effect has become a major issue in coding quality.

### 3.1.2. Hot–spot

In a number of domains, the capacity of a component may be exceeded. This can lead to saturation, but for a number of physical phenomena also breakdown will occur. This is most known in the temperature domain, where too much local heat can bring the material to disrupt. Over and by, these are irreversible processes that can bring a permanent fault. In the following, we will list some in the domains temperature, voltage and current.

**Current breakdown.** In all those cases where a current flows through a conductor, this current will cause heat dissipation according to: $P = I^2R$. The density of the local dissipated power is defined as: $\delta P/(\delta x \delta y \delta z) = J^2\rho$. Combining them will lead to the formula:

$$P = \int_{x,y,z} J^2(x,y,z)\rho(x,y,z)\delta x \delta y \delta z$$

For many materials the resistivity has a positive temperature coefficient. As a result, local power dissipation will result in a higher local temperature which leads to a higher local power dissipation. If this cycle can't be stopped, the component will finally melt. So local power dissipation must be avoided.

**Power breakdown** (thermal cracks). The highest temperature a component can have is until it rises up to a level where physical processes will change the properties of the component. Above a temperature of about 350 degrees Celsius silicon components no longer function. In practice the thermal/mechanical structure of a component will lead to a crack in the structure because of the different thermal expansion coefficients of the materials used.

**High voltage breakdown** occurs at the moment when a current flows through an otherwise isolating layer of material. It is possible to distinguish three types of breakdown: Impact ionization, Avalanche and Zener breakdown and the Electron–trap ionization. A common aspect is the presence of an electric field enabling charge carriers (p.e. electrons) to gain energy required for transfer to the conduction band. The difference is between how the actual conduction is achieved.

1. **Impact ionization:** This is the simple case in which, by collision, the electrons can escape the valence band to the conduction band. The effects are destructive.
2. **Avalanche breakdown**: In the case of an Avalanche breakdown a collision between electrons will result in a hole and an electron. This process won't lead till immediate destruction of the component. It might trigger other latent errors.
3. **Zener breakdown:** In the case of a Zener breakdown, the field is able to raise the energy of electrons, which will result in a hole and an electron. For the rest it's totally equal to the Avalanche breakdown.
4. **Electron–trap ionization:** Materials with impurities might have electrons hopping from one impurity to another. This model is highly temperature dependant. It's one of the major breakdown mechanism of MOS gate oxides.

**Pulse power effects:** Switching of a bipolar semiconductor from reverse biased diode to a forward biased diode might lead to overcurrent or overpower problems. Switching a forward

biased diode to a reverse biased diode might lead in considerable reverse currents through the diode. High–voltage diodes have these problems more as low–voltage diodes. This problem is directly related to the power–dissipation in a device.

**Second breakdown:** In high power and high frequency applications within maximum current and voltages ranges catastrophic failures occur. These failures manifest themselves by a sudden collapse of the collector–emitter voltage and loss of control of the base drive. This results in such a heat that both the crystal and silicon will melt. This failure mechanism is found under both forward– and reverse bias conditions.

In software, a hot spot can be defined as register overflow or missing synchronization. Here, the amount of data or their rate exceeds the capacity of the system and important information may be lost.

### 3.1.3. Wear–out

Next to temporary effects (cross–talk) and permanent effects (breakdown) we will often encounter slowly varying processes. These are commonly also referred to as "ageing". Their common explanation is a reversible physical process of which the remnants add up to an eventual fault.

**Corrosion:** Corrosion may occur due to a combination of moisture, D.C. operating potentials and CL– or NA+ ions. Absence of one of these aspects will inhibit corrosion. A typical example is a leaky package, for which the defective isolation may admit contaminants to creep in during life–time. Corrosion is so much random dependent, that even one accurate formula for this failure mechanism is impossible.

**Electromigration:** The continuous impact of electrons on the material atoms, may cause a movement of the atoms in the direction of the electron flow. Especially aluminium metallization tracks on semiconductors show this failure mechanism. Finally at one end of the track there are no atoms left, and they are piled up at the other end. A commonly accepted formula for this behavior is:

$$\textit{Mean lifetime} = A \times J^{-n} \times e^{\frac{\delta E}{k \times T}}$$

where
- J: current density
- n: constant
- T: average temperature of the conductors
- k: Boltzmann's constant

Though technology has drastically improved over time, the gate oxide will always be slightly contaminated. Their fixed charges may move towards the gate/channel interface driven by the constantly changing signals and eventually change the threshold–voltage over time.

**Secondary diffusion** may occur when atoms diffuse themselves into other ones. This reallocation process is at room temperatures only relevant over very long time–periods. When primary or secondary diffusion effects are used for electrical programming (such as in Electrically Erasable Programmable Read–Only Memories: EE–PROM), the physical type of operation will limit the amount of re–programming.

In software we also have the problems of ageing. Programmers still get educated in the languages of the 70's like Cobol or even assembly, because these programs still exist and have to be maintained. These programmers will have to work themselves into the code and fixing errors is troublesome because of the readability of the code. Also there is the problem of hardware. The hardware might not be produced anymore at the time something breaks down and finding re-

placement is sometimes impossible. So the software might still be functioning but the hardware is not available to support it. This is nowadays known as "the legacy problem".

In contrast to the former categories, wear–out can be unavoidable. It may be part of a maintenance scheme to regularly check for old components and replace them before they actually fail. It must be part of the design considerations to ensure that an ageing part will not affect the remainder of the system adversely. In short, it is required that the risk on a failure is evaluated in its fullest consequence.

## 3.2. Structural faults

Structural faults need to be abstracted from the previously discussed physical ones. Little effort in this area has been spent on analog circuitry, as here the notion of abstraction has been hardly used. Instead design centering has been widely applied to ensure the largest safety margin between the typical design and its extreme situations. This will be further discussed in section 3.4.1.

More effort has been devoted to model faults in digital circuitry as such circuits tend to be larger and more uniform. Conventionally the faults are then cast into changes of the network topology, or to put it more precisely into changes of the wiring. In line with [1] we can distinguish three categories of structural faults:

1. **Fabrication faults**
   Some faults are caused by the vulnerability of the fabrication technology for mask and environment failures, causing:
   - wiring fault: An incorrect connection between 2 modules.
   - crosstact fault: A crosstact fault is caused by the presence of some unintended mask programming.

2. **Primary faults**
   Most of the physical component faults can be mapped into a constant false value for a connection, irrespective of the changing signal it should carry in the case of a fault–free circuit. The used fault model depends strongly on the abstraction level and has a deep historical value.
   - stuck–at fault: A line or gate is erroneously carrying the constant logic value 1 or 0.
   - stuck–on fault: A transistor has permanently the logic value 1 or 0 including a potential memory effect resulting from the occurrence of floating nodes.

3. **Secondary faults**
   The last, but most difficult, category consists of faults with a changing (but still false) value. Under such conditions, the fault usually implies two or more wires with a mutual dependence.
   - short/bridging: A line got connected to another line in the network, creating a short–cut in the network.
   - coupled: A pair of memory cells, i and j, is said to be coupled if a transition from x to y in one cell of the pair changes the state of the other cell.

Despite the variety in potential fault models, we will largely pay attention to the primary component faults, emphasizing the historical development in this area.

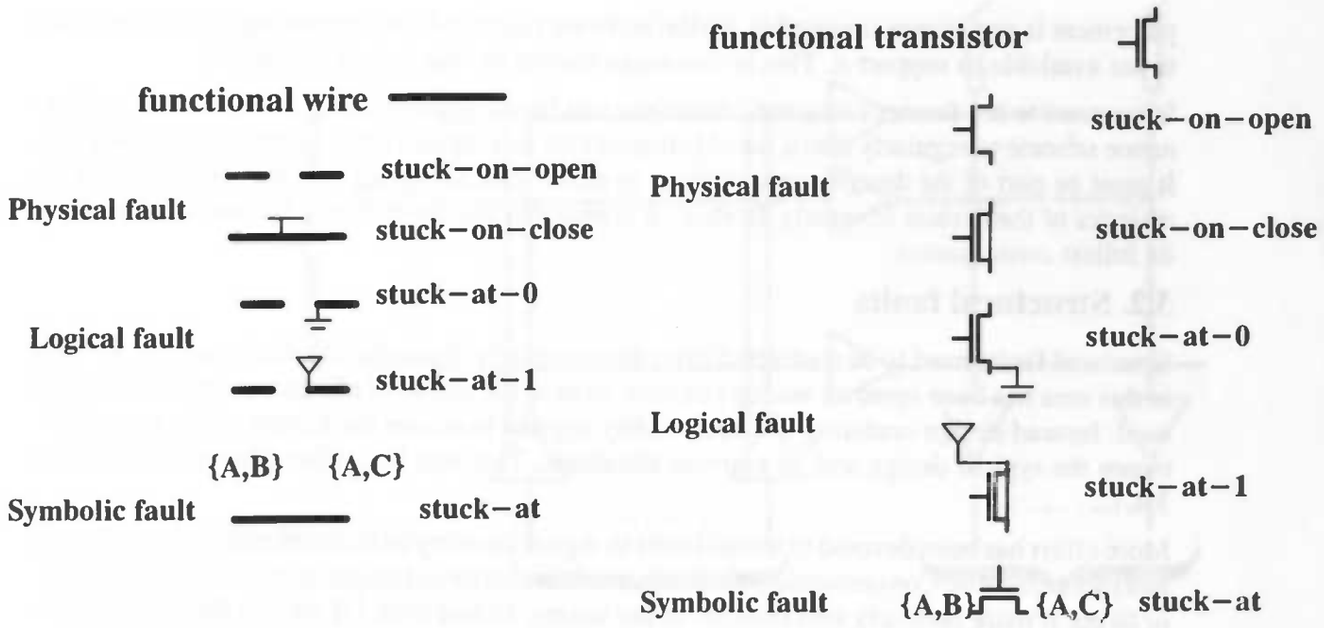
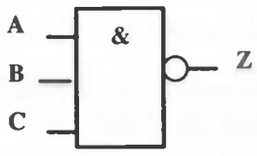



*Fig. 5: Different stuck faults and their origin.*

### 3.2.1. Logic gate–level models

The logic gate–level model is the most ancient in this area and relies solely on the **stuck–at fault** model. It has the following advantages/disadvantages:
– usable for wear–out faults, not for 'shorts' and 'opens.'
– simple in use and design.
– copes with most of the faults.
– usable information for statistical analysis.
– faults at transistor level have no correspondence with faults at logic level.
– Modern systems are so complex, that the gate–level representation might result in a prohibitive number of faults, which have to be considered for the system.

The primary thought behind the **stuck–at fault** model is that, due to physical effects or faults, the lines at the logic gate level or in the circuit are permanently stuck at the logic value 0 or 1. In practice only single faults in the circuit are studied. I'll give an example of this model with a 3 input NAND:




| A | B | C | Z | Fault Class |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | A–>0, B–>0, C–>0, D–>1 |
| 0 | 1 | 1 | 1 | A–>1, D–>0 |
| 1 | 0 | 1 | 1 | B–>1, D–>0 |
| 1 | 1 | 0 | 1 | C–>1, D–>0 |

*Fig. 6: A 3–input NAND gate.*

Suppose the line at A is permanently stuck at 1. An input of A: 0, B: 1 and C: 1 should logically give the output Z a value of 0. But because of the stuck–at 1 fault, Z shows the value 1. So the fault A stuck–at 1 gives also Z stuck–at 0. There are totally 4 lines, with 3 input lines, and thus there can exist a maximum of 8 possible stuck–at faults. However if an output is stuck at 1, it is impossible to find the corresponding value of an input stuck at 0. So these faults are said to be equivalent. So we get the table shown in Fig. 6. Here, A–>0 means A stuck–at 0. Also notice that if there is a detection of the lines A, B or C stuck on a value, you get automatically that D

is stuck also. This is called dominance. A–>0 dominates D–>1. In practice it's difficult to apply dominance and equivalence to more complex circuits.

### 3.2.2. Transistor–level fault model

In the transition from microelectronic design based on primitive logic functions towards logic structures from transistors, and especially with the coming of age of the CMOS technology, the need arose for a further improvement on the stuck–at model:

– to cope with more faults as the gate–level fault model.
– to have more complex elements than the gate–level fault model.
– to be more accurate as gate–level fault model.
– to be usable for wear–out faults.
– to be used for 'shorts' and 'opens' faults.

**Stuck–On fault model:** With a transistor permanently being turned on, both the P and N networks in a CMOS circuit may conduct, and the fault may or may not be detectable upon the resistance of the stuck–on device. If you have a valid stuck–on test, it also tests for corresponding stuck– open faults, while the converse is not true.
If P–transistor B is not present, the output node will be floating for the input condition AB=10. As a result, the output will carry for some time the result of the previous gate activation. If this is Z=1, then the fault may pass unnoticed. In other words, for the fault of P–transistor B missing to be notices, the output must be previously set to Z=0.

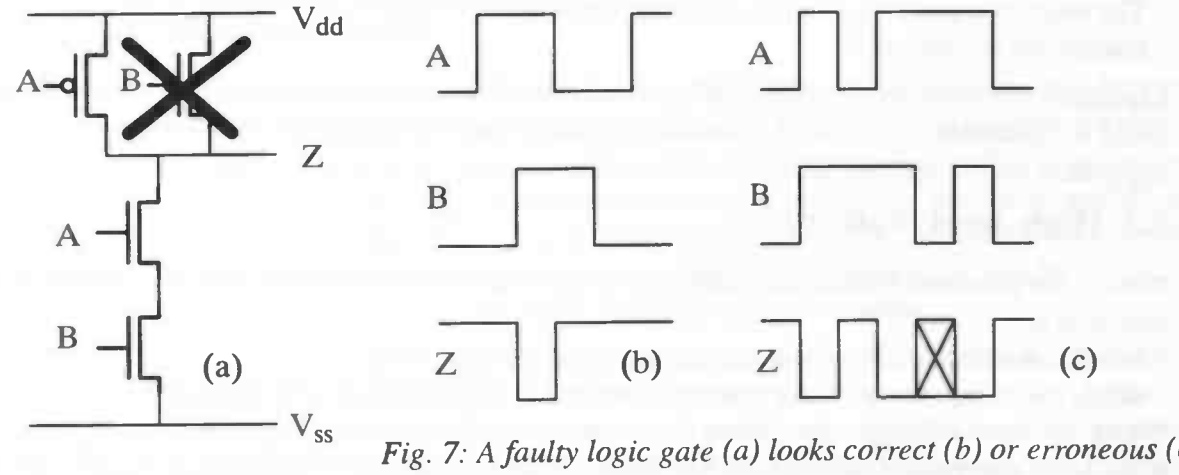



*Fig. 7: A faulty logic gate (a) looks correct (b) or erroneous (c).*

### 3.2.3. Matrix fault model.

Sofar we assumed the use of so–called irregular logic: logic of which the topology has no regular structure. Such a regular structure will often be a matrix of components and because of the inherent regularity some other fault models can be applied here. For the larger part, these fault models have to do with the mask–based personalization of a general–purpose template or with the very close location of the respective components.

Faults in the PLA could include stuck–at, shorts as well as cross–point faults. It has been shown that cross–point faults dominate all the other possible faults. A cross–point fault on a row causes the corresponding product term to grow, shrink or disappear. A cross–point fault on the output–line causes the appearance cq. disappearance of a new product–term.

As memory is a major part of the system, several models have been created to conquer the possible memory faults. Faults in a memory part may cause it to have changes in some electrical parameters or in timing. Faults will also cause memory to be functionally incorrect. The faults

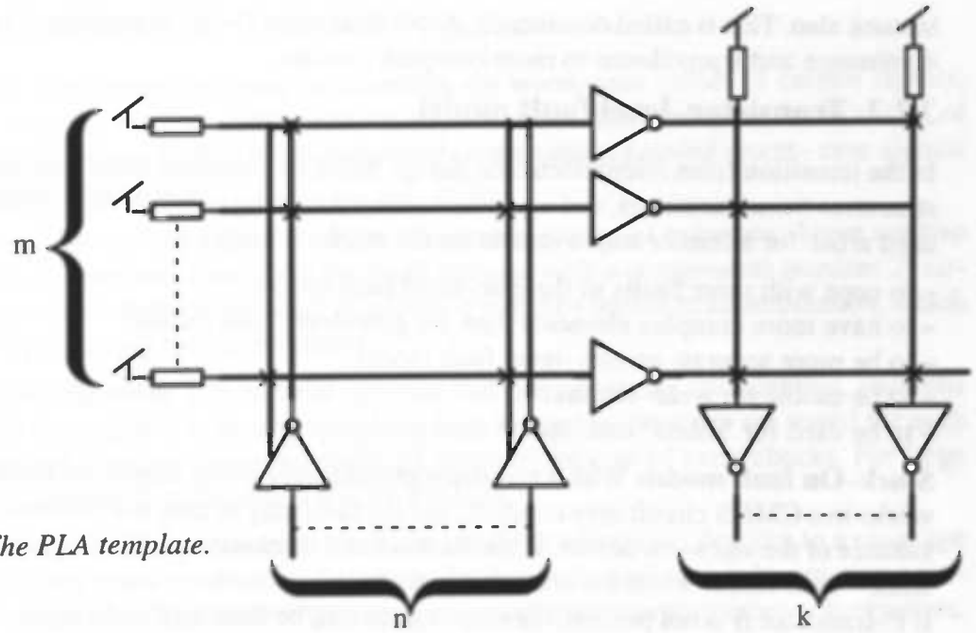



*Fig. 8: The PLA template.*

are:
- One or more cells in the system are stuck at the logic value 1 or 0.
- Two or more cells are coupled.
- The state of a memory cell can be altered as a result of certain patterns of 1s and 0s or transitions in other cells.

Especially for memory structures, a large number of special–purpose algorithms exist, that attempt to condition the matrix for the occurrence of specific faults. These faults tend to be very dependent on the applied fabrication technology.

## 3.3. High–level fault models

Next to the previous abstraction, fault models have also been developed on higher levels of abstraction to:
- handle more complex systems, at the expensive of accuracy.
- affect the components of the system at module level instead of transistor/gate level.

When the level of abstraction rises, the description of a model becomes like a program. The high–level component description will therefore be very similar to the low–level software description, both being typical behavioral utterings. We follow again [1].

### 3.3.1. Timing model.

As digital systems are still highly synchronized, timing information is critical for correct execution. As the test will often be performed on a special machine with analog accuracy, the test clock speed will be low and many timing faults may pass unnoticed. A faster tester will prove to be necessary but this will also place high demands on the available technology. Modern testers are highly specialized machines that use techniques on the frontiers of what is technologically possible. There are basically two types of timing fault models:

1. the **delay fault** where the detailed switching behavior may be dynamically different;
2. the **overrun fault** where parts of the messages may disappear.

Some failures only produce timing errors. Using circuit simulation to study the effects of faults is computationally very expensive. A compromise is to derive an approximation to circuit de-

lays in less computer time. An MOS fault simulator, developed by Shih [49], uses a table lookup of the transistor I–V characteristics to simulate both the fault–free circuit and many faulty circuits in one pass. The simulator can either plot voltage waveforms or extract both the logic and delay values for both the fault– free circuit and the fault circuits from the waveform. Simulation of tests generated under the assumption of zero–delay through the transistor and lines, would not detect some of the faults, demonstrating the need for using delay information.

### 3.3.2. Microprocessor fault model.

Microprocessors are the building blocks of very complex systems. Even though these modules are quite complex themselves, there exists quite effective fault models. The following fault–effects can be noticed:

– One or more microorder are inactive, therefore the order is not executed completely.
A micro–order consists of several microinstructions.
– Microorders which are normally inactive become active.
– Another microorder is active in addition to the normal microorder.
– Another microorder instead of the intended order is active.

This can cause incorrect values at the external pins or the destruction of internally stored data.

Within this same category falls the RTL fault model. Here the description is based on the transport of data between storage locations (a microprocessor could thus be described). Relevant faults refer to changes in the transport paths in the same manner as in lower–level models changes in the signal paths occurred.

### 3.3.3. Function fault model.

Given a combinational function with N inputs, the general fault model assumes that this function can be transformed into any other combination function with N inputs. So it requires the testing of $2^N$ possible inputs. This is very expensive to test. However if a function with a large number of inputs is implemented using a regular interconnection of subfunctions, each with a relatively small number of inputs, then the subfunction can be tested individually. The obvious way to provide such algorithmic descriptions is by a functional programming language. The basic mechanism is the function map; faults will therefore look like erroneous maps.

A related fault type acts on the communication protocol instead of on the structural implementation of the transport. A variety of mechanisms are known to specify communication. In the Petri–net the system is given as a pool of processes with a abstracted request–acknowledge protocol. Typical examples of what can be studied on this level are dead–lock, hang–up and liveliness. An alternative rendering is by a Process Algebra.

## 3.4. Observation of reliability

Reliability can be based on the existence of fault models and the assumption and/or measurement of their distribution. A well–known technique to guarantee maximum robustness for analog circuits is Design Centering. Here, the worst–case values are assumed and the design is typically set such that its operational margins are equally divided. In other techniques, this is not so easy to formulate. For instance, the timing behavior of a logic gate is strongly dependent on the number of active input signals and an operational mid–point would then be highly susceptible on the operational profile: the average number of active inputs for a typical program.

In order to design for reliability and/or to observe its impact a number of techniques have been developed [3]. We will first look into the design technique. Then, after a short discussion on the rendering of measurement results, we will try to summarize this chapter from a software perspective.

### 3.4.1. Worst–case analysis

Main purpose of this method is trying to determine the worst–case values of certain signals. Therefore you need the extreme component values, which are those values at the extremes of the parameter distribution. To find those parameter combinations causing worst– case signals the following possibilities exist.

**Vertex analysis:** This method explores all possible combinations of extremes. There are two disadvantages to this method. First, even for small circuits, with n components requires $2^n$ circuit analysis steps. Second, the worst–case signal value might be related to a combination, which is not detected by checking the vertex points.

**Parameter regionalization:** This method assumes that one parameter combination, also tells something about the region it is in, and the whole area within the extremes are tested for each region. This method gives an improved coverage factor at the cost of extra checks. For large circuits the total number of calculations will be too high.

**Monte Carlo analysis:** The previous methods are totally deterministic, leading to a coverage of the entire tolerance space systematically. A non–deterministic approach is by creating a random testset to be tested for. This is the only technique available with large number of parameters.

### 3.4.2. Pass/fail diagrams

This method uses the Monte Carlo Analysis and circuit simulation to generate a stressor set for a given failure mechanism of a component. Any circuit where the susceptibility exceeds a predetermined limit is marked as failed. All the other ones are marked as passed. Then the number of passes and fails is plotted against the initial parameter value. A disadvantage is that a medium sized circuit has too many parameters to have a straight–forward solution.

**Reliability optimization using the Center of Gravity.** First of all the pass/fail diagrams are calculated for the parameters. After that the center of the pass and fail observations are calculated according the following formulas:

$$Cp = \frac{\sum_{i=1}^{passes} pass_i}{passes} \quad , \quad Cf = \frac{\sum_{i=1}^{fails} fail_i}{fails}$$

The center of gravity method moves the nominal value along the axis between $C_p$ and $C_f$ in the direction of $C_p$. This leads till the determination of the Center of Gravity with the following formula:

$$CoG_{new} = CoG_{old} \times k \times (C_p - C_f)$$

There are 3 policies in determining the $CoG_{new}$:

1) $k = 1$
2) $CoG_{new} = C_p$
3) $k = 1 - \frac{passes}{passes + fail}$

At this moment the third option is the most used one.

### 3.4.3. In summary

In this chapter, we have reviewed some concepts in system reliability from a fault model perspective. In rough outline, it starts with the assumption of a small number of different components and/or a highly regular composition scheme. Then by creating a lengthy list of potential

faults we may eventually end up with a fault model for the circuit. At occasion we have even ventured some analogy between the hardware and the software concepts.

Still we have to stress the impact of the above assumptions, as in software we have neither of them available: (a) there are many different components, and (b) there is no regular composition. Hence, as hardware reliability techniques run into problems when the system complexity rises, software already starts at such a hard–to–manage level.

Typically we have sofar used the approach, that if a fault occurs, we want to know which fault and where. When this is no longer achievable, we might suffice with the simple question: how many faults are present? In this case, we do not want to know what the fault is. The simple fact that a fault exists makes the system already unusable. We are dealing here with the Heisenberg dilemma, which states that from electrons around a kern it's possible to calculate their place or speed, but not both [44].

A next difference between hardware and software is the fact that we can look into the operation of software when it is running, but we can not look into the actual integrated circuit. This allows us to split the fault detection from the fault identification phase. For fault identification we have on–line debuggers available, but for the fault detection we still need a production oriented reliability scoring technique. In the next chapters, we will largely focus on this part of the overall "design for manufacturability" problem.

# 4. System development models

During the development of a system the main concern is quality. But as discussed before quality has many aspects. We will limit ourselves here mainly to the production process in going from an agreed specification to the executable code and/or the runnable system. On receiving a developed system, the question arises of how much faults are latent. In other words, we wonder about the quality per se. During the use of a developed system we regularly run into a failure and again start wondering about the quality. When will the next fault occur? Similarly we might wonder about the up–time of the system. Faults need to be repaired and if faults are still present the full functionality can not be exploited.

In literature we find different names and different definitions for all this. In general we will use the notions quality and reliability interchangeably. For specific aspects, as outlined above, we will talk about quality assurance, reliability engineering and risk assessment in full knowledge that a lack of clear standards will still give rise to much confusion: inside this text and foremost in comparison with other texts. For these different phases in system evaluation a number of much publicized reliability models will be presented. Each will provide a mathematical formulation of reliability but in a different setting. In a later chapter we then continue to apply the here developed techniques in a general frame–work to estimate the risk of running potential faulty software [16].

## 4.1. Assurance

The first question on reliability is on how far the system has been developed. Initially the number of latent faults is unknown and must be found in order to quantify the system quality. From the known quantity of detected faults, the number of latent faults may be predicted. This is obviously a simplified rendering of the problem. Faults may be clustered and the distribution of clusters may be corrupted by the clustering of behavioral expressions into functions, procedures or other means of soft encapsulation.

Two different strategies are treated here: **seeding** and **probing**. Seeding is based on actively inserting new faults to find existing ones. In this light it looks like throwing marbles into the room to find previously lost ones. Probing assumes the potential faults and provides elaborate means to find them, if present. Both have application in software and hardware technology, and can therefore be viewed as general–purpose techniques.

### 4.1.1. Seeding

In this model a known number of faults are seeded into the program that is assumed to have an unknown number of indigenous (naturally occurring) errors. During testing the number of detected (seeded and indigenous) faults are counted. From this count an estimate of the fault content of the program prior to seeding is obtained and used to assess software reliability and other relevant measures. It assumes that seeded and indigenous faults are randomly distributed in the software and have equal probabilities of being detected. The following example will demonstrate this method.

Lets assume that 100 faults are seeded into a program containing an unknown number of inherent faults. After a period of debugging, 20 seeded and 10 inherent faults have been discovered. As the discovered seeded faults amount to 20 percent of the total, we may assume that the same is true for the inherent ones. Therefore, 20 percent of the inherent faults are discovered; so the total of inherent faults should be around 50 according this method. Examples: Mills Seeding [33], eventually with Lipow [30] or Basin [4] modifications.

In order to be successful seeded faults must be distributed over sensitive points in the code. To establish these "nerve centers" we need to introduce the concepts of controllability and observability [7]. A place in the program is **controllable** if it can be steered from the program input; it is **observable** if it can be viewed on the program output. Both are not hard facts but degrees and the enumeration of both can be used to establish the testability. As an example we will treat here the test scoring technique in hardware.

The underlying idea of testscoring is, that the testability of a circuit node can be quantified on the basis of quantifiers for its controllability and observability. As ***Controllability*** (CY) concerns itself with the ease, by which the current output state of a gate is determined from its inputs, a controllability transfer factor (CTF) can be proposed, which enumerates the different ways N(1) and N(0) to produce respectively a '1'and a '0' at the output, as:

$$CTF = 1 - \frac{|N(0) - N(1)|}{N(0) + N(1)}$$

For example for a 3–input NOR–gate, 8 different input signal combinations are possible, of which only 1 leads to an output value of '1'. Hence the CTF is : 1–6/8 and thus .25. Propagating controllability through a series of gates then repeats this calculation recursively. Arbitrarily it is assumed, that the controllability of the inputs is the average controllability ACY of the steering output nodes. Then it follows, that

$$CY = CTF * ACY$$

In a similar way, **observability** can be enumerated. Observability (OY) quantifies the ease, by which a node can be observed from a circuit output. There are N(SP:IO) different ways to set a sensitive path, i.e. a path from input to output such that the output value is solely dependent on the value of the input. The signals set to create a sensitive path are called the supporting signals. Vice versa there are N(IP:IO) ways to set an insensitive path, i.e. the output is not dependent on the value of the input signal. Then the observability transfer factor (OTF) is given by

$$OTF = \frac{N(SP:IO)}{N(SP:IO) + N(IP:IO)}$$

For example for a 3–input NOR–gate, a path from input to output is sensitized by one combination of the supporting signals and insensitized by three. Hence the OTF is .25. Propagating observability through a series of gates then repeats this calculation recursively, but for a single sensitive path only. Of importance is again the average controllability ACY1 but this time only of the supporting signals. Per gate leads this to

$$OY = OTF * OY(sensitiveinput) * ACY1$$

For all the sensitive paths though one gate, only the lowest OY value has significance. Testability can now be quantified on a per output node basis, as the product of observability and controllability. As boundary conditions serve the input and output signal of the circuit as a whole. Circuit inputs are optimally controllable, while circuit outputs are optimally observable. From the calculated values one can calculate, at which node a testpoint can be inserted to improve the Fig. 9.

In practice, testscoring has not been overwhelming successful. Though it aids the backtracking process, the benefits are not that large to counteract the connected overhead in computing time. Therefore one often suffices with *levelizing*, where the gates are placed in an order that if evaluated sequentially all predecessors of a gate are evaluated before the gate itself is evaluated. In Fig. 9, the gates are already depicted in such an order. The strong relation between decrease/increase of the testscoring values with the gate order is apparent.

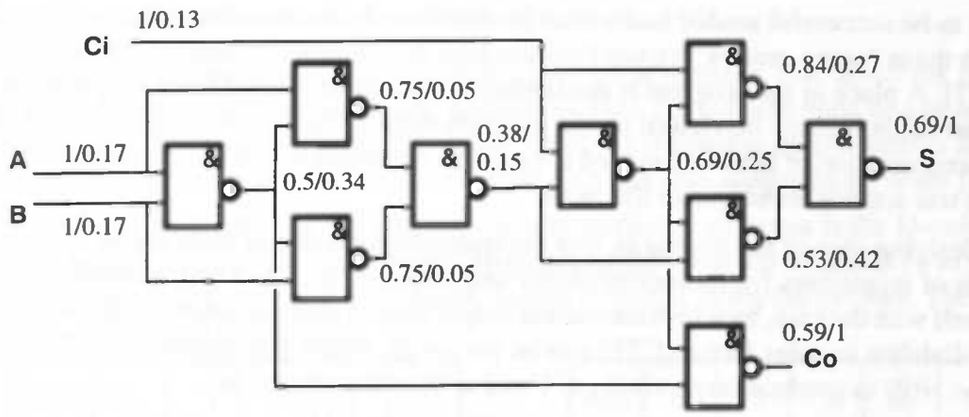



*Fig. 9: Controllability and observability figures for a Full–Adder circuit.*

### 4.1.2. Probing

A representative set of test cases from an input distribution is created and partitioned into a set of equivalence classes. Each class is associated with a logical execution path, an object or a module. An estimate of program reliability is based on the number of failures observed during the application of the test cases sampled from the input domain. Also an estimate can be made of the fault–prone module, objects or execution path. It assumes that the input distribution is known, and that this distribution can be partitioned into equivalent classes. Examples: Nelson [40], Ramamoorthy [42] and Bastani [5], Fuzzy [6]. There are also models that help to find the reliability score after the system is already operational. This doesn't mean, you can't use it before the system is in operation. These models are derived from [16].

To elaborate the principle of probing, we dwell some more on the **Nelson model**. A computer program p is defined as a specification of a computable function F on the set E, where $E = (E_i : i = 1..N)$ is the collection of all input data values, that are needed to make the program run. The set E defines all possible computations that the program p can produce. The execution of p produces, for each input $E_i$, the function value $F(E_i)$. Due to imperfections in the implementation of p, p specifies the function F' and not F. For some $E_i$, the actual execution output $F'(E_i)$ is within an acceptable tolerance $\Delta_i$ from $F(E_i)$. Or in formula:

$$|F'(E_i) - F(E_i)| \leq \Delta_i.$$

For all other $E_i$, (a subset $E_e$ of E) the execution of p does not produce an acceptable output:
– The execution output $F'(E_i)$ is not within the acceptable tolerance $\Delta_i$. Or: $|F'(E_i) - F(E_i)| > \Delta_i$.
– The execution of p terminates prematurely.
– The execution of p fails to terminate.
The number N of input data $E_i$ is very large, but finite because only a finite number of different values fit in a fixed–sized computer word.

The process of presenting $E_i$ to p and executing p to produce the output $F'(E_i)$ or an execution failure is called a run of p. On the assumption that every $E_i$ has the same chance of occurrence, the probability P that a run of p will result in an execution failure is therefore equal to the probability that the input $E_i$ used in the run will be chosen from the subset $E_e$.

$$P = \frac{\#E_e}{N} \qquad (1a)$$

And thus $$R = 1 - P \qquad (1b)$$
is the probability that a run of p will produce an acceptable output.

When this assumption isn't taken, then there exists a probability distribution $p_i$ according to some operational requirement. The set of $p_i$ is called the **operational profile** of E. P may then be expressed in the terms of $p_i$ by defining an execution variable $y_i$, which is assigned the value 0 if a run with $E_i$ computes an acceptable output and the value 1 otherwise. Then

$$P = \sum_{i=1}^{N} (p_i \times y_i)$$

is the chance that a run of p with input $E_i$ chosen according to the probability distribution $p_i$ will result in an execution failure. And

$$R = 1 - P = \sum_{i=1}^{N} (p_i \times (1 - y_i))$$

And thus the probability of no execution failures in n runs with each of the inputs selected independently according to $p_i$ is:

$$R(n) = R^n = (1 - P)^n$$

In operational use, the inputs for n runs are not selected independently, but in a definite sequence. The operational profile $p_i$ has to be redefined to $p_j$, which is the probability that $E_i$ is chosen as input to the $j^{th}$ run in a sequence of runs. Then $P_j = \sum_{i=1}^{N} (p_j \times y_i)$ *and* $R(n) = \prod_{i=1}^{N} (1 - P_j)$.

Written in exponential form: $R(n) = e^{\sum_{j=1}^{N} \ln(1 - P_j)}$

Some rules: *If* $P_j = P$, *then* $R(n) = e^{-P \times n}$ *or* *If* $P_j << 1$, *then* $R(n) = e^{-\sum_{i=1}^{N} P_j}$

R(n) may be expressed in terms of execution time t by making the following substitutions:

$\Delta t_j$ denotes the execution time for run j

$t_j = \sum_{i=1}^{N} \Delta t_i$, the cumulative execution time through i=1 run j.

$$R(n) = \exp\left(- \sum_{j=1}^{n} \Delta t_j \times \frac{-\ln(1 - P_j)}{\Delta t_j}\right)$$

If $\Delta t_j$ is considered to approach zero as n becomes large, the sum in the exponential becomes an integral, producing the formula:

$$R(t) = e^{-\int_0^t h(u).du} \qquad \text{where} \quad h(t_j) = -\ln\frac{(1 - P_j)}{\Delta t_j}$$

In case $P_j << 1$, $h(t_j)$ may be interpreted as the hazard function, which when multiplied by $\Delta t_j$ is the conditional failure probability during the interval $(t_j, t + \Delta t_j)$ given no failure prior to $t_j$.

**Measurement of Software Reliability with the Nelson model**. In order to carry out a measurement of R, the operational profile $p_i$ must be determined. This is done by dividing the ranges of the input variables into subranges and assigning probabilities that an input will be chosen from each subrange. When the n runs of the measurement are made, it's probable that a failure will occur. The measurement process should not be stopped and the error corrected, but the entire sample of n runs should be done. From this you can calculate the reliability according to formula 1a and b.

### 4.1.3. Testpattern assembly

In order to get from the assumed set of faults to the set of patterns, that detect the presence of these faults, the following steps must be performed. First the patterns must be assembled, then the patterns must be verified. In the following, we give a method to derive testsets in hardware engineering and assume that with some modification this also applies to software [8].

**Testpattern generation.** The old and popular way to find testpatterns is the D–calculus. It is based on an extension of the conventional Boolean algebra with the values D (1 in the correct circuit; 0 in the faulty one) and /D (0 in the correct circuit; 1 in the faulty one). The algebra is defined as illustrated in Fig. 10.

| OR | 0 | 1 | x | D | /D |
|---|---|---|---|---|---|
| 0 | 0 | 1 | x | D | /D |
| 1 | 1 | 1 | 1 | 1 | 1 |
| x | x | 1 | x | x | x |
| D | D | 1 | x | D | 1 |
| /D | /D | 1 | x | 1 | /D |

| AND | 0 | 1 | x | D | /D |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x | D | /D |
| x | 0 | x | x | x | x |
| D | 0 | D | x | D | 0 |
| /D | 0 | /D | x | 0 | /D |

| NOT | 0 | 1 | x | D | /D |
|---|---|---|---|---|---|
| | 1 | 0 | x | /D | D |

*Fig. 10: The D–calculus.*

Based on the D–calculus, the testpattern for an assumed fault can be determined as follows:

Step 1 [Fault Insertion]
A stuck–at fault is imitated by giving the wire the logic value D if 1 in the fault–free circuit or /D if 0 in the fault–free circuit.

Step 2 [Fault Propagation]
The influence of the inserted fault is driven towards the circuit outputs.

Step 3 [Fault Implication]
The setting of the network obtained sofar is asserted by driving these values back to the circuit inputs.

In Fig. 11 a typical example is given. Assume a stuck–at–1 must be detected at the lower input of NAND gate G5. In order to propagate this fault to the output of this gate, the other inputs must be 'high', while the input to be tested must be low. In other words, the /D at the input must be driven to the output and will appear there as a /D value. This process is repeated for all subsequent gates, until the D–value appears at a circuit output. Then the settings must be asserted, and this is where the trouble really starts. There are many ways in which a reasonable setting can be achieved, but some may be conflicting. This involves an intelligent search process, and the major algorithmic variations on the D–algorithm (FAN, PODEM) essentially attack this problem.

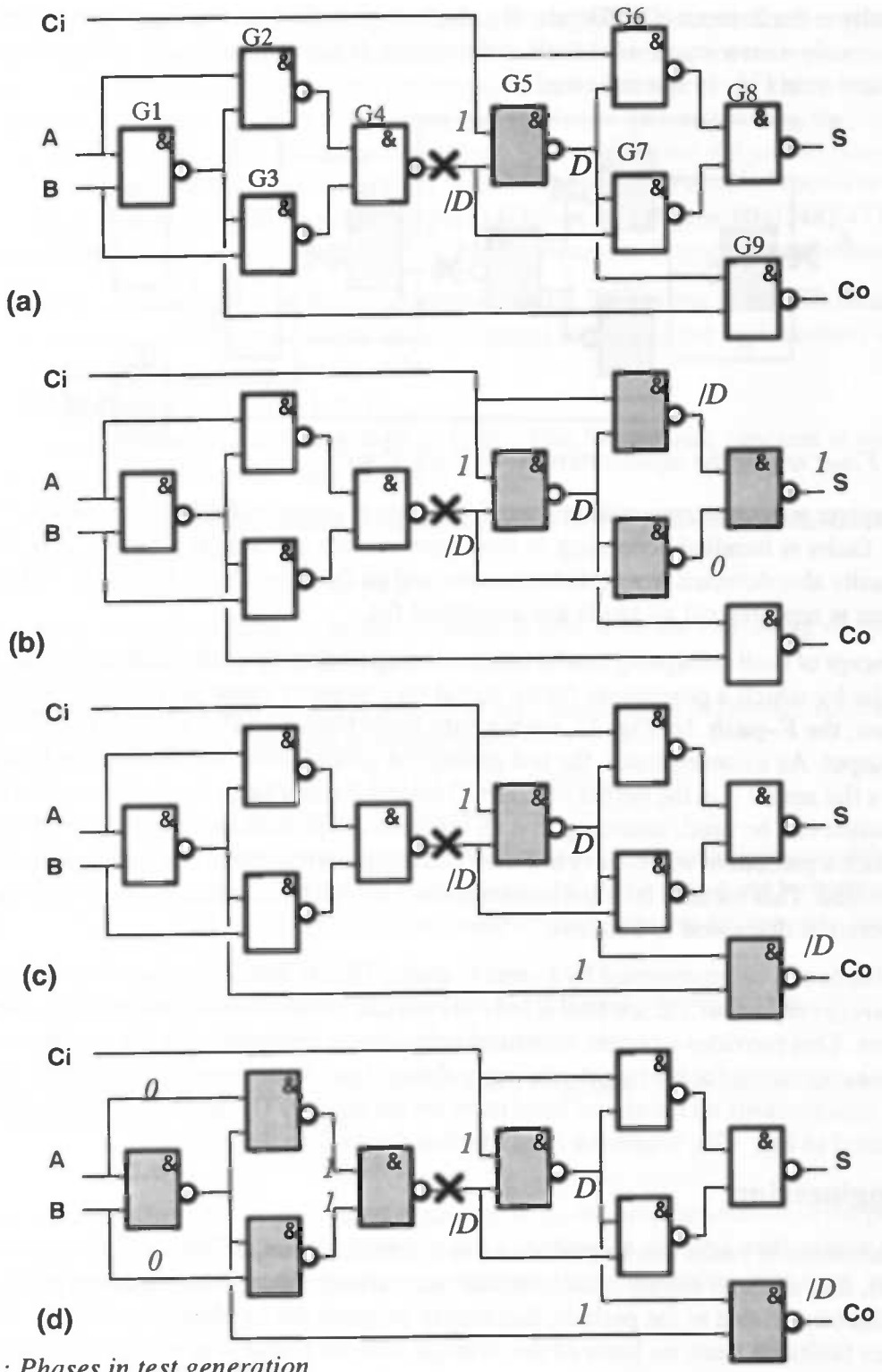



*Fig. 11: Phases in test generation.*
(a) Fault Insertion at gate G2; (b) Attempt to drive fault to S output; (c) Idem for C output; (d) Fault Implication

**Fault collapse.** The above procedure establishes a testpattern for a single fault at a time. Assuming, that always a single fault will appear, it is clear that a number of single faults could be found by the same testpattern. Therefore it is of advantage to check on the coverage of other faults, when a new testpattern is found. This way the fault set decreases more rapidly and the entire process of fault generation becomes more efficient. A simple example of such a fault collapse

opportunity is the 2–input NAND gate. If a stuck–at–1 fault at an input can be determined, then simultaneously also a stuck–at–0 fault at the output is found. In Fig. 12, all faults covered by the testpattern in Fig. 11 are indicated.

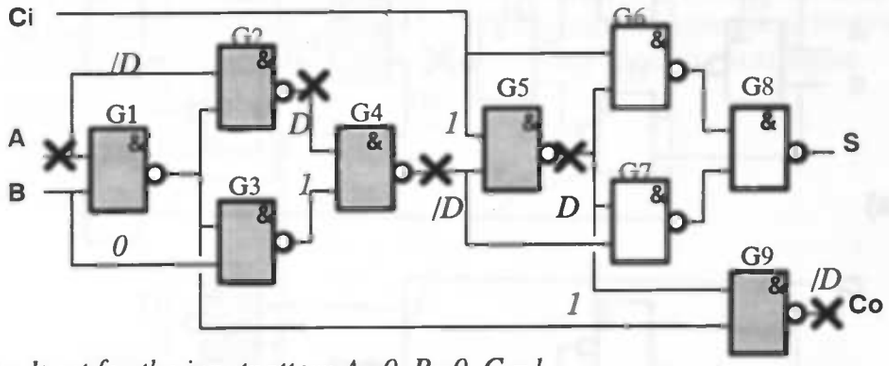



*Fig. 12: Fault set for the inputpattern A=0, B=0, $C_i$=1.*

The complete testpattern generation starts from a set of potential faults to be detected. Then one of these faults is handled according to the above outline procedure. Ensuing it is determined which faults also detected from this testpattern and all these faults are deleted from the set. This procedure is repeated till all faults are accounted for.

This concept of fault collapsing can be taken one step further by noting that apparently inputpatterns exist by which a potentially faulty signal on a primary input port will be driven through the circuit, the **F–path**. In Fig. 12, such a path leads from signal A via G2, G4, G5 and G9 to the C–output. As a consequence, the test generation process does not necessarily have to be applied to a flat netlist (i.e. the netlist wherein all instances have been merged), but also the hierarchical netlist can be used, assuming that all instances have been represented by their I– and F–paths. Such a procedure will clearly be faster than if operating on the flat netlist, as less data have to be handled. This method is called ***macro testing***, which by its name contrast to the gate–level test generation discussed in this text.

Not all faults can be represented by I– and F–paths. This is basically caused by the appearance of ***reconvergent fanout***, i.e. a signal is led over several parallel paths through the network to the same gate. This provides a certain structural redundancy, that precludes the generation of a test, as the test conditions for the target gate (once driven back to the source signal) may lead to conflicting requirements on the signal level to be set for testing. The adder is a notorious example, as appeared in Fig. 12b, where we tried to drive the fault to the S–output.

## 4.2. Engineering

The assessment of faults can be performed in a number of ways. There are two closely related concepts, that allow to estimate the potential occurrence of faults during future execution. The first is directly related to the periods, that elapse between the localization of faults. The second estimates faults on basis on basis of the average number found in specific time intervals.

The time between 2 failures follows a distribution whose parameters depend on the number of faults remaining in the software. Observed times between fails provide the basis for estimates of the MTTF (Mean Time To Failure) and reliability. It assumes an independence of times between failures, an equal probability of exposure of the faults, faults are independent of each other, faults are removed perfectly after each occurrence. Examples: Jelinski and Moranda De–Eutrophication [26][38], Schick and Wolverton [46], Goel and Okumoto Imperfect Debugging [17][18], Littlewood–Verall Bayesian [31][32].

The number of failures are recorded in specified time intervals. These numbers can be accumulated. The failure counts are assumed to follow a known stochastic process with a time dependant discrete or continuous failure–rate. Observed failure counts provide also a basis for estimates of the MTTF and reliability. It assumes that the faults detected during the intervals are independent, and the intervals are also independent. Examples: Goel–Okumoto Non–homogeneous Poisson Process [19], Goel Generalized Non–homogeneous Poisson Process or S–shaped Poisson Process [15][14], Musa Execution Time [34], Shooman Exponential [48][47], Generalized Poisson [2], IBM Binomial Poisson [10], Musa–Okumoto Logarithmic Poisson [37].

**Jelinski and Moranda (JM) De–Eutrophication model**. This is one of earliest time to failure models. It assumes a failure–rate function proportional to the current fault content of the program. This function is given by the formula:

$$\lambda(t_i) = \alpha \times (N - i + 1)$$

where $\alpha$ is a proportionality constant (0.01 to 0.02). This failure–rate function is constant between failures and decreases in steps of size $\alpha$.

**Schick and Wolverton (SW) model**. This is a variation on the JM Model with the failure–rate function modified as follows:

$$\lambda(t_i) = \alpha \times (N - i + 1) \times t_i$$

In some papers $t_i$ has been taken to be the cumulative time from the beginning of testing. This interpretation is inconsistent with the original paper.

**Littlewood–Verall Bayesian model**. Littlewood and Verall use a parameter function for the failure–rate. The parameter function is as follows:

$$f(\lambda_i, \psi(i)) = \frac{(\psi(i))^{\alpha} \times \left[\lambda_i^{\alpha - 1} \times \exp^{-\psi(i)\lambda_i}\right]}{\Gamma\alpha}$$

where $\alpha$ is a fit–constant, and the $\psi(i)$ a value indicating the skill of programmer and the difficulty of the programming task. The range of these values should be found out by testing, because I couldn't find them. This result–value is then used in the following failure–rate function:

$$f(t_i, \lambda_i) = \lambda_i \exp^{-\lambda_i t_i}$$

They claim that with this model failure phenomena in different environments can be explained by taking different values of $\psi(i)$.

**Shooman Exponential model**. This model is also a modification on the JM Model. Here the following failure–rate function is used:

$$\lambda(t) = \alpha \times \left[\frac{N}{I} - n_c(\tau)\right]$$

where t is the operating time of the system measured from its initial activation, $\alpha$ the proportionally constant, N the total number of failures found, I the total number of instructions in the program, and $n_c(\tau)$ the total number of failures corrected during the debugging time, normalized to I.

### 4.2.1. The S–shaped model

The failure process is modeled by Goel and Okumoto as a non–homogeneous Poisson process

$$P\{N(t) = n\} = \frac{H(t)^n \times e^{-H(t)}}{n!} \qquad n = 1, 2, 3, \ldots$$

with as the mean–value H(t) function and the failure intensity function d(t):

$$H(t) = a \times (1 - e^{-bt}) \;,\; d(t) = \left(\frac{dH(t)}{dt}\right) = a \times b \times e^{-bt}, \quad a, b > 0$$

H(t) is the expected number of failures observed at time t, a is the total number of failures initial in the software and b is the fault detection rate per fault.

**The S–shaped model** is a modified Goel–Okumoto model. When I started thinking of the reliability of software, and was thinking about a figure comparing the time spend finding the errors and the cumulative number of errors found, I was thinking about the next figure:

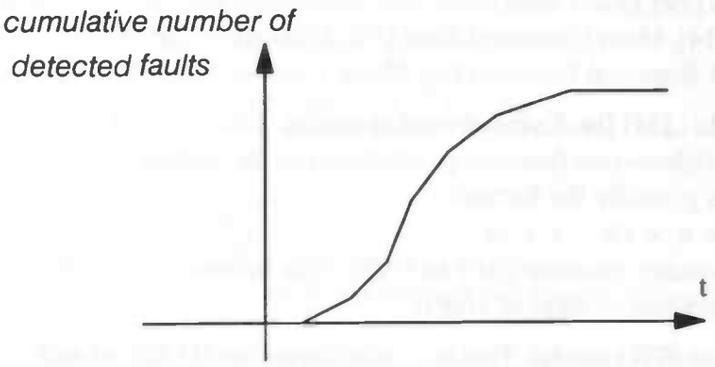



*Fig. 13: The S–shaped fault distribution.*

Of course this first attempt has a lot of assumptions: (a) all failures are independent, (b) some errors are never found, (c) no changes are made to the program except repairing errors.

Because the S–shaped model primarily states the S–shape in the model, I also expected that this model would be a nice test to investigate if I was right or wrong on my first idea. Second first ideas are most of the time the best ideas.

There are 2 versions. One with infliction and one without infliction. This is being documented in [51]. Let collection CP be a counting process representing the cumulative number of errors detected up to testing time t. Thus:

$$CP = \{N(t), t > 0\}$$

Define the statistically expected value of N(T), called a mean value function, as H(T). Then the S– shaped model can be formulated, based on the Goel– Okumoto non–homogeneous Poisson process [19], as:

$$Prob\{N(t) = n\} = \frac{e^{-H(t)} \times H(t)^n}{n!}, \quad n = 0, 1, 2, 3, \ldots$$

The mean value function H(t) is usually assumed to be a nondecreasing function with the boundary conditions: H(0) = 0 and H(∞) = a, where a is the initially total number of errors in the software.

The mean value function H(t) without infliction is defined as:

$$H(t) = a \times (1 - (1 + b \times t) \times e^{-b \times t}), \quad b > 0 \tag{2}$$

In this formula is b the discovery ratio.

The mean value function H(t) for infliction is $H_i(t)$:

$$H_i(t) = \frac{a \times (1 - e^{-b \times t})}{1 + c \times e^{-b \times t}}, \quad with\ b > 0, c > 0 \tag{3}$$

In this formula is b the discovery ratio, and c is the infliction factor. The failure intensity function is defined as:

$$\lambda(t) = (\frac{dH(t)}{dt}) \times \frac{1}{(a - H(t))}$$

Then the error detection rate per error for each stochastic S–shaped model without infliction is:

$$\lambda(t) = \frac{b^2 \times t}{1 + b \times t}, \quad \textit{delayed S–shaped}$$

And the error detection rate per error with infliction is:

$$\lambda_i(t) = \frac{b}{1 + c \times e^{-b \times t}}, \quad \textit{inflection S–shaped}$$

The following 3 definitions characterize an error detection process, or a reliability growth process in terms of the failure intensity functions.

1. If λ(t) is increasing in t, then H(t) is an increasing error detection rate function (IEDR).
2. If λ(t) is decreasing in t, then H(t) is a decreasing error detection rate function (DEDR).
3. if λ(t) is constant in t (t >= 0), then H(t) is a constant error detection rate function (CEDR).

It can be shown that the error detection rate per error of (2) and (3) are both IEDR. This means that the detectability of an error increases with the progress of software testing.

### 4.2.2. The Musa model

The basic execution model according to Musa assumes the failure intensity function for execution time as:

$$\lambda(\tau) = \lambda_0 \times e^{\frac{-\lambda_0 \times \tau}{\nu_0}}$$

with $\nu_0$: total number of failures
$\tau$: execution time
$\lambda_0$: initial failure intensity

This leads to the formulation of the failure intensity function for the number of failures as:

$$\lambda(\mu) = \lambda_0 \times (1 - \frac{\mu}{\nu_0}) \quad \text{whereby } \mu(\tau) = \nu_0 \times (1 - e^{\frac{-\lambda_0 \times \tau}{\nu_0}})$$

From this formulation we can now derive for the additional number of failures to be found to reach the failure intensity objective:

$$\Delta\mu = \frac{\nu_0}{\lambda_0} \times [\lambda_p - \lambda_f]$$

with $\lambda_p$: present failure intensity
$\lambda_f$: wanted failure intensity

The additionally needed execution time to reach the failure intensity objective is

$$\Delta\tau = \frac{\nu_0}{\lambda_0} \ln(\frac{\lambda_p}{\lambda_f})$$

This brings us finally to the formulation for the reliability:

$$R(\frac{\tau'}{\tau}) = e^{(-\nu_0 \times e^{\frac{-\lambda_0 \times \tau}{\nu_0}}) \times (1 - e^{\frac{-\lambda_0 \times \tau'}{\nu_0}})}$$

with $\tau$: execution time
$\tau'$: execution time measured from the present

**Musa–Okumoto logarithmic Poisson model.** Here the failure intensity function for execution time is:

$$\lambda(\tau) = \frac{\lambda_0}{\lambda_0 \times \tau \times \theta + 1}$$

with θ: failure intensity decay parameter (for instance 0.02)

Failure intensity function for the number of failures

$$\lambda(\mu) = \lambda_0 \times e^{-\theta\mu}$$

Number of failures expected versus execution time

$$\mu(\tau) = \frac{1}{\theta}\ln(\lambda_0 \times \tau \times \theta + 1)$$

$$\nu_0 = \frac{1}{\theta}\ln(\lambda_0 \times \tau \times \theta + 1)$$

$$\nu_0 \times \theta = \ln(\lambda_0 \times \tau \times \theta + 1)$$

$$e^{\nu_0 \times \theta} = \lambda_0 \times \tau + 1$$

$$\tau = \frac{e^{\nu_0 \times \theta} - 1}{\theta \times \lambda_0}$$

Additional number of failures to be found to reach the failure intensity objective

$$\Delta\mu = \frac{1}{\theta}\ln\frac{\lambda_p}{\lambda_f}$$

Additional execution time needed to reach the failure intensity objective

$$\Delta\tau = \frac{1}{\theta} \times \left(\frac{1}{\lambda_f} - \frac{1}{\lambda_p}\right)$$

$$MTTF : \theta(\tau) = \frac{\theta}{1-\theta} \times \left(\lambda_0 \times \tau \times \theta + 1\right)^{1-\frac{1}{\theta}}$$

$$Reliability : R(\tau'/\tau) = \left(\frac{\lambda_0 \times \tau \times \theta + 1}{\lambda_0 \times \theta \times (\tau' + \tau) + 1}\right)^{\frac{1}{\theta}}$$

**Ramamoorthy and Bastani model**. The authors of this model are concerned with the reliability of critical, real–time, process control programs. In such systems no failures should be detected during a reliability estimation phase, so that the reliability estimate is one. The model provides a best guess of the conditional probability that the program is correct for all possible inputs, given it is correct for a specified set of inputs.

P{program is correct for all points in [t, t + V] | it is correct for test cases $x_j$ j=1..n} =

$$\exp^{-\lambda V} \times \prod_{j=1}^{n-1} \frac{2}{1 + \exp^{-\lambda x_j}}$$

where λ is a parameter which implies the complexity of the code.

## 4.3. Assessment

We will refrain here from socio–/economical considerations and assume that risk is directly related to the (degree of) availability of the system. If a fault is found and not repaired, something may go wrong that has an impact on the outside world. Hence partly based on the previous theoretical models, we will derive a formulation for the system availability.

We assume that the system is constructed from components, that may fail and get stuck independent of one another. This can be modeled as a component that begins in the operational state and will finally fail, and go over in the failed state. As the component can't be repaired, it will automatically go over into the broke state.

A transition diagram of a (non)–repairable component is:

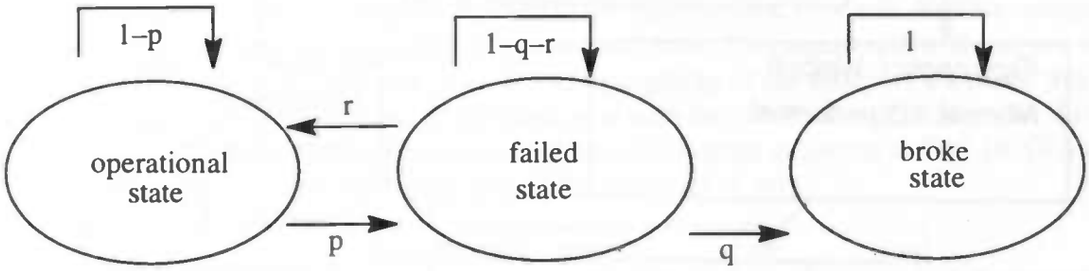



*Fig. 14: Transition diagram of a component.*

p : chance that the component fails
q : chance that the component can't be repaired
r : change that the component can be repaired

It is impossible to predict the component's operational time, since we consider this as a random variable. That's why we consider this component as a sample from a large process, that consists of many of these components, with each own failure time. The next table shows a population of 315.982 people who are still alive at the age of 75. As we have a set of components with a random failure status, the mean time between failure can be quantified through a survival estimate. Lets assume a survival distribution (or reliability function) R(t) = L(t) / N, whereby
– R(t) is the chance to have survived upto time t;
– L(t) is the number of components still alive at time t;
– N is the initial number of components

| t | L(t) | R(t) | F(t) | n(t+Δ)–n(t) | f(t) | r(t) |
|---|---|---|---|---|---|---|
| 0 | 315.982 | 1.0000 | 0.0000 | 134.217 | 0.0850 | 0.0850 |
| 5 | 181.765 | 0.575 | 0.425 | 103.554 | 0.0655 | 0.1139 |
| 10 | 78.221 | 0.248 | 0.752 | 56.634 | 0.0358 | 0.1444 |
| 15 | 21.577 | 0.0683 | 0.9317 | 18.556 | 0.0118 | 0.1728 |
| 20 | 3.011 | 0.0095 | 0.9905 | 2886 | 0.0023 | 0.2421 |
| 24 | 125 | 0..0004 | 0.9996 | 125 | 0.0004 | 1.000 |

So age equals to 75+t. We now check what happens to this population of people, if we improve t. This gives us the table L(t). From this data we can calculate the reliability R(t), the failure chance F(t), the failure density f(t) and the failure–rate r(t). The input for this part are the failure data.

### 4.3.1. Mean Time To Failure

The function R(t) can be interpreted as the probability that the component experiences no failure during the time interval (0,t] given that the component is operational at time t=0. From the list of times on which components have started to fail, we can now derive the MTTF according to Fig. 15.

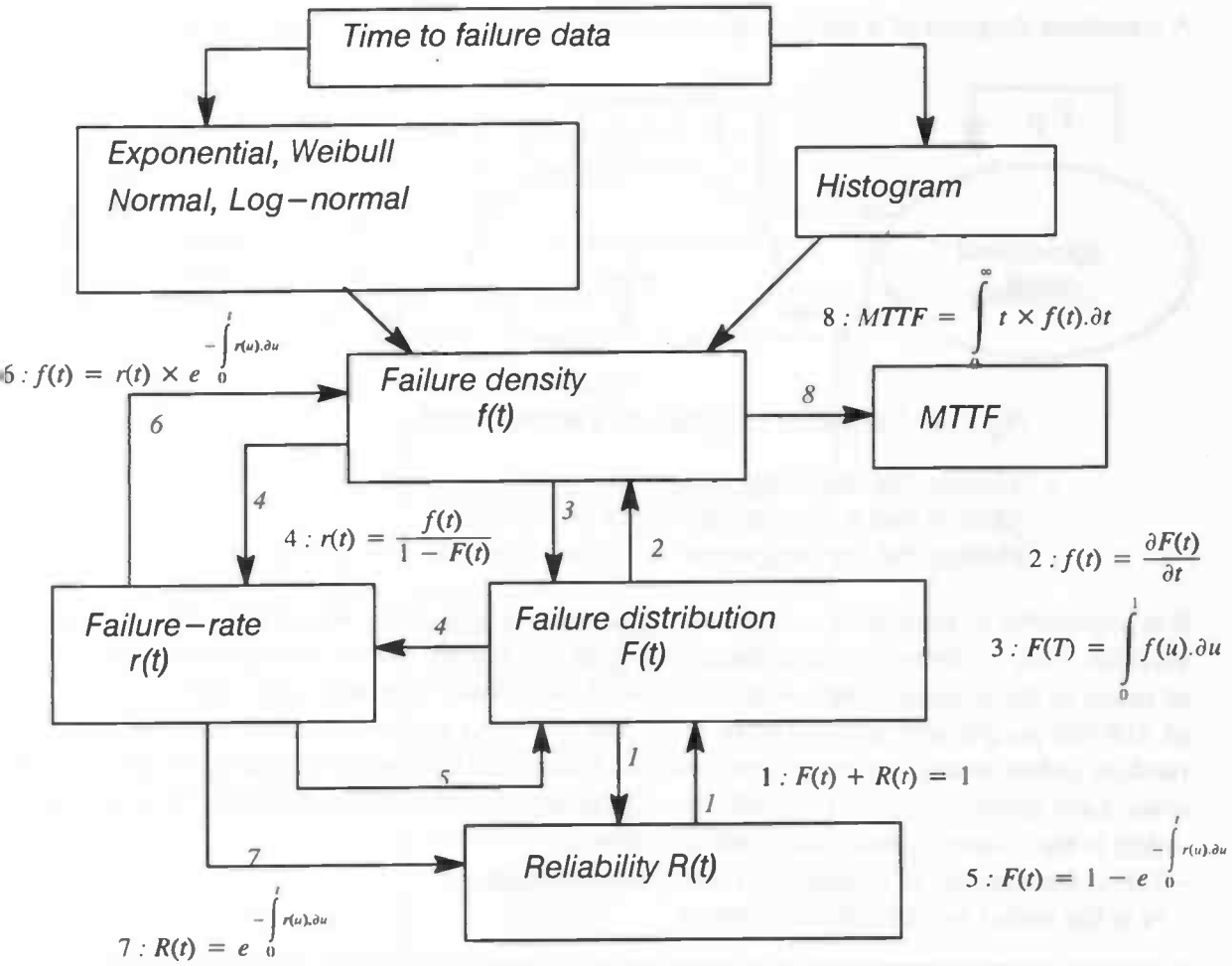



*Fig. 15: Computational scheme for the MTTF.*

*Some useful approximations of the formulas:*

$$f(t) \simeq \frac{L(t) - L(t + \Delta)}{N \times \Delta(t)} \qquad MTTF \simeq \sum_{t=0}^{N-1} (t + \frac{\Delta(t)}{2}) \times f(t) \times \Delta(t)$$

*further* $MTTF: \frac{1}{r(t)}$, *if failure rate constant for all t.*

This computation of the MTTF is based on a closely related set of functions:

- **Failure Distribution** F(t): the probability that the component experiences at least one failure during [0,t) given that it was operational at time t=0.
- **Failure Density** f(t): the first–order derivative of F(t).
- **Failure Rate** r(t): the probability that the component fails per unit time at time t given that the component was operational at time t=0.

The data we need for studying the whole process is the MTTF, because a repairable component experiences repetitions of repair–to–failure and failure–to–repair. Therefore we need to calculate the MTTR as well. In most cases MTTR << MTTF and can thus be neglected, but in some cases it isn't, and therefore you need to add it to your model. That's why the definitions in the models look sometimes not exactly as what is written in the standard literature. The component can get in operational state in 2 ways. It was already in the operational state in the previous phase and didn't experience a failure or it got repaired in the previous phase. The start of the process doesn't even assume that all components are in the operational phase.

## 4.3.2. Mean Time To Repair

Now the process where the component starts in the failed state at time t=0. And the component can be repaired so that it can go into the operational state. This means that in the figure of the whole process that: p>0, q>0 and r>=0. For the simplicity of the model, we assume that r=0. We'll introduce equal definitions of functions as in with the failure data. The input for this process are the repair data. From this we can do our calculations as shown in Fig. 16. G(t) is the equivalent of F(t) for the failure data, g(t) of f(t) and m(t) of r(t).

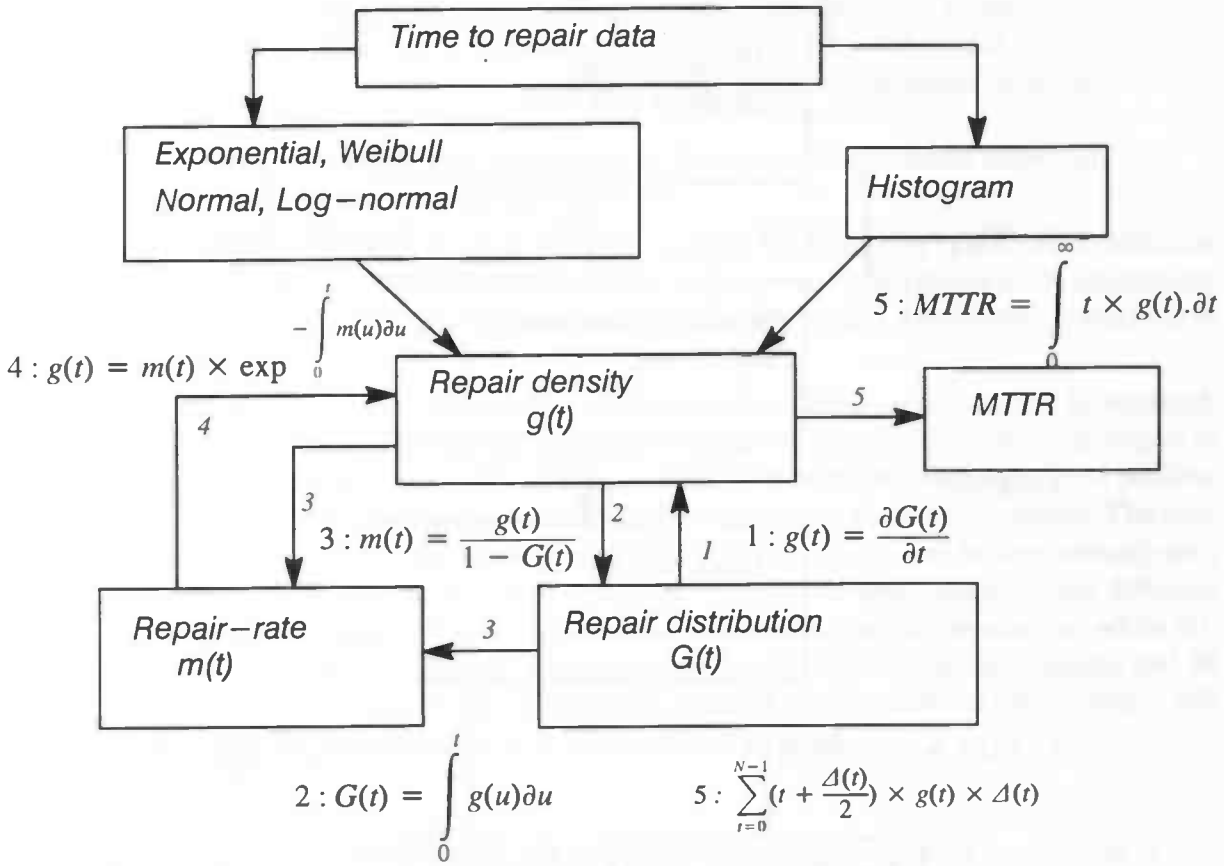



*Fig. 16: Computational scheme for the MTTR.*

*Some useful approximations of the formulas:*

$$G(0) = 0,\ G(\infty) = 1,\ G(t) = \frac{\textit{Number of completed repairs until time } t}{N}$$

$$g(t) = \frac{\partial G(t)}{\partial t} \simeq \frac{G(t + \Delta) - G(t)}{\Delta}$$

$$MTTR: \int_0^\infty t \times g(t)\, \partial t \simeq \sum_{t=0}^{N-1} (t + \frac{\Delta(t)}{2}) \times g(t) \times \Delta(t)$$

This computation of the MTTR is based on a closely related set of functions:

- **Repair Probability** G(t): the probability that the repair is performed before time t, given that the component failed at time t=0.
- **Repair Density** g(t): the first–order derivative of G(t).
- **Repair–rate** m(t): the probability that the component is repaired per unit time at time t, given that the component failed at time t=0 and remained so ever since.

### 4.3.3. Availability

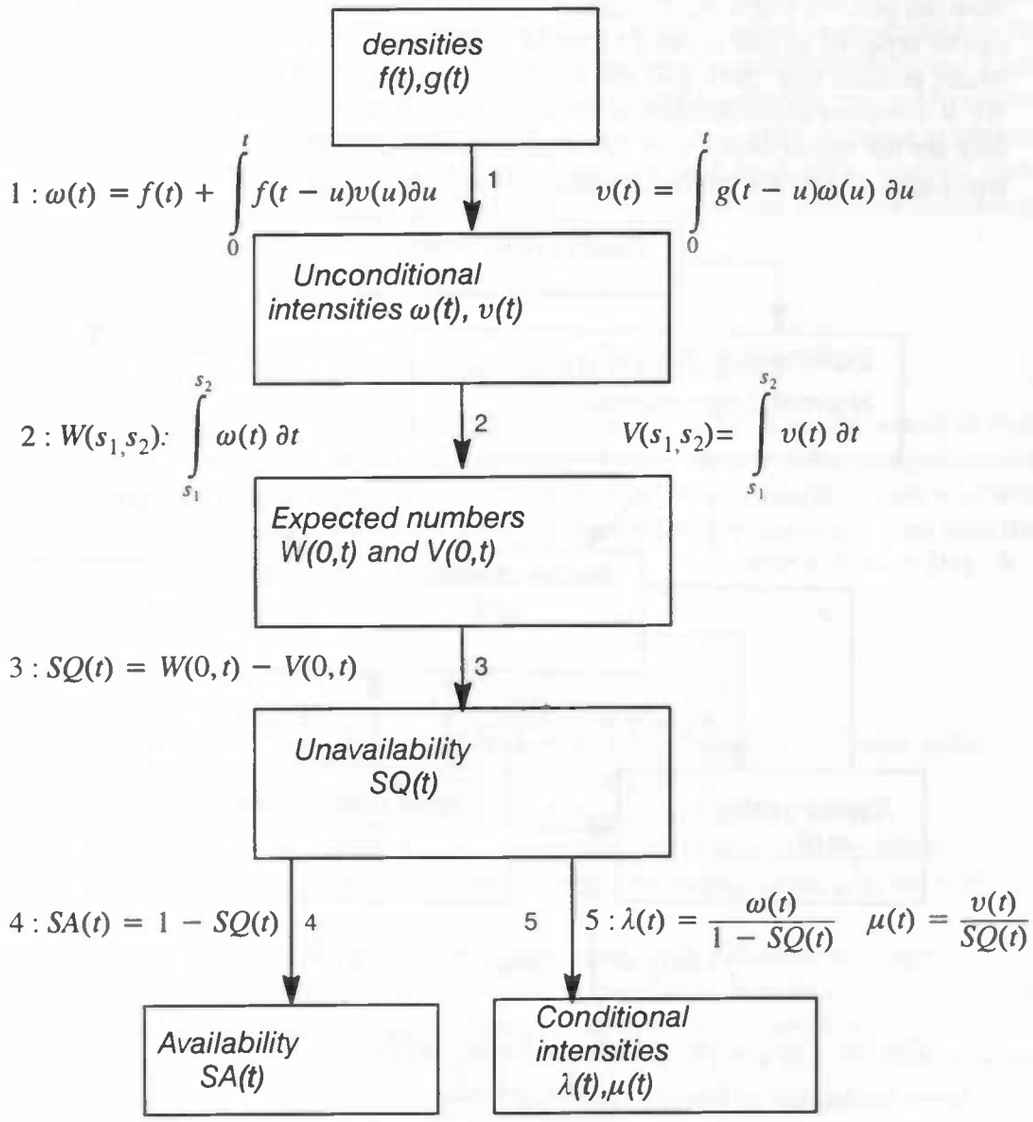



*Fig. 17: Computational scheme for the System Availability.*

*Some useful approximations of the formulas:*

$$v(t) \simeq \frac{\textit{Number of repairs at time } t}{N} \qquad \omega(t) \simeq \frac{\textit{Number of failures at time } t}{N}$$

$$W(s_1,s_2) \simeq \sum_{t=0}^{N-1} \omega(t) \times \Delta(t) \qquad V(s_1,s_2) \simeq \sum_{t=0}^{N-1} v(t) \times \Delta(t)$$

From the MTTF and MTTR we can attempt to quantify the availability of the system as depicted in Fig. 17. A different expression with the same purpose is Mean–Time–Between–Failure (MTBF) or Mean–Time–Between–Repair (MTBR).

Availability is not identical to Reliability, as we will discuss in a next paragraph.

This computation of MTBF = MTBR = MTTF + MTTR is based on a closely related set of functions:

- **System Availability** SA(t): probability that the system will be operational.
- **System Unavailability** SQ(t): probability that the system will not be operational.
- **Unconditional Repair Intensity** $\nu(t)$: probability that the process can be repaired per unit time at time t.
- **Unconditional Failure Intensity** $\omega(t)$: probability that the system fails per unit time at time t.
- **Failure Expectation** $W(s_1, s_2)$: Expected number of failures between $s_1$ and $s_2$.
- **Repair Expectation** $V(s_1, s_2)$: Expected number of repairs between $s_1$ and $s_2$.
- **Conditional failure intensity** $\lambda(t)$: proportion of the system components expected to fail per unit time at time t.
- **Conditional repair intensity** $\mu(t)$: proportion of the system components expected to be repaired per unit time at time t.

Reliability generally is different from availability, because the reliability requires the continuation of the normal state over the whole interval (0,t]. A component contributes to the availability SA(t), if it failed before time t, got repaired and is available at time t, but doesn't contribute to the reliability.

This means that: SA(t) >= R(t). The equality holds only when the component can't be repaired, or in the case of the figure: q=0. The Availability of a non–repairable component decreases to t=0, for t very large, but the Availability of a repairable component decreases to a fixed positive number. For the unavailability the inequality: SQ(t) <= F(t) holds, for similar reasons. The conditional failure intensity $\lambda(t)$ is the equivalent with r(t). The unconditional failure intensity $\omega(t)$ and the conditional $\lambda(t)$ refer to the failure per unit at time t. These quantities are different though, because they assume different populations. $\omega(t)$ takes the whole population, while $\lambda(t)$ only takes those who were functioning at the start of t. So if we have 100 components, and 30 are in the failed state, and 70 in the functioning state at time t, and from the 70 functioning 7 will fail in the time $(t,t+\Delta]$, then $\lambda(t)dt=7/70\ dt$ and $\omega(t)dt = 7/100\ dt$.

### 4.3.4. In summary

This chapter has attempted to structure the reliability research area. To find the amount of embedded faults, we can start from the number of known & located faults or from the potential fault coverage by the test input. Both approaches are feasible but imprecise and cumbersome. The reason is that software is not a mere list of statements but has an ever changing structure. One way to avoid this problem is by looking only at large lumps of code (say 1000 lines at a time) and hope that from this far away viewpoint the internal structure has become irrelevant.

From afar, we can now interpret the software production process as a group effort where not so much the internal structure as well as the impact of faults becomes visible. Major mistakes are more visible than small ones and moreover major mistakes tend to imply a lot of small ones. So we can assume a **reliability growth model** that displays how the process will improve over time. This model is still very arbitrary and for the moment serves only to allow for a mathematical formulation of the problem.

From an ever wider perspective, the software production process is merely a matter of survival. Faults are events that come into existence and live until they are repaired. We then assume a distribution of life times which seems to be valid for any production process. Assuming the system is complicated enough the assumption that this model is also applicable to software seems not too farfetched. Therefore our current bet will be on this management–oriented technique.

# 5. Discussion

In this final chapter we will attempt to wrap up the mass of details and hints from the previous chapters. Firstly we will select from all the reliability models treated before a simplified one for the simple reason that there is no evidence that a more elaborate model will function better. Then we discuss a first attempt to provide some guidance to the software production process by a simple management tool. We have refrained from a more complicated program because at present we need more evidence of what works and what not. Lastly we look back at the original topic of "Embedded Systems" and discuss how Hardware Test Engineering and Software Reliability Engineering can be blended together and extended to aid

*the creation of quality Embedded Systems in the shortest time.*

## 5.1. Analysis of the constant failure and repair–rate model

In the previous chapter we finally came to the conclusion that a survival model would fit best to our purposes at the current time. We can further simplify this model by the straightforward assumption that the rate of failure and repair is constant. To perform this simplification we make some additional assumptions about where we are in the development process and then treat the failure, repair and availability parts in sequence.

### 5.1.1. Constant failure–rate

The assumption of a constant failure–rate is valid only if:

– The component is not at the extreme ends of the learning curve.
  In other words, the code should already be compilable but still have some faults that there is a decent chance in meeting them.

– The component is stable in its construction, as for instance a solid–state device.
  In other words, the faults do not change the system structure significantly. Practice shows that structural changes place the endeavor back to the start of the learning curve.

– The component is a large one with many sub–components, with different rates/ages.
  In other words, the effect of internal structure can be removed by standing off.

– The data are so limited that elaborate mathematical treatments are unjustified.
  In other words, the conditions for playing a survival game should be adhered to.

The functions from Fig. 15 regarding the failure–rate r will simplify as follows:

$$F(t) = 1 - e^{-rt}$$

$$R(t) = e^{-rt}$$

$$f(t) = r \cdot e^{-rt}$$

$$MTTF = \int_0^\infty t \cdot r \cdot e^{-rt} \cdot \delta t = 1/r$$

*Fig. 18: The constant failure model.*

When the mean time to failure MTTF is known, we can find the failure distribution at time MTTF as:

$$F(t = MTTF) = 1 - e^{-1} \simeq 0.632$$

### 5.1.2. Constant repair–rate

The same assumptions of the failure–rate apply here too. The functions from Fig. 16 regarding the repair–rate s will simplify as follows:

$$G(t) = 1 - e^{-st}$$
$$g(t) = s \cdot e^{-st}$$
$$MTTR = \int_0^\infty t \cdot s \cdot e^{-st} \cdot \delta t = 1/s$$

*Fig. 19: The constant repair model.*

When the repair distribution is know, we can find the MTTR by solving the equation:

$$G(t) = 1 - e^{-1} \simeq 0.632$$

### 5.1.3. Constant repair–rate and failure–rate

The assumptions for the constant failure–rate r and repair–rate s must hold. Also it assumes that the program can't be operational while in repair. Then with the help of some Laplace transform [21], the functions of Fig. 17 will change as follows:

The unavailability function SQ(t) and availability function SA(t) of the system become:

$$SQ(t) = \frac{r}{r+s}(1 - e^{-(r+s)\cdot t}) \qquad SA(t) = \frac{s}{r+s} + ((\frac{r}{r+s}) \cdot e^{-(r+s)\cdot t})$$

Likewise we can express the conditional unavailability ω(t) and the conditional availability function ν(t):

$$\omega(t) = \frac{r \cdot s}{r+s} + ((\frac{r^2}{r+s})e^{-(r+s)\cdot t}) \qquad \upsilon(t) = \frac{r \cdot s}{r+s}(1 - e^{-(r+s)\cdot t})$$

The expected number of failures from 0 to t is W(0,t)

$$W(0,t) = \frac{(r \cdot s \cdot t)}{r+s} + ((\frac{r^2}{(r+s)^2})(1 - e^{-(r+s)\cdot t}))$$

and the expected number of repairs is V(0,t):

$$V(0,t) = \frac{(r \cdot s \cdot t)}{r+s} - ((\frac{r \cdot s}{(r+s)^2})(1 - e^{-(r+s)\cdot t}))$$

It is clear that the failure–rate function λ(t) = r ∀ t and the repair–rate function μ(t) = s ∀ t . This gives the following results:

$$MTBF = r + s$$
$$SQ(\infty) = \frac{r}{r+s}$$
$$SA(\infty) = \frac{s}{r+s}$$
$$\omega(\infty) = \nu(\infty) = \frac{r \cdot s}{r+s}$$

*Fig. 20: The constant failure and repair model.*

Suppose we have a repair–rate of 4 days, and a failure–rate of 20 days. How many errors can you expect in 200 days. Well, r is in this case 1/20 == 0.05 and s is in this case 1/4 == 0.25. The formula W(0,t) gives then:

$$W(0,200) = (0.05 * 0.25 * 200)/(0.05 + 0.25) + ((0.05 * 0.05)/(0.05 + 0.25)) * (1 - e^{-1*(0.05+0.25)*200})$$
$$= 8.33333 + 0.00833 * 1 = 8.341$$

System Availability will be: 0.25/(0.25+0.05) = 83%.
System Unavailability will be: 0.05/(0.25+0.05) = 17%.

## 5.2. The experiment

As last part of my graduation, I've made a program that can be used to manage and control a development–process. It uses Time to Failure Data and Time to Repair data to determine the Mean Time to Failure and Mean Time to Repair for the different phases in the development process. Also the Mean Time of Non–Repair is being calculated of the failures that didn't get repaired. Every error that gets found outside it's development–phase gets a penalty equal to the INFLICTION_FACTOR defined in 'failure.h', in this experiment 3. The inserted phase will be accounted for this failure. Also it is possible to administer different accounting–groups, with their own development–phases, at the same time.

### 5.2.1. Input–definition

The program uses it's own scanner, which is word–oriented. It uses the definition of the scan_table in 'init.c' to know where what is standing on a line. The word and line oriented scan can be shown according to the following stripped code of the function scan() in 'scan.c'.

```
fp = fopen( DATAFILE, "r");
while (!feof( fp )) {
   line_nr++;
   nr_word = 0;
   ...
   while (!feol( fp )) {
      word = fread_word(....);
      switch( scan_table[nr_word].name ) {
         case SCAN_ACC ...
         ..
      } /* switch */
      nr_word ++;
   } /* feol */
   fread_to_nl( fp );
   insert_date if accounting_group is found and no errors
} /* feof */
```

The function fread_word in 'scan.c' is made so that it doesn't read pass the end of line symbol '\r' or '\n'. If a premature eol symbol is being parsed, then the empty string will be returned to fread_word. The function feol has been written by me, because it doesn't exist in the language c in which this program has been written. The DATAFILE has been specified in the file 'failure.h' and can be changed there as well. In this experiment the DATAFILE has been called 'data.txt'. The DATAFILE is a comma separated file. This can be changed in 'scan.c' in the function fread_word where the ',' has to be changed in the new type of separation.

An example of correct input is dependent on the definition of the scan_table in 'init.c'. In this case it looks as follows:

23,"1995–1–23","1995–2–12","CLO",,"scan","larger","DEV1","DEV1"

The first variable is the error number, the second the date that the error got found, the third is the date that the error got repaired. The fourth is the status of this fault for the first accounting group, and the fifth for the second accounting group. The variables "scan" and "larger" explain in which module and function the error got found. Finally the first "DEV1" is the phase the error got found and the second is the phase the error got inserted.

The input for the file is being defined in the file 'init.c' and in the type scan_table, as mentioned before. The scan_table is an array of the struct scan_word and declared as follows:

const struct scan_word scan_table[MAX_SCAN_WORDS];

The scan_table explains how many words have to be scanned and what is on the position of the scanned word. The first scan_word in the scan_table is the first word on a line in the DATAFILE (data.txt). The second scan_word in the scan_table is the second word on a line in that file and so on. The declaration is as follows:

```
struct scan_word {
        int name;
        int group
};
```

The struct scan_word contains an int for the name because in C a switch on strings is impossible (See scan() in scan.c). Now the switch is being done on the following statement: scan_table[nr_word].name. If you have different ones of the same name, then you can use the second variable to identify them uniquely. At this moment this is only been used by SCAN_ACC. All the other types will ignore this second integer.

SCAN_ERROR_NUMBER: the error number. This type will be skipped. It's only for tracing errors back in the data–file.

SCAN_FOUND_DATE: the date on which the error got found. It has the syntax of "1971–6–29" and the year should be larger as the year 1900, because of the system–call mktime.

SCAN_REPAIR_DATE: the date on which the error got repaired. Same date syntax definition as above

SCAN_FOUND_DATE: If the date didn't get filled in, then I'll assume the error didn't get fixed and the time until now gets calculated and added to the non–repair time. It also means that errors must be searched up later to add the failure–time. Therefor I've added the function name and module name. This gives together with the fault–number a unique key to find the error.

SCAN_ACC: the accounting group for this error. This group is responsible for the error. The second integer of the struct scan_word (int group) matches with the second integer of the accounting_table struct account_name. This struct is declared as follows:

```
struct accounting_name {
        char *name;
        int num;
};
```

So if you a struct scan_word as follows: {SCAN_ACC, 3} and the list of the accounting_table looks as follows: { {"GROUP_A", 1}, {"GROUP_B",2},{"GROUP_C",3}} then the group identified on this position on the line by SCAN_ACC is GROUP_C. This has been done this way to have multiple reports of accounting_groups and also not to be dependent on the position on the line to identify the groups. The empty string as input will be considered as a group not responsible for this fault.

SCAN_MODULE_FOUND: indicates in which module the error got found. An empty string as input will result in skipping this error.

SCAN_FUNCTION_NAME: indicates what the function name of this error is. An empty string will result in skipping this error. Linenumbers are not used because of changes to the code also results in changes to linenumbers.

SCAN_PHASE_DETECTED: tells in which development–phase the fault has been found. An empty string will result in a value of 0 and the inserted phase will be used as the detected phase.

SCAN_PHASE_INSERTED: tells in which development–phase the fault has been inserted. An empty string will result in a value of 0 and the error will be skipped.

The accounting–groups are being defined in the accounting_table. In this case they are GROUP1 and GROUP2. The number behind the variable identifies this group unique and matches with the numbers of SCAN_ACC in the scan_table.

The possible status of the fault is being defined in the project_table. In this case they can be NUL and CLO. A NUL error is actually a nullified error and will be treated as an error that hasn't occurred. In effect an empty string.

The possible phases of an error are being defined in the phase_table. The value of 0 is being reserved of a phase that didn't get filled in.

The working days are being defined in an array with 7 elements named working_days. In case of 0, it indicates that the day is not a working day and a 1 otherwise. The count starts from Sunday, matching with the system–call mktime.

Holidays are being defined in the the type holidays. They indicate the day and month. In this case New Year celebration, Queen's celebration of her birthday and Xmas.

If these tables are being changed, then the variables MAX_... should be modified in the file 'failure.h' as well.

### 5.2.2. Creating a list of the input.

The input is being recorded in a failure_table. Every accounting group has an index into this table with the total number of failures and a list of dates in ascending order. This is being done by the function insert_date of the file 'scan.c'. One of these items of the list looks as follows (definition can be found in 'failure.h'):

```
struct failure_date {
   int date_found[3];        /* datum found */
   int date_repaired[3];     /* datum repaired */
   char *module_name;        /* module name */
   char *func_name;          /* function name */
```

```
    int phase_inserted;      /* phase error got inserted */
    int phase_detected;      /* phase error got detected */
    int delta_date           /* # working days till next fault*/
    RP delta_repair;         /* # working days it took to repair
                                the fault if it got repaired,or
                                the number of working days the
                                fault didn't get repaired till
                                now (maximum of 200 days).*/
    FAILURE_DATA *next;      /* next error */
}
```

The variables delta_date and delta_repair are being filled in by the function calc() in the module 'calc.c'. The rest is being done by scan() in the file 'scan.c'. This list itself is part of the failure_table. Together with this list the number of the failures is kept in the variable nr_failures.

### 5.2.3. The results

The results are shown by the function show_results() in output.c. In case you want it written to a file, use the UNIX method: 'progname >& outputfile'. This function produces a monthly score for every accounting–group. At the end of the last month availiable a total score for the group is being produced for all errors. The following output–variables are in use:

```
#F           -> Amount of failures.
#R           -> Amount of failures that got repaired.
#NonR        -> Amount of failures that didn't get repaired.
Ftime        -> Total time of days that all faults accumulate.
Rtime        -> Total time of days that all repairs accumulate.
NonRtime     -> Total time of days
                that faults didn't get repaired.
AvgFtime     -> Average workdays till next failure.
AvgRtime     -> Average workdays failures got repaired.
AvgNonRtime -> Average workdays failures didn't
                get repaired until now.
```

The results are being shown for every development phase. The selection is being done on the inserted phase, so if you are in phase 3 and have results in phase 1, then you're not happy :–). A short result–file has been included:

```
Failure data for the group: GROUP2
Accounting-group      GROUP2 has   50 failures.
>From [ 5- 1-1996] to [ 5- 2-1996]:
Phase:#F FTime #R Rtime #N Ntime AvgFtime AvgRtime AvgNonRtime
 DEV1 11   25   11   44   0    0     2.273    4.000     0.000
 DEV2  0    0    0    0   0    0     0.000    0.000     0.000
 DEV3  0    0    0    0   0    0     0.000    0.000     0.000
 DEV4  0    0    0    0   0    0     0.000    0.000     0.000
>From [ 5- 2-1996] to [ 5- 3-1996]:
Phase:#F FTime #R Rtime #N Ntime AvgFtime AvgRtime AvgNonRtime
 DEV1  3   12    3   21   0    0     4.000    7.000     0.000
 DEV2  5    9    5   32   0    0     1.800    6.400     0.000
 DEV3  0    0    0    0   0    0     0.000    0.000     0.000
```

```
 DEV4  0   0    0    0   0   0     0.000    0.000    0.000
...
(A lot more months being listed, and after the last month
shown:)

Overall result for this group:
Phase:#F FTime #R Rtime #N Ntime AvgFtime AvgRtime AvgNonRtime
 DEV1 16  39   16  104   0   0     2.438    6.500    0.000
 DEV2 17  58   16  105   1  32     3.412    6.562   32.000
 DEV3 10  24   10   39   0   0     2.400    3.900    0.000
 DEV4 7   13    6   26   1  41     1.857    4.333   41.000

A listing of all the modules and number of faults.
         Module    Function Amount
        MODULE1    FUNC2      2
                   FUNC3      6
                   FUNC1      5
        MODULE3    FUNC1      5
                   FUNC4      3
                   FUNC2      1
                   FUNC3      6
        MODULE2    FUNC3      5
                   FUNC1      3
                   FUNC4      1
                   FUNC2      4
        MODULE4    FUNC1      3
                   FUNC2      2
                   FUNC4      4
```

---

The file main.c calls all functions that needs to be handled in their order. It's not that shocking and I'll leave it here to my comments.

The file bug.c takes care of the output to the BUGFILE of all errors that got found with the scanner.

The file failure.h contains some global variables. I'll discuss them shortly:

INFLICTION_FACTOR
: A penalty factor that gets multiplied with the number of repair days if an error got produced out of phase.

MAX_HOLIDAYS
: Used to indicate the maximum value for the array holidays.

MAX_SCAN_WORDS
: Used to indicate the maximum number of words that have to be scanned. A change of this value results a change to scan_table and a possible new to defining type named SCAN_... and a possible new entry in the switch of scan() in scan.c

MAX_ACC
: Used to indicate the maximum number of accounting–groups.

MAX_STATUS
: Used to indicate the maximum number of status an error can be in.

MAX_PHASE
: Used to indicate the maximum number of phases a typical development–project has.

BUGFILE
: This is the file–name where all the bugs are written to.

DATAFILE
: This is the file–name where the input data is listed.

SCAN_.....
: Used for the switch of scan() in scan.c. That's why all those different values.

The rest of the file are typedefs and struct–types and extern definitions of some variables and functions used in the modules.

## 5.3. Provisional conclusions

It is too early to draw hard conclusions. The research reported here has a strict time limit of 6 months and can not linger on till indefinitely. In these 6 months we have hopefully kicked off the research in the right direction. But the project is in scope larger than what can be accomplished in this limited time–period, so all we can do is make some intermediate observations. For my program only the formulas in sections (see 5.1.1. and 5.1.2.) can be used. If you have a constant failure–rate and repair–rate, use of 5.1.3. seems also possible, but this is not valid here as the program can be operational at fault occurrence. From the MTTF reports the r and s are calculated as follows: r = 1/MTTF and s = 1/MTTR.

We have tried to list notions and techniques from three related, but distinct fields: hardware engineering, software engineering, and quality control. In each of these fields the research targets and the background of the researchers are slightly different, which leads to a bewildering set of names to partially similar things. In the next subsections we aim to materialize a proposal for future work.

### 5.3.1. The role of statistics

Much of the quality–related research is of a statistical origin. When such results are used for Software Reliability Engineering (SRE), the question arises whether it is permitted to use statistics. There are numerous examples of the unjustified application of statistics and in SRE this danger is always on the lurk. The basic requirement is the independence of choice; in this case choice can be interpreted as to have an error, fault, failure or none of these. It is questionable whether this independence really exists on the code–line level. A false interpretation of the specification can have many effects and then gives rise to numerous dependent faults. In this case, a formal proof w.r.t. the specification would be better but this requires the presence of a complete specification: a rare event in practice.

Additional measures are needed to provide an environment wherein the conditions for formal proof are guaranteed. A first measure that in practice is applied as an alternative is project review. It is claimed that by visual inspection and/or mechanized compilation 70–80% of the errors can be removed [28]. The abundance of errors that result from the straight encoding makes this number believable. Another technique that is more fundamentally justified is complexity management either through formal specification or by object–oriented programming. Especially the latter approach promises to be a worthwhile addition.

Reliability Engineering comes into play when a runnable system is present but still contains a sizable number of faults. The conditions for using statistic may be present here but granularity is of importance, as discussed before. Granularity can either be forced on the lowest level by using object–oriented programming or be abstracted by taking some distance from the low–level code. In practice hardly more than the latter approach seems feasible, but the preferred granularity is debatable.

The moment that faults have become rare, the system can come into actual (beta) use. The rare failures can then best be measured as failures in time. This is typically the area of risk management where the quality of the system as a whole is at stake. In other words, granularity has become of little importance.

### 5.3.2. HW/SW comparison

The most well–known types of verification go under the name *test* for hardware and *debug* for software. The difference is solely in the flexibility of a software realization to place its observation points after the realization has taken place, while for hardware such points have to be inserted beforehand by multiplexers, scanregisters and so on. Overall this is not of major concern here, as we restrict ourselves to the quality of the specification and refrain from realizability issues. As indicated in Fig. 21, such a specification is a software program that forms the input to the subsequent technology mapping.

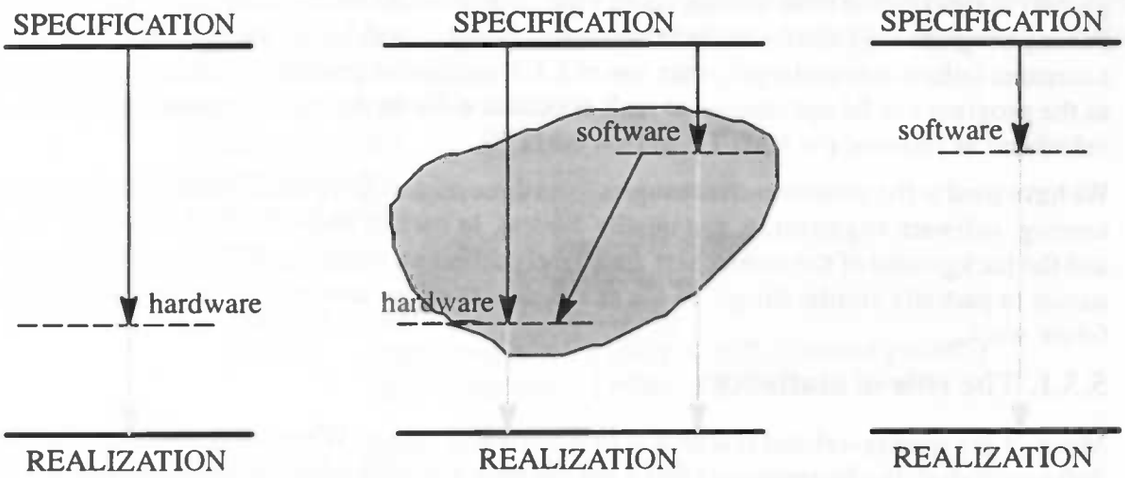



*Fig. 21 The fuzzy CoDesign implementation.*

The alternative is transformational design or, as usually named in software technology, **program proof**. It is based on a mathematical set of re–writing rules, such that a notation of a derived specification can be shown to be equivalent to the initial one. A rudimentary, often used, form is the straightforward run–time check on adherence to some primitive syntax and semantic rules; for the full–fledged program proof the problem space is a severely limiting factor, that has kept this technique far from practice for a long time.

In analog design, the reliability of a circuit is enhanced by **design centering**. Assuming the circuit to be influenced by parameters with a maximum and minimum value, the typical operating point of the circuit is placed in the middle of the inscribing circle of the area spanned by the worst–case design parameters. This equals the use of **software assertions**, i.e. the procedural definition of every hardware descriptive macro can be guarded by checking on the plausibility of the result w.r.t. the worst–case computation.

To qualify a design during progress, statistical measures have been introduced in software engineering that are principally based on the famous learning curve. It assumes the existence of programming errors and attempts to obtain a prediction of remaining errors based on the repair effort spent sofar. **Fault insertion** is often used to make the unknown, indigenous fault come to the surface. In hardware, fault insertion is usually applied to pin–point fault–sensitivity and then to generate fault–patterns for these specific faults. In software, the insertion is randomized and from the statistical sensitivity it is concluded how much overall faults can still be expected to be present. The obvious reason why software does not attempt to achieve a 100% fault coverage is the infinite number of cases that should be inspected. With the growing complexity of hardware, this reasoning is also applicable but, on the other hand, this is clearly not acceptable.

In current practice, the hardware equivalent to the C–programming language is often used to develop the system functionality. Then a manual transition to VHDL is made. This transition is obviously error–prone and we will venture to study such errors by software reliability techniques. It is usually conjectured that software reliability measurements can not be applied to assembler–level software programs. This seems to rule out the use for HDL verification. However, the implied complexity in counting the primitives in the target technology is immense as the primitives themselves are at a much lower abstraction level. By reference of the abstractional primitives, the various techniques do have a lot in common (Fig. 22).

| | SOFTWARE | | HARDWARE | |
|---|---|---|---|---|
| | object | means | object | means |
| abstraction | file | reliability | design | ??????? |
| decomposition | function | proof | core module | transformation |
| change in view | macro | assert | macro cell | generation |

Fig. 22 Overview of HW/SW verification techniques.

The premise of our research is the recognition that complex systems can not be extensively studied in a short time. Then to produce microelectronic products of the highest quality requires either the development of a new design style, oriented towards formal proof techniques and rigorous transformations, or a sense of quality must be integrated into the accepted simulation–driven design approach. This takes its inspiration from the software area, where also large and complex systems are constructed on basis of a non–targeted Compile–Link–Execute approach. Too much variation in function and usage forbids any extensive validation of the program's correctness.

Though the complexity of current software and hardware systems is not too different, the popular validation styles are wide apart. For reason of the cost–efficiency provided by mass–production the systems–on–silicon are still simulated, while further on the development track systems are to be assembled on fewer, but thoroughly tested, parts. These separate developments will be integrated for CoDesign and therefore it becomes the more urgent to unify the terminology and techniques in these areas. In other words, we will try to use the best of both worlds in order to answer their combined challenge.

In order to monitor the potential success of completing the specification, we assume that the current version is likely to be completed; the small lack in completion can be viewed as a set of faults. As these faults are unknown, the task is not so much as to identify them as to quantify their existence. Finding these faults is difficult enough; finding non–existing faults is a total

waste of effort. To this purpose we need (a) the detection of fault–effects, and (b) a mechanism to insert faults in order to (c) predict the potentially present number of indigenous faults.

In the following, we will provide arguments for a libral introduction of assertions for the in–line observation of simulation results. Further we introduce reliability scoring to verify completeness while simulating the HDL specification. The overall simulation effort will be diminished by fault seeding techniques, guided by compositional profiles.

### 5.3.3. To each his own

In the next discussion, the different ways to perform verification during the design of a microelectronic system will be illustrated by examples in a hardware–oriented dialect of the C–language called MDL. For the purpose of illustration, we go bottom–up and introduce first the biological neuron according to van der Malsburg with a function that can be informally rendered as:

```
repeat: if x>y then y=y+x;
  else if x<y then y=y/2;
```

This straightforward combinatorial function can be directly cast into a silicon–compatible structure. The only additions are the function header and the subsequent connect–statements, that relate internal to external variables. The hardware implementation is assumed to operate asynchronously in the X–input as present, as the hardware implementation will not start nor stop. The equivalence between the above algorithm and the below description can be formally proven through a series of re–writing rules.

```
module neuron(data Xext[8], data Oext[8]);
{       instance of cmp()       m1;
        instance of sub()       m2;
        instance of shift()     m3;
        signal                  less, neq, diff[8];
        variable                x[8], y[8]=0;
        { connect x=Xext; m1(x, y, neq, less);
          switch{   case "neq.less": { m2(y, x, diff); connect y=diff; }
                    case "neq.!less": { m3(y, diff); connect y=diff;}
                  }
          connect Oext=y;
        };
}
```

This description can classically be further processed to derive an ASIC–compatible netlist.

A first complication arises when the repeat–statement has real significance, i.e. when it implies a start and stop condition for the hardware implementation to operate. This situation is present in the Greatest Common Divider algorithm for which the informal pseudo–code is given below:

```
repeat: if x>y then x=y−x;
  else if x<y then y=x−y;
till x==y;
```

As a result of the repeat–statement, the operations have to be timed, which leads to the occurrence of states. The equivalence between the above algorithm and the below description can be formally proven through a series of re–writing rules.

```
module gcd(control enable, control done, data Xext[8], data Yext[8]);
{       instance of cmp()       m1;
        instance of sub()       m2;
```

```
        signal                          less, neq, diff[8];
        variable                        x[8], y[8], s1=1, s2, s3;
    for s1 { connect x=Xext; connect y=Yext;} next if "enable" then s2;
    for s2 { m1(x, y, neq, less);
          switch{     case  "neq.less": { m2(y, x, diff); connect y=diff; }
                      case "neq.!less": { m2(x, y, diff); connect x=diff; }
                 }
        } next if "!neq" then s3;
    for s3 set done, {connect Yext=y;} next if "!neq.enable" then s3
                                       else if "!enable"      then s1;
}
```

This description can classically be further processed to derive an ASIC–compatible netlist.

A more involved instance of the same computational template is the successive–approximation A/D–conversion. Here the unknown analog value is successively compared with a known analog value, that is created from a continuous digitally–controlled update. When both values are equal, the serial digital control sequence represents the digital notation for the analog value. In pseudo–code this is notated as:

```
y=0; z=2**width;
repeat: wait-on(x-y);
        if x>y then y=y+(z=z/2);
   else if x<y then y=y- (z=z/2);
till x==y;
```

In addition to the GCD–implementation discussed above, now we also have to consider the untimed cooperation between the analog and the digital world. Though in microelectronic design such issues are usually tested during simulation, we adhere to the software reliance on verification and make such requirements part of the hardware specification. Dependent on the development phase, such assertions are checking on the conformity between the functional kernel and the specific demands of the current specificational view. For the current structural view we find:

```
module sadc(control enable, control done, data Xext[8], data Yext[8]);
{       instance of cmp()               m1;
        instance of approx()            m2;
        signal                          less, neq;
        variable                        y[8], z[8], s1=1, s2, s3;
   for s1 { connect y=0; connect z=10000000;} next assert(T1) and if "enable" then s2;
    for s2 {  m1(Xext, y, neq, less);
          switch{     case  "neq.less": { m2(y, z, min); }
                      case "neq.!less": { m2(y, z, plus); }
                 }
        } next assert(T2) and if "!neq" then s3;
    for s3 set done, {connect Yext=y;} next if "!neq.enable" then s3
                                       else if "neq"         then s1;
}
```

This description can classically be further processed to derive an ASIC–compatible netlist.

The fundamental message from the above examples is, that for a reasonable design a range of verification techniques must be used. Their appearance in the hardware descriptions is always momentarily, but aids to get the transitions between the design views into grip. Despite these precautions we can never be better than the previous version of the descriptions.

### 5.3.4. Future work

Looking back we have regularly identified some unsolved problems. Overall we find that Reliability Engineering has little fault identification power, but there is hardly any other option. On closer inspection, we find that the reasons are in a software production process that aims to do everything right the first time. A more realistic approach would be to enforce a systematic work methodology that allows for an easy diagnosis and repair. We see three developments as the basic ingredients for such a design style.

**Software Encapsulation** should be such that faults and failures can be easily localized. This calls for Functional Programming and/or Object–Oriented Programming to be further developed in this direction. Especially the use of Java could be a remarkable enhancement.

**Engineering Unification** should lead to the collaborative development of hardware and software, such that design changes will not introduce new errors, hardly any faults and maybe some failures. Re–use of specification may here improve re–use of hardware and/or software.

**Tooling homogeneity** should lead to using the best of the different approaches. We suggest here ultimately the blending of formal proof using object–oriented specifications with rigidly proven transformations and a final risk monitoring.

# Suggested further reading

J.A. Abraham, *Fault Modeling in VLSI*, in: T.W. Williams (ed.), *VLSI Testing*, Advances in CAD for VLSI 5 (North–Holland, Amsterdam, 1986) pp. 1–28.

T. DeMarco, *Controlling Software Projects* (Dorset House Publishing, New–York, 1982).

A.L. Goel, *Software Reliability Models: assumptions, limitations and applicability*, IEEE Trans. on Software Engineering **SE–11**, (Dec. 1985) pp. 1411–1423.

E.J. Henley and H. Kumamoto, *Probabilistic Risk Assessment* (Prentice Hall, 1981).

H. Kopetz, *Software Reliability* (Mac–Millan, London, 1979).

J.D. Musa, A. Iannino and K. Okumoto, *Software Reliability* (Mc. Graw–Hill, 1993).

J.H. Poore, *Planning and Certifying software system reliability*, IEEE Software (January 1993) pp. 88–99.

# References


[1] J.A. Abraham, *Fault Modeling in VLSI*, in: T.W. Williams (ed.), *VLSI Testing*, Advances in CAD for VLSI **5** (North–Holland, Amsterdam, 1986) pp. 1–28.

[2] J.E. Angus, R.E. Schafer and A. Sukert, *Software reliability model validation*, Proceedings Ann. Reliability and Maintainability Symposium (San Fransisco, January 1980) pp. 191–193.

[3] J. Arabian, *Computer–Integrated Electronics Manufacturing and Testing* (Marcel Dekker, New York, 1989).

[4] S.L. Basin, *Estimation of software error rate via capture–recapture sampling* (Science Applications, Plao Alto, 1974).

[5] F.B. Bastani, *An input domain based theory of software reliability and its application*, Ph.D. Thesis (Berkeley, 1980).

[6] F.B. Bastani, *On the uncertainty in the correctness of computer programs*, IEEE Trans. Software Engineering **SE–11** (September 1985) pp. 857–864.

[7] R.G. Bennetts, *Design of Testable Logic Circuits* (Addison–Wesley, Reading, 1984).

[8] P.S. Bottorff, *Test Generation and Fault Simulation*, in: T.W. Williams (ed.), *VLSI Testing*, Advances in CAD for VLSI **5** (North–Holland, Amsterdam, 1986) pp. 29–64.

[9] A.C. Brombacher, *Integrating reliability analysis in the design process of electronic circuits and systems*, Ph.D. Thesis (Enschede, 1990).

[10] W.D. Brooks and R.W. Motley, *Analysis of discrete software reliability models*, Rep. RADC–TR–80–84 (April 1980).

[11] T. Coe and P.T.P. Tang, *It takes six ones to reach a flaw*, Proceedings 12th Symposium on Computer Arithmetic (Bath, England, 1995).

[12] T. DeMarco, *Controlling Software Projects* (Dorset House Publishing, New–York, 1982).

[13] S. Gerhart and L. Yelowitz, *Observations of fallibility in applications of modern programming languages*, Software Eng. **SE–2** (May 1976) pp. 195–207.

[14] A.L. Goel, *Software reliability modelling and estimation techniques*, Rep. RADC–TR–82–263 (October 1982)

[15] A.L. Goel, *A guidebook for software reliability assessment*, Rep. RADC–TR–83–176 (August 1983)

[16] A.L. Goel, *Software Reliability Models: assumptions, limitations and applicability*, IEEE Trans. on Software Engineering **SE–11**, (Dec. 1985) pp. 1411–1423.

[17] A.L. Goel and K. Okumoto, *An analysis of recurrent software failures in real–time control systems*, Proceedings Ann. Technical Conf. ACM (Washington, 1978) pp. 496–500.

[18] A.L. Goel and K. Okumoto, *A Markovian model for reliability and other performance measures of software systems*, Proceedings Nat. Comp. Conf. **48** (New York, 1979) pp. 769–774.

[19] A.L. Goel and K. Okumoto, *Time–dependent error detection rate model for software and other performance measures*, IEEE Trans. on Reliability **R–28**, (Aug. 1979) pp. 206–211.

[20] J. Harauz and M. Azuma, *Software engineering standards and methods*, ISO–IEC JTC1/SC7/WG6 (January 1993).

[21] E.J. Henley and H. Kumamoto, *Probabilistic Risk Assessment* (Prentice Hall, 1981).

[22] B. Hetzel, *The Sorry State of Software Practice: measurement and evaluation*, hand–out CSR 10th Annual Workshop (Amsterdam, 1993).

[23] B. Hetzel, *Measuring our measurements: the technology for software practice evaluation*, hand–out CSR 10th Annual Workshop (Amsterdam, 1993).

[24] IEEE Standards Board, *IEEE Standard Glossary of Software Engineering Terminology* (IEEE, New–York, 1983).

[25] Y. Iizuka, *A new paradigm for software quality,* hand–out CSR 10th Annual Workshop (Amsterdam, 1993).

[26] Z. Jelinski and P.B. Moranda, *Software Reliability Research*, in: W. Freiberger (ed.) *Statistical Computer Performance Evaluation* (Academic Press, New–York, 1972) pp. 465–484.

[27] R.J. Jorna and J.L. Simons, *Kennis in Organisatie* (Coutinho, Muiderberg, 1992).

[28] S. Keene, J.D. Musa, and T.W. Keler, *Developing Reliable Software in the Shortest Time*, Video Tutorial Course (IEEE, 1995).

[29] H. Kopetz, *Software Reliability* (Mac–Millan, London, 1979).

[30] M. Lipow, *Estimation of software package residual errors*, Rep. TRW–SS–72–09 (TRW, Redondo Beach, 1972)

[31] B. Littlewood and J.L. Verrall, *A Bayesian reliability growth model for computer software*, Appl. Statist. **22** (1973) pp. 332–346.

[32] B. Littlewood, *Theories of software reliability*, IEEE Trans. Software Eng. **SE–6** (1980) pp. 489–500.

[33] H.D. Mills, *On the statistical validation of computer programs, Rep. 72–6015* (IBM FSD, Gaithersburg, 1972).

[34] J.D. Musa, *A theory of software reliability and its application*, IEEE Trans, Software Eng. **SE–1** (1971) pp. 312–327.

[35] J.D. Musa, A. Iannino and K. Okumoto, *Software Reliability* (Mc. Graw–Hill, 1993).

[36] J.D. Musa and W.W. Everett, *Software reliability engineering*, IEEE Software (November 1990) pp. 36–43.

[37] J.D. Musa and K. Okumoto, *A logarithmic Poisson execution time model for software reliability measurement* (IEEE CS Press, Los Alamitos, 1984) pp. 230–238.

[38] P.B. Moranda, *Prediction of software reliability during debugging*, Proceedings Ann. Reliability and Maintainability Symp. (Washington, January 1975) pp. 327–332.

[39] G.J. Myers, *Software Reliability* (Wiley, New York, 1976).

[40] E. Nelson, *Estimating software reliability from test data*, Microelectronics Rel. **17** (1978) pp. 67–74.

[41] J.H. Poore, *Planning and Certifying software system reliability*, IEEE Software (January 1993) pp. 88–99.

[42] C.V. Ramamoorthy and F.B. Bastani, *Software reliability*, IEEE Trans Software Eng. **SE–8** (July 1982) pp. 359–371.

[43] J. Reynolds, *The craft of programming*, (Prentice–Hall, Englewood Cliffs, New York, 1981).

[44] B. Russell, *Human knowledge* (George Allen & Unwin Publishers, London, 1948).

[45] S.R. Schach, *Classical and Object–Oriented Software Engineering* (Irwin Inc., Chicago, 1996).

[46] G.J. Schick and R.W. Wolverton, *Assessment of software reliability*, Proc. Oper. Res., (Physica Verlag, Wirzberg–Wien, 1973) pp. 395–422.

[47] M.L. Schooman, *Software reliability measurement and models*, Proceedings Ann. Reliability and Maintainability Symp. (Washington, January 1975) pp. 485–491.

[48] M.L. Schooman, *Probabilistic models for software reliability prediction*, in: W. Freiberger (ed.) *Statistical Computer Performance Evaluation* (Academic Press, New–York, 1972) pp. 485–502.

[49] H.–C. Shih, J.T. Rahmeh, and J.A. Abraham, *An MOS fault simulator with waveform information*, Digest ICCD (November 1985).

[50] L.A. Skantze, *Air Force systems command software quality indicators*, AFSC Pamphlet 800–14 (January 1987) pp. 21–36.

[51] S. Yamada, M. Ohba, and S. Osaki, *S–shaped reliability growth modeling for software error detection,* IEEE Trans. on Reliability **R–32**, nr. 5 (Dec. 1983) pp. 475–478.

[52] K. Yasuda and K. Koga, *Product Development and Quality Assurance in the Software Factory*, hand–out CSR 10th Annual Workshop (Amsterdam, 1993).

# List of Figures

# Acknowledgements

I like to express my gratitude to Prof. dr. ir. L.J.M. Nieuwenhuis of KPN Research for providing much information and background on Software Reliability Engineering. Furthermore I like to acknowledge the discussions and cooperation with P.J. van der Gaag of Ericsson Business Mobile Networks. I also would like to thank my advisor Prof. dr. ir. L. Spaanenburg who has supported me with his knowledge in hardware & software. Also he was always available to provide me with guidance, when I needed it. Lastly praise is due to the Dutch Ministery of Education as well as to my parents for their patience.

# Appendix

## Makefile

```
#############################################################################
#
#             MAKEFILE FOR MTTF/MTTR/NMTTR/LISTING PROGRAM
#       by: Peter Smeenk
#
#       EXECUTABLE      is the name of the executable to be created
#       SRC             is a listing of all source-files
#       LANGUAGE        which language is being used
#       ANSI_CC         ansi c compiler name
#
#       Following arguments can be supplied:
#       None:   compiling all not up to date c-files to object files
#               and then linking them to the executable
#       clean:  remove all object files, the executable and
#               the bug-file(s)
#
#############################################################################

EXECUTABLE    = main
SRC             = calc.c bug.c init.c scan.c main.c output.c


LANGUAGE        = ANSI C

ANSI_CC         = gcc
ANSI_CFLAGS     = -Wall -g
CFLAGS          = -DSYSV  -DXOPEN_CATALOG
LIBS            = -lm

OBJ = $(SRC:.c=.o)

$(EXECUTABLE): $(OBJ)
              @echo Linking    $(EXECUTABLE)
              $(CC) $(OBJ) -o $(EXECUTABLE)
              @echo "Done"


.SUFFIXES: .o .c

.c.o:
              @echo Compiling $< [$(LANGUAGE)]
              $(CC) -c $(CFLAGS) $< -o $@

CC = \
@`if [ "$(LANGUAGE)" = "ANSI C" ]; then echo $(ANSI_CC) $(ANSI_CFLAGS); fi`


clean:
              rm -f *.o $(EXECUTABLE) bugs.txt
```

## Failure.h

```
/* File: failure.h

   The author of this software makes no warranty of any kind, express
   or implied, with regard to this software or documentation. nor shall
   he be liable for incidental or consequential damage in connection
   with the furnishing, performance or use of this software or
   documentation.
   This program may not be used in a commercial environment, or by a
```

```
   commercial organisation without a license. Permission is granted to
   make a limited number of copies of this software for educational or
   research purposes only.

   Peter Smeenk
   University of Groningen
*/

#define MAX_REPAIR_DAYS 200  /* We don't count further as MAX_REPAIR_DAYS    */
#define INFLICTION_FACTOR 3  /* How worse is it that a failure got found of  */
                             /* a previous phase. This will be multiplied    */
                             /* with the time of repair. The formula will be */
                             /* days *( INFLICTION_FACTOR power difference)  */
#define MAX_HOLIDAYS 4       /* How many holiday days do we know             */
#define MAX_SCAN_WORDS 9     /* The data file consits of 19 positions        */
#define MAX_ACC 2            /* Total number of accounting groups            */
#define MAX_FOUND_DATE 1     /* Day on which the error was found             */
#define MAX_REPAIR_DATE 1    /* Day on which the error was repaired          */
#define MAX_STATUS 2         /* Total number of states an error can have     */
#define MAX_PHASE 4          /* Total number of phases the project can be in */
#define BUGFILE "bugs.txt"   /* The file were the errors will be reported    */
#define DATAFILE "data.txt"  /* The file where the data is listed            */


#define SCAN_FOUND_DATE 1     /* These SCAN definitions determine what is    */
#define SCAN_REPAIR_DATE 2    /* are used in the switch of the function scan */
#define SCAN_ACC 3            /* in the file scan.c. See init.c scan_table   */
#define SCAN_MODULE_FOUND 4   /* for what is exactly standing on the places. */
#define SCAN_FUNCTION_NAME 5
#define SCAN_PHASE_INSERTED 6
#define SCAN_PHASE_DETECTED 7
#define SCAN_ERROR_NUMBER 8

/* short-hand form of some types that are being used. */
typedef struct failure_date     FAILURE_DATA;
typedef struct tm               TM;
typedef struct repair_time      RP;
typedef struct global_time      GT;
typedef struct module_list      ML;

/* Used in output.c to show how many errors a function has in every module */
struct module_list {
  char *func_name;
  char *module_name;
  int amount;
  ML *next;
};

/* Definiton of how 1 scan_word looks in the scan_table */
struct scan_word {
        int             name;
        int             group;
};

/* Definition of what is standing in the struct global_time */
struct global_time {
  int ftime[MAX_PHASE];
  int rtime[MAX_PHASE];
  int nonrtime[MAX_PHASE];
  int famount[MAX_PHASE];
  int ramount[MAX_PHASE];
  int nonramount[MAX_PHASE];
};

/* Accounting group name */
struct accounting_name {
        char *          name;
```

```
        int             num;
};

struct project_state {
        char *          name;
        int             num;
};

struct repair_time {
        int             repaired;
        int             time;
};

/* A list of failure dates */
struct failure_date {
        int  date_found[3];      /* date when the fault got found            */
        int  date_repaired[3];   /* date when the fault got repaired         */
        char *module_name;       /* 2 strings defining the module and        */
        char *func_name;         /* function name of where the fault occured */
        int  phase_inserted;     /* 2 integers for explaining in which phase */
        int  phase_found;        /* the fault got inserted and found.        */
        int  delta_date;         /* The number of days  till next fault      */
        RP   delta_repair;       /* The number of days a fault got not       */
                                 /* repaired if delta_repair.repaired == 1   */
                                 /* or the number of days the fault didn't   */
                                 /* get repaired until today                 */
        FAILURE_DATA *next;      /* Pointer to the next failure_date         */
};

/* An accounting group failures. For every accounting group, 1 failure_group */
struct failure_group {
        FAILURE_DATA *  date;
        int             nr_failures;
};

/* A day and month struct for general holidays */
struct holiday_group {
  int day;
  int month;
};

/* A table of all scan_words */
extern const struct scan_word           scan_table [MAX_SCAN_WORDS];

/* A table in which all accounting groups get their names. */
extern const struct accounting_name     accounting_table  [MAX_ACC];

/* A table for every possible project_status */
extern const struct project_state       project_table     [MAX_STATUS];

/* A table for every possible phase a project can be in */
extern const struct project_state       phase_table       [MAX_PHASE];

/* for every accounting group a list of failure dates */
extern struct failure_group             failure_table   [MAX_ACC];

/* Helplist for deteriming day in the month */
extern const int                        days_in_month   [12];

/* Helplist for determing if a day is a working day */
extern const int                        working_days    [7];

/* Helplist for determing if a day is a holiday, only the ones that
   come back every year at the same day.
*/
extern const struct holiday_group               holidays [MAX_HOLIDAYS];

/* Append an error string named bug_str to the BUGFILE */
void bug_data(const char *func, const char *bug_str,
              long line_nr, const char *file);
```

```
/* Append a function-error to the BUGFILE */
void bug_string      (const char *func, const char *bug_str);

/* The scanner for the datafile */
void scan            (void);

/* The calculator that does the actually work */
void calc            (void);

/* The results for all the groups */
void show_results    (void);

/* The function which determins if the first date is larger as the second */
int larger           (const date1[3], const date2[3]);
```

## Main.c

```
/* FILE: main.c

   The author of this software makes no warranty of any kind, express
   or implied, with regard to this software or documentation. nor shall
   he be liable for incidental or consequential damage in connection
   with the furnishing, performance or use of this software or documentation.
   This program may not be used in a commercial environment, or by a
   commercial organisation without a license. Permission is granted to
   make a limited number of copies of this software for educational or
   research purposes only.

   Peter Smeenk
   University of Groningen
*/


#include "failure.h"
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

/*
 * This is the main procedure. It calls the necessary functions in
 * a pre-set order :)
 */
void main() {
   scan();
   calc();
   show_results();
   return;
}
```

## Bug.c

```
/* FILE: bug.c


   The author of this software makes no warranty of any kind, express
   or implied, with regard to this software or documentation. nor shall
   he be liable for incidental or consequential damage in connection
   with the furnishing, performance or use of this software or documentation.
   This program may not be used in a commercial environment, or by a
```

```
   commercial organisation without a license. Permission is granted to
   make a limited number of copies of this software for educational or
   research purposes only.

   Peter Smeenk
   University of Groningen
*/

#include "failure.h"
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>


/*
 * Get the time of day in the form of a string: Wed Aug 21 14:39:20 1996
 */
char *get_date() {
        char *strtime;
        struct timeval tp;
        struct timezone tzp;
        time_t timer;

        gettimeofday( &tp, &tzp );
        timer = (time_t) tp.tv_sec;
        strtime = ctime( &timer );
        if (strtime[0] == '\0')
          strcpy( strtime, "UNKNOWN" );
        else
          strtime[strlen(strtime) -1] = '\0';
        return strtime;
}

/*
 * Give a bug message about a file with the line_nr where the
 * bug occured.
 */
void bug_data(const char *func, const char *bug_str, long line_nr,
                    const char *file) {
  FILE *fp;

  if ((func[0] == '\0') || (bug_str[0] == '\0')) return;
  if ((fp = fopen( BUGFILE, "a" )) != NULL) {
     fprintf( fp, "[%s]\nFunction %s, found error in file %s on line %ld:\n"
                          "%s\n\n",
                    get_date(), func, file, line_nr, bug_str );
     fclose( fp );
  }
  return;
}

/*
 * Give a bug message and report the function in which it occured.
 *
 */
void bug_string(const char *func, const char *bug_str) {
   FILE *fp;
   if ((func[0] == '\0') || (bug_str[0] == '\0')) return;
   if ((fp = fopen( BUGFILE, "a" )) != NULL) {
      fprintf( fp, "[%s]\nFunction %s reported error:\n%s\n\n",
                     get_date(), func, bug_str );
      fclose( fp );
   }
```

```
    return;
}
```

## Init.c

```
/* File: init.c

   The author of this software makes no warranty of any kind, express
   or implied, with regard to this software or documentation. nor shall
   he be liable for incidental or consequential damage in connection
   with the furnishing, performance or use of this software or documentation.
   This program may not be used in a commercial environment, or by a
   commercial organisation without a license. Permission is granted to
   make a limited number of copies of this software for educational or
   research purposes only.

   Peter Smeenk
   University of Groningen
*/


#include "failure.h"
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

/*
 * Every word position for every line are named here.
 * Reverence to this table as follows:
 * scan_table[4].name ==  SCAN_MODULE_FOUND
 * scan_table[4].group == 1
 *
 */
const struct    scan_word       scan_table          [MAX_SCAN_WORDS] = {
  { SCAN_ERROR_NUMBER, 0},
  { SCAN_FOUND_DATE, 1},
  { SCAN_REPAIR_DATE, 1},
  { SCAN_ACC, 0},
  { SCAN_ACC, 1},
  { SCAN_MODULE_FOUND, 1},
  { SCAN_FUNCTION_NAME, 1},
  { SCAN_PHASE_DETECTED, 0},
  { SCAN_PHASE_INSERTED, 0},
};

/* The names of all accounting groups. The second integer revers to
 * the position in the table.
 *
 */
const struct    accounting_name accounting_table    [MAX_ACC] = {
   {"GROUP1", 0},
   {"GROUP2", 1},
};

/*  The possible state an error can have in my system are:
 *  NUL -> nullified, thrown away, no error
 *  CLO -> closed, solved etc
 *  Free to think up on new nice error states. Only nullified errors
 *  Will be not counted. Other errors will, even with new names.
 */
const struct    project_state  project_table    [MAX_STATUS] = {
```

```
    {"NUL",       1},
    {"CLO",       2},
};

/* The phases a project can be in. From development to system test,
 *  Having this in uppercase is important!
 *  The integer is the unique key. 0 is reserved for no phase specified.
 */
const struct    project_state  phase_table     [MAX_PHASE] = {
  {"DEV1",      1},
  {"DEV2",      2},
  {"DEV3",      3},
  {"DEV4",      4},
};

/* Every month has a specific number of days. The code checks the
   month February for a leap year or not.
*/
const int days_in_month[12] = {
  31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
};

/* Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday. */
const int working_days[7] = {
  0, 1, 1, 1, 1, 1, 0,
};

/* New year, Queen's day, Xmas */
const struct holiday_group holidays[MAX_HOLIDAYS] = {
  {1,1},
  {30,4},
  {25,12},
  {26,12},
};
```

## Scan.c

```
/* File scan.c


   The author of this software makes no warranty of any kind, express
   or implied, with regard to this software or documentation. nor shall
   he be liable for incidental or consequential damage in connection
   with the furnishing, performance or use of this software or documentation.
   This program may not be used in a commercial environment, or by a
   commercial organisation without a license. Permission is granted to
   make a limited number of copies of this software for educational or
   research purposes only.

   Peter Smeenk
   University of Groningen
*/

#include "failure.h"
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

/* Definition of all failures for every group */
struct failure_group failure_table [MAX_ACC];
```

```
/*
 *  function which determines if date1 is larger as date2
*/
int larger(const date1[3], const date2[3]) {
    /* Assumption that the date1 and date2 are in the correct format:
       Day-Month-Year of the form: 01-12-1995
    */
    if (date1[2] > date2[2]) return 1;
    if ((date1[1] > date2[1]) && (date1[2] == date2[2])) return 1;
    if ((date1[0] > date2[0]) && (date1[1] == date2[1]) &&
        (date1[2] == date2[2]))
      return 1;
    return 0;
}

/*
 *  insert a new date into the FAILURE_DATA list of that specific accounting
 *  group
*/
FAILURE_DATA *insert_date(FAILURE_DATA *list, const int date_found[3],
                          const int date_repaired[3], const char *module_name,
                          const char *func_name, const int phase_inserted,
                          const int phase_found, const long line_nr) {
    /* Assumption that the dates in the list and date are in correct format:
       Day-Month-Year of the form: 01-12-1995
    */
    FAILURE_DATA *templ,*hup;

    if ((date_found[0] <= 0) || (date_found[1] <= 0) || (date_found[2] <= 0)) {
      bug_data("insert_date","Invalid date found received.",line_nr,
                    "data.txt");
      return 0;
    }
    if ((date_repaired[0] < 0) || (date_repaired[1] < 0) ||
              (date_repaired[2] < 0)) {
      bug_data("insert_date","Invalid date repaired received.",line_nr,
                    "data.txt");
      return 0;
    }
    if (func_name[0] == '\0') {
      bug_data("insert_date","No func_name specified.",line_nr,"data.txt");
      return 0;
    }
    if (module_name[0] == '\0') {
      bug_data("insert_date","No module_name specified.",line_nr,"data.txt");
      return 0;
    }
    hup = malloc( sizeof( *hup ) );
    if (hup == NULL) {
      bug_data("insert_date","MALLOC failed.",line_nr,"data.txt");
      return list;
    }
    hup -> date_found[0] = date_found[0];
    hup -> date_found[1] = date_found[1];
    hup -> date_found[2] = date_found[2];
    hup -> date_repaired[0] = date_repaired[0];
    hup -> date_repaired[1] = date_repaired[1];
    hup -> date_repaired[2] = date_repaired[2];
    hup -> module_name = strdup(module_name);
    hup -> func_name = strdup(func_name);
    hup -> phase_inserted = phase_inserted;
    hup -> phase_found = phase_found;
    hup -> delta_date = 0;
    hup -> delta_repair.repaired = 0;
```

```
    hup -> delta_repair.time = 0;
    hup -> next = NULL;
    if (list == NULL) return hup;
    temp1 = list;
    if (larger(temp1 -> date_found, date_found)) {
      hup -> next = temp1;
      return hup;
    }
    while (temp1 -> next != NULL) {
       if (!larger(temp1 -> next -> date_found,date_found)) {
           temp1 = temp1 -> next;
           continue;
       }
       hup -> next = temp1 -> next;
       temp1 -> next = hup;
       return list;
    }
    if (larger(temp1 -> date_found,date_found)) {
       hup -> next = temp1;
       list = hup;
    }
    else temp1 -> next = hup;
    return list;
}


/*
 * Syntax check of the date, this can be modified, if necessary.
 *
 */
int error_date(const char* date,const int line_nr, int ret_date[3]) {
    int day, month, year,i;

    for (i=0;i<2;i++) ret_date[i] = 0;
    i = sscanf(date,"%d-%d-%d", &year, &month, &day);
    if (i != 3) {
      bug_data("error_date","Rejected the date on wrong syntax",
               line_nr,"error_date");
      return 0;
    }
    if ((month < 1) || (month > 12)) {
      bug_data("error_date","rejected date on month",line_nr,DATAFILE);
      return 0;
    }
    if ((day < 1) || (day > days_in_month[month-1])) {
      bug_data("error_date","rejected date on days in month",line_nr,DATAFILE);
      return 0;
    }
    if (year < 1900) {
      bug_data("error_date","rejected date on year",line_nr,DATAFILE);
      return 0;
    }
    ret_date[0] = day;
    ret_date[1] = month;
    ret_date[2] = year;
    return 1;
}
/*
 * Read till the beginning of the next line
 */
void fread_to_nl( FILE *fp ) {
    char c;
    c = getc(fp);
```

```
    while ( (c != '\n') && (c != '\r' ))
    {
        c = getc( fp );
    }
    while ((c == '\n') || (c == '\r'))
    c = getc( fp );
    ungetc( c, fp );
    return;
}

/*
 * Read 1 string/number
 */
char *fread_word( FILE *fp, int word_nr, long line_nr ) {
    static char word[100];
    char *pword;
    char c;

    /* Before calling this function the invariant !feol holds    */

    c = getc( fp );
    if (c == '"') c = getc( fp );
    pword = word;
    for ( ; pword < word + 100; pword++ ) {
      if ((c == ',') || (c == '"') || (c == '\r') || (c == '\n')) {
        if (c == '"') c = getc( fp );
        if ((c == '\n') || (c == '\r')) ungetc( c, fp );
        *pword = '\0';
        return word;
      }
      *pword = c;
      c = getc( fp );
    }
    bug_data( "fread_word", "Fread_word: word too long.\n", line_nr, DATAFILE);
    printf("A serious error has occured, A word is longer as 100 chars.\n");
    printf("Please check your BUGFILE for more information on which line.\n");
    printf("Using emergency exit and closing down program.\n");
    exit( 1 );
    return NULL;
}

/*
 *  Return the the characters of input in upper_case
 */
char *upper( const char *input,long line_nr) {
  static char word[100];
  int i;

  for (i=0; (i < strlen(input)) && (i < 100); i++) {
   if ((input[i] >= 'a') && (input[i] <= 'z'))
     word[i] = input[i] - 'a' + 'A';
   else
     word[i] = input[i];
  }
  if (i >= 100) {
    bug_data( "upper", "Upper: word too long.\n", line_nr, DATAFILE);
    printf("A serious error has occured, a word is longer as 100 chars.\n");
    printf("Please check your BUGFILE for more information on which line.\n");
    printf("Using emergency exit and closing down program.\n");
    exit( 1 );
  }
  word[i] = '\0';
  return word;
}
```

```
/*
 * function returns 1 if end of line has been reached.
 */
int feol(FILE *fp) {
  char c;
  c = getc(fp);
  ungetc(c, fp);
  if ((c == '\n') || (c == '\r')) return 1;
  return 0;
}

/*
 * Function to match *name with the phase
*/
int get_phase(const char *name) {
  int i;
  for (i=0;i<MAX_PHASE;i++) {
   if (!strcmp(name,phase_table[i].name))
     return phase_table[i].num;
  }
  return 0;
}


/*
 * scan the DATAFILE for failure_data
 */
void scan() {
   FILE *fp;
   char *word,buf[100],*mod_name,*func_name;
   int accs[MAX_ACC],found_date[3],repair_date[3];
   int i,nr_word,error,phase_detected,phase_inserted,correct_date;
   long line_nr;

   if ((fp = fopen( DATAFILE, "r")) == NULL) {
             bug_string("scan","Could not open the DATAFILE for reading.");
             exit(1);
   }
   for (i=0;i<MAX_ACC;i++) {
     failure_table[i].date = NULL;
     failure_table[i].nr_failures = 0;
     accs[i] = 0;
   }
   line_nr = 0;
   while (!feof(fp)) {
      line_nr++;
      nr_word = 0;
      correct_date = 1;
      for (i=0;i<MAX_ACC;i++) accs[i] = 0;
      for (i=0;i<3;i++) {
             repair_date[i] = 0;
        found_date[i] = 0;
      }
      while (!feol(fp)) {
      word = strdup( fread_word( fp, nr_word, line_nr ) );
      switch(scan_table[nr_word].name) {
          case SCAN_ERROR_NUMBER:
          break;
          case SCAN_ACC  :
               if ((word[0] != '\0') && strcmp(upper(word,line_nr),"NUL"))
                  accs[scan_table[nr_word].group] = 1;
          break;
          case SCAN_FOUND_DATE :
               if ((word[0] != '\0') && (correct_date))
```

```
                correct_date = error_date(word,line_nr,found_date);
            break;
            case SCAN_REPAIR_DATE :
              if ((word[0] != '\0') && (correct_date))
                correct_date = error_date(word,line_nr,repair_date);
            break;
            case SCAN_MODULE_FOUND :
              mod_name = strdup(upper(word,line_nr));
            break;
            case SCAN_FUNCTION_NAME :
              func_name = strdup(upper(word,line_nr));
            break;
            case SCAN_PHASE_INSERTED :
              phase_inserted = get_phase(upper(word,line_nr));
            break;
            case SCAN_PHASE_DETECTED :
              phase_detected = get_phase(upper(word,line_nr));
            break;
            default         :
              sprintf(buf,"Unexpected type in scan_word: %d on line: %ld",
                  scan_table[nr_word].name,line_nr);
              bug_string("scan",buf);
              error = 1;
            break;
        } /* switch */
        nr_word++;
        } /* feol(fp) */
        fread_to_nl( fp );
        if (nr_word < MAX_SCAN_WORDS - 1)
          bug_data("fread_word","Fread_word: Unexpected End of Line.\n",
                              line_nr, DATAFILE);
        error = 1;
        for (i=0;i<MAX_ACC;i++) {
          if ((accs[i] == 1) && correct_date) {
           failure_table[i].date =
                  insert_date(failure_table[i].date,found_date,repair_date,
                              mod_name,func_name,phase_inserted,phase_detected,
                              line_nr);
           failure_table[i].nr_failures++;
           error = 0;
          }
        }
        if (error)
          bug_data("scan","Incorrect input: No accounting group supplied.",
                          line_nr,DATAFILE);

    } /* feof(fp) */
    fclose( fp );
    return;
}
```

## Calc.c

```
/* File: calc.c

    The author of this software makes no warranty of any kind, express
    or implied, with regard to this software or documentation. nor shall
    he be liable for incidental or consequential damage in connection
    with the furnishing, performance or use of this software or documentation.
    This program may not be used in a commercial environment, or by a
    commercial organisation without a license. Permission is granted to
```

```
   make a limited number of copies of this software for educational or
   research purposes only.

   Peter Smeenk
   University of Groningen
*/

#include "failure.h"
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <float.h>

/*
 *  Determine if the day is a day on which people work or not.
 *  - wday is an integer ranging from 0-6, and the 0 equals a sunday
 *    and the 6 equals saturday
 *  - mday is an integer ranging from 1-31, indicating the month's day.
 *  - month is an integer ranging from 1-12, indicating the month
 */
int working_day(const int wday, const int mday, const int month) {
  int i,j;

  j = 1;
  if (!working_days[wday]) j = 0;
  for (i=0;i<MAX_HOLIDAYS;i++)
    if ((holidays[i].day == mday) &&
              (holidays[i].month == month)) j = 0;
  return j;
};

/*
 * Return a pointer to the structure TM, in which from 3 integers,
 * with:
 * - arg[0] equals the days in the month [1..31],
 * - arg[1] equals the month [1..12],
 * - arg[2] equals the year in 4 digits. eg 1996
 * The system call mktime, should do the work. Some conversions
 * need to be done to get the values right.
 *
 */
TM *calc_time(const int arg[3]) {
  TM *tm_ptr;

  if ((arg[0] <= 0) || (arg[1] <= 0) || (arg[2] <= 0)) return NULL;
  tm_ptr = malloc( sizeof( *tm_ptr ));
  if (tm_ptr == NULL) {
    bug_string("calc_time","Malloc failed in calc_time");
    printf("Malloc failed, sorry taking emergency exit.\n");
    exit(1);
    return NULL;
  }
  tm_ptr -> tm_sec  = 0;
  tm_ptr -> tm_min  = 0;
  tm_ptr -> tm_hour = 12;
  tm_ptr -> tm_mday = arg[0];
/* Month of the year goes from 0..11 */
  tm_ptr -> tm_mon  = arg[1]-1;
/* Year count starts from 1900 */
  tm_ptr -> tm_year = arg[2] - 1900;
  mktime(tm_ptr);
  return tm_ptr;
}
```

```
/*
 * Determine the number of days between 2 days in the same month.
 * wday is the working day of day1 and ranging from 0-6, and the
 * 0 is the sunday.
 */
int calc_days(const int day1, const int wday,
              const int day2, const int month) {
  int i,j;
  j = 0;
  for (i=day1+1;i<=day2;i++)
    if (working_day((i-(day1+1)+wday) % 7,i,month)) j++;
  return j;
}

/*
 * Determine the number of days between 2 structs of type TM.
 * Distinction between 2 cases:
 * 1) month and year equal
 * 2) not 1, count days till end of month and recurse for
 *    the next month, not forgetting the first day of the
 *    next month.
 *
 */
int num_days(const TM *tm_ptr1, const TM *tm_ptr2) {
  if ((tm_ptr1 == NULL) || (tm_ptr2 == NULL)) return 0;
  if ((tm_ptr1 -> tm_year == tm_ptr2 -> tm_year) &&
      (tm_ptr1 -> tm_mon  == tm_ptr2 -> tm_mon)) {
    if (tm_ptr1 -> tm_mday > tm_ptr2 -> tm_mday)
      return (-1 * num_days(tm_ptr2,tm_ptr1));
    else if (tm_ptr1 -> tm_mday == tm_ptr2 -> tm_mday)
      return 0;
    else return calc_days(tm_ptr1->tm_mday,tm_ptr1->tm_wday,
                          tm_ptr2->tm_mday,tm_ptr1->tm_mon+1);
  }
  else {
      int i;
      int mon[3];
      TM *tmp;
      int result;
      if ((tm_ptr1 -> tm_year > tm_ptr2 -> tm_year) ||
          ((tm_ptr1 -> tm_year == tm_ptr2 -> tm_year) &&
           (tm_ptr1 -> tm_mon > tm_ptr2 -> tm_mon)))
        return (-1 * num_days(tm_ptr2,tm_ptr1));
      if (tm_ptr1 -> tm_mon == 1) {
/* Every year modulo 4 is a leap year, except once in the 400 year. */
        if ((((tm_ptr1 -> tm_year + 1900) % 400) != 0) &&
            (((tm_ptr1 -> tm_year + 1900) % 4) == 0)) i = 29;
        else i = 28;
      }
      else i = days_in_month[tm_ptr1 -> tm_mon];
      result = calc_days(tm_ptr1->tm_mday,tm_ptr1->tm_wday,
                         i,tm_ptr1->tm_mon+1);
      mon[0] = 1;
      if (tm_ptr1 -> tm_mon == 11) {
        mon[1] = 1;
        mon[2] = 1900 + tm_ptr1 -> tm_year + 1;
      }
      else {
        mon[1] = 1 + tm_ptr1 -> tm_mon + 1;
        mon[2] = 1900 + tm_ptr1 -> tm_year;
      }
```

```
        tmp = calc_time(mon);
        if (tmp == NULL) {
           bug_string("num_days","Malloc failed in else part");
           return 0;
        }
        /* THIS IS A BUG FROM MK_TIME. It's one day of at the first
           day of the month at this moment :-)
        */
        if (tmp -> tm_wday == 0) i = 6;
        else i = tmp -> tm_wday - 1;
        return (result + (working_day(i,1,tmp -> tm_mon+1) ? 1 : 0) +
                num_days(tmp,tm_ptr2));
      }
}

/*
 * Main function of this module.
 * It determines the number of days of a repair in case the fault
 * got repaired (variable data -> date_repaired), otherwise determines
 * the number of days the fault didn't get repaired.
 * It determines the number of days till the next fault. It assumes
 * that fault-days are recorded ascending in the structure data.
 */
void calc() {
  FAILURE_DATA *data;
  TM *day_time,*this_fault_found,*next_fault_found,*this_fault_repair,*temp;
  int i,current_date[3];
  struct timeval tp;
  struct timezone tzp;
  time_t timer;

  gettimeofday(&tp,&tzp);
  timer = (time_t) tp.tv_sec;
  day_time = localtime(&timer);
  if (day_time) {
  current_date[0] = day_time -> tm_mday;
  current_date[1] = day_time -> tm_mon + 1;
  current_date[2] = day_time -> tm_year + 1900;
  }
  else for (i=0;i<3;i++) current_date[i] = 0;
  temp = calc_time(current_date);
  for (i=0;i<MAX_ACC;i++) {
    data = failure_table[i].date;
    if (data == NULL) continue;
    this_fault_found = calc_time(data -> date_found);
    for ( ;data != NULL;data = data -> next) {
      this_fault_repair = calc_time(data -> date_repaired);
      if (data -> next == NULL) {
             data -> delta_date = 0;
             if ((this_fault_found != NULL) &&
                 (this_fault_repair != NULL)) {
               data -> delta_repair.time = num_days(this_fault_found,
                                           this_fault_repair);
               if (data -> delta_repair.time > MAX_REPAIR_DAYS)
                 data -> delta_repair.time = MAX_REPAIR_DAYS;
               data -> delta_repair.repaired = 1;
             }
             else {
               if (temp)
                 data -> delta_repair.time = num_days(this_fault_found,temp);
               else data -> delta_repair.time = -1;
               if (data -> delta_repair.time > MAX_REPAIR_DAYS)
                 data -> delta_repair.time = MAX_REPAIR_DAYS;
               data -> delta_repair.repaired = 0;
```

```
            }
        }
        else {
            next_fault_found = calc_time(data -> next -> date_found);
            data -> delta_date = num_days(this_fault_found,next_fault_found);
            if ((this_fault_found != NULL) &&
                (this_fault_repair != NULL)) {
              data -> delta_repair.time = num_days(this_fault_found,
                                        this_fault_repair);
              data -> delta_repair.repaired = 1;
            }
            else {
          if (temp)
              data -> delta_repair.time = num_days(this_fault_found,temp);
              else data -> delta_repair.time = -1;
              data -> delta_repair.repaired = 0;
            }
            this_fault_found = next_fault_found;
        }
      }
    }
    return;
}
```

## Output.c

```
/* File output.c

   The author of this software makes no warranty of any kind, express
   or implied, with regard to this software or documentation. nor shall
   he be liable for incidental or consequential damage in connection
   with the furnishing, performance or use of this software or documentation.
   This program may not be used in a commercial environment, or by a
   commercial organisation without a license. Permission is granted to
   make a limited number of copies of this software for educational or
   research purposes only.

   Peter Smeenk
   University of Groningen
*/


#include "failure.h"
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <float.h>

GT *init_gt() {
  int j;
  GT *init;

  init = malloc(sizeof(*init));
  if (init == NULL) {
    bug_string("init_gt","Malloc failed. Taking emergency exit.");
```

```
    exit(1);
    return NULL;
  }
  for (j=0;j<MAX_PHASE;j++) {
    init -> ftime[j] = 0;
    init -> rtime[j] = 0;
    init -> nonrtime[j] = 0;
    init -> famount[j] = 0;
    init -> ramount[j] = 0;
    init -> nonramount[j] = 0;
  }
  return init;
}


/*
 * Determine average of 2 integers. A bit more safe.
 */
float avg(const int i,const int j) {
  if (j) return (((float) i)/((float) j));
  return 0;
}




void show_gt(GT *show) {
  char buf[100];
  int j;
   sprintf(buf,"Phase:    #F  FTime    #R  Rtime    #N  Ntime    AvgFtime    AvgRtime
AvgNonRtime\n");
  printf(buf);
  for (j=0;j<MAX_PHASE;j++) {
    sprintf(buf,"%5s  %4d %5d %4d %5d %4d %5d   %8.3f   %8.3f    %8.3f\n",
                phase_table[j].name,
                show -> famount[j],show -> ftime[j],
                show -> ramount[j],show -> rtime[j],
                show -> nonramount[j],show -> nonrtime[j],
                avg(show -> ftime[j],show -> famount[j]),
                avg(show -> rtime[j],show -> ramount[j]),
                avg(show -> nonrtime[j],show -> nonramount[j]));
    printf(buf);
  }
}


void module_listing(int group) {
  FAILURE_DATA *data;
  ML *list, *temp, *templ;
  int found_module;
  char *last_module;
  char buf[100];


/* Start building the list */
  list = NULL;
  if ((data = failure_table[group].date) == NULL) return;
  for ( ; data != NULL; data = data -> next) {
    temp = list;
    found_module = 0;
    while ((temp != NULL) && (!found_module)) {
      if ((!strcmp(data->module_name,temp -> module_name)) &&
```

```
            ((!strcmp(data->func_name,temp -> func_name)))) {
          temp -> amount += 1;
          found_module = 1;
      }
      temp = temp -> next;
    }
    temp = NULL;
    if (!found_module) {
      temp = malloc(sizeof(*temp));
      if (temp == NULL) return;
      temp -> module_name = strdup(data -> module_name);
      temp -> func_name = strdup(data -> func_name);
      temp -> amount = 1;
      temp -> next = NULL;
      templ = list;
      while ((templ) && (!found_module)) {
            if (!strcmp(templ->module_name,temp->module_name)) {
              temp -> next = templ -> next;
              templ -> next = temp;
              found_module = 1;
            }
            else templ = templ -> next;
      } /* while */
      if (!found_module) {
       temp -> next = list;
       list = temp;
     }
    }
  } /* for */

/* Print out the list to .. */
  temp = list;
  printf("A listing of all the modules and number of faults.\n");
  sprintf(buf,"%15s %15s %5s\n","Module","Function","Amount");
  printf(buf);
  last_module = NULL;
  while (temp != NULL) {
    if ((last_module == NULL) ||
        (strcmp(last_module,temp -> module_name))) {
      sprintf(buf,"%15s %15s %5d\n",
                  temp -> module_name,
                  temp -> func_name,
                  temp -> amount);
      printf(buf);
      last_module = strdup(temp -> module_name);
    }
    else {
      sprintf(buf,"%15s %15s %5d\n","",temp -> func_name,temp -> amount);
      printf(buf);
    }
    temp = temp -> next;
  }
}


/*
 * Display for every month what the results are.
 * Don't forget the infliction factor!
 */
void init_graf(int group,GT *all_times) {

  FAILURE_DATA *data;
  GT *show;
```

```
int j,inflict,next_month[3],this_month[3];
char buf[100];


if ((group < 0) || (group >= MAX_ACC)) return;
if ((data = failure_table[group].date) == NULL) return;
for (j=0;j<3;j++) this_month[j] = data -> date_found[j];
next_month[0] = data -> date_found[0];
if (data -> date_found[1] == 12) {
  next_month[1] = 1;
  next_month[2] = data -> date_found[2] + 1;
}
else {
  next_month[1] = data -> date_found[1] + 1;
  next_month[2] = data -> date_found[2];
}
while (data != NULL) {
  if ((show = init_gt()) == NULL) return;
  while ((data != NULL) && (!larger(data -> date_found,next_month))) {
    if (data -> phase_inserted <= 0) continue;
    j = data -> phase_inserted-1;
    if ((data -> phase_inserted) &&
              (data -> phase_inserted < data -> phase_found) &&
              (INFLICTION_FACTOR > 1))
            inflict = INFLICTION_FACTOR *
              (data -> phase_found - data -> phase_inserted);
    else
            inflict = 1;
    show -> ftime[j] += data -> delta_date;
    show -> famount[j] += 1;
    if (data -> delta_repair.repaired) {
            show -> rtime[j] += data -> delta_repair.time * inflict;
            show -> ramount[j] += 1;
    }
    else {
            show -> nonrtime[j] += data -> delta_repair.time;
            show -> nonramount[j] += 1;
    }
    data = data -> next;
  }
  for (j=0;j<MAX_PHASE;j++) {
    all_times -> ftime[j] += show -> ftime[j];
    all_times -> famount[j] += show -> famount[j];
    all_times -> rtime[j] += show -> rtime[j];
    all_times -> ramount[j] += show -> ramount[j];
    all_times -> nonrtime[j] += show -> nonrtime[j];
    all_times -> nonramount[j] += show -> nonramount[j];
  }
  sprintf(buf,"From [%2d-%2d-%4d] to [%2d-%2d-%4d]:\n",
              this_month[0],this_month[1],this_month[2],
              next_month[0],next_month[1],next_month[2]);
  printf(buf);
  show_gt(show);
  for (j=0;j<3;j++) this_month[j] = next_month[j];
  next_month[0] = next_month[0];
  if (next_month[1] == 12) {
    next_month[1] = 1;
    next_month[2] = next_month[2] + 1;
  }
  else {
    next_month[1] = next_month[1] + 1;
    next_month[2] = next_month[2];
  }
 }/* while */
```

```
  return;
}

/*
 *  Do the output for all groups
 */
void show_results() {
  GT *all_times;
  int i;
  char buf[100];

  for (i=0;i<MAX_ACC;i++) {
    if ((all_times = init_gt()) == NULL) return;
    sprintf(buf,"Failure data for the group: %s\n", accounting_table[i].name);
    printf(buf);
    sprintf(buf,"Accounting-group %10s has %4d failures.\n",
                 accounting_table[i].name,
           failure_table[i].nr_failures);
    printf(buf);
    init_graf(i,all_times);
    printf("\n\n\n");
    printf("Overal result for this group:\n");
    show_gt(all_times);
    module_listing(i);

printf("--------------------------------------------------------------------------------\n
");

    printf("\n\n\n");
  }
  return;
}
```