



TRITA-MMK 1996:13
ISSN 1400-1179
ISRN KTH/MMK/R--96/13-SE
rev 3

Safe and Reliable Computer Control Systems Concepts and Methods

Henrik Thane



Stockholm 1996

Mechatronics Laboratory
Department of Machine Design
Royal Institute of Technology, KTH
S-100 44 Stockholm
Sweden

Safe and Reliable Computer Control Systems

Concepts and Methods

6 September 1996
Henrik Thane
Mechatronics Laboratory
The Royal Institute of Technology
Sweden
henrikt@damek.kth.se

ABSTRACT

The introduction of computers into safety-critical control systems lays a heavy burden on the software designers. The public and the legislators demand reliable and safe computer control systems, equal to or better than the mechanical or electromechanical parts they replace. The designers must have a thorough understanding of the system and more accurate software design and verification techniques than have usually been deemed necessary for software development. This document presents existing concepts and methods, relating to the design of software in safety-critical computer control systems. The concepts and methods dealing with fault avoidance, fault removal, fault tolerance, hazard analysis and safe design will be penetrated in detail. The document will enlighten the reader of what kind of realistic expectations the designer can form regarding the reliability and safety of a system's software. The author of the document has concluded that it is not enough to, for example, solely rely on formal verification in order to achieve the necessary level of safety and reliability needed, but to use several concepts in union, like formal specification, testing, fault tolerance and hazard analysis.

Key words: dependability, formal methods, test, fault tolerance, hazard analysis and safe design.

TABLE OF CONTENTS

1 AIM & STRUCTURE	4
2 BACKGROUND	5
2.1 How computers and software differ from other machine elements	5
3 BASIC VOCABULARY	8
3.1 Definitions	8
4 COMPUTERS AND RELIABILITY	11
4.1 What causes system failures?	11
4.2 The nature of software and its environment	12
4.3 Construction of Dependable Systems	12
4.3.1 Fault Avoidance	13
4.3.2 Fault Removal	15
4.3.3 Fault Tolerance	16
4.4 Remarks	21
5 COMPUTERS AND SAFETY	22
5.1 Distinctions between reliability and safety	22
5.2 Software and hazards	23
5.3 Hazard analysis	23
5.3.1 The iterative hazard analysis process	23
5.4 Design of safe systems	31
5.4.1 Design techniques	31
6 CONCLUSION	35
7 ACKNOWLEDGMENTS	36
8 REFERENCES	37

1 AIM & STRUCTURE

In this document the author will try to enlighten the reader of the concepts and methods that exist for design of safe and reliable computer control systems. Although this report is biased towards embedded computer control systems, many problems and solutions apply to more general computer based systems.

The document can be viewed as a state of the art report. The contents of this paper, however filtered through the author, is mainly a compilation of previous work done in the field of safe and reliable computer software design. The discussions and conclusions are nonetheless the authors. As the subjects of reliability and safety are penetrated in detail the author will try to highlight the shortcomings of current methods and what yet needs to be devised.

It is important to point out that this report will not, to any great extent, cover the subject of security. Security is generally, in the open computer based systems' sense, a quality regarding protection against maliciously intended external actions or protection of information. Issues that come to mind are hackers and cryptography. Security in embedded systems, which by definition are closed, is of less importance than safety. Safety is a quality inherent to a system, which during its lifetime does not subject people or property to harm. Computer control systems tend more often to be safety-critical than, for example, a fund transferring bank computer system.

The paper might be perceived to have a negative contents, but the intention is solely to make the reader form realistic expectations of what kind of reliability and safety a computer based system can be designed to have.

The outline

This report begins with a **background** explaining why computer software is non trivial to design. Then, in order to form a terminology foundation, on which the rest of the document is based, a **basic vocabulary** is defined.

The first issue to be considered is: **computers and reliability**. The concepts of fault avoidance, fault removal and fault tolerance are penetrated. Lastly **computers and safety** is discussed. Hazard identification, causes of hazards and safety by design will be covered in depth.

2 BACKGROUND

The computer and its software is sometimes called a machine-element. Machine-elements can, for example, in a car be a screw, a gearbox or a computer; that somehow has a function in the system – the machine. Computers do, however, not behave like usual physical machine elements.

The computer is a general purpose machine that when given a program to execute, in effect becomes the machine it is designed (programmed) to be.

Trivially explained: using a computer we no longer need to build intricate mechanical or analog devices to control, for example, a washing machine. We simply need to write down the "design" of the controller using instructions to accomplish the intended goal. These instructions are then made available to the computer, which while executing the instructions in effect becomes the washing machine controller. The entire manufacturing phase¹ has been eliminated from the life cycle of the product. Simply put, the physical parts of the system, i.e. the computer, can be reused, leaving only the specification, design and verification phases [Lev95].

The immense flexibility of computers and their ability to perform complex tasks efficiently, have led to a great increase in their use, not even saving their introduction into potentially dangerous systems.

2.1 How computers and software differ from other machine elements

Software has a not so flattering reputation of always being late, expensive and wrong. A question is often asked: "what is so different about software engineering? Why do not software engineers do it right, like traditional engineers?"

These unfavorable questions are not entirely uncalled for; the traditional engineering disciplines are founded on science and mathematics and are able to model and predict the behavior of their designs. Software engineering is more of a craft, based on trial and error, rather than on calculation and prediction.

However, this comparison is not entirely fair, it does not acknowledge that computers and software differ from regular machine-elements on two key issues:

- (1) They have a discontinuous behavior and
- (2) software lack physical restrictions (like mass, energy, size, number of components) and lack structure or function related attributes (like strength, density and form).

The sole physical entity that can be modeled and measured by software engineers is *time*. Therefore, there exists sound work and theories regarding modeling and verification of systems' temporal attributes [Pus89, Ram90, Xu90].

The two properties (1) and (2) are both blessings and curses. One blessing is that software is easy to change and mutate, hence the name software – being soft. A curse is that complexity easily arises. Having no physical limitations, complex software designs are possible and no real effort to accomplish this complexity is needed. Complexity is a source for design faults. Design faults are often due to failure to anticipate certain interactions between a system's components. As complexity increases, design faults are more prone to occur when more interactions make it harder to identify all possible behaviors.

Software is often found in control systems, which makes it infeasible to compensate for design faults by providing a design margin in the same manner as a physical system can be designed to have safety margins. A bridge can be designed to withstand loads far greater than any it should encounter during

¹ That is, the actual physical component – the computer (CPU, Memory and I/O) needs not to be manufactured again for every new software design. Comparatively this is not possible considering, for example, an airplane design. The design will not be functional until manufactured.

normal use. Software on the contrary, must for example, if controlling a brain surgery robot do things exactly right. This adheres to software not being a physical entity (it is pure design) it cannot be worn-out or broken. All system failures due to errors in the software are all design-flaws; built into the system from the beginning. Physical designs can of course also suffer from design flaws, but since complexity is the key factor here they are more likely to occur in software.

A plausible counterpart, nevertheless to overengineering in the software context, is defensive programming using robust designs. Every software module has a set of pre-conditions and post-conditions to ensure that nothing *impossible* happens. The pre-conditions must be valid when entering the software module and the post-conditions must be valid at the end of execution of the module. If these conditions are violated the program should do something *sensible*. The problem is that if the *impossible* do happens, then the design must be deficient and a *sensible* local action might have a very suspect global effect. This is intrinsic to all interacting complex systems. A local event might have a very global effect.

Another overengineering method that does not work in the context of software design, as it does in the physical world, is fault-tolerance by redundancy. Physical parts can always succumb to manufacturing defects, wear, environmental effects or physical damage. Thus, it is a good idea to have spares handy that can replace defective components. In order for a redundant system to function properly it must by all means avoid common mode failures. For example, two parallel data communication cables were cut 1991 in Virginia, USA. The Associated Press (having learned from earlier incidents, had concluded that a spare could be a good idea) had requested two separate cables for their primary and backup circuits. Both cables were cut at the same time, because they were adjacent [Neu95,p.16].

Design faults are *the* source for common mode failures, so fault tolerance against design faults seems futile. An adaptation of the redundancy concept has, however been applied to software, it is called *N*-version programming and uses *N* versions of dissimilar software produced from a common specification. The *N* versions are executed in parallel and their results are voted upon. Empirical studies have, unfortunately concluded that the benefit of using *N*-version programming is questionable. More on this issue later, but a clue is common mode faults in the requirements specification and the way humans think in general.

The application of defensive programming and redundancy may sometimes be of benefit but mostly the system designer is left only with the choice of eliminating all design faults or at least those with serious consequences. In addition some evidence must also be produced that assures that the design faults have been successfully eliminated.

The task of considering all behaviors of a system and all the circumstances it might encounter during operation might be intractable. Physical systems can be tested and measured. There often exist piece-wise continuous relationships between the input and the output of a system. Only a few tests, for each continuous piece, needs to be performed. The behavior of the system intermediate to the samples can be interpolated. Thus the number of behaviors to be considered is reduced. This property does sadly not apply to computers' software. Here is the second case (page 4), where computers and software differ from regular physical systems. They are discontinuous or discrete. Every input is either discrete or is made discrete. The input then affects a multitude of discrete decisions (the software) or computations. The software's execution path changes for every decision depending on whether or not a condition is true. The output is discrete and depends totally on which execution path the software takes. For example, a simple sequential list of 20 if-statements may, in the *worst* case, yield 2^{20} different behaviors due to 2^{20} possible execution paths. A small change in the input can have a severe effect on which execution path is taken, which in turn may yield an enormous change in output. Since all computations are discrete (discontinuous) it is not possible in general to assume that there exists a continuous relationship between the input and the output. Quantization² errors are propagated and boundaries to the representation of numbers can affect the output.

Approaches to deal with the problem of having to consider all behaviors have evolved. They either try to statistically model and test the software or use some analytical method in conjunction with testing. An example of an analytical method is fault tree analysis, which is used to show that the program does nothing seriously wrong. These issues will be dealt with in more detail later in this document.

² Research in the field of control theory have however dealt with the problem of quantization errors.

Other important issues are:

Software, which is verified to comply with the requirements (100% correct), can still fail when the requirements themselves are incomplete or ambiguous.

If software, against all odds, could be designed error free, the host of the software, the computer, could still experience failures induced by the environment and subsequently fail the system. Here fault-tolerance can come in handy; dealing with transient and permanent hardware faults.

When a computer system is used in a safety-critical application, the system can still be unsafe although a 100% correct specification and design is verified. This can occur when the requirements have omitted to state what should not happen regarding safety.

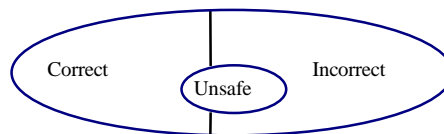


Figure 1. The system's state-space. Of the total number of states in the system some are correct and some erroneous, and of these states some are hazardous.

Since computer related problems, relating to safety and reliability, have just recently been of any concern for engineers, there exists no holistic engineering knowledge of how to construct a safe and reliable computer based system. There exists only bits and pieces of engineering knowledge and no silver bullets that can handle everything. Some people do nonetheless, with an almost religious glee, decree that their method, principle or programming language handles or kills all werewolves (bugs). It is certain that more research and contemplation over these issues are needed.

3 BASIC VOCABULARY

The development of safety-critical systems is a multi-disciplinary task, and it is therefore imperative that a common vocabulary is defined. Computer scientists often redefine standard engineering terms with confusion, misunderstanding and indirectly accidents as a result.

The scope of this section is to define some basic terms that are used throughout this document. The vocabulary is intended to be consistent with engineering terminology and may conflict with some computer science definitions[Tan95][Neu95,p.12]. The definitions are extracts from [Lev95,Lap92] and are compliant with the standard IEC1508.

3.1 Definitions

Definition. Reliability is the probability that a piece of equipment or component will perform its intended function satisfactorily for a prescribed time and under stipulated environmental conditions [Lev95].

Reliability is often quantified by MTTF – Mean Time To Failure. Say, 115 000 years or as a frequency 10^{-9} failures/hour.

Definition. Availability is the probability that the system will be functioning correctly at any given time.

Availability is usually quantified by $1 - MTTR/MTTF$, where *MTTR* is the mean time to repair the system and *MTTF* the mean time to failure.



Figure 2. Cause consequence diagram of fault, error and failure.

Definition. A **failure** is the nonperformance or inability of the system or component to perform its intended function for a specified time under specified environmental conditions [Lev95].

A failure is a behavior or an event.

Failures in physical devices are often divided into two categories:

- **Systematic failures.** A systematic failure is caused by a design flaw; a behavior that does not comply with the intended, designed and constructed systems goal.
- Failures that are the result of a **violation** upon the original design. Such failures may be caused by environmental disturbances, wear or degradation over time.

Definition. An **error** is a design flaw or deviation from a desired or intended state [Lev95].

An error might lead to a failure unless something constructive is done. A failure in turn might lead to a new erroneous state.

Abstractions or designs that do not execute, but have states like models, diagrams, programs, etc., do not fail, but can be erroneous. Failures occur when the designs are realized into concrete machines and the machines operated.

Software does not fail; it is a design for a machine, not a machine or physical object. The computer can however fail, either due to failures in the hardware or due to errors (bugs) in the design (program) when it is executed.

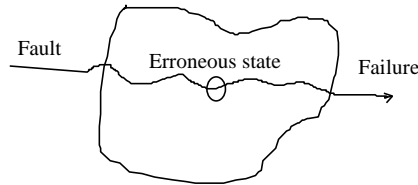


Figure 3. An execution thread passes through an error that results in a failure.

Definition. A **fault** is the adjudicated cause for an error [Lap92].

Generally a failure is a fault, but not vice versa, since a fault does not necessarily lead to a failure.

A distinction can be made between primary faults, secondary faults and command faults [Lev95, pp. 173]:

- **Primary faults** (and failures).
A primary fault (failure) is caused by an error in the software so that the computer output does not meet its specification.
- **Secondary faults** (and failures).
A secondary fault occurs when the input to a computer does not comply with the specification. This can happen when the computer and the software is used in an environment not designed for.
- **Command faults.**
Command faults occur when the computer delivers the correct result but at the wrong time or in the wrong order. This is typically a real-time fault.

The temporal behavior of faults can be categorized into three groups:

- **Transient faults.** A transient fault occurs once and subsequently disappears. These faults can appear due to electromagnetic interference, which may lead to *bit-flips*.
- **Intermittent faults.** An intermittent fault occurs time and time again - and disappears. These faults can happen when a component is on the verge of breaking down or, for example, due to a glitch in a switch.
- **Permanent faults.** A permanent fault that occurs, stays until removed (repaired). A fault can be a damaged sensor or a systematic fault – typically a programming fault.

Definition. Dependability is the trustworthiness of a system such that reliance can justifiably be placed on the service it delivers [Lap92].

Dependability is an abstraction that comprises the measures:

- **Reliability**³
- **Availability**⁴. This measure is not relevant in systems where no safe-state exists. For example, an airplane pilot will not benefit from an availability measure of 99% if the system goes down during in-flight.
- **Safety**⁵
- **Security** The concept of security deals with maliciously intended actions, whereas the safety concept deals with well intended actions, although with dangerous consequences. The scope of this document does not encompass security, as stated in section 1.
- and perhaps some other envisionable measure.

It can be convenient to have a common abstraction, like dependability, but it has been argued that by combining these measures (or qualities) understanding and control of the system is inhibited. Although there might be some commonalties and interactions, the qualities should be treated separately from each other or otherwise required tradeoffs may be hidden from view. For example, a reliable system is not necessarily safe (a computer guided missile may reliably destroy a friendly airplane.) A safe system is not necessarily available (the safest airplane is the one that never flies.) It is possible to redesign a time

³ Definition on page 8.

⁴ Definition on page 8.

⁵ Definition on page 10.

driven system to an event driven system in order to gain better responsiveness. As a consequence the risk for overload will increase and system safety decrease. (Contemplate, e.g., a fly-by-wire aircraft where this would not be a satisfactory solution.)

Definition. An **accident** is an undesired and unplanned (but not necessarily unexpected) *event* that results in (at least) a specified level of loss [Lev95].

Example. During the 1991 Gulf War, the software controlling the Patriot missile's tracking system failed, preventing the missiles from seeking and destroying incoming Scud missiles. The failure led to the death of 28 American soldiers. The computer software was performing according to specifications. These specifications assumed that the system would be restarted often enough to avoid accumulated inaccuracies in the time keeping mechanism. But the system was used in an unintended way, being left up too long without restart, leading to the tragic failure [Wie93].

Definition. A near miss or **incident** is an *event* that involves no loss (or only minor loss) but with the potential for loss under different circumstances [Lev95].

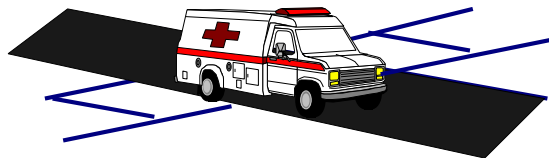


Figure 4. An ambulance is passing through a railroad crossing.

For example, a car is passing through a railroad crossing. Due to a failure in the controlling computer, the signaling system is not working (stuck on green light) and the level-crossing gate is open. It should be shut and the signal red. No train is passing – luckily, so the driver can unwittingly keep his car and life.

To constructively hinder accidents we need to know something about the precursors, the hazards and control these hazards.

Definition. A **hazard** is a *state* or a set of conditions of a system (or an object) that, together with other conditions in the environment of the system (or object), will lead inevitably to an accident (loss event) [Lev95].

A hazard is defined with respect to a system's or a component's environment.

For example, consider the railroad crossing again. The car is in the middle of the crossing and the signaling system has failed. The state is now hazardous or dangerous.

A hazard has two properties:

- **Severity** – the worst accident that can happen.
- **Likelihood** of occurrence.

The two properties combined is called the **hazard level**.

Definition. **Risk** is the hazard level combined with (1) the likelihood of the hazard leading to an accident (sometimes called *danger*) and (2) hazard exposure or duration (sometimes called *latency*) [Lev95].

For example, consider the railroad crossing again given the same situation. The car stops in the middle of the crossing. The risk thus increases according to (2), which is trivially true.

Definition. **Safety** is the freedom from accidents or losses [Lev95].

Safety is here meant to be absolute. Although nothing can be totally safe it is more constructive to aim for total safety.

4 COMPUTERS AND RELIABILITY

Software, which is a frozen thought, a manifestation of intellectual work will during the development become flawed. Physical systems suffer from environmental damage and material imperfections; software only suffers from mental imperfections.

When a computer executes a faulty program the evidence of the fault, the error could lead to a failure. When a system cannot meet its specification it fails. The program may *crash*, enter an erroneous state and untimely terminate. The program can also *hang*, i.e., appearing to freeze up or come to a grinding stop. The undesired occurrence of a failure is an event, yielding an output not complying with the requirements. The program may continue, and the succeeding results could be perfectly correct.

The reliability of software is deemed worse, the higher the failure rate.

4.1 What causes system failures?

1. Faults in the requirement's specifications, design specifications ...

This is inherently a software problem. More than half of all failures can be traced to the requirements and specifications [Voa95,Lev95,Lut92].

Misunderstandings between the developer and the customer may lead to an awry understanding of the problem or the customer's expectations. The developer may incorrectly interpret those expectations into technical requirements.

The customer asks for a vehicle. The customer wants a motorcycle; the developer builds a car.

2. Faults introduced in the design and implementation phases.

- The requirements could be inadequately interpreted. These interpretations could be performed by either man or tool. For example, a compiler.
- The design may be deficient. Due to mental discrepancies the design, which should solve the problem, is flawed.
- The implementation – the program, may suffer errors. For example *typos* or wrongful use of syntax.

A common measure of program quality is the number of encountered faults per 1000 lines of code (Kloc). These faults do, however not state anything about the consequences of their existence.

3. Faults in the hardware.

The risk of using contemporary high level languages for design of hardware (VHDL), is that systematic⁶ faults will burden the hardware; just like software. There are empirical evidence showing that VHDL code has about 1 fault per 8000 lines of code.

4. Mistakes by the operator.

System failures can be caused by human-error. This is of course trivially true since all systems are operated and built either directly or indirectly by man.

⁶ Definition of systematic failures on page 8. Analogous to systematic failures there are systematic faults. Remember fault – error – failure ...

4.2 The nature of software and its environment

Achieving reliability and safety is hard, but what is even tougher is assessing those qualities [Voa95].

Software which is pure design is deterministic, i.e., for specific circumstances (input, time, environment and internal state) the computer will always deliver the same output. If an error is afflicting the software it will always lead to a failure if the wrong circumstances arise. The occurrence of a circumstance is however not deterministic, but related to a stochastic process – namely the sequence of inputs and interactions with the environment. The manifestation of an error – the failure, does thus lend it self to be assigned a probability of occurrence. To say that a piece of software has a failure probability rate of maximum 10^{-9} failures per hour is to say that the probability for a specific input sequence to occur, leading to a failure is less than 10^{-9} per hour.

The number 10^{-9} failures/hour is frequently used as a measure by which safety-critical systems must comply. For example, catastrophic failure conditions in aircraft, preventing safe flight and landing, must be extremely improbable. That is, so unlikely that they are not anticipated to occur during the entire operational life of all air planes of one type [FAA88,§6.h,§9.e]. A little arithmetic suggests 10^7 hours as the operational life of an aircraft fleet, and hazard analysis might typically reveal ten potentially catastrophic failure conditions in each of ten systems on board the aircraft – so the maximum allowable failure rate for each is about 10^{-9} per hour [Lt82,p.37][Rus95b,p.15]. The probability is applied to complete (sub)system failure, not to any software the system might contain. Numerical estimates of reliability are not assigned to airborne software, but gives an idea of the quality needed[DO178B,sub. 2.2.3].

There are limits to the extent that experimental statistical methods allow us to gain confidence in computer based systems. The smallest failure rates that practically can be asserted are several magnitudes off the 10^{-9} that is required in safety-critical systems[Rus95b].

It is difficult to construct good test-profiles. Test scenarios used to evaluate reliability must be good approximations of the operational profile, i.e., the types, frequencies and distributions of inputs that a system receives during operation.

- To evaluate systems with, for example a 10^{-9} requisite, millions of very rare scenarios that each and every one only occur once every billion times must be devised. The reason is that catastrophic failures usually happen when several very rare events occur at the same time. The mismatch between the test- and operational profiles in these remote areas can lead to an awry understanding of a systems reliability.
- The difficulty to reproduce the operational profile for rare events, and the time required to perform fault-injection and "all-up" tests, limits the failure rates that can be verified empirically to about 10^{-4} . This is several magnitudes off the 10^{-9} required for safety-critical systems [Rus95b].

The problem of assessing the quality of software has lead people to believe in approved processes, i.e., processes producing correct software. The problem however, is how to assess the process. It might be harder to verify that the process always produces the correct software than to verify the correctness of the process product — the software. There are mathematical proofs stating that an automata (a program) cannot decide if any arbitrarily chosen automata will ever terminate [Tur36]. This has the implication that it is not possible to verify that a process will always produce the correct result given any problem.

4.3 Construction of Dependable Systems

How can we build dependable systems such that reliance can justifiably be put on them.

There are three main approaches as described by Laprie [Lap92]:

- **Fault avoidance** – How to avoid faults by design, i.e., try to build the system correctly from the beginning
- **Fault removal** – How to reduce, by verification, the presence of faults.
- **Fault tolerance** – How to provide correct function despite presence of faults.

In order to be compliant with earlier definitions in this document these approaches should be named: error avoidance, error removal and error tolerance, but since their names and concepts are so well established we use them anyway. Be aware of the difference though.

Laprie has in addition defined a concept of **fault forecasting**, which concerns the assessment of reliability in an already existing product. This concept is also intended to include the evaluation of the consequences of faults. In this document the consequences of faults will be covered in the computers and safety section.

Most safety-critical systems demand a failure rate of less than 10^{-9} failures per hour. Software testing (fault removal) may alone uncover and reduce the failure rate of computer systems to about 10^{-4} failures/hour. It has been argued that perhaps only an improvement factor of 10 may be possible using fault-tolerant technique on software such as N-version programming [Bow92]. The actual benefit of using N-version programming has also been vividly debated [Kni90]. This gives us a figure of about 10^{-5} , which is four magnitudes off the 10^{-9} .

In general, to construct an ultra-dependable system or as often referred to in this document: safety-critical systems (10^{-9} failures per hour or less), it is believed that a combination of these approaches must be used.

4.3.1 Fault Avoidance

Fault avoidance implies the set of methods used for producing programs that are, by design free of faults. Fault avoidance includes the use of formal methods, semi-formal methods, structured methods and object-oriented methods. What they all got in common is to impose discipline and restrictions on the designers of a system. More or less effectively they introduce virtual physical laws that hinders the designers from making too complex designs and provide means to model and predict the behavior of their designs.

Just as traditional engineers can model their designs with different kinds of continuous mathematics, formal methods attempt to supply the computer software engineers with mathematical logic and discrete mathematics as a tool.

Definition. A **formal method** is a method that has a complete basis in mathematics.

The other methods are well defined but do not have a complete basis in mathematics for description of the designs behavior.

Technically the most significant difference between, for example, formal methods and structured methods⁷ is that formal methods permit behavior to be specified precisely where as structured methods only allow structure to be specified precisely [Bar92].

Formal Methods

Formal methods can be put to use in two different ways [Bar92]:

1. They can be used as a syntax to describe the semantics of specifications which are later used as a basis for the development of systems in an ordinary way.
2. Formal specifications can be produced as stated by (1) and then used as a fundament for verification (proof) of the design (program).

In (1) the mathematics establish a framework for documentation. The benefits of such a formalism include: precision, abstraction, conciseness and manipulability. Consistency checking, automatic generation of prototypes or animation, and derivations by means of proof are different kinds of manipulations that can be performed. In the second case the same benefits accumulate, but with the possibility to prove equivalency of program and specification, i.e., to prove that the program does what it is specified to do. This stringency gives software development the same degree of certainty as a mathematical proof [Bar92].

⁷ Structured methods does in computer science correspond to rules of thumb for guidance of design decisions.

[Bar92] distinguish between five classes of formal methods:

- **Model based approaches** such as Z and VDM (Vienna Development Method) allow explicit definitions of the system state and operations which transform the state but gives no explicit representation of concurrency .
- **Algebraic approaches** allow implicit definitions of the internal behavior of systems but do not define the state of the system (i.e., allow formal proof of algorithms) and do not again, give any explicit representation of concurrency.
- **Process algebras** (e.g., CSP, CCS) allow modeling of concurrency in systems and represent behavior by means of constraints on allowable observable communication between processes.
- **Logic based approaches** (e.g., temporal and interval logic, Q-model) allow specification of timing behavior and a low level specification of the system in terms of agents and actions.
- **Net based approaches** (e.g., Petri-nets and Predicate transition nets) gives an implicit concurrent model in terms of data flow, and conditions that allow data flow about a network.

Unfortunately cannot a proof (when possible) guarantee correct functionality or safety. In order to perform a proof the *correct* behavior of the software must first be specified in a formal, mathematical language. The task of specifying the correct behavior can be as difficult and error-prone as writing the software [Lev86,Lev95]. In essence the difficulty comes from the fact that we cannot know if we have accurately modeled the "real system", so we can never be certain that the specification is complete. This distinction between model and reality attends all applications of mathematics in engineering. For example, the "correctness" of a control loop calculation for a robot depends on the fidelity of the control loop's dynamics to the real behavior of the robot, on the accuracy of the stated requirements, and on the extent to which the calculations are performed without error.

These limitations are however, minimized in engineering by empirical validation. Aeronautical engineers believe fluid dynamics accurately models the air flowing over the wing of an airplane, because it has been validated in practice many times [Rus95a,pp.284]. Validation is an empirical pursuit to test that a model accurately describes the real-world. The same dual process applies to computer science where it is better known as verification and validation (V&V). Verification and validation goes hand in hand and does not exclude one another.

Nonetheless, using formal methods to verify correspondence between specification and design does seem like a possible pursuit to gain confidence. The fact that more than half of all failures can be traced to the requirements and specifications [Lev95, Lutz92, DeM78, Ell95] gives the application of formal methods some weight. Using mathematical verification of software to any great extent is however currently intractable, but will probably be feasible in the future. Formal methods, as of today, are exceedingly abstract and non intuitive, requiring extensive training of the users. There exists over 300 kinds of formal methods and notations, so there definitively is a lack of consensus in the formal methods community [Sha96]. Most safety-critical systems are in essence multi-disciplinary, which requires all stakeholders of a project to communicate with each other. Formal methods must thus be made more easy to comprehend and use.

Further, disadvantages with current formal methods are their inability to handle timing and resource inadequacies, like violation of deadlines and overload. There are yet again, other methods that can complement formal methods with verification of timing, e.g., execution time analysis [Pus89] and scheduling [Ram90,Xu90].

There are standards that advocate the use of formal methods. For example, [MoD95, MO178B, IEC1508].

Formal methods are no silver bullets. They are not *the single solution*. Formal methods in general, like other informal methods (SA/SD, OOA/OOD) does impose discipline on the users and make them think in rational and disciplined ways – helping to increase understandability, finding problems and in boosting confidence.

4.3.2 Fault Removal

Fault removal implies the set of methods used to remove faults from existing programs. Fault removal includes, for example, the combined use of formal specification and dynamic verification.

Software does not wear out over time. It is therefore reasonable to assume that as long as errors are uncovered reliability increases for each error that is eliminated. This notion of course rely on maintenance not introducing new errors. Littlewood et.al. [Lit73] elaborated on this idea and developed their reliability growth model. According to the reliability growth model the failures are distributed exponentially. Initially a system fails frequently but after errors are discovered and amended the frequency of failures decreases. A problem with this method is that it would take years to remove a sufficient amount of errors to achieve a critical standard of reliability. For safety-critical systems where the failure rate is required to be lower than 10^{-9} failures per hour this is an intractable task [Fen95] since testing has to be performed for at least 115 000 years.

If we have a reliability requisite of 10^{-4} and testing is undertaken two things can occur:

- (1) failures are observed in operation within a reasonable amount of time,
- (2) no failures are observed in operation within a reasonable amount of time.

If (1) happens the system is apparently flawed and must be re-designed. If the other case (2) occurs, i.e., no observations are made (which is more positive, one may think), it tells us ludicrously little of the system's reliability. If a system has been operational for n hours then all we can conclude is that the reliability of the system for the next n hours is 0.5 [Lit91]. So if a system has been fully operational for 10 000 hours without failure the best we can predict about the future is that the system has a probability of 0.5 of not failing in the next 10 000 hours, i.e., we can only be 50% confident that the 10^{-4} requirement is met.

What makes matters even worse is the fact that more than half of the errors in a system are due to ambiguous or incomplete requirement specifications. The intention of testing is often to verify that a specific input will yield a specific output, defined by the *specification*. Thus the confidence gained by testing software is limited.

If the failure rate of software is plotted, a graph could look like this (figure 5).

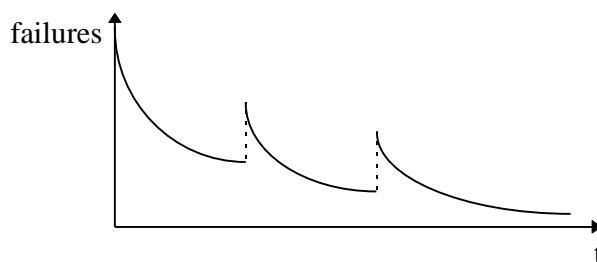


Figure 5. The failure rate of software decreases when errors are removed. New errors are introduced when the software is changed

The graph (figure 5) shows that initially there is a high failure rate, but after a period of *burn-in* or *debugging* the failure rate decreases. Every new change to the software will require a new period of burn-in since in reality new errors may be introduced during maintenance.

Just like formal methods, testing cannot be the *sole* means to gain assurance in a safety-critical system. When dynamic verification (testing) and static verification (e.g., formal methods) are used to augment each other, testing should focus on the aspects not covered sufficiently by the static analysis. For example, timing and overload issues should be addressed by testing, which are difficult to verify in static analysis [Lev95].

4.3.3 Fault Tolerance

A vast array of techniques exists for increasing system reliability. Each technique is paired with an assumption about the potential faults that need to be accommodated for, i.e., a fault hypothesis. The most trivial assumption deals with transient faults that will occur sporadically and then disappear; in such cases, error detection and retry may be adequate. At the other extreme of the spectrum, are the assumptions of essentially unrestricted failure modes, namely the Byzantine failures [Lam85]. Fault tolerance has also been applied to accommodate for design deficiencies.

Fault tolerance involves design of a system or a subsystem in such a way that, even if certain types of faults occur, a fault will not propagate into the system. Several techniques have been devised and used to improve reliability and specifically to provide fault-tolerance.

Fault tolerant designs can be divided into two categories robust designs and redundant designs:

Robust designs

Robust systems are designed to cope with unexpected inputs, changed environmental conditions and errors in the model of the external system. A robust design can for example, be a servo feedback loop in a control system.

Generally, software is not designed to be robust since focus is primarily on what the system should do, and not on what it should not do; as a consequence does testing (usually) not cover abnormal inputs or actuations.

In order for a system to be robust, the events that trigger state changes must satisfy the following [Lev95]:

1. Every state must have a behavior (transition) defined for every possible input.
2. The logical *OR* of the conditions on every transition out of any state must form a tautology. A tautology is a logically complete expression, i.e., always true.
3. Every state must have a software behavior (transition) defined in case there is no input for a given period of time (a time-out or exception upon violation of deadline).

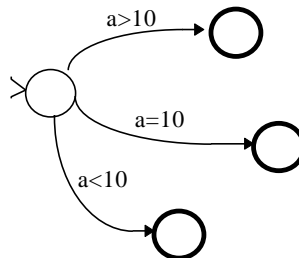


Figure 6. This state-machine forms a tautology for an event **a** with a requirement that **a** must be greater than 10.

Applying robust designs to accommodate for design flaws is, however not satisfactory. It can be applied however: every software module have a set of pre-conditions and post-conditions to ensure that nothing *impossible* happens. The pre-conditions must be valid when entering the software module and the post-conditions must be valid at the exit of the module.

For example,

```
    assert <pre-condition>;
        action 1;
        .
        .
        .
        action n;
    assert <post-condition>;
```

If these conditions are violated the program should do something *sensible*. The problem is that if the *impossible* does happen, then the design must be deficient and a *sensible* local action might have a very suspect global effect. This is intrinsic to all interacting complex systems. A local event might have a very global effect.

This method is however excellent when debugging a system trying to find residual faults.

Redundant designs

Physical parts can always succumb to manufacturing defects, wear, environmental effects or physical damage. Thus, it is a good idea to have spares handy that can replace defective components.

Redundancy in computers had its infancy with the first *real-time* computer used in the Whirlwind project at MIT in 1944. The large number of vacuum tubes used in the computer, had to be replaced every 20 minutes because they had a very short lifetime. This led to the use of fault-tolerance to increase reliability. They designed the system so that they could detect weak tubes before they failed and re-directed signals to other components, and then replace the weak tubes; thus they could continue operating despite occurrence of hardware failures [Bow92]. The concept of redundancy has ever since been a means for the computer community to gain better reliability or availability.

Redundancy has been found to be most efficient against random⁸ failures and less effective against design errors.

A design can use redundancy in [Lev95]:

- *information*
- *time*
- *space* and
- *model*.

For example, checksums or double-linked lists are/make use of redundant *information*. Data structures that make use of redundant information are usually referred to as robust data structures. If, for example, a double linked list is used – and one link is corrupted, the list can be regenerated using the other link.

Redundancy in *time* can be realized for example, by allowing a function to execute again if a previous execution failed. *Redundancy* in space is called *replication*. The concept is founded on the assumption that parts that are replicated fail independently. A common use of replication is for example, to use several sensors, networks or computers in parallel. *Model-based* redundancy uses properties of a known model, e.g., physical laws. If for example, a revolution counter for a wheel, in a four wheel drive vehicle fails, it is possible to estimate the revolution speed based on the other wheels' speeds.

⁸ That is, failures that are not due to design errors but rather due to failures induced by the environment. For example, bit-flips.

General steps

Tolerance of component failures can be handled by performing these steps [Kim94]:

- Firstly, to be able to recover from a fault before it leads to a failure, the manifestation of it, the erroneous state must first be detected.
- Secondly, the cause to the erroneous state must be isolated at the level of replaceable or repairable components. The cause may be found in hardware or software. Since software runs on hardware it is logical to check the hardware for causes first. If a hardware module is found to be lacking in performance (malfunctioning) appropriate reconfiguration or repair actions can be taken by utilizing spares.
- Thirdly, the system needs to be restored to a state where the system can resume execution of the application.

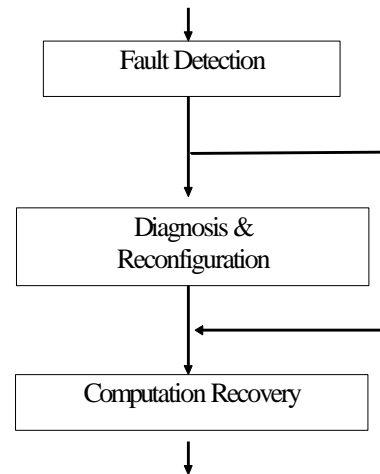


Figure 7. Steps to achieve fault-tolerance.

Design principles for redundancy

In order for a redundant system to function properly it must by all means avoid common mode failures. For example, the loss of all the hydraulic systems in the Sioux City DC-10 crash; the separate and redundant hydraulic systems were routed through a single space near the tail engine and all were severed when that engine disintegrated [Rush95,p.11].

Here are a few basic principles on how to realize redundancy.

Triple Modular Redundancy (TMR)

The essence of *triple modular redundancy scheme* is to use three copies of a component. They may be nodes in a network, sensors, actuators, CPUs in a single node, tasks or functions. The results from a homogenous triple of them are voted on and a majority decision is made [Kim94]. The components are assumed to fail independently.

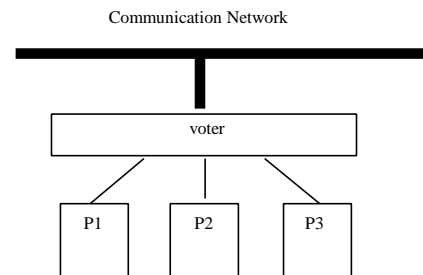


Figure 8. Triple modular redundancy realized by the use of three components and a voter

The Pair-of-comparing-pairs (PCP) scheme

In this approach all critical components like, for example processors and memory are duplicated. Two such pairs are used, one primary and one secondary. The output from the duplicates are compared in order to detect discrepancies. If a discrepancy is detected in the primary pair the secondary pair is given the command. This is realized by letting the comparators communicate with each other [Kim94].

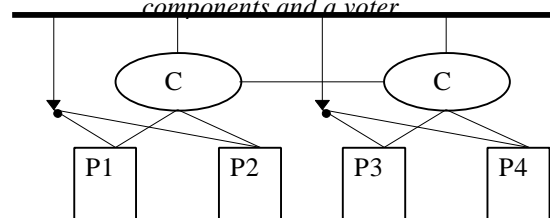


Figure 9. The Pair of comparing pairs scheme – PCP.

The voting N-version scheme – NVP

A method that does not work in the context of design, as it does in the physical world, is fault-tolerance by redundancy.

Design faults are *the* source for common mode failures, so fault tolerance against design faults seems futile. An adaptation of the redundancy concept has, however been applied to software, it is called N-version programming and uses N versions of dissimilar software. A program specification is given and implemented in N different ways. The same input is given to all N versions, and the results from the N versions are compared. If the result is deemed valid it is transmitted to the output. The technique does not eliminate residual design faults but it provides a means to detect and mask design faults before they can affect safety in a negative manner.

Different voting strategies can be applied depending on the application [IEC1508a-7-b17]:

- If the system has a safe state, then it is reasonable to require a consensus between the N versions, otherwise a fail-safe state should be entered. For a simple trip system (a system that should force the system to a safe state when appropriate) the vote should be biased towards the safe state. The safe action is to trip whenever a version demands a trip. This approach typically uses two versions ($N=2$).
- For systems with no safe state, majority votes can be used. For cases where there exists no consensus, probabilistic methods can be utilized in order to maximize the probability of choosing the right input value.

Empirical studies have, however concluded that the benefits of using N-version programming are questionable. Knight and Leveson [Kni86] performed an experiment using N-version programming. The result suggested that the asserted reliability was not as good as the reliability models indicated, because it was found that the versions were not independent of each other. The experiment involved two different universities in two different states in the USA and over 20 programming teams participated. The result indicated that common mode failures do occur and thus independence of failures cannot be assumed. The study showed that errors are to a great extent dependent on the original requirements and specifications. An interesting observation was made that even though all the programming teams used different programming languages and algorithms they made the same mistakes – all adhering to the fact that people think and reason in similar ways. Knight and Leveson, having employed statisticians, based their proof on statistical analysis of the confidence interval. The statisticians tested the hypothesis that failures occurring on the same input are independent. The test failed at the 99% confidence interval which implied that independence cannot be assumed. The believed levels of reliability that could be achieved were way off the factual estimate.

The cost of producing N versions is not only N times the cost of producing one version but also N times the cost for maintenance. Perhaps some benefits can somehow be gained by applying N version programming, but would it not be more cost effective to produce one version of software with the resources allocated for N versions?

Design principles for recovery

Whenever a fault is detected (and perhaps amended) there needs to be a computational recovery. There are two general principles for recovery: forward and backward .

Forward Error Recovery

Forward error recovery means detecting an error and continuing on in time, while attempting to mitigate the effects of the faults that may have caused the error. It implies the constructive use of redundancy. For example, temporally or spatially replicated messages, may be averaged or compared to compensate for lost or corrupted data. By utilizing redundancy: Transient faults inherent to the environment or permanent faults due to, e.g., a severed communication cable could be overcome. Another approach is to use error-correcting codes or robust data structures⁹ to correct faults in memory, errors in communications, or even errors in arithmetic operations.

A benefit of using forward recovery is that interactions made with the environment can be dealt with.

⁹ See page 16, redundant designs.

Backward Error Recovery

Backward error recovery means detecting an error and retracting back to an earlier system state or time. It includes the use of error detection (by redundancy, comparing pairs or error-detecting codes) [Kim94] so evasive action can be taken. It subsumes rollback to an earlier version of data or to an earlier system state. Backward error recovery may also include fail-safe, fail-stop and graceful degradation modes that may yield a safe state, degraded performance or a less complete functional behavior.

There can be problems if backward recovery is used in real-time systems. One problem is that interactions with the environment cannot be undone. Another problem is how to meet the time requisites. Here forward recovery is more apt.

A way to implement backward recovery is to use recovery blocks.

The Recovery Block Approach – RBA

The recovery block approach relies just like N -version programming on design diversity and does thus try to compensate for discrepancies in the individual designs.

A recovery block is made up of a number of dissimilar functions produced from a common specification. These functions are named *try-blocks*. The try blocks are executed sequentially. After a try block has executed it must pass an acceptance test. If it does not pass the test the next try block is executed. When a recovery block begins executing, its or the systems state is stored. The point where the state is stored is called a *recovery point*. If a fault is detected either interior to a try block or if the acceptance test fails the system state is restored to what it was at the recovery point and the next try block is executed [Kim94].

```
For example[Kim94],
ensure (acceptance test)
by
  <try block 1>
else by
  <try block 2>
.
.
.
else by
  <try block n>
else
  error
```

Comparison

Similarities with N -version programming (NVP) are striking. The main difference is that N -version programming executes its try blocks in parallel (forward recovery) while the recovery block approach (RBA) executes its try-blocks sequentially. There are benefits with both approaches. The recovery block approach, for example, can be more economical with respect to use of the CPU when only one version needs to be executed at the same time.

Not only is it hard to roll back the system state for, e.g., a mechanical device that has been affected by an undetected erroneous output, but an erroneous software module may have passed information to other software modules, which then also have to be rolled back. Procedures to avoid domino effects in backward recovery are complex and thus error prone, or they require performance penalties such as limiting the amount of concurrency that is possible. In distributed systems, erroneous information may propagate to other nodes and processors before the error is detected [Lev95,p.438].

Both approaches have the same problem regarding specification and design of reasonableness checks. The NVP needs threshold checks in order to make voting possible and the RBA needs reasonableness checks in the acceptance tests. It is hard to decide what is reasonable. Perhaps only for simple mathematical calculations it is possible to find the inverse of a function or a matrix which, when applied on the result and compared with the input can deem the result correct.

The two approaches need not, in practice, be mutually exclusive but can complement each other. For example, envision a pilot flying a not fully functional computer controlled unstable fighter plane. The pilot is trying to avoid incoming missiles while concurrently backward recovery of the armaments computer is being attempted. In such an instant forward recovery is needed to keep the plane flying.

Forward recovery is deemed necessary when [Lev95,p.438]:

-
- Backward recovery procedures fail.
 - Redoing the computation means that the output cannot be produced in time (of vital importance in real-time systems.)
 - The software control actions depend on the incremental state of the system (like using a stepping motor) and cannot be recovered by a simple check point and rollback.
 - The software error is not immediately apparent and incorrect outputs have already occurred.

Both approaches do however, suffer from the same major flaw: they try to compensate for design faults using diverse design. They will thus not be able to recuperate from faults in the requirements specification and are likely to be afflicted by common mode faults relating to how people think in general.

4.4 Remarks

The concepts and methods of fault-avoidance, fault-elimination and fault-tolerance are by them selves not sufficient to produce adequate designs and assurance that a system complies with the requisites set for safety-critical systems. In combination, however, they seem to be able to provide adequate designs and the assurance required. The task of applying them is tough though and requires plenty of the process and from the people applying them.

What have been considered so far has been biased towards reliability. The consequences of failures have not been considered. The subject of safety will follow next.

5 COMPUTERS AND SAFETY

“The most likely way for the world to be destroyed, most experts agree, is by accident. That’s where we come in; we computer professionals. We cause accidents.” [Bor91]

5.1 Distinctions between reliability and safety

Safety is a property of a system just like reliability. When estimating software reliability every failure is considered.

Reliabilitywise these are equivalent:

- A failure resulting in an 'A' on the computer screen instead of a 'B'.
- A failure in a control system for an aircraft causes the plane to crash.

Reliability only quantifies the frequency of failures, disregarding the consequences of a failure. From a safety point of view it is important to consider the consequences of failures, especially the failures that lead to hazards.

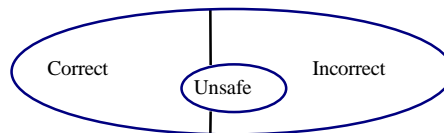


Figure 9. The system’s state-space. Of the total number of states in the system some are correct and some erroneous, and of these states some are hazardous.

Usually, software that is deemed reliable does what it shall.

Software that is deemed safe, does not do what it shall not do, i.e., does nothing that can lead to accidents.

Example. A weapon system should destroy and kill. The property that the system destroys and kills is a reliability property. The property that the software in the weapon system does not destroy and kill friendly forces is a safety property [Voa95].

Example. Assume that the system properties described in previous example are valid. If a torpedo is not launched it is safe but not reliable. If a torpedo is fired, misses the foe and does a U-turn and destroys the submarine which launched it, it is reliable but not safe [Voa95].

It can be argued however, that the requirements are faulty in the examples if this is allowed to happen. Usually requirements are stated in terms of shall-criteria and not in shall-not-criteria.

Anecdote. In order to avoid that a torpedo destroys the submarine that fired it, an American arms supplier was asked to design a torpedo that should self-destruct if it made a 180 degree turn. The problem however, was that when the new torpedo was put to test, it got stuck in the barrel and the commander of the submarine ordered the vessel back to port. It made a U-turn...

Test and specifications

The intention of testing is often to verify¹⁰ that a specific input will yield a specific output, defined by the *specification*. Thus the confidence gained by testing software is limited. It is not possible to disclose faulty or incomplete specifications by testing.

The specification can be hazardous, when requirements regarding safety have been omitted.

Preferably all safety requirements would be included in the specifications. It would then be possible to show that all accidents are failures. Safety related failures would then be a subset of all system failures.

¹⁰ There is a distinction between verification and validation. Laprie [Lap92] means that verification is the process of asserting that we build or design the system correctly, while validation is the process of asserting that we build or design the correct system.

But yet again, the specifications can be incorrect. Software can be 100% correct yet still cause serious accidents.

5.2 Software and hazards

Software by it self is not hazardous. It can be conceived that software will be hazardous when executed on a computer, but even then there exists no real danger. A computer does actually nothing else than generating electrical signals¹¹. In reality it is first when the computer and the software starts monitoring and controlling physical components that hazards can occur. *Thus it is appropriate to define safety as a system property and not as a software property.*

5.3 Hazard analysis

In order to design or assert that a safety-critical system is not only correct with respect to functionality but also safe, a hazard analysis must be undertaken. The goals of safety analysis are related to these tasks [Lev95,289-290]:

1. **Development:** the examination of a new system to identify and assess potential hazards and eliminate and control them.
2. **Operational management:** the examination of an existing system to identify and assess hazards in order to improve the level of safety.
3. **Certification:** the examination of the planned and/or existing system to demonstrate its level of safety and to be accepted by the customer, authorities or the public.

The two first tasks are intended to make the system safer, while the third task has the goal of convincing management and government that the system is safe.

These steps should be performed during the **Development** and **operational management** of the system:

1. Define scope, i.e., which components and data that are to be subjected to the analysis.
2. Identify hazards that singly or in combination could cause an accident.
3. Rank the hazards in order to know, which hazard to attack first.
4. Evaluate the causal factors related to the hazards: (a) Determine how the hazards could occur, their nature, and their possible consequences. (b) Examine the interrelationships between causal factors.
5. Identify safety design criteria, safety devices, or procedures that will eliminate or minimize and control the identified hazards.
6. Find ways to avoid or eliminate specific hazards.
7. Determine how to control hazards that cannot be eliminated and how to incorporate these controls into the design.
8. Evaluate the adequacy of hazard controls.
9. Provide information for quality assurance – quality categories, required acceptance tests and inspections, and items needing special care.
10. Evaluate planned modifications.
11. Investigate accidents and near-miss reports. Determine whether they have validity and, if they do, the cause of the problem.

Certification of the system:

1. Demonstrate the level of safety achieved by the design.
2. Evaluate the threat to people from the hazards that cannot be eliminated or avoided.

5.3.1 The iterative hazard analysis process

The hazard analysis process is both iterative and continuous over time. The process begins with the identification of hazards during the conceptual phase of the system and continues on through out the construction and operation of the system.

¹¹ It is however, possible to think of a scenario where a computer catches fire or falls down on some one.

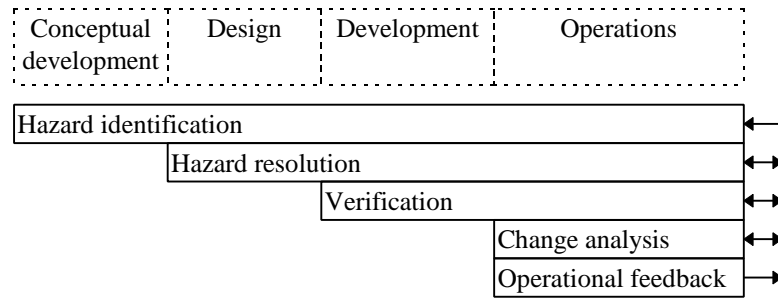


Figure 10. The hazard analysis is iterative and continual [Lev95,p. 293].

It is imperative to start early, so design decisions regarding safety can be made inexpensively and effectively. The task of considering hazard elimination and reduction should begin as soon hazardous problems are discovered. Ideally this should occur before hazardous features are fix parts of the design. As the project continues and the designs are evolving, more in depth tests and analysis may uncover new hazards or eliminate old ones from the scope of consideration.

The safety of the system during operation depends on the correctness of the assumptions and models forming the foundation for the design process. It must be asserted that the system is constructed, operated and maintained in the manner the designers intended it to be. Further it must be asserted that the basic models and assumptions forming the system are not violated by changes in the system. Operational feedback from incidents and accidents should initiate reanalyzes of the system.

If a change is proposed or an unplanned change is monitored during operation then the effects of this change must be analyzed with regard to safety. The analysis must start at the highest level where the changes become visible. Then it must be ensured that the change does not introduce new hazards or affect a hazard that has already been resolved, or increase the severity level of a known existing hazard [Lev95].

The Hazard analysis process has several names, namely: preliminary hazard analysis, system hazard analysis and subsystem hazard analysis. However, individually these analyses only represent different phases in the iterative and continuous process.

Hazard identification

Already in the conceptual stages of a project hazard identification should be initiated. The lists of identified hazards are continually updated with new information about new and old hazards. This process continues through the entire lifetime of the system.

The output from the hazard identification process is used in developing system safety requirements, preparing performance, resource and design specifications, test planning, preparing operational instructions, and management planning [Lev95,p.295].

There are a few techniques for identifying hazards in a system. They make people think in structured and rational ways and make them ask a lot of *what-if* questions.

When a computer controls something, typical hazards can occur due to [Voa95]:

1. Release of energy

- Electromagnetic energy: atomic energy, radio waves, sound, microwaves, light, heat, electric currents, magnetic energy, electrostatic energy, x-rays.
- Stored energy: geopotential energy (e.g., falling objects), strain energy (e.g., springs), chemical energy and mechanical kinetic energy. May, for example, cause loss of control of a vehicle, making a vehicle and an obstacle collide.

2. **Release of toxins.** Harmful substances: poisons, pathogens, environmental pollutants (e.g., affecting air, soil or water) and overdoses. Actually anything can be poisonous when excessively dosed.

3. **Interference with life support:** asphyxia, exposure to contaminants, dehydration, emaciation; pacemaker failure or failure of other life-sustaining equipment.

-
4. **Deception of trusting people:** software can in a subtle way cause hazards by promoting misleading information. A simple example is a traffic light system consisting of red, yellow and green light signals. These lights are not intrinsically hazardous, but when the drivers depend on the traffic light signals, the wrong color at the wrong time can have catastrophic consequences.

Techniques that exist for identification of hazards are for example, HAZOP – Hazards and Operability Analysis or FMEA – Failure Modes and Effects Analysis. These methods are covered on page 27.

Ranking of hazards

It is often desirable to quantify risk to allow ranking of hazards. Ranking helps to prioritize hazards so a sound judgment can be made regarding in which order hazards should be eliminated, reduced or ignored.

For example, if the unit of loss is *the probability for a person to die per year*, it is possible to show (in theory¹²) that the risk of dying by lightning is 5/4 of the risk of dying by a giant hail.

On the other hand if the unit of loss is measured by monetary means an example might look like this: If the probability for an accident is 0.00005, which results in a loss of 100 000 S.kr. then the risk is (according to definition of risk) $0.00005 * 100\ 000 = 5$.

If the probability for another accident is 0.00025, which results in a loss of 40 000 S.kr. then the risk is $0.00025 * 40\ 000 = 10$.

The risk ratio is thus 1:2, in favor of the former.

In many cases it is not possible to express risk solely by monetary means, instead some other non monetarian measure must be used. In those cases it is possible to express risk as an ordered pair: (Probability for accident to occur, Consequence of accident).

The worst possible consequences and probabilities of occurrence of hazards are often categorized. Several standards address this issue among them DoD STD-882, NHB 5300.4 and DOE 5481 (military, NASA and nuclear standards respectively). Here is a compiled example [Voa95] (see next page):

¹² In theory – according to David Parnas – means not really.

CONSEQUENCE

Catastrophic:	People	;death.
	Facilities	;system loss, cannot be repaired, requires salvage or replacement.
	Environment	;severe environmental damage.
Critical:	People	;severe injury/illness; requires medical care (lengthy convalescence and/or permanent impairment)
	Facilities	;major system damage, loss of mission
	Environment	;major environmental damage.
Marginal:	People	;minor injury/illness; requires medical care but no permanent impairment.
	Facilities	;loss of non-primary mission
	Environment	;minor environmental damage.
Negligible:	People	;superficial injury/illness; little or no first aid treatment
	Facilities	;less than minor system damage; disabled less than one day.
	Environment	;less than minor environmental damage.

Figure 11. This is a classification of hazards' consequences.

LIKELIHOOD

Frequent	-Likely to occur frequently during system's life time.
Probable	-Will occur several times during system's life time.
Occasional	-Likely to occur sometime during system's life time.
Remote	-Unlikely to occur during system's life time.
Improbable	-Extremely unlikely to occur during system's life time.
Impossible	-Probability equal to zero.

Figure 12. This is a classification of hazards' likelihood of occurrence.

Example. Risk quantification (frequent, marginal).

Sometimes, or actually frequently, it is hard or not possible to quantify risk (especially in the software context). This might simply make the analyst ignore the risks that are tough to quantify and simply concentrate on the risks that are simple to quantify. In some cases the analysts have come up with numbers like: the probability for the risk to occur is 10^{-9} or even 10^{-12} and have yet ignored hard to quantify safety problems that can occur during installation and maintenance with probabilities of 10^{-2} or 10^{-3} .

Safety is usually a relative measure, defined by acceptable loss. This definition does naturally vary due to whom is subjected to the loss. For example, an employee is probably more likely to have a lower level of acceptable risk than the employer. It used to be that way in the old days anyway. Before the civil war in the USA a worker on the factory floor was considered less valuable than a good slave¹³.

Safety is meant to be absolute. There is however, nothing that is absolutely safe. It is possible, at least in theory, do get killed by a meteorite. But, by not defining safety as absolute, some hazards may not be discovered, when in practice it might actually be possible to eliminate those hazards.

¹³ This was actually the case. The owner had to pay good money for slaves, and feed and cloth them. The factory workers were cheap labor and there were multitudes of them. If anyone got hurt they could quickly be replaced [Lev95].

Causal factors

When hazards have been identified the next step is to conclude what causes and effects are associated with each hazard. It is necessary to trace backwards, from effect to cause, and from effect to cause, and so on.

A cause is a set of circumstances or events that are sufficient for a certain effect to occur. For example, an event is the occurrence of a signal or a transition between two states. The failure of a component is, for example, an event. The component goes from a correct state to an erroneous state. A circumstance is an already existing state that a component is in; the event that caused the transition to the state has already occurred.

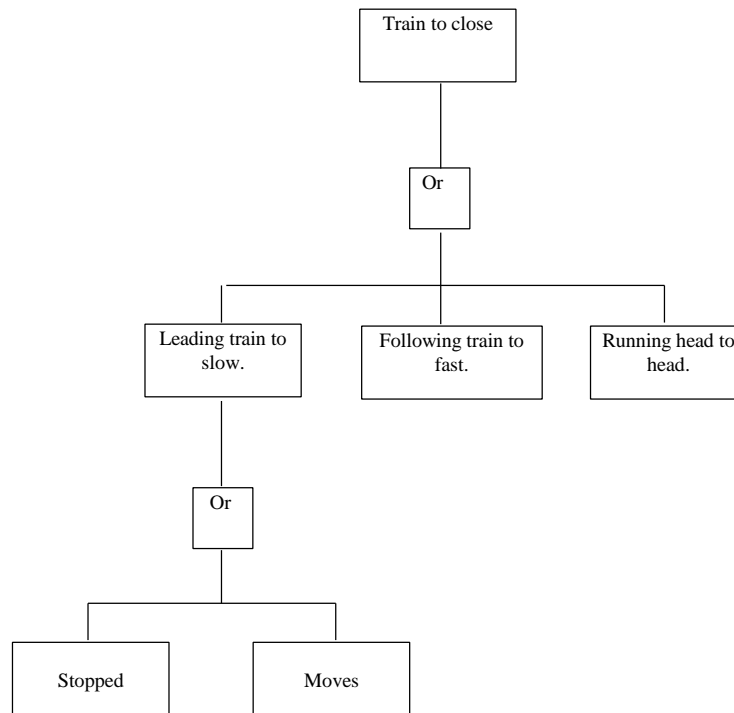


Figure 13. The causes of the hazard: the trains are to close, arranged as a tree of contributing causes [Voa95,p. 118].

The analysis can be performed on system- or subsystem basis. As mentioned previously the hazard analysis process is iterative and continual. The following analyses are performed in order of appearance.

System hazard analysis

This analysis consider the system as a whole and identifies the behavior of the system relating to the interface between components and how the interface between the system and the operator can contribute to hazards.

Subsystem hazard analysis

This analysis consider the individual subsystems' operating and failure modes impact on the system hazards.

Typical subsystems are:

- Power – electricity, pneumatic pressure, ...
- Control – computers and software
- Operators
- Communication – Computer networks
- Sensors
- Actuators
- Propulsion – Engines, Gas,..

This analysis evaluates: primary faults, secondary faults (faulty input) and command faults (timing issues).

Methods

There are several methods that try to identify and/or find the causes of the hazards¹⁴.

- Checklists

The hard earned experience gained from projects are often distilled into a list of common hazards or into a list of questions. A question could be: how does the system handle transient errors due to electromagnetic interference? The list can help stimulate the designers thoughts or what to check for first when debugging a system

- Fault Tree Analysis – FTA

Fault tree analysis is primarily a method for finding causes of hazards, not identifying hazards. The method has a background of wide use in the electronics, aerospace and nuclear industries.

The FTA tries to reduce the number of behaviors that need to be considered in order to get assurance that the design is safe. FTA is an analytical method using proof by contradiction. An undesirable state is specified – a hazard. The system is then analyzed, in a top-down manner, with the objective to find all possible paths that can result in the undesirable state. Boolean logic is used to describe how combinations of single faults can lead to a hazard. It is like detective work in the spirit of Sherlock Holmes. If the deduction cannot find any causes the assumed hazard cannot occur. If causes are identified then preventive actions can be taken. If a quantitative measure is desired, the probability for the top event – the hazard can be computed out of the boolean expression represented by the fault tree. All atomic root causes must then be assigned a probability of occurrence. This is however, a questionable pursuit since not all causes' probability of occurrence can be quantified. For example, how can design errors be quantified? Program errors remain until removed, no design errors appear or disappear during runtime. The use of FTA is better suited when applied qualitatively.

- Several more methods exist like:

- Event Tree analysis – ETA
- Cause Consequence Analysis – CCA
- Hazards and Operability Analysis – HAZOP
- Failure Modes and Effects Analysis – FMEA
- Failure Modes, Effects and Criticality Analysis – FMECA
- Fault Hazard Analysis – FHA
- State Machine Hazards Analysis – SMHA

More information can be found in Leveson [Lev95, chap. 14]

¹⁴ See page 23 for why they should be applied at all.

Here is an example of a fault tree (figure 14). The hazardous state is assumed to be microwave exposure by a microwave oven. The figure 15 below shows the notation used.

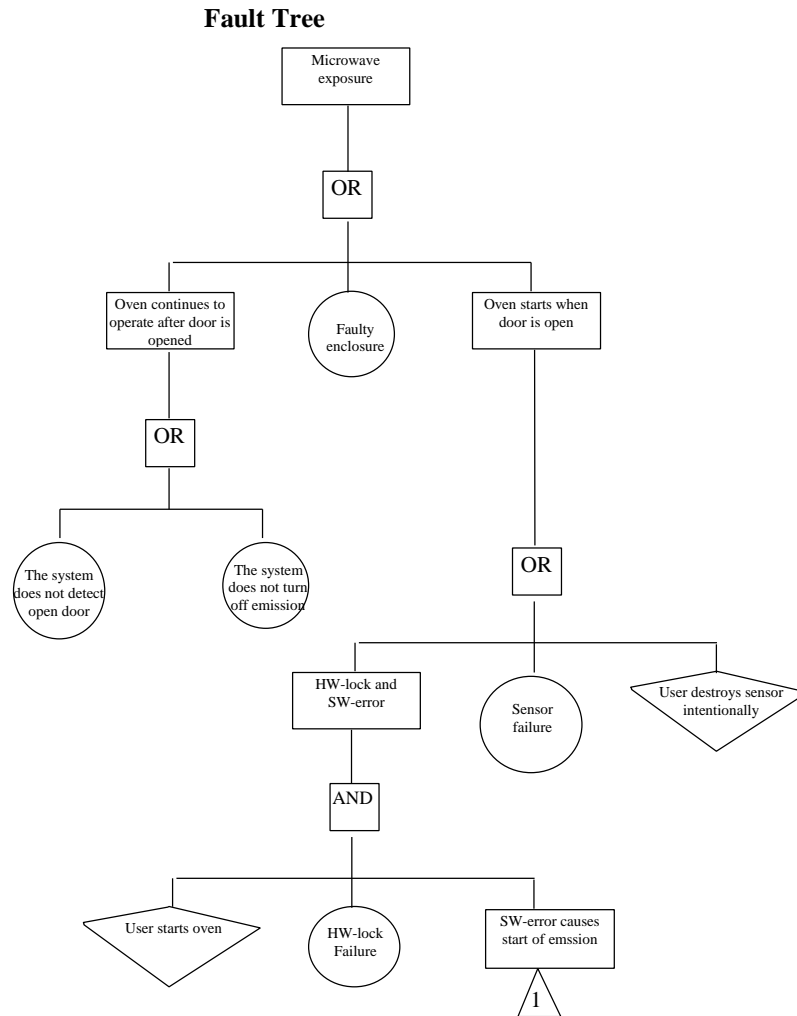


Figure 14. A fault tree for the hazardous condition: microwave exposure [Gow94].









	An event that results from a combination of events through a logic gate.		A condition that must be present to produce the output of a gate (may be used for synchronization).
	A basic fault event that requires no further development.		Transfer
	A fault event that is not developed further, either because the event is not consequential or the necessary information is not available.		OR gate
	An event that is expected to occur normally.		AND gate

Figure 15. Fault tree notation symbols.[Lev95,p. 319]

Software hazard analysis

Software is just like any other component in a system if it while executing can control other components. Software must therefore be part of the system hazard analysis and subsystem hazard analysis phases. The computers behavior and interface towards the rest of the system must be examined for potential system hazards.

If such a hazard is identified, that can be traced to the software, the next step is to identify potentially hazardous designs and code in the software on which a more in depth going analysis can be made.

Later, the software must be evaluated to determine if the software specifications have satisfied the design criteria and if the implementation has improved or degraded system safety or introduced new hazards.

Software Fault Tree Analysis – SFTA

Fault tree analysis can be applied on software. It is however, used in a slightly different manner. SFTA is used for verification, since the code must be written already in order to make a fault tree. Software is often changed and for every new change a new SFTA must be made. Thus the fault trees can with advantage be generated by a tool.

Usually when doing a fault tree analysis, the elements are assigned a probability of failure and using the fault tree the probability for the hazard to occur can be calculated. This is however useless for software. If a design fault is found in a program it is rather eliminated than assigned a probability of causing a failure. This might be useless anyway since software is always deterministic (although the input may behave stochastic). For a specific input and a specific internal state the program will always yield the same result (output).

Here is a piece of sample code and a corresponding fault tree (next page). A hazard is assumed to exist when $x > 100$.

Code

```
if(a > b) {  
    x = f(x);  
}  
else {  
    x = 10;  
}
```

Examining the fault tree it is apparent that the right sub-tree cannot possibly happen. The problem is thus transformed into three sub-problems. The analysis can stop here and the hazards can be avoided by introducing a control when $a > b$ and $f(x) > 100$.

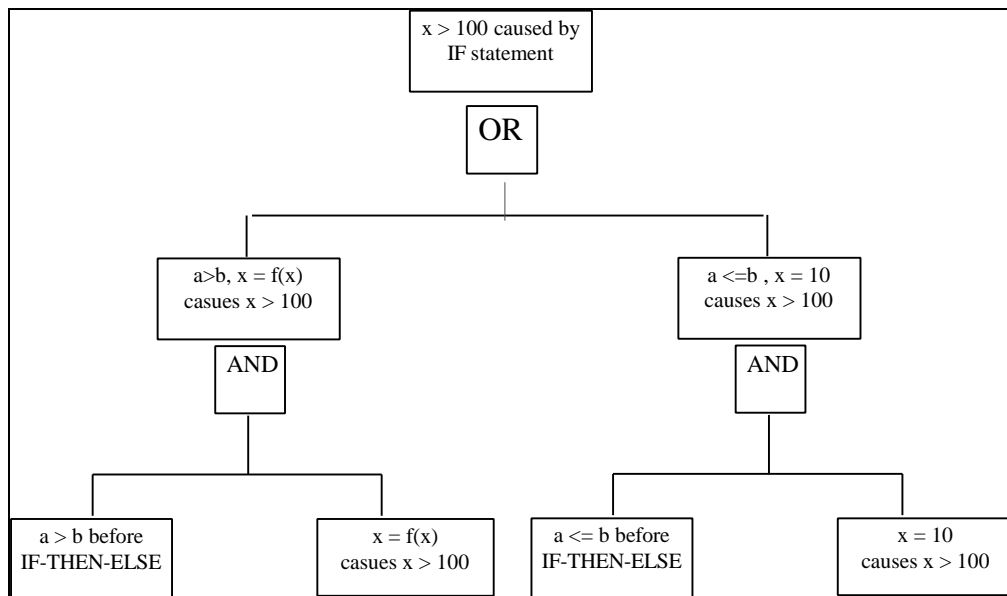


Figure 16. Software Fault Tree [Lev95,p. 502]

5.4 Design of safe systems

To design a safe system it is not sufficient to only identify the hazards in a system, the knowledge of their existence must also be taken advantage of during system design. The goal is to eliminate the existence of hazards to the lowest reasonably practical level (the ALARP principle – As Low As Reasonably Practical). When the hazards cannot be eliminated, reduce the impact of their existence, or when that is not possible at least try to control the hazards.

Here is a compilation of techniques to eliminate or control hazards [Lev95,p. 402]. They will all be addressed in the following section¹⁵. These techniques are not mutually exclusive but complementary. They are all necessary since the elimination of all hazards might not be practical, due to tradeoffs regarding functionality versus safety, or simply too expensive. The effectiveness of the methods are in order of appearance. If hazards are eliminated or reduced there will be less or no need for control measures. These control measures may include people who make mistakes or protection devices that are prone to fail, maintained negligently, turned off, or simply ignored in an emergency [Lev95]. It is easier and more cost efficient to build safety into a system from the beginning than to add safety on.

1. Hazard Elimination

- Substitution
- Simplification
- Decoupling
- Elimination of specific human errors
- Reduction of hazardous materials or conditions

2. Hazard Reduction

- Design for controllability
 - Incremental control
 - Intermediate modes
 - Decision aids
- Barriers
 - Lockouts
 - Lockins
 - Interlocks
- Failure minimization
 - Redundancy

3. Hazard Control

5.4.1 Design techniques

Hazard elimination

A design can be made intrinsically safe by eliminating hazards. Hazards can be eliminated, either by removing the hazardous state entirely from the systems operation or by eliminating the negative consequences of the hazards' existence. Thus are the hazards eliminated by definition; if a state does not lead to a loss it is not a hazard.

Substitution

A possible way to eliminate hazards or minimize the consequences of their existence could be accomplished by substituting a hazardous substance or an energy source for a less harmful one.

Example. A railroad crossing for cars and people. If a viaduct is built instead of a railroad crossing the hazard of getting overrun by a train is eliminated. A new hazard has nonetheless occurred: the hazard of

¹⁵ Leveson has a more thorough and in depth description of these techniques. The interested reader is advised to read [Lev95,ch. 16]

the viaduct caving in. The probability of the viaduct caving in does however seem more remote than a failure in the signaling system or getting stuck on the railroad track.

Example. If the meat from lions was good to eat, some farmer would certainly consider breeding lions. The farmer would have to build cages and take safety measures to keep the lions in check. Sometimes would, nonetheless a lion sneak out... – but why keep lions when sheep could do as well?[Kle84].

Simplification

When designing a computer based system, implement only the functionality absolutely necessary. Software is notorious for having *yet another cool feature*. In a real-time system, use static scheduling instead of priority based scheduling. This will simplify verification and allow reproduction of exactly the same execution order, time and time again.

Decoupling

Accidents in tightly integrated systems occur due to unplanned interactions. This interdependency can lead to a system hazard when a failure in a component affects all the other components, in a falling domino bricks manner.

Example. Decoupling can for instance be realized by design of a fire-wall.

High safety requirements in a computer based system might require safety-critical parts of the system to be separated from the non safety-critical parts. The safety-critical parts are then assembled and reduced in order to decrease the number of interfaces to other software parts. A significant benefit of isolating the safety-critical parts in a program is that the resources dedicated to verification of safety can be used more wisely, i.e. no resources need to be wasted on non safety-critical software.

When the safety-critical modules of a software system have been identified it is possible to protect them with so called fire walls. The fire walls can be physical or logical . Physical fire walls can be provided by, e.g. memory management units or separate computers. Logical fire walls can, for example, in the programming language C be implemented in the following fashion[Voa95,pp. 123]:

- Every safety-critical module (SM) has got its own source file.
- All code that is to be protected by a fire wall should be put in a function declared as *static*. By doing that, the code will be invisible outside the file and can thus not be executed by some other function outside the module. Global variables in a SM that is intended to be interior to a fire wall should be declared as global and *static*. The data will then be invisible outside the file and can therefore not be used by a function residing outside the module.
- Every function's local data should be declared as static or automatic. This data will then automatically end up behind the fire wall.
- The functions that shall act as ports or interfaces to the other modules are declared without the static directive. These functions can then be linked to other functions in an ordinary way. By reducing the number of interfaces unintended coupling can be avoided and the verification will be simpler.

Elimination of human errors

Humans are always fallible. We may directly or indirectly cause accidents, either through erroneous operation or maintenance, or by designing a hazardous system. Operation and maintenance errors can be remedied by designing the system in such a way that there are few opportunities to make mistakes. In the aircraft industry the components are designed in such a way that incorrect assembly is difficult or impossible. For example, electrical connectors on similar or adjacent cables are made in different sizes or forms, which makes it virtually impossible to make hazardous connections [Lev95, chap. 16].

Users of certain programming languages like for instance C have been found to be especially prone to error. Hard learned experiences with language constructs like pointers, gotos and breaks, implicit type conversions, global variables and overloading have deemed them hazardous.

A language should be simple and encourage the production of simple, readable and understandable programs. The semantics of the language must be precisely defined, static type checking employed and conditional statements should be robust, i.e., all branches on a condition shall be defined.

The application of subsets of languages have been found successful, where only a subset of the least hazardous language constructs are used. There are safety-critical standards that advocate the use of language subsets, for example, the new standard for safety-critical systems IEC 1508 [IEC1508]. An approach to make the C language safer, by defining a subset, has been presented in a book by Les Hatton named Safer C [Hat95].

Reduction of hazardous conditions

Software should only contain code that is absolutely necessary. For example, electromagnetic interference or cosmic radiation can make programs do inadvertent jumps in execution. Unused memory should therefore be initialized to *illegal instruction* or something similar that takes the computer to a safe state if executed. Likewise should memory boundaries be guarded against overwrites.

This defensive strategy has the implication of making Commercial Of-The-Shelf-Software (COTS) almost unusable in safety-critical applications. Most COTS are made in such a general fashion that they can be used in a wide array of applications. But, most applications will not make use of all the functionality provided; this is hazardous. However, the main point of using COTS is to increase reliability¹⁶, so there is a trade-off here. On one hand high reliability is desirable and on the other hand safety. As previously discussed in this section, just because software is deemed reliable does not mean that it is safe. Another catch with COTS is that they are proprietary and are thus not likely to have all functionality documented. The documentation of a software module's behavior is imperative, in order to make a hazard analysis of a system. This documentation might very well have to cover the requirements specifications, designs, implementations and all the way down to the machine code level.

Hazard reduction

It is possible to decrease the frequency of hazards occurrence. The probability for an accident to occur will decrease if the precursing or contributing hazard is less likely to happen.

Different kinds of safety devices can be used, passive or active.

Passive safety devices improves safety by:

- its sheer presence. Like shields, safety belts, lifejackets, barriers, fences,...
- or by failing into a safe state. Valves can, for example, be designed to fail in an open or closed position, or landing gear can be designed to fail in the extended state.

Active safety devices improves safety by reacting to some stimuli, in order to give protection.

Typically a state must be monitored, a set of variables needs to be read, some analysis needs to be done and finally an actuation must take place. Thus a control system must be present – a computer. For example, an airbag in a car and an anti brake system are active safety devices.

Design for controllability

A feasible way to reduce hazards is to make the system easier to control (for both man and machine). For example, it is easier and safer to rely on an anti brake system (ABS) than on manual pump braking.

Incremental control

In software, control devices can be realized as control loops, i.e., by letting critical actuations be performed in increments instead of just an *on* or *off*. This allows for corrective measures to be taken, if an actuation is incorrect, before serious damage is a fact.

¹⁶ Another main point is of course "time to market" – the shorter "turn around time" the better. This "time" is however in reference to reliability since it takes a certain time (and costs money) to achieve sufficient reliability. Managers and engineers do therefore want to buy finished COTS that are well proven and save them the time and effort designing the software.

Intermediate modes

Safety can also be attained by using intermediate modes. The system can be designed in such a manner that the operator, for example, has more choices than just to turn off a process entirely or let it keep on running. Different levels of functionality can be used for each mode. Modes can, for example, be full functionality, reduced functionality, or emergency modes. This is a must in an airplane where there is no safe state to go to, i.e., operation is required¹⁷.

Decision aids

Another means for computers and software to contribute to safety is by supplying decision aids. For example, decision aids can structure alarms in an airplane or in a nuclear power plant. All in the best interest of man – lessening stress and making the system easier to control.

Barriers

A feasible way to decrease the probability of a hazards to occur is to erect barriers. The barriers can be physical objects, incompatible materials, system states, or events[Lev95, p.421].

Lockouts

A lockout¹⁸ prevents someone, something or an event to make the system enter a hazardous state. The most basic lockouts are physical objects that hinders access to hazardous states.

Typically they are used for:

- Avoidance of electromagnetic interference (EMI) like radio signals, electrostatic discharges or electromagnetic particles – for example, alpha or gamma rays, can be implemented by using shielding or filters.
- Limitation of authority. For example, the controlling computer in an airplane may restrict the pilots maneuvers so that they are always inside the safety margins for what the planes wings hold for. Such limitations must however be carefully designed so that they do not restrict maneuverability in emergency or otherwise extreme situations.
- Lockouts in software can take the form of limited access to safety-critical variables and code. This is imperative since unintended execution of safety-critical code at the wrong time may be catastrophic. Likewise may unintended corruption of data have lethal consequences. This can be avoided by employing techniques borrowed from the security community. When an action is to take place it must be authorized by several modules. For example, the ability for software to arm and launch a missile may be severely limited if several confirmations are needed. Here is a trade-off again. This has the impact of lowering reliability for software since more parts are required to function correctly. It will however be safer. Other techniques have also been borrowed from the security community, like access rights (read, write and execute) that function in the same manner as in a protected file system (Unix), limiting the access rights for modules or people.

Lockins

Lockins try to keep the system in a certain safe state. For example, containment of harmful substances like toxins, pathogens or radioactive materials, or to put harmful objects out of range, like cages around robots.

Software must be designed in such a way that even though non nominal, erroneous or out of order inputs are received, the system will not enter a hazardous state. This was discussed in the fault-tolerance section and defined as robustness.

¹⁷ When a systems must keep on running despite failures it is called *fail operational*.

¹⁸ A lockout is intended to keep a hazardous entity locked out while a lockin intends to keep a hazardous entity locked in.

Interlocks

Interlocks can be used to force operations to be performed in a certain order or to keep two events separated in time.

Interlocks ensure [Lev95,p.426]:

- That event A does not occur inadvertently. For example, by requiring two separate events to occur – like pushing of button A and pushing of button B.
- That event A does not occur while condition C exists. For example, by putting an access door over high voltage equipment so that if the door is opened, the circuit is broken.
- That event A occurs before event D. For example, by ensuring that a tank will fill only if a vent valve is opened a priori.

The two first are referred to as inhibits and the last one as a sequencer.

Failure Minimization

Redundancy and recovery

As discussed in the fault-tolerance section of this document, redundancy and recovery can provide means for increasing reliability of a system. This is also the case for safety to some extent. Particularly when safety is dependent on a reliable function. For example, in a fail operational¹⁹ system (like an airplane) it is of utmost importance that the most vital functions are up and running. Redundancy in a system, where safe states exist, is not as important for system safety as it is for reliability.

Redundancy has been found to work most beneficially against transient errors imposed by the environment. Redundancy against design faults has not proven to be equally successful.

Redundancy and recovery techniques increase complexity in a system and will therefore afflict the safety of the system, as design errors are more prone to occur.

Redundancy and recovery can, when designed and applied carefully, provide higher reliability and safety.

Hazard Control

If hazard control is employed then the probability of residual hazards leading to accidents is reduced.

Hazard control can be implemented in such a way that when the system detects an unsafe condition it transfers the system to a safe state. For example, a panic button can cut the power to a system and subsequently turn it off, making it safe.

For an accident to occur more conditions than just the hazard must be present. Control over a hazard might be manifested as a reduction of the exposure to the hazard. For example, decreasing the time that the hazardous state exists will in turn lead to a reduced probability of an accident occurring.

Isolation is another measure that is often used in order to reduce the exposure to risk. Nuclear reactors or chemical plants are often not situated in highly populated areas. Likewise are airports and military testing grounds located far from densely populated areas.

6 CONCLUSION

In the introduction of this document it was stated that even if we could make a perfectly safe and reliable system it was not enough, a proof had also to be provided, asserting the reliability and safety of the system. As of today several methods and programming languages are individually claimed to be *the* solution. Alleged silver bullets are, e.g., formal methods, fault tolerance and testing. However, there are

¹⁹ Fail operational – the system must operate despite failures.

empirical evidence and logical arguments implying that they by themselves are not sufficient in providing the assurance level required of safety-critical systems.

The next logical step would thus be to apply them in some combination, or even all at once. Then we certainly would achieve the required level of confidence!?

David Parnas have stated what should be required of safety-critical software. He calls it the tripod[Par95]:

1. Precise, organized, mathematical documentation and extensive systematic review.
2. Extensive testing.
 - Systematic testing – quick discovery of gross errors.
 - Random testing – discovery of shared oversights and reliability assessment.
3. Qualified people and approved process

According to Parnas the three legs are complementary, although the last one is the shortest.

One could actually stop here and say that the way to go is as Parnas and Bowen [Bow92] proclaim: use a combination of fault avoidance, fault tolerance, fault removal. Then and only then will the desired level of reliability be achieved (however excluding safety). But by doing so we lend ourselves the comfort of belief again. We do not know – at least not yet, what is the ultimate combination of approaches and methods.

This necessity of belief (not knowledge) in a concept or method shows that software engineering is unconsummated as a discipline. Software engineering is still a craft.

There are two stages in the early years of a new technology [Lev92]: (1) the exploration of the space of possible approaches and solutions to problems, i.e., invention and (2) the evaluation of what has been learned by this trial and error process to formulate hypotheses that can be scientifically and empirically tested in order to build the scientific foundations of the technology. Most of the efforts done within the software engineering discipline has so far been emphasized on the first stage, i.e., invention.

We do not need more specification languages; what we need are means by which we can validate underlying principles and criteria for design of specification languages. We do not need yet another tool for specification of designs; we need to devise and validate fundamental design principles, and to understand the tradeoffs between them. We do not need to create more languages to specify processes; we need to study effects of different types of software development processes in real organizations and under different conditions [Lev92].

We need a science, we need to know the nature of software, and the nature of the methods and processes producing the software. In order to do so we probably need to establish a set of measures and metrics for software which has an empirical foundation, i.e., based not on frivolous assumptions, but on empirical data. We need to compare different designs' attributes and judge whether a design is better (more reliable, more reusable, more testable, ...) or worse than another. If this works we will be able to analyze the alleged silver bullets and perhaps refute or confirm them once and for all. And, finally relevant design principles might emerge.

Summing up: If we have a theoretical foundation it can provide [Lev92]: (1) Criteria for evaluation, (2) means of comparison, (3) theoretical limits and capabilities, (4) means of prediction, and (5) underlying rules, principles and structure.

If we can establish this foundation we will finally be able to model, compare and predict the behavior of our designs and thus transform software engineering from a craft into an engineering discipline.

7 ACKNOWLEDGMENTS

A great thanks to Christer Eriksson for being extremely supportive of this work.

Other thanks go to Martin Törngren and Jan Wikander.

This work has been made possible by funding from SCANIA.

8 REFERENCES

- [Bar92] L Barroca and J McDermid. Formal Methods: Use and Relevance for Development of Safety-Critical Systems. *The Computer Journal*, Vol. 35, No. 6, 1992.
- [Bor91] N.S Borenstein. *Programming As If People Mattered: Friendly programs, Software Engineering, and other noble delusions*. Princeton University Press, Princeton, New Jersey, 1991.
- [Bow92] J Bowen and V Stavridou. Safety-Critical Systems, Formal Methods and Standards. *Software Engineering Journal*, 1992.
- [Bur89] A. Burn and A. Wellings. *Real-Time Systems and Their Programming Languages*. Addison Wesley 1989.
- [DeM78] T DeMarco. *Structured Analysis and System Specification*. Yourdon Press 1978. ISBN 0-917072-07-3
- [DO178B] Software Considerations in Airborne Systems and Equipment Certification. Available from RTCA. Also the European joint standard: EUROCAE ED-12A, December 1992.
- [Ell95] A. Ellis. Achieving Safety in Complex Control Systems. Proceedings of the Safety-Critical Systems Symposium. Pp. 2-14. Brighton, England, 1995. Springer-Verlag. ISBN 3-540-19922-5
- [FAA88] System Design and Analysis. Federal Aviation Administration. June 21, 1988. Advisory Circular 25.1309-1A.
- [Fen91] N E Fenton. *Software Metrics: A rigorous approach..* Chapman & Hall, 1991. ISBN 0-412-40440-0.
- [Fen95] N E Fenton. The Role of Measurement in Software Safety Assesment. CSR/ENCRESS Annual conference, Sept. 1995.
- [Gow94] L. Gowen. Specifying and verifying safety-critical software systems. IEEE 7th symposium on computer based medical systems. June, 1994.
- [Hat95] L. Hatton. *Safer C*. McGraw-Hill, 1995. ISBN 0-07-707640-0.
- [IEC1508] Draft IEC1508 – Functional safety: safety related systems. June 1995
- [Kim94] K.H. Kim. Design of Real-Time Fault-Tolerant Computing Stations. *Real-Time Computing* ,Edt W. Halang A. Stoyenko, Springer Verlag, Nato ASI series, volume 127, page 31-46.
- [Kle84] T. A. Kletz. *Myths of the chemical industry*. The institution of Chemical Engineers. Rugby, Warwickshire, United Kingdom, 1984, pp. 66.
- [Kni90] Knight J. C. and N. G. Leveson. A reply to the criticism of the Knight and Leveson experiment. *ACM SIGSOFT Software engineering Notes*, 15, p. 25-35, January 1990.
- [Lam85] L Lamport et.al, "Synchronizing clocks in the presence of faults", *journal of the ACM*, Vol. 32, No. 1, Jan 1985.
- [Lap92] J.C. Laprie. "Dependability: Basic Concepts and Associated Terminology" in *Dependable Computing and Fault-Tolerant Systems*, vol. 5, Springer Verlag, 1992.
- [Lev86] N. G. Leveson. Software safety: What, why and How. *Computing surveys*, 18(2),1986
- [Lev92] N. G. Leveson. High Pressure Steam Engines and Computer Software. The Int. Conference on software engineering, Melbourne, Australia, May 1992.
- [Lev95] N. G. Leveson. *Safeware System, Safety and Computers*. Addison Wesley 1995. ISBN 0-201-11972-2.
- [Lit73] B Littlewood et. al. A Bayesian Reliability Growth Model For Computer Software. *Journal of the Royal Statistical Society, Series C*, No. 22, p 332-346, 1973
- [Lit91] B Littlewood. Limits to evaluation of software dependability. *Software Reliability and Metrics*, 1991.
- [Lt82] E. Loyd and W. Tye. *Systematic Safety: Safety Assessment of Aircraft Systems*. Civil Aviation Authority, London, England, 1982. Reprinted 1992.
- [Lut92] R. R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *software requirements conference*, IEEE, January 1992.

-
- [MoD95] Ministry of Defence UK. The Procurement of Safety Critical Software in Defence Equipment. Draft Defence Standard 00-55, August 1995.
- [Neu95] P G Neuman. Computer Related Risks. ACM Press, Addison-Wesley, 1995. ISBN 0-201-55805-x.
- [Par95] D Parnas. Connecting theory with practice. Communication Research Laboratory, Software engineering group. McMaster University, Canada, November 4, 1995.
- [Pus89] P. Puschner, C. Koza. Calculating the maximum execution time of real-time programs. Journal of Real-time systems 1(2), Pp. 159-176, September, 1989.
- [Ram90] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In 10th international conference on distributed computing systems. Pp. 108-115. 1995.
- [Rus95a] J Rushby. Formal Specification and Verification for Critical systems: Tools, Achievements, and prospects. Advances in Ultra-Dependable Distributed Systems. IEEE Computer Society Press. 1995. ISBN 0-8186-6287-5.
- [Rus95b] J Rushby. Formal methods and their Role in the Certification of Critical Systems. 12th Annual CSR Workshop, Bruges 12-15 September 1995. Proceedings, pp. 2-42. Springer. ISBN 3-540-76034-2.
- [Sha96] N. Shankar. Unifying Verification Paradigms. At the 4th international symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems. Uppsala, Sweden, September 1996. Lecture Notes in Computer Science, no. 1135. Springer. ISBN 3-540-61648-9
- [Tan95] A. Tanenbaum. Distributed Operating Systems. Prentice Hall 1995. ISBN 0-13-595752-4.
- [Tur36] A. M. Turing. On Computable Numbers with an Application to the entscheidungs problem. Proceedings London Mat. Society, 2:42, pp. 23-265, 1936.
- [Wie93] L. Wiener. Digital Woes: Why We Should Not Depend on Software. Addison-Wesley, 1993
- [Voa95] J M Voas, et.al. Software Assessment: Reliability, Safety, testability. Wiley Interscience, 1995. ISBN 0-471-01009-x.
- [Xu90] J. Xu, D. Parnas. Scheduling process with release times, deadlines, precedence and exclusion relations. IEEE transactions on software engineering, vol. 16, no. 3, March 1990.