DIPLOMARBEIT

Safety Supervision Layer

Fault Detection for Operating Systems in Fail-Safe Environments

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Diplom-Ingenieurs unter der Leitung von

o. Univ. Prof. Dr. Dietmar Dietrich
und
Dipl.-Ing. Andreas Gerstinger
als verantwortlich mitwirkendem Universitäts-Assistenten am
Institutsnummer: E384
Institut für Computertechnik

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Georg Hartner
Mtr.Nr. 9902458
Pulverturmgasse 7/32
1090 Wien

Wien, 20. Jänner 2008

Abstract

This work deals with fail-safe applications in a railway context and describes a way to produce safety-relevant software more efficiently. Fail-safe systems in this context have to be shut down if they produce erroneous results in order to guarantee that neither material nor people are harmed. Those errors have to be detected. Safety-critical applications run on operating systems, operating systems provide interfaces to the hardware those systems run on. Hence not only those applications themselves, but also operating systems as well as hardware are possible sources of failure. As they are potential sources of failure, they have to be validated and verified, their functionality has to be monitored, which is costly in terms of man hours and increases time-to-market for the considered products.

Currently whole operating systems and their specific functions have to be verified line by line by experts. Nowadays operating systems get more and more complex and change rapidly. For every system or version change they have to be evaluated again. Complex hardware tests have to be written to match the different architectures. When used hardware changes, tests have to be re-written and re-verified.

The approach used in this work is to provide end-to-end checking by creating a layer between the safety-relevant applications and the operating system interface. This safety supervision layer offers the applications functions enriched with fault detection features. The ultimate goal of the layer is to provide the application developer with the guarantee that every fault situated in the layers below – may it either be somewhere in the operating system or the hardware – is detected and the system is shut down in order to prevent hazardous situations.

The end-to-end checking approach is a generic way to check for failures as it checks if a given result is plausible respective to a given specification. To stay on this generic track the layer is intended to be above the Portable Operating System Interface, which is implemented by many systems. As there are faults that may not be checked in that end-to-end way directly when calling a function, a watchdog structure is suggested in this work.

Kurzfassung

Diese Arbeit behandelt Fail-Safe Eisenbahn-Applikationen. Fail-Safe Systeme kennen einen sicheren Zustand, in den sie im Fehlerfall gebracht werden und abgeschaltet werden können, ohne Menschen oder Material zu gefährden. Diese Fehler müssen von Seiten des Systems erkannt werden. Sicherheitskritische Anwendungen laufen auf Betriebssystemen, Betriebssysteme laufen auf Hardware. Daher sind nicht nur die Anwendungen selbst, sondern auch Betriebssystem und Hardware potentielle Fehlerquellen. Weil sie mögliche Fehlerquellen darstellen, müssen sie sorgfältig verifiziert und validiert werden, ihr Funktionieren überwacht. Dies ist sehr teuer, was den Personalaufwand betrifft, und erhöht die Entwicklungszeit für die betreffenden Produkte.

Zur Zeit müssen ganze Betriebssysteme und ihre Funktionen Zeile für Zeile von Experten verifiziert werden. Heutzutage werden diese immer komplexer und ändern sich schnell. Für jede neue Version oder Änderung wird der Code erneut überprüft. Komplexe Hardware-Test werden entwickelt werden, um unterschiedlichen Architekturen zu genügen. Ändert sich die zu überprüfende Hardware, müssen die komplexen Tests neu geschrieben werden.

Als Lösungsvorschlag beschreibt diese Arbeit einen Safety-Layer zwischen den Applikationen und dem Betriebssystem. Diese Schicht bietet den Zugriff auf herkömmliche System Calls über Wrapper-Funktionen, die mit Features zur End-to-End-Fehlerdetektion angereichert sind. Das ultimative Ziel dieses Layers ist, dem Entwickler die Garantie zu bieten, dass alle Fehler, die ihren Ursprung unterhalb dieser Schicht haben, verlässlich erkannt werden. Die Anwendung wird davon mittels Signalen in Kenntnis gesetzt und kann das System in einen sicheren Zustand bringen, um Gefahrensituationen zu vermeiden.

Dieser End-to-End-Ansatz überprüft, ob ein geliefertes Ergebnis plausibel im Sinne einer Spezifikation ist. Um diesen allgemeinen Anspruch weiter zu führen, setzt die Schicht auf dem Portable Operating System Interface auf. Zusätzlich zu den Fehlererkennungs-Funktionen wird eine Watchdog-Struktur für generelle Systemfehler vorgeschlagen.

Acknowledgements

This work has been elaborated from April to December 2007 under the supervision of Andreas Gerstinger at TU Vienna's Institute for Computer Technology, to whom I owe a lot for the great input on safety, fault-detection and diversity and the profound support during this year. Thanks a lot for your guidance and advice. I got a lot of tips from the colleagues at the project partner that supported me in developing the Safe Message Passing layer – Nicolas McGuire and Klaus Reichl. Thank you for the important hints and guidelines. Furthermore thanks to Prof. Dietrich and my fellow students at the ICT's diplomate seminar for the important criticism and tricky questions.

I learned a lot from my fellow students and friends that accompanied me during my time at university. You taught me a lot about what one should and should not do. Special thanks to Josef Trojer who introduced me to my advisor and my friends from my home area.

My family spent me a solid education and always supported me during my studies even when things went bad. Thank you a lot.

Abbreviations

CCP	Consistent Comparison Problem
CENELEC	European Committee for Electro-technical Standardization
COTS	Commercial-Off-The-Shelf
FTA	Fault Tree Analysis
FMEA	Failure Modes and Effects Analysis
gdb	GNU Debugger
IPC	Inter-Process Communication
LTP	Linux Test Project
MTBF	Mean Time Between Failure
OTS	Off-The-Shelf
POSIX	Portable Operating System Interface
RAMS	Reliability, Availability, Maintainability, Safe
SIL	Safety Integrity Level
SLoC	Source Lines of Code
SMP	Safe Message Passing
SOUP	Software of Unknown Pedigree
V&V	Verification and Validation

Table of Contents

1 Introduction

Currently and in the future people get more and more influenced by electromechanical systems. They have conquered their space in our everyday life and this space is about to grow in the coming years and decades. Those systems provide outstanding possibilities and may radically change a society's lifestyle as the car, the airplane and train revolutionized old traffic systems. But as we learn from history every technological innovation introduces new dangers and – obviously – resulting catastrophes, while the most serious of them have occurred in the past 30 years [Lev95, p.7].

In the United States, technological hazards account for 15 to 25 percent of human mortality and have significantly surpassed natural hazards in impact, cost and general importance [Lev95, p.4]. We generally tolerate those risks because in most cases the advantages that come with a new technology exceed the risks they bring. Nevertheless this circumstance creates new challenges for engineers to build safer systems and for governmental organizations to decide, whether a system should be admitted to public use or not. IEC 61508 describes these triangular relationships and mutual influences as can be seen in Figure 1.

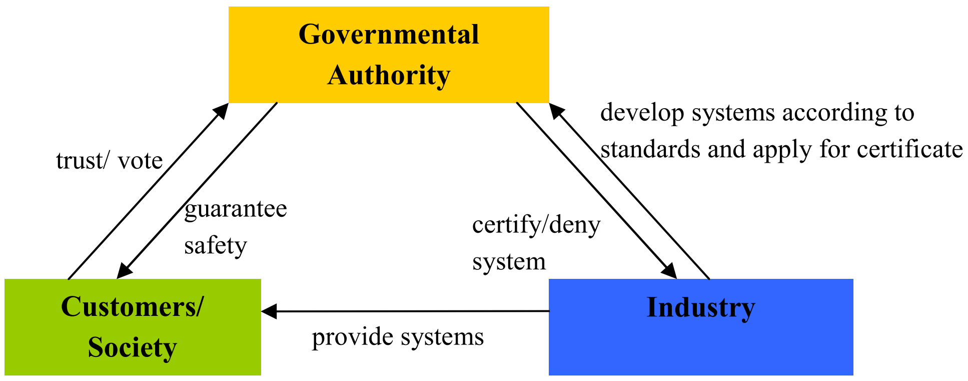


Figure 1: Triangular safety relationship.

One of the most important steps in a safety-critical development process is to act pro-actively. Hazard analysis techniques have to be applied to identify possible risks and find series of events that may possibly lead to accidents. They have their place in the centre of a safety-related system as shown in Figure 2. The figure is taken from [Lev95, p.288] and was amended with the development bubble, as requested by my colleagues at the project partner. A more detailed description of hazard analysis techniques is given in the safety chapter.

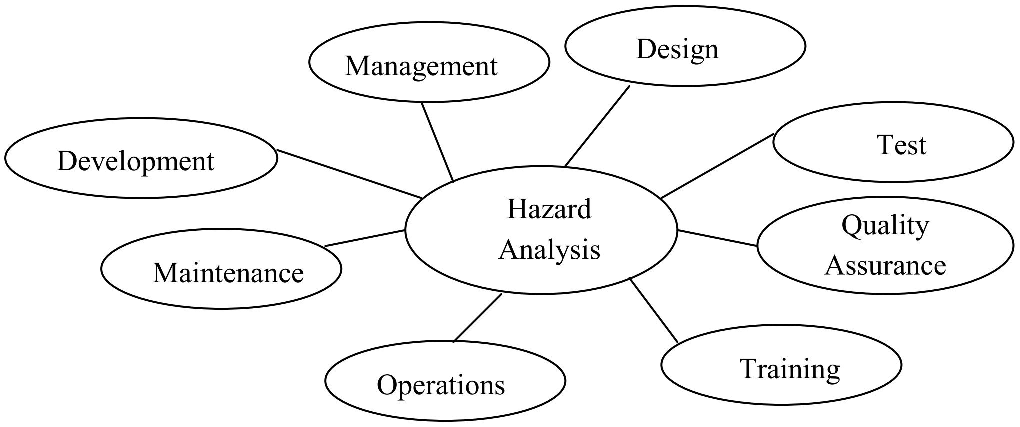


Figure 2: Hazard analysis as a central entity in a safety assurance program [Lev95, p.288].

So Hazard Analysis is really a crucial part to avoid accidents and control incidents. Nevertheless people had to learn that there is no risk-free life and the only thing we can do is just to reduce the risks to an acceptable level. On the other hand there's always a big outcry when serious accidents occur. Companies as well as authorities are brought to trial and safety guidelines are discussed, scrutinized and adapted which leads us to a spiral of safety increase and an incremental evolution of safety standards whose fulfilment consumes a major part of the development budgets and time-to-market. Figure 3 visualizes the relationships that lead to a circular improvement of observed safety-critical standards.

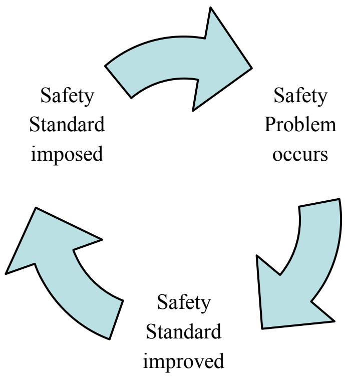


Figure 3: Safety standard improvement circle *after* incidents or accidents have occurred.

Systems have to comply those standards, although their implementation gets more and more expensive and often has to be rewritten and rechecked as a whole if small changes according hard- or software are undertaken.

So while complexity continuously increases the industry yaws for technologies that help to contain the exploding costs. One considerable option to reach this goal is to use Commercial-Off-The-Shelf components and Open Source software – like operating systems – to generate reusable solutions. My work sketches up a portable, reusable and testable safety layer that's based on the standardized "Portable Operating System Interface" that contains functionality and patterns to solve identified hazards to enforce safety criteria and save time and – depending on the environment and applicability of the components - money.

1.1 Problem Statement

As systems grow in complexity and size it is desired to build them out of previously written software. Reuse and the use of COTS Software can increase product quality and reduce development time [YBB99, p.1]. Those components must not influence the whole system in a safety-critical way.

One big chunk of possibly being reused software is operating systems. Operating systems suitable for safety-critical environment are expensive, relatively rare and often they don't provide the desired features. As there are many Off-The-Shelf operating systems available, open source as well as commercial ones, it would be necessary if developers could just use those as platforms for they safety-critical applications without spending too much work on the layers below. They could explicitly concentrate on the application logic of the actual critical project itself and wouldn't have to worry about which OS version they're working with and if the hardware they use working according to specification.

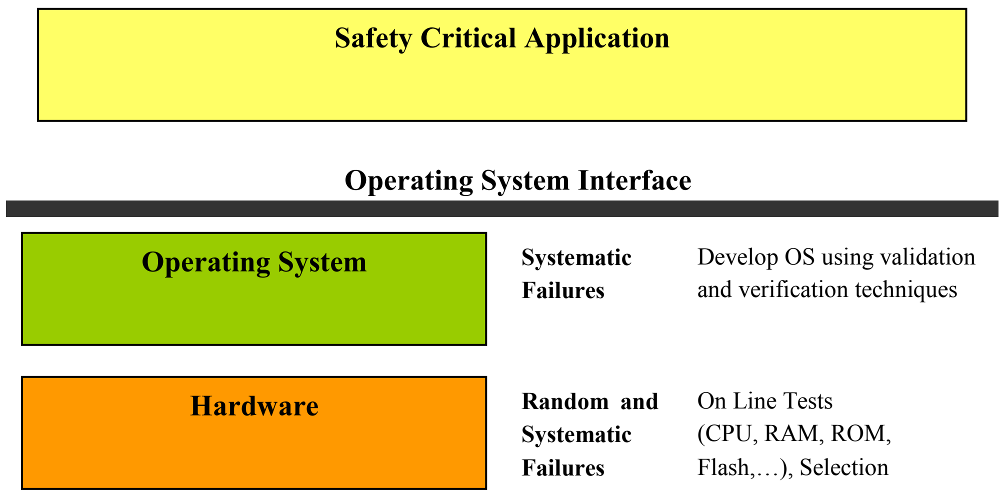

Safety Critical Application
Operating System Interface
Operating System
Systematic Failures
Develop OS using validation and verification techniques
Hardware
Random and Systematic Failures
On Line Tests (CPU, RAM, ROM, Flash,…), Selection

Figure 4: Current safety assurance techniques for operating systems.

Currently engineers have to put a lot of work into securing the layers below the actual application layer. Figure 4 shows that in order to prevent systematic failures hardware functionality is continuously tested. The components are checked while the system is running by so-called On-Line-Tests (OLT) that inspect RAM, ROM, Flash, CPU functionality, etc.. Those online tests are hard to write and have to be at least revalidated, if not rewritten if there are slight changes in the hardware layer.

Operating systems in use have to be developed according the safety standards. Quality Assurance tools like validation, heavy testing, code reviews have to be applied to the system. This counts for every new version of a system as well, so the trend is that the cost for the validation of those low level layers increases immensely. A good approach would be to enrich the operating system interface with the capability of automatic error detection. Nevertheless there exists a study initiated by UK Health and Safety Executive of 2002 [HSE02] that concludes under certain circumstances and constraints: "*On the basis of evidence from widespread use, some numeric reliability data, observed reliability growth, the existence of test projects and the limited analysis carried out by this study, it is concluded that "vanilla" Linux would be broadly acceptable for use in safety related applications of SIL 1 and SIL 2 integrity*". They even state that it might be feasible to certify Linux to SIL 3 [HSE02, p.8], [Skram05, p.2] in an IEC 61508, 61511 context. For other safety contexts systems have to be considered separately according to the specific prescriptions. Nevertheless the approach selected in [HSE02] is suitable for other standards as well.

1.2 Related Work

To paint a big picture of what work has been done so far, those theses may be classified into three categories of research that this text is based on. Its mission is to combine theories of required general safety features of component based software, common pitfalls and best practices, as well as techniques of how operating systems used in safety related environments and pictures of hardware systems are based on has to be proved functional during a product's lifetime.

1.2.1 General Dependability Issues

One starting point for this work was "Fundamental concepts of dependability", a commonly known paper by Laprie et al. [Avi01] that defines common terms and techniques that may be used to construct dependable computing systems. It sketches the picture of how relevant fields of research like safety, security, fault tolerance and others work together to guarantee working systems.

Very useful for this work was "Safeware" [Lev95], a book written by Nancy Leveson in 1995, which gives a comprehensive picture of safety and provides an intensive historical overview of safety design drivers and how generic techniques have evolved over the last 50 years. It shows as well what happens when system architects aren't aware of grown technologies and that safety is more than just a technical, but often more an organizational issue, numbering and explaining series of commonly known industrial accidents.

For the practical part I used the book Software fault tolerance techniques and implementation [Pul01] by Laura Pullum. This book explains approaches for reaching fault tolerant systems by software techniques, which I intensively made use of in designing the safe message passing and watchdog functions in the safety supervision layer.

1.2.2 Off-the-Shelf Components in Safety-Related Environments

[Aas07] gives a number of definitions according Off-The-Shelf (OTS) components and explains potentials and possible pitfalls for systems making use of them. Apart from that it shows possibilities of how to guarantee Safety Integrity Levels in those systems. [Skram05] deals with the reuse of Commercial-Off-The-Shelf (COTS) components and Software of Unknown Pedigree (SOUP). The fact that safety standards have strict requirements to how software has to be developed and tested, and most do not explicitly allow the use of COTS components or SOUP is stated and ways to deal with these circumstances are stated. [Ye04b] gives a schematic overview of what has to be considered, when selecting components for safety-related systems from market monitoring to COTS Acquisition contract. [Ye04a] examines ways to analyze the criticality of COTS software components using commonly known technologies like System Hazard and Fault Tree Analysis.

The papers stated above deal with software safety. Safety Supervision Watchdog deals with ways to check for failures that cannot or hardly be detected directly when using the respective functionality. Many of them deal with hardware and low-level operating system faults. [Tam07] and [Arl02] deal with testing, error detection and fault analyses for those.

1.2.3 Requirements to Safe Operating Systems

[HSE02] states requirements to safe operating systems assessing Linux. It states techniques and issues heavily using proven-in-use that have to be used when deciding whether a given operating system may be in safety critical environments. They also show how the results of such assessments can be mapped to required Safety Integrity Levels. Other more generic sources are provided by the CENELEC standards stated below that exactly prescribe required features in the railway problem domain.

1.3 Motivation

“There is an increasing interest in acquiring commercial-off-the-shelf (COTS) functionality for safety-critical applications” [Ye04b, p.1]. It is commonly known that many advantages come along with Software Reuse. Depending on the surrounding circumstances it can reduce time-to-market and probably reduce costs. Those do not emerge on their own, as a component based approach may also lead to pitfalls like faulty components, incomplete specifications and interface mismatches. Nevertheless an advantage that may be generated by consequently using carefully selected COTS Software is that those pieces of software get proven-in-use. Evolution may take place improving the components’ code, documentation and proneness to errors.

Release	*Operating System*	*Million SLOC*
1998-07-24	Debian Linux 2.0 Hamm	37.41 [16]
1999-03-09	Debian Linux 2.1 Slink	70.6 [16]
2000-08-15	Debian Linux 2.2 Potato	55-59 [LoCDeb2.2, LoCDeb3.x]
2002-07-19	Debian Linux 3.1 Woody	104 [LocDeb3.x]
2005-06-06	Debian Linux 3.2 Sarge	215 [LocDeb3.x]

Table 1: Growth of debian linux distributions in SLOCs.

Furthermore the starting point for this work was seeing whole operating systems as OTS components and probably find ways to reduce costs for writing tests and fault detection facilities and expensive verification and validation efforts. Currently operating systems have to be checked for each version, function by function, line by line by experts. As operating systems constantly grow in size, these efforts get more and more expensive and impractical. Source Lines of Code is a reasonable complexity metric for software just counting the lines the system is compiled of. The size of the Linux kernel itself has continuously increased from 10,239 SLoCs in September 1991 to 5,929,913 in December 2003. Table 1 shows the growth in size in terms of SLoCs of the debian linux distributions to visualize this evolution.

In safety-relevant systems every version and configuration has to be checked in depth, which – in times of rapidly emerging software – gets more and more costly. Hence there would be numerous advantages in finding structures to decouple safety-critical applications from the layers used by the application such as hardware and operating system. This work presents an approach without a claim for completeness but considers itself to be a step in the desired direction.

2 Domain Issues

To get a picture of what this work is all about, the reader needs some basic knowledge of the work an thesis taken by this work as starting points. Generally those items are dependability, fault hypotheses and fault tolerance as well as basic safety-relevant techniques and technical standards. When it comes to fault tolerance, the main focus is on fault detection, as in fail-safe systems, it is rather important to simply detect faults and shut down a system und consideration than to compensate the detected faults and keep the system running.

2.1 Dependability

When we talk about Safety, it is crucial to specify exactly what we mean, which is often not easy in this problem domain. Remember that e.g. Fault Tolerance deals with unintended system behaviours which are often hard to describe, or faults in Human-Machine-Interface design. Hence it is important to clearly define the terminology that is used in Safety literature and in my work.

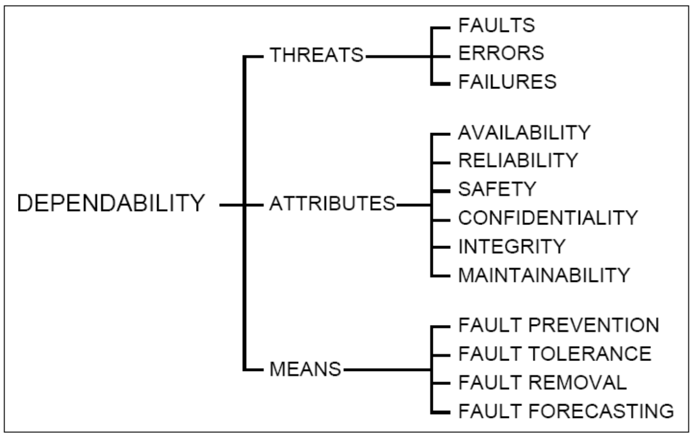


Figure 5: The dependability tree [Avi01].

However it is important to understand the environment the safety research domain is situated in, as it is a part of the Dependability discipline. [Avi01] gives a concise overview of the concepts, techniques and tool that have evolved over the past 40 years in the field of dependable computing and fault tolerance. Figure 5 gives a systematic exposition of the concepts of dependability in three parts: threats, attributes and means. According to [Avi01] Computing systems are characterized by five fundamental properties: functionality, usability, performance, cost, and dependability. It defines dependability of a computing system as the ability to deliver service that can justifiably be trusted, while a service delivered by a system in this context is meant defined as its behaviour as it is perceived by its users. Safety is considered as a sub-discipline and described as "absence of unauthorized disclosure of information" [Avi01].

As this context is a rather abstract one to talk about, it is especially important to clearly define the terms and paradigms that are under consideration. When we talk about fault tolerance, the basic approach is, to distinct between *faults*, *failures* and *errors*.

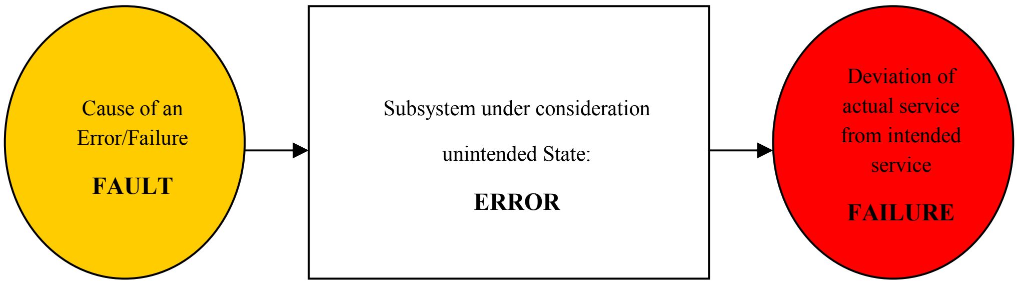

Cause of an Error/Failure
FAULT
Subsystem under consideration
unintended State:
ERROR
Deviation of actual service from intended service
FAILURE

Figure 6: Relationship between faults, failures and errors.

According to [Lev95, p.172] a *fault* is: "A requirements, design, or implementation flaw or deviation from a desired or intended state. " "A fault is the adjudged or hypothesized cause of an error. A *fault* may be *active* when it produces an error, otherwise it is *dormant*" according to [Avi01]. Figure 6 shows the relationship between faults, failures and errors. In developing fault-tolerance it is important to know what faults, more precisely, what kind of faults we want to tolerate or detect, as we generally don't know what fault will exactly influence our system. Otherwise we could eliminate it a priori. Therefore it is important to specify generically what we want to prevent to be able to implement suitable counter-measures.

Basically faults can be classified in design faults, physical faults and interaction faults. Faults can be described by their attributes. The following points are taken from [Avi01]:

- Phase of creation or occurrence: Determines if the fault originates from the development or operational phase.

- System boundaries: Specifies whether an operational fault is caused from somewhere within the system or brought in from outside like attacks or input mistakes. Developmental faults are by nature considered as internal faults.
- Domain describes whether it is a software or hardware fault.
- Phenomenological Cause: Is the fault a human-made mistake or attack or is it unforeseeably influencing the system by nature or vis maior.
- Intent: Does some intelligence stand behind a fault or is it caused by unintended. If it is intended, we have to distinguish between attacks and actions that are undertaken bona fide but result in a fault.
- Persistence: Faults may happen every time a certain service is performed – permanently - or they can happen several times, but don't occur on other pseudo-randomly - transient. Transient errors may be unexpected hardware errors originating in older chips or human-made faults by e.g. tired operators that perform a task in a false way from time to time, while taking the right actions most of the time.

Figure 7 provide a detailed overview of how fault classes and their attributes may be linked to a consistent concept.

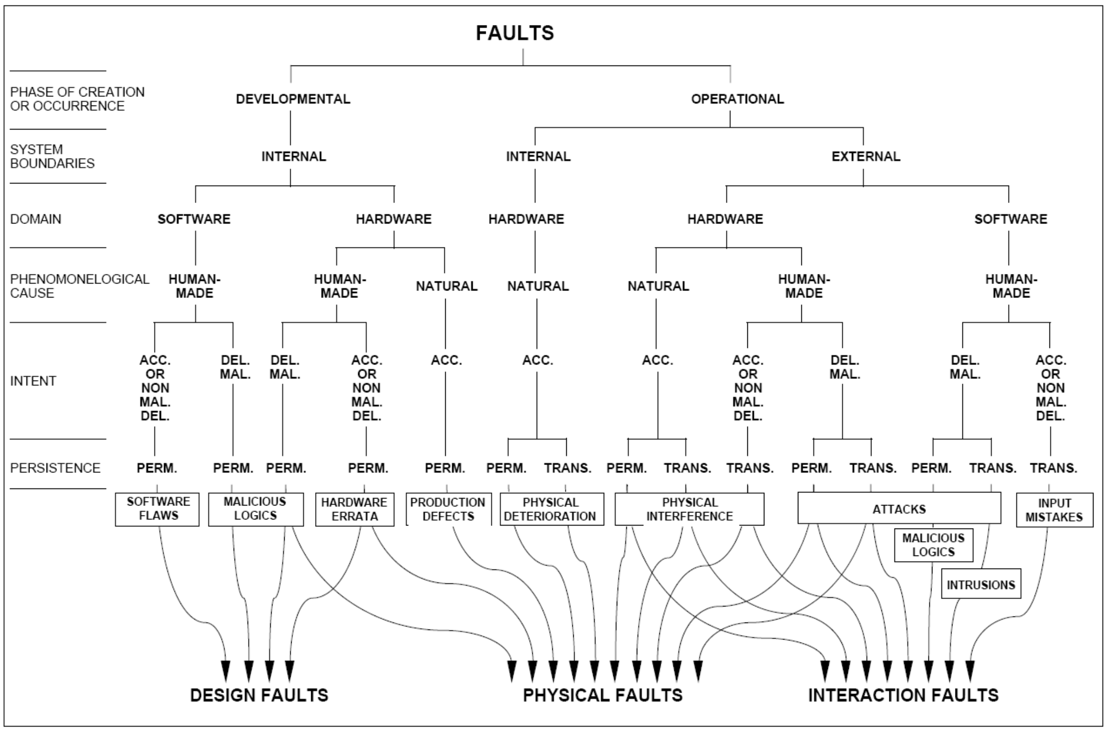


Figure 7: Combined fault classes [Avi01, p.6].

An **error** is a design flaw or deviation from a desired or intended state [Lev95, p.172]. In this context it is important that an error is just the deviation from a system's internal state and may not necessarily produce improper service. When an error reaches the service interface and alters the service, it produces a *failure* [Avi01, p.3].

According to [Lev95, p.172] a *failure* is the non-performance or inability of the system or component to perform its intended function for a specific time under specified environmental conditions. This is quite similar to the [IEEE 610.12-1990] definition: “The inability of a system to perform its required functions within specified performance requirements. “ In this work Leveson’s definition is used, as it is more detailed and it announces temporal and environmental constraints namely.

As a system not always fails the same way, [Avi01, p.3] states the classification of *failure modes* that can be ranked by *failure severities*. The paper characterizes incorrect service according to four viewpoints:

- failure domain
- controllability of failures
- consistency of failures, when a system has two or more users
- consequences of failures on the environment.

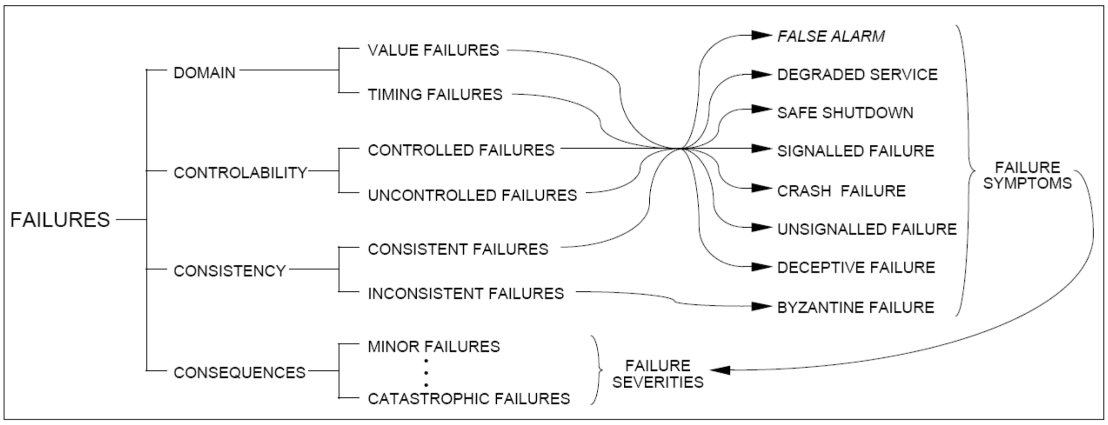


Figure 8: Failure modes according to [Avi01, p.3].

A special case of a failure symptoms are so-called Byzantine failures. They are inconsistent, malicious and asymmetric: In a multiple receiver scenario, the different receivers see differing, possibly incorrect, possibly correct, results. Those failures are especially a problem because they are hard to detect. An example of a Byzantine failure would be a clock sending one component the right time while sending another an incorrect timing value.

As stated above a fault may be dormant, which means that it doesn’t cause unintended changing of the state. This can happen, if e.g. there is a fault in a function but the execution branch the fault is contained in is never called or it is correct for data that is handled over most of the time. If this execution branch is then called with the data it is improper for, it leads to improper result, the *fault* is so-called *activated*. This leads to an *error* in the result computed or the action taken. If this erroneous state is passed through to the service interface of the system or component we call this a failure of a system or component. Failures of components – e.g. an algorithm is used and yields the wrong result – can be the *cause* for a *fault* in another component that relies on the faulty one. These phenomena build a chained action as sketched by Figure 9. It is a fundamental goal of fault tolerant systems to detect those errors a supervise component states in order to break that chain.

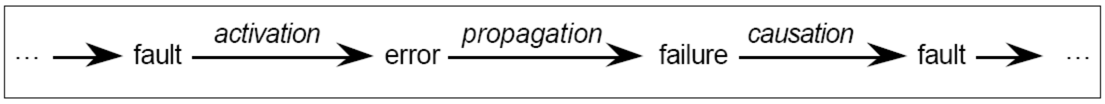


Figure 9: The fundamental chain of dependability threats [Avi01, p.14].

A system consists of a set of interacting components, therefore the system state is the set of its component states [Avi01, p.14]. As components often rely on each other faults occurring in one component are often handled over to the next component that relies on computations done or actions taken by the failed. As operating systems may be seen as so-called Off-The-Shelf components one crucial mission of my safety supervision facility for operating systems is amongst others to detect erroneous operating system functionality and prevent it from leading to wrong and possibly hazardous malfunctions of applications relying on it. Figure 10 shows the relationship between component and system failures.

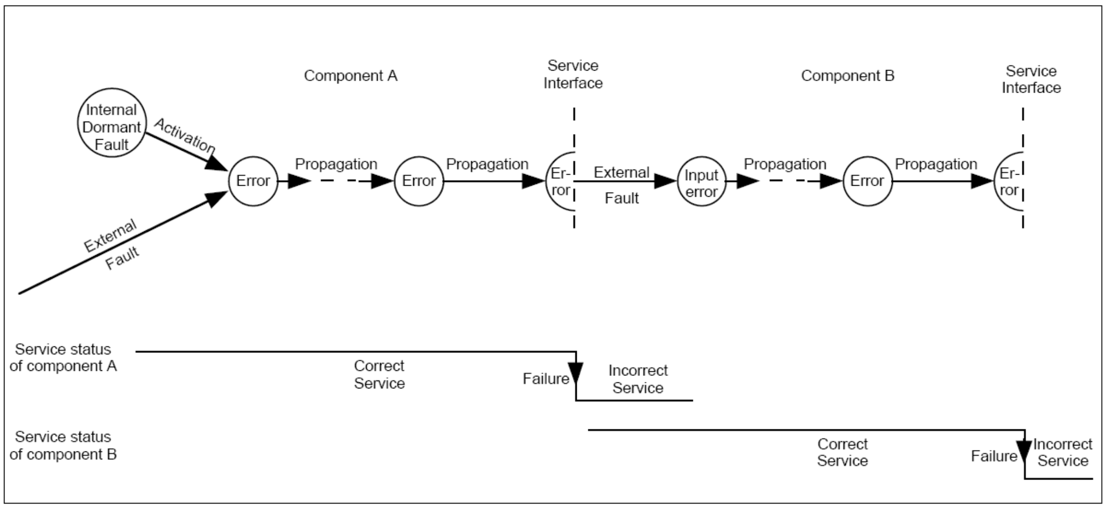


Figure 10: Error propagation in component-based systems [Avi01, p.13].

2.2 Fault Hypothesis

To be able to detect failures they have to be clearly stated and specified in the system specification. This has to be done by producing a so-called Fault Hypothesis. The Fault

Hypothesis has to state the assumptions about the type and number of faults that the fault-tolerant system has to tolerate [Kop06]. An important property of a Fault Hypothesis is the assumption coverage, which states to what extent these assumptions are met by reality. The assumption coverage limits the dependability of a perfect fault-tolerant system [Kop06]. Hence a fault-tolerant system can never be better than its Fault Hypothesis intentionally.

[Kop06] also states that without a precise specification of the Fault Hypothesis it is impossible to

- investigate whether the assumption coverage is realistic
- and to test the correctness of implemented fault-tolerance mechanisms.

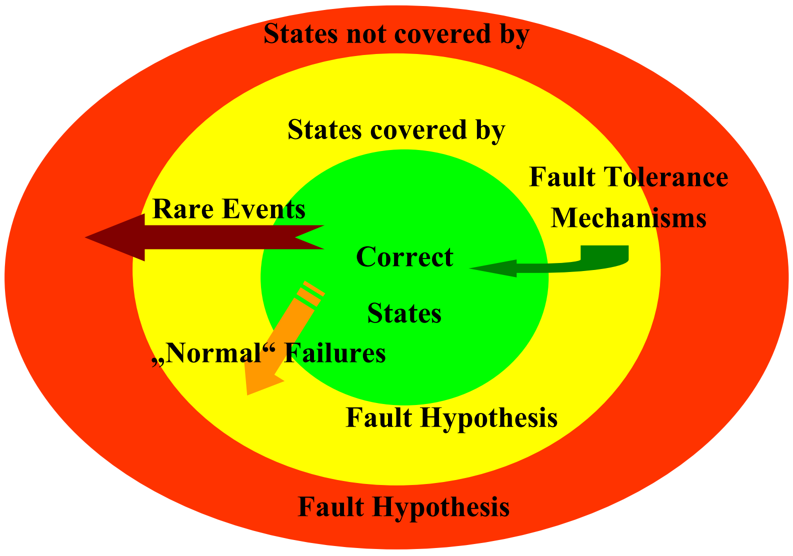


Figure 11: System states of a fault tolerant system [Kop06].

Basically, in this context faults are classified into three types: primary faults, secondary faults and command faults [Lev95, p.173].

- In a *primary fault* (and failure) the component fails within the design envelope or environment. Or - in other words – it is covered by the fault hypothesis and Fault-Tolerance mechanisms can be used to hoist the system back into the current state.
- *Secondary faults* are problematic because they aren't covered by system recovery mechanisms. Secondary faults occur when components fail because of excessive environmental stresses that exceed the requirements specification or design environment [Lev95, p.173]. Events that lead to secondary faults have to be rare events.

- *Command faults* involve the inadvertent operation of the component because of a failure of a control element-the component operates correctly, but at the wrong time or place [Lev95, p.173].

Another classification can be found according to [Kop06] in Level-1 and Level-2 faults whilst the first are covered by the Fault-Hypothesis and the second are not. [Kop06] also states that if during the test and installation phase it is found out that Level-2 faults are not rare events

- either the fault-hypothesis is wrong
- or the implementation is deficient.

2.3 Fault Tolerance

To increase the reliability of a system beyond the intrinsic reliability of a technology there are two strategies described [Kop06].

- Use of ultra-high quality components. This attempt has the advantage that no other structures have to be built in, but the major disadvantage that the cost increase in relation to component quality is exponential (e.g., early space program).
- Use of fault tolerance. This has the disadvantage that further ostensibly unproductive structures have to be built in. But it has the big advantage that the reliable system may be built of less-reliable components. Quite early – in the 1950s – J. von Neumann, E. F. Moore, and C. E. Shannon, and their successors developed theories of using redundancy that proved that reliable logical structures can be built of less reliable components [Avi01].

In high reliability systems a combination the above strategies may need to be used.

As the need for reliability and robustness are crucial parts of safety-critical systems fault-tolerant mechanisms have to be implemented. Nearly every technical system fails in its period of use and it often can’t be prevented when it fails in what way. There are studies dealing with fault prevention and failure rates. An important model that came out of those studies is the bathtub curve describing failure rates of electro-technical circuits. The bathtub curve consists of three periods: an infant mortality period with a decreasing failure rate followed by a normal life period (also known as "useful life") with a low, relatively constant failure rate and concluding with a wear-out period that exhibits an increasing failure rate [4].

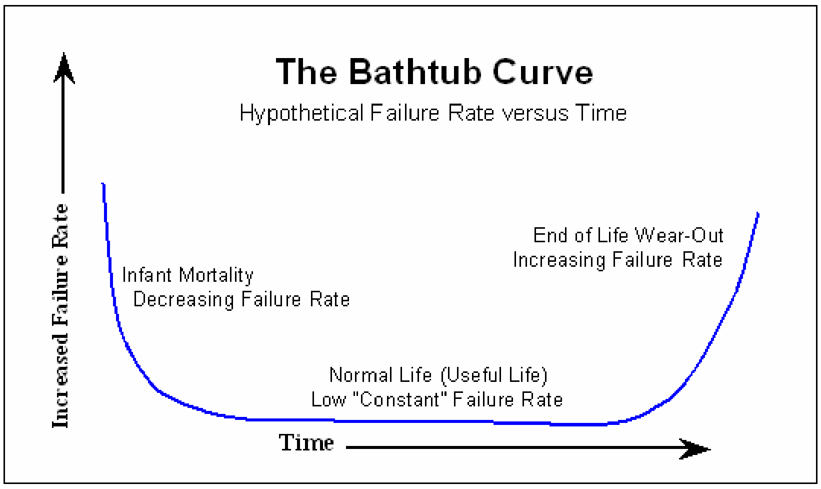


Figure 12: The bathtub curve taken from [4].

2.3.1 Fault-Tolerant Programming Paradigms

Development of fault tolerant system is not a trivial task. The trend toward increasing complexity and size, distribution on heterogeneous platforms, diverse accidental and malicious origins of system failures, the consequences of failures, and the severity of those consequences combine to thwart the best human efforts at developing these applications [Pul01]. In the environment of Fault-Tolerance a series of technologies have evolved able to compensate faults of system component. This chapter explains the most important of the paradigms and techniques used in fault-tolerant system development.

To understand why certain technologies are invented one has to understand the problems they try to solve. One of them is the so-called *Similar Errors or Lack of Diversity* [Pul01, p.60-62] problem. As faults may be nested in applications themselves, just copying or replicating the application will just replicate the error as well, so it is not sufficient for detecting such faults. Hence there is a need for diversity. Diversity allows the system to be able to detect faults using multiple versions of software and an adjudicator [Pul01, p.44-45]. An adjudicator is used to compare the compare the results to check them for errors. Nevertheless there is the possibility that 2 versions of an algorithm yield the same wrong result. [Pul01, p.60] states typical floating-point arithmetic calculations that may offer similar results within certain tolerance borders. In order to prevent this, the versions have to divert heavily in their method of resolution. There are two failure modes in this relationship defined by: If the variants fail on the same input case, than a *coincident failure* is said to have occurred. "If the actual measured probability of coincident variant failures is significantly different from what would be expected by chance occurrence of these failures, then the observed coincident failures are *correlated* or dependent" [Pul01, p.61]. In fault-tolerant systems – while it is crucial to detect failures – it is important to avoid such dependent failure modes.

Another basic problem is the *Consistent Comparison Problem* (CCP). The CCP limits the generality of the voting approach for error detection. The problem occurs as a result of finite-precision arithmetic and different paths taken by the variants based on specification-required computations [Pul01, p.62-68]. When N versions operate independently it is not clear that they consistently make the same decisions.

Another problem that is faced in backward recovery systems is the *domino effect*. This refers to the successive rolling back of communicating processes when a failure is detected in any one of the processes [Pul01, p.68-70]. As this is not a big problem in primitive sequential systems it gains importance in concurrent, distributed systems. Here these effects may cause the system to fall into an inconsistent state. A consistent state allows the system to achieve an error-free state that leads to no contradictions and conflicts within the system and its interfaces. All communications between processes and their order of occurrence are taken into account. To support consistency, some restrictions on the communication system must be enforced.

The following paragraphs describe some of those constraints and common programming constructs used in reliability engineering.

2.3.1.1 Assertions

An executable assertion is a statement that checks whether a certain condition holds among various program variables, and, if that condition does not hold, takes some action [Pul01, p.78-80]. This definition shows that assertions are a construct that allows us to check if a system state satisfies certain reasonableness conditions, which define a correct, consistent system state.

2.3.1.2 Checkpointing

Checkpoints are points in time where a consistent system state is stored. If there is an error detected after that checkpoint it is possible to perform a rollback of the system state to it and perform the desired action again. Checkpoints may be part of a never-give-up strategy and are not explicitly used in my practical work as it rather deals with error detection than with fail-operational environments.

2.3.1.3 Atomic Actions

Atomic Actions are sets of commands that are performed fully or not. They are used preferably in concurrent distributed systems for error recovery. There are three characteristics describing this constructs. They are according to [Pul01, p.84-88]:

- *Indivisible*: Either all the steps in the atomic action complete or none of them dows, that is, the “all-or-nothing” property.

- *Serializable*: All computation steps that are not in the atomic action either precede or succeed all the steps in the atomic action.
- *Recoverable*: The external effects of all the steps in the atomic action either occur or not; that is, either the entire action completes or no steps are completed.

2.3.2 Design Diversity

Design diversity is the provision of identical services through separate design and implementations [Pul01, p.29-35].

2.3.2.1 Manual Design Diversity

This denotes techniques to detect failures by trying to reach the desired functionality by implementing a system or subsystem more then once. In the *N-Version Programming* technique [Pul01, p.12-14] software is produced in diverse ways by different people and – ideally – using different techniques. The certain versions are executed in parallel and their results are compared and adjudicated. There is a lot of literature comparing advantages and disadvantages of putting all effort into the production of "one good version" versus using an N-Version approach. What can be said is that these techniques yields the best results if it is applied from the very beginning – before the specification – to the very end of a component's development cycle.

2.3.2.2 Automated Design Diversity

This denotes that diversity is reached through redundant structures according the control flow and the constructs used. [PUL01, p.29-35] gives some examples for automated design diversity:

- Utilization of different processor registers in different versions;
- Transformation of mathematical expressions;
- Different implementations of programming structures;
- Different memory usage;
- Using complementary branching conditions in the variants by transforming the branch statements;
- Different compilers and libraries;
- Different optimization and code-generation options.

2.3.3 Data Diversity

Data diverse techniques use different representations of data to perform computations on them. Data re-expression algorithms are used in determined points of time and decision algorithms are used to adjudicate the output. Examples for reasonable fields of use of data diversity are:

- Detecting data transmission errors like bit flips;
- Detecting errors in floating point computations;
- Rounding errors;

Apart from data and design diversity there exist several other diversity techniques like temporal diversity, but those are not relevant for my work.

2.4 Hazard Analysis

In order to prevent catastrophic events pragmatic approaches are needed and dangerous situations and their precursors have to be identified to find measures to prevent them from leading to serious losses. To reach this goal, safety engineers commonly use the term *Hazard*.

2.4.1 Accident and Incident

A straight forward approach to engineering safety is to classify events by their result concerning the potential damages they cause. [Lev95, p.175-176] differentiates those events into *accidents* and *incidents*.

She defines an *accident* as "an undesired and unplanned (but not necessarily unexpected) event that results in (at least) a specified level of loss".

The first part of this definition "undesired and unplanned" is quite common what natural language means by an accident. But natural language means something unexpected as well which could be fatal in a safety engineering context. In a safety related environment it is crucial to identify unexpected, but possible, events that may lead to accidents as well in order to be able to provide counter-measures to minimize damages caused by an accident.

The second part of this definition says that an accident always relates to a "specified level of loss". This is an important topic for damage reduction to determine what level of concern has to be taken in order to prevent such an event. Typically those situations are classified into minor, serious and catastrophic events [Lev95, p.175].

The definitions of accidents are very important as they strongly influence the way counter-measures are taken to control those situations and increase safety. "For example, if an accident is defined as an unwanted or uncontrolled release of energy, then prevention methods should focus on energy controls and barriers between possibly harmful energy flow and the things damaged by it." [Lev95, p.176] This states that engineers are able to provide measures to minimize damage if they know what can happen.

Another term in this context is *incident*. "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, p.176] The concept of the incident is very important to find failures that probably lead to serious accidents.

Note that the difference between incident and accident is just the result. The prerequisite of an incident not being an accident is that just the circumstances under which a failure has happened didn't lead to a catastrophic event. For example the steering of an airplane is transiently defiant so that the pilot is not able to pull up for 5 minutes flying in a height of 9 kilometres. This would be an incident that has to be investigated and the failure has to be found and eliminated, as an event like this would certainly lead to an accident if it happened when the plane would have been flying just in front of a mountain.

2.4.2 Hazards

To prevent accidents and near misses designers have to know what they have to be aware of. In other words, they have to know the detailed precursors that lead to situations that are potentially dangerous in specific environments. This is what the safety community created the term *hazard* for.

[Lev95, p.177] provides the following definition: A *hazard* is a state or a set of conditions of a system (or an object) that, together with other conditions in an environment of the system (or the object), will lead inevitably to an accident (loss event).

There are two things to note about that definition:

- A hazard is always defined in relation to its environment
- What constitutes a hazard depends upon where the boundaries of the system are drawn (Lev95, p.177).

Another term that is important in this context is *risk*. *Risk* is the hazard level combined with the likelihood of the hazard leading to an accident (sometimes called *danger*) and hazard exposure or duration (sometimes called *latency*) (Lev95, p.179), while hazard level is itself a combination of hazard severity and likelihood of its occurrence.

2.4.3 Hazard Analysis Process

The goals of safety analysis are related to three general tasks (taken from [Lev95, p.289]), while the first two are basically to making systems safer and the last is to convince management and authorities that a system is safe:

- Development: the examination of a new system to identify and assess potential hazards and eliminate or control them.
- Operational management: the examination of an existing system to identify and assess hazards in order to improve the level of safety, to formulate a safety management policy, to train personnel, and to increase motivation for efficiency and safety of operation.
- Certification: the examination of a planned or existing system to demonstrate its level of safety and to be accepted by the authorities or the public.

2.4.4 Relevant Fault Analysis Models

In literature a large number of fault and hazard analysis techniques can be found [Lev95, p.344-346, p.317-326, p.497-507] [Avi01] [7] [8]. Many of them may be used alternatively, but mostly some of them are used additionally to ensure all possible faults are examined. The main difference of the structured techniques is the starting point – simplistic: the cause or the effect. Such analyses may be reasonably augmented with certain problem domain specific checklist approaches to ensure that nothing known is overseen. For example such checklists exist for aviation, chemical industry and many more. For this work, especially for the safe message passing library, two common analysis techniques are used: Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA). FMEA is used in an adapted form as numerical analyses are hard to perform on software.

2.4.4.1 Fault Tree Analysis

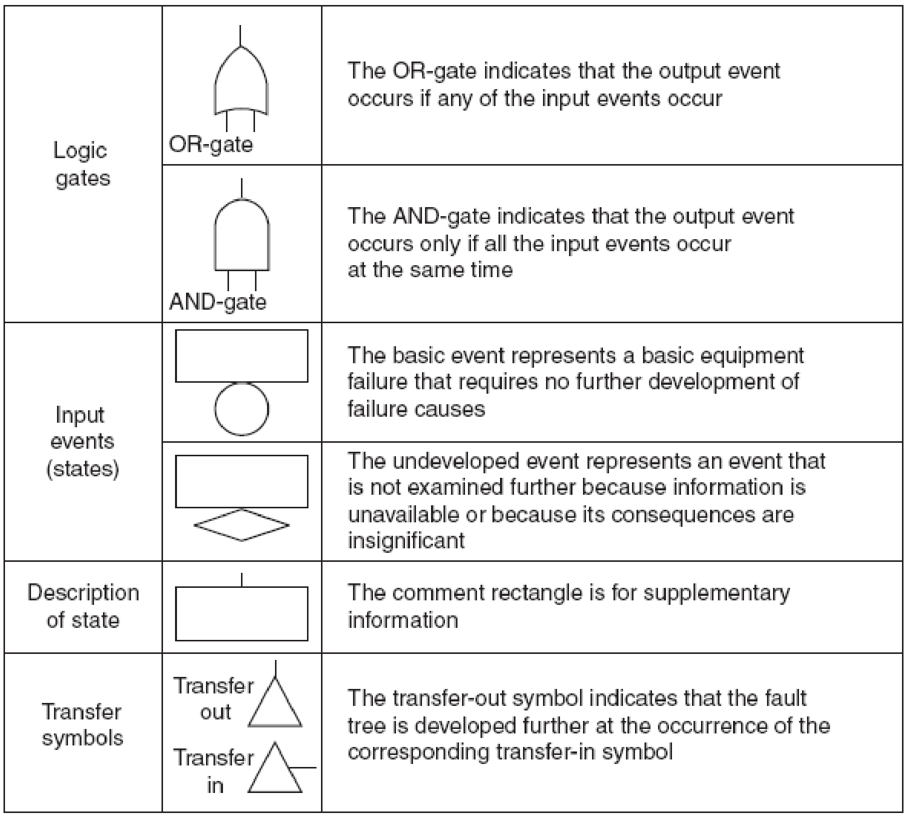
Fault Tree Analysis (FTA) is a technique for reliability and safety analysis. Bell Telephone Laboratories developed the concept in 1962 for the U.S. Air Force for use with the Minuteman system. It was later adopted and extensively applied by the Boeing Company [7]. Fault Tree diagrams are logical block diagrams that represent the state of a system as a combination of its component states using logical Boolean gates. They are built so-called top-down strategy from the undesired event connecting the causes that lead to the causing base events through logical gates.

[8] as well as other references in literature describe the *fault tree analysis process* as follows:

- Definition of the system, the *top* event (the potential accident), and the boundary conditions
- Construction of the fault tree

- Identification of the minimal cut sets
- Qualitative analysis of the fault tree
- Quantitative analysis of the fault tree
- Reporting of the results

Figure 13 show the items such diagrams consist of.



Logic gates	OR-gate	The OR-gate indicates that the output event occurs if any of the input events occur
	AND-gate	The AND-gate indicates that the output event occurs only if all the input events occur at the same time
Input events (states)		The basic event represents a basic equipment failure that requires no further development of failure causes
		The undeveloped event represents an event that is not examined further because information is unavailable or because its consequences are insignificant
Description of state		The comment rectangle is for supplementary information
Transfer symbols	Transfer out Transfer in	The transfer-out symbol indicates that the fault tree is developed further at the occurrence of the corresponding transfer-in symbol

Figure 13: Fault tree diagram symbols [8].

In order to be able to generate a Fault Tree analysis it is important to have fully understood how the system under consideration works. Usual Figures to reach this are system block diagrams. Figure 14 shows how a fault tree is built for a subsystem consisting of a valve and two redundant water pumps of a fire pump system.

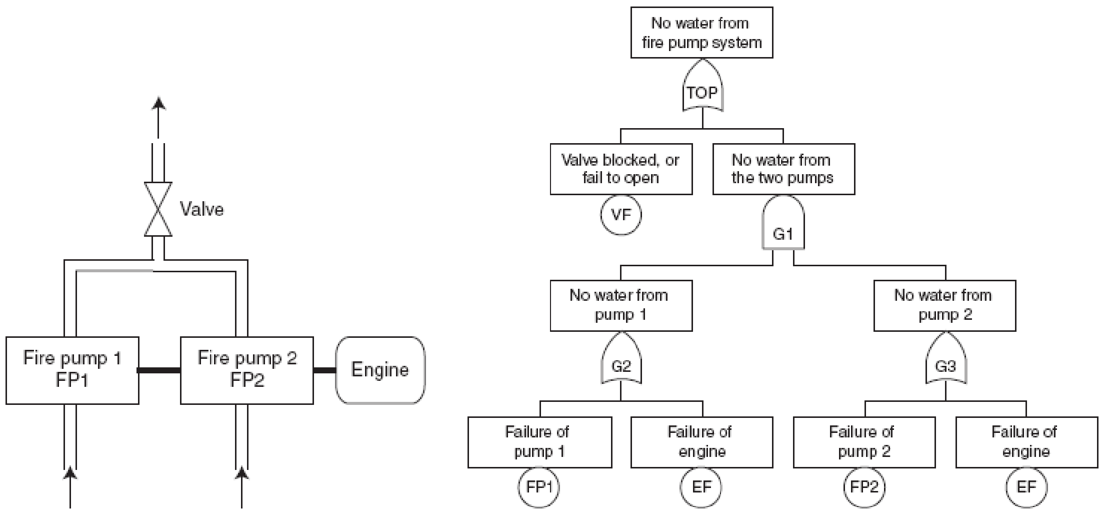


Figure 14: Fault tree diagram example given by [8].

Fault trees show relationships between events and are snap shots of a system's state. [Lev95, p.317] states, that they have originally been developed to calculate quantitative probabilities, although it is more commonly used qualitatively. The book also notes that "simply developing the tree, without analyzing it, forces system-level examination beyond the context of a single component or subsystem". Nevertheless they are a good visualization and provide a solid base for further analyses and solution development.

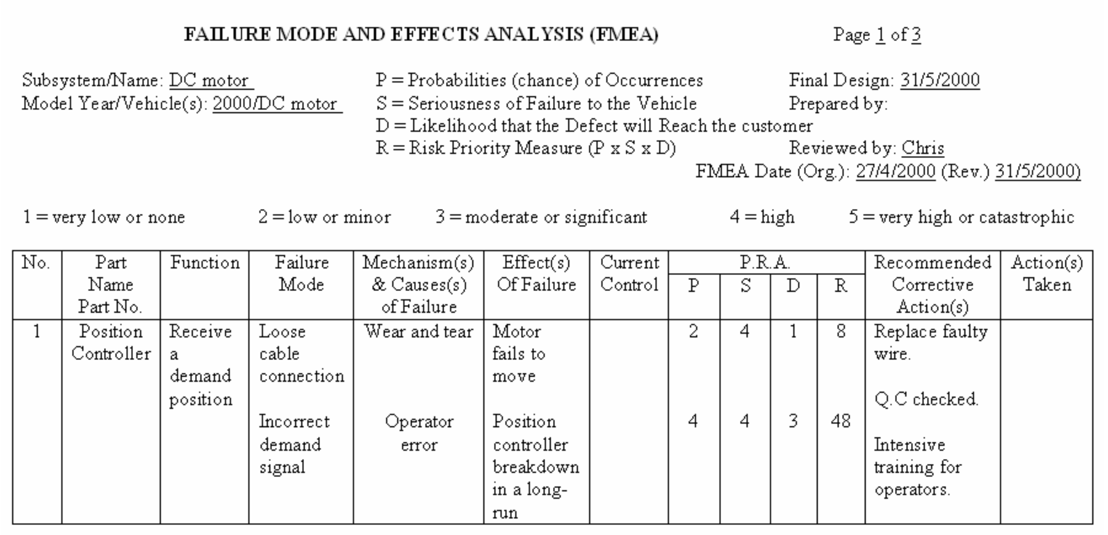
2.4.4.2 Failure Modes and Effects Analysis

Failure Mode and Effects Analysis (FMEA) is risk assessment technique for systematically identifying potential failures in a system or a process. It is commonly used in the industry in various phases of the product life cycle. *Failure modes* means the ways, or modes, in which a system under consideration might fail. Failures are any errors or defects, especially ones that affect persons or material. *Effects analysis* refers to studying the consequences of those failures.

The FMEA process was originally developed by the US military in 1949 to classify failures "according to their impact on mission success and personnel/equipment safety". FMEA has been used on the 1960s Apollo space missions. In the 1980s it was used by Ford Motor Company to reduce risks after a car model suffered a design flaw that failed to prevent the fuel tank from rupturing in a crash, leading to the possibility of the vehicle catching on fire.

FMEA is effective for analyzing single units or single failures to enhance individual item integrity. It can be used to identify redundancy and fail-safe design requirements, single-point failure modes, and inspection points and spare part requirements [Lev95, p.341-343]. It was originally developed to be able to generate mathematical predictions about failure rates. Statistical data about software failure rates is hard to get in my field of work so the failure modes and effects analysis in this thesis concentrates on the effects, leaving the expected component failure rates besides.

Basically Failure Modes and Effects Analyses contain a tabular compilation of the possible failure modes and their effects. Consequentially the possible failure modes are given IDs and counter measures are proposed. In general those documents contain numerical data for probability and criticality quantifiers, but those data are hard, sometimes impossible to get for software. There is no standardized procedure to generate those Analyses as their form changes from company to company. An example is shown below in Figure 15.



FAILURE MODE AND EFFECTS ANALYSIS (FMEA) Page 1 of 3

Subsystem/Name: DC motor
Model Year/Vehicle(s): 2000/DC motor

P = Probabilities (chance) of Occurrences
S = Seriousness of Failure to the Vehicle
D = Likelihood that the Defect will Reach the customer
R = Risk Priority Measure (P x S x D)

Final Design: 31/5/2000
Prepared by:
Reviewed by: Chris
FMEA Date (Org.): 27/4/2000 (Rev.) 31/5/2000)

1 = very low or none 2 = low or minor 3 = moderate or significant 4 = high 5 = very high or catastrophic

No.	Part Name Part No.	Function	Failure Mode	Mechanism(s) & Causes(s) of Failure	Effect(s) Of Failure	Current Control	P.R.A.				Recommended Corrective Action(s)	Action(s) Taken
							P	S	D	R		
1	Position Controller	Receive a demand position	Loose cable connection	Wear and tear	Motor fails to move		2	4	1	8	Replace faulty wire. Q.C checked.	
			Incorrect demand signal	Operator error	Position controller breakdown in a long-run		4	4	3	48	Intensive training for operators.	

Figure 15: Failure Modes and Effects Analysis example taken from [9].

2.5 The Railway Domain

In my work I concentrated on railway specific problems in depth, as it is the project partner's main interest. Most safety related railway systems are so called fail-safe, which basically means that we can always switch to a safe state – mostly "system is shut down" – in case of a failure where the system stays stable and there's the guarantee that no safety-critical situation can be initiated. The circumstance that we can switch to this means that the most important part is error detection, as just turning off the system isn't very complicated. The opposite of fail-safe are fail-operational systems where one would have to guarantee a specific service level under which the system must never get, e.g. an airplane has to reach the next airport in case of an error.

A crucial part of my work is to guarantee safe message passing, because the platform I am talking of is a network of distributed control nodes which mustn't lose communication between each other.

2.6 CENELEC

CENELEC - *European Committee for Electro-Technical Standardization* - is one of the three biggest standardization organizations in Europe. It has evolved from a series of earlier European standardization organizations since the second half of the 19th century and was actually founded in 1973. “CENELEC is developing and achieving a coherent set of voluntary electro-technical standards as a basis for the creation of the Single European Market/European Economic Area without internal frontiers for goods and services”. [1] Besides other issues it prescribes the safety requirements technical systems have to fulfil to be admitted for public use in most of Europe.

2.7 CENELEC 50159 Standard

As for other problem domains it worked out CENELEC 50159 which defines requirements to methods for message passing, where [CEN50159-1] specifies the target in so called closed systems, CENELEC 50159-2 is related to open systems.

In case of faults that may compromise safety, the fault must be detected an a suitable reaction has to be initiated. It says that for the safety requirements defined in the standard a single physical transmission path is sufficient. Safety critical faults are detected through safety procedures and redundant safety codes that interact above the unsafe protocol of the transmission system. Reliability is not a topic in the standard but has to be reminded in order to build safe systems. In our use case the application of [CEN50159-1] for closed systems is applicable.

According to the standard the following prerequisites to a communication system have to be fulfilled for it to be considered as a closed system:

- Only authenticated access is allowed.
- There exist a known maximal number of communication participants.
- The transmission media is known and specified properly.

The standard explicitly omits the definition of the transmission system, the devices plugged to the transmission system, specific solutions or which kinds of data are safe from a signal technical point of view.

2.7.1 The Grey Channel

The essence of the standard in short is to sketch how signal technical transmission functions may be plugged on a signal technical transmission channel to provide fault detection. The goal is to see the communication system as a *grey channel*. This means that the EN 50159 standard allows the attachment of signal technical unsafe communication media to signal technical safe systems. In such a case when an unsafe media is used for safe communications the transmission functions have to ensure correct communication. It has to be guaranteed that the failures done by the unsafe communication channel are detected by the safe transmission functions and safety reactions are initialized. Therefore the standards define common failure modes and requirements to those functions. Figure 16 shows the concept of how to wrap a possibly unsafe communication channel at the end points using techniques like checksums sequencing, checking of timely constraints and other common methods.

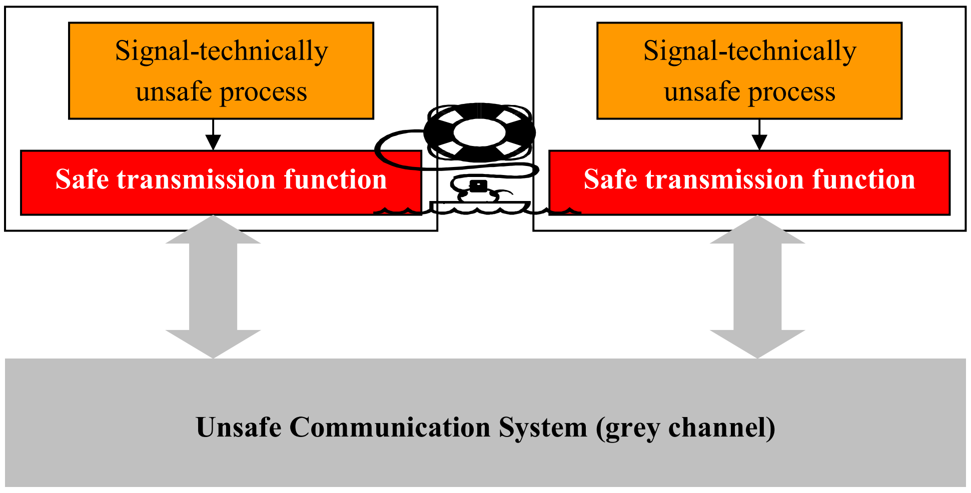

Signal-technically unsafe process
Safe transmission function
Signal-technically unsafe process
Safe transmission function
Unsafe Communication System (grey channel)

Figure 16: Wrapping an unsafe communication channel with safety techniques.

In our case everything below the POSIX layer can not be classified in any CENELEC Safety Integrity level as the pedigree is not known and the code has not run through validation and verification. So for the Safe Message Passing chapter everything starting from the message passing system calls below is seen as a grey channel and the Safe Message Passing Library implements the techniques demanded by [CEN50159-1] for closed communication systems.

2.7.2 Safety Assurance Technologies Covered by the Standard

[CEN50159-1] defines the following cases that are mandatory to be covered by a standard compliant system.

- Sender error (authentication)
- Data type error
- Data value error
- Discovery of old data or data that has not been received in specified timely constraints.
- Detection of communication losses
- Assurance of functional independence between the unsafe transmission media layer and the safe transmission functions

So as the list shows there are generally two types of errors identified - Data errors and timing errors. Data errors cover all types of influence according the information of sent messages, such as electro-magnetic influence, errors in header fields such as the descriptor specifying the originator of a message, etc., whereas timing errors describe problems according points of time messages are sent or received, respectively. This can be errors detected by the timestamps attached or discovered discontinuities in the sequence of the messages.

Overall six functional as well as fourteen safety requirements are formulated by [CEN50159-1]. These requirements are described in depth and possible technologies are enumerated. The project partner’s railway safety platform is a system for which Safety Integrity Level 4 is required. Although the standard does not define any concepts for guaranteeing reliability, it provides methods for calculating the resulting system’s estimated Mean Time between Failure (MTBF). A comprehensive discussion of how I implemented the specifications required by the standard is provided in the “Safety Library” Chapter.

In 5- Safe POSIX Message Passing a full Failure Modes and Effects Analysis is provided and compared to the standard. This yields strong arguments for justifying the Safe Message Passing library.

2.8 Safety Integrity Levels

Safety Integrity Levels describe standards systems have to be built after and are an indicator of the possible divergence of the systems behaviour from the intended. They are introduced by standardization or governmental organizations and embody a measure of whether the system is to be admitted or not. There is no common set of SILs over all standards that deal with safety-critical system. A number of standards define their own requirements, as the risks and needs crucially depend on the problem domain. SILs do not only describe what measures have to be undertaken, but also state certain requirements to the development and risk assessment process.

In [IEC 61508] and in some other standards, mappings are provided between the tolerable hazard occurrence rate (for continuous or high demand systems) and the target Safety Integrity Level [HSE02]. Safety related functions are given extensive reliability requirements to provide risk reduction. As stated above Components used in a system may have higher or lower reliability that the overall system. In the second case reliability has to be hoisted by implemented redundant structures described in the fault-tolerance chapter.

The following description is given for this relationship by [HSE02, p.6]: "For a software component (either a complete software system running on one computer, or one element in a partitioned software system), the SIL of the software is determined by its apportioned failure rate using a mapping from target ranges of failure rate to SIL. In some standards such as DO-178B the equivalent of the SIL is derived from failure consequence rather than risk, but this makes little difference to the overall principle."

2.9 Other Industry Standards for Railway Systems

Beneath other standards there are a family of CENELEC standards that define terms and landmarks for railway systems. They define a series of technical terms and suggest a number of measures to be taken. For this work, besides the 50159-1 suggestions the following three standards had to be taken into account [EN50126], [EN50128], [EN50129].

2.9.1 EN 50126

This subsumes proposals for the specification and approval of Reliability, Availability, Maintainability and Safety (RAMS) for railway applications. The first paragraphs give definitions of important safety terms, the following prescribe a process based on the system lifecycle and its activities concerning RAMS management. The paper announces specific railway problems, but does not prescribe rules or processes for the certification or process for approval of railway components.

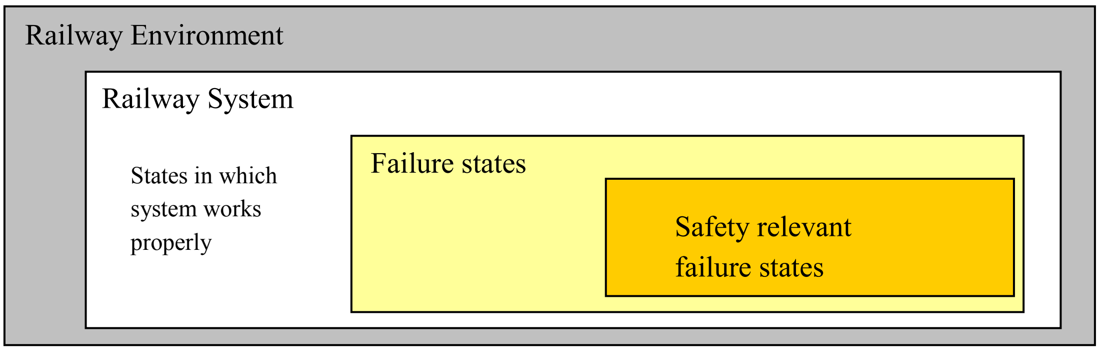

Railway Environment
Railway System
States in which system works properly
Failure states
Safety relevant failure states

Figure 17: Safety relevant failure states [EN50126].

Figure 17 shows how failures are seen within the operational context. The elements of RAMS for railway systems are identified as safety, dependability, maintainability and operation. Factors influencing it are classified under system, operation and maintenance conditions. Furthermore a risk assessment technology is proposed for railway systems, linking the commonness of the failure to the hazardousness and effect of the state it initiates.

2.9.2 EN 50128

This standard for telecommunications, signalling and data processing systems defines an own set of five safety integrity levels for safety-critical software in railway system, of which level 0 means that the software is not safety-critical at all and level 4 systems are of most criticality. In accordance with [EN50126] and [EN50129] it prescribe the creation of a requirements documentation, specification of a system's safety requirements, description of a system's architecture and a system safety plan. Figure 18 shows how the stated SILs are determined for a system under consideration.

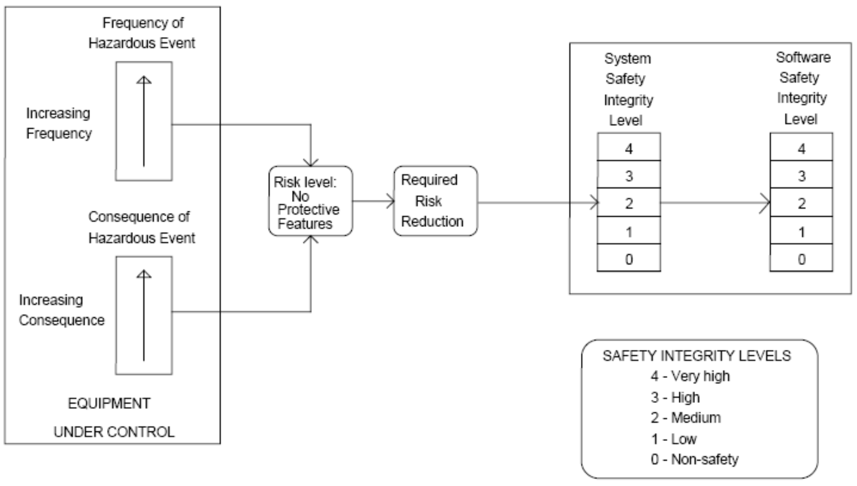


Figure 18: Safety Integrity Levels [EN50128].

It enumerates and describes the development phases and the items that have to be produced in each phase. The document makes use of usual models like the V-model of software development and maps the certain product items to development stages. Furthermore mandatory personnel and roles are required depending on what safety integrity level is desired. Figure 19 gives a glance of what activities are prescribed to be done in which phase of the system's lifecycle.

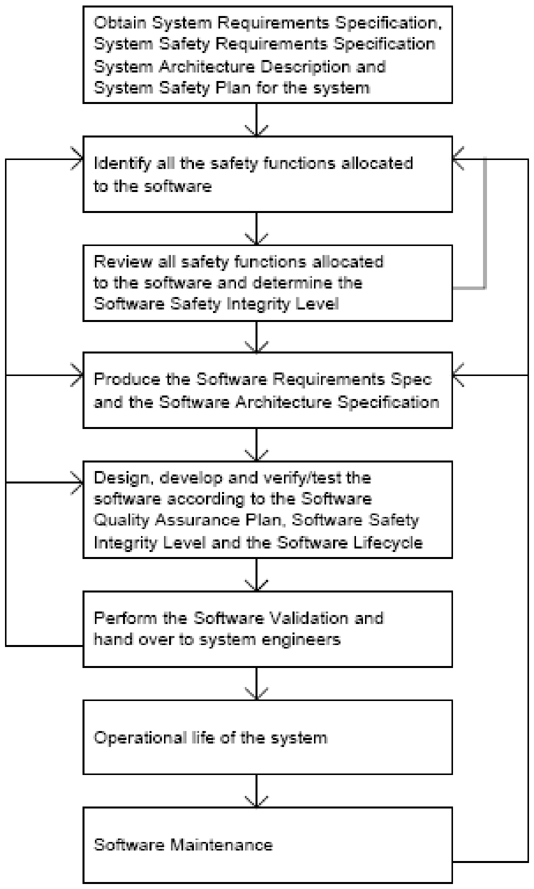

Obtain System Requirements Specification, System Safety Requirements Specification System Architecture Description and System Safety Plan for the system
Identify all the safety functions allocated to the software
Review all safety functions allocated to the software and determine the Software Safety Integrity Level
Produce the Software Requirements Spec and the Software Architecture Specification
Design, develop and verify/test the software according to the Software Quality Assurance Plan, Software Safety Integrity Level and the Software Lifecycle
Perform the Software Validation and hand over to system engineers
Operational life of the system
Software Maintenance

Figure 19: Software safety route map [EN50128].

2.9.3 EN 50129

[EN50129] defines the conditions a safety-relevant electronic railway system, subsystem or system under consideration is accepted as reasonably safe basically requiring three items:

- **Demonstration of Quality Management**, which must show that the quality of the system can be guaranteed because of the measures that have been undertaken during development, and those that are planned to be performed during operation and maintenance.

- **Demonstration of Safety Management**, which has to declare measures like safety plan, test planning, organizational structure, hazard log strategy, reports and plans for safety reviews and safety argumentation. In short this declares that the process a product is developed and maintained after was suitable.
- **Demonstration of functional and technical Safety.** Contains a demonstration of functional safety, specification of failure effects, external influence, application conditions and the proof that the system has been tested under operational conditions.

Furthermore the definition of the system declaring precisely what the system does, as well as version number and change state, and requirements, design and application documentation. Also contained in the safety demonstration documentation are relations to other safety demonstrations for the subsystems.

3 Operating System Issues

The proposed layer will be based upon the operating system layer. It is supposed to detect errors that occur below the operating system interface. Hence it is important to delimit the items and functionality that shall be tested. This chapter describes the operating systems and interfaces the Safety Supervision Layer has to be aware of. Furthermore important tasks according Linux operating systems and systems safety are elaborated.

3.1 POSIX

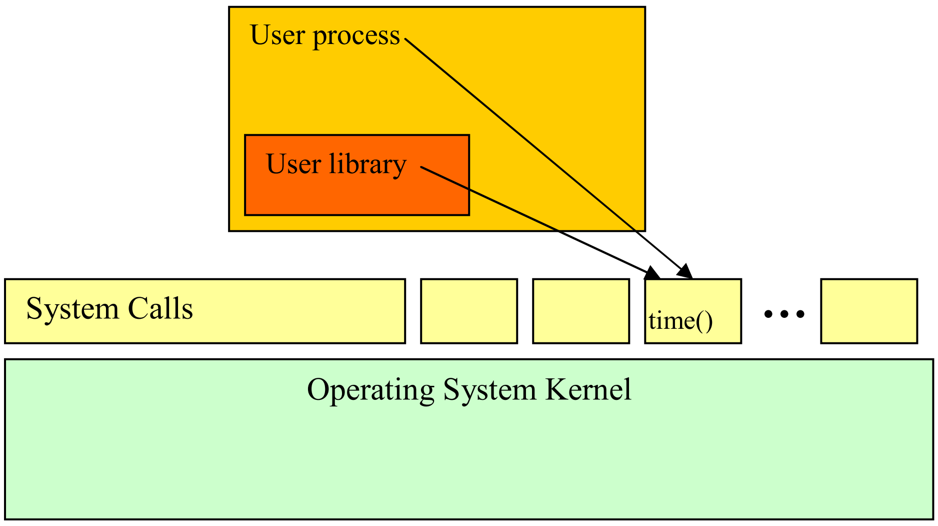


Figure 20: POSIX Portable Operating System Interface concept.

As most software does many UNIX/ Linux applications use system calls to provide their functionality. System calls provide an interface to the underlying operating system. So a change of a system call would mean that one would have to change the kernel, which is obviously bad for reusability, increasing the cost of change and a the avoidance of undesired side effects gets hard to handle.

Figure 20 shows POSIX's concept of encapsulation of the actual underlying operating system distribution by a standardized interface.

This circumstance is all the more problematic as we know that even in so-called UNIX-like operating systems particularly these system calls may differ from distribution to distribution, which makes it hard to guarantee compatibility.

In the 1980s several initiatives have been raised to master the rank growth of slightly differing UNIX versions and create standards that software developers could rely on. In 1986 the Institute for Electrical and Electronic Engineers created some standards out this called Portable Operating System Interface. POSIX – the short name - is the collective name of a family of related standards specified by the IEEE to define the application programming interface for software compatible with variants of the UNIX operating system [2]. The first version of IEEE POSIX 1003.1 has been applied in 1988. Currently most UNIX operating systems support POSIX, but there exist others that support the standard as well. The family of POSIX standards is formally designated as *IEEE 1003* and the international standard name is ISO/IEC 9945.

POSIX documentation is available at IEEE and is divided into three parts (taken from [2]):

- **POSIX Kernel APIs** (which include extensions for POSIX.1, Real-time Services, Threads Interface, Real-time Extensions, Security Interface, Network File Access and Network Process-to-Process Communications)
- **POSIX Commands and Utilities** (with User Portability Extensions, Corrections and Extensions, Protection and Control Utilities and Batch System Utilities)
- **POSIX Conformance Testing**

A test suite for POSIX accompanies the standard. It is called PCTS or the POSIX Conformance Test Suite.

The POSIX standard family has emerged continuously since its introduction in 1985. It has experienced important extensions and upgrades. Among those are some prerequisites for safety critical systems. The list below shows important POSIX upgrades. It is taken from [2]:

- POSIX.1, Core Services (incorporates Standard ANSI C)
    - Process Creation and Control
    - Signals
    - Floating Point Exceptions
    - Segmentation Violations
    - Illegal Instructions
    - Bus Errors

    - Timers
    - File and Directory Operations
    - Pipes
    - C Library (Standard C)
    - I/O Port Interface and Control
- POSIX.1b, Real-time extensions
    - Priority Scheduling
    - Real-Time Signals
    - Clocks and Timers
    - Semaphores
    - Message Passing
    - Shared Memory
    - Asynch and Synch I/O
    - Memory Locking
- POSIX.1c, Threads extensions
    - Thread Creation, Control, and Cleanup
    - Thread Scheduling
    - Thread Synchronization
    - Signal Handling

Currently a significant number of commonly used operating systems comply to the IEEE 1003 family. In order to avoid and eliminate portability problems – which is actually POSIX's main purpose – system vendors have to take a rigid and expensive certification process to get their product officially certified. Currently certified common products are e.g. Microsoft Windows NT Kernel based systems, a number of Linux distributions, Solaris and a number of others.

Since it is possible now to switch from one operating system to another without rewriting the code above, this standard is a huge leap forward for code reusability. We can simply change a POSIX compatible operating system with another POSIX compatible without being forced to change the application.

In our case the situation is a bit more complicated. As every bite of code has been verified and validated and further analyses have to be provided, the mission in case of an operating system change would be to do the safety proof for the whole operating system again, which is not a convenient solution. Operating systems get more and more complex, in Linux operating systems memory management testing get more and more costly, time consuming and complicated as cache level amount and lines of code increase.

So a Safety Supervision Layer above the POSIX system call layer but below the application layer is desirable that encapsulates all the safety testing and possible faults that all POSIX operating systems have in common. This end-to-end approach is expected to save a lot of costs for verification and validation having the potential to provide better, because more understandable and comprehensive, safety solutions.

3.2 Safety Problems faced using Open Standards

So those are key advantages of using systems that are compliant to a broadly spread standard. But as one can reuse code across platform one often lacks good knowledge of the system implementation. You can never actually know if there are still systematic errors below your application level. So this circumstance is a possible safety threat and the issue where the supervision layer comes into play. It checks the required functionality in an End-to-End way, as independent of the implementation as the application that utilizes it.

3.3 OTS Issues in Safety-Related Environments

The main goals of off-the-shelf component based architectures are lower cost, increased quality, lower time-to-market and decreased development time. An idealistic solution is to use the components as a black box. As OTS development decreases development time it increases the amount of time to be invested into integration of the components. Apart from that it gets more difficult to demonstrate that a component fulfils given safety requirements and often it is impossible to give proof of a mature development process of a component, as the buyer of a commercial or user of an open source component often does not have insight to how the software evolved. ‘Proven in Use’ is often used as evidence for safety-critical components and their ability to fulfil the safety requirements when they are used. [Aas07]

Another term that can be found in the literature is SOUP – Software of unknown pedigree: “Software, the pedigree of which is unknown or uncertain, which could in any way affect the correct operation of a safety-related system” [HSE01]. Increased quality as one main OTS architecture goal does not automatically give safer systems, but can give increased reliability. Hence, if OTS components are used as they were intended in the system and the system context, this can give safer systems. [Aas07]

The most common safety standards, such as [IEC61508], [DO-178B], and [EN50128], have strict requirements to how software should be developed and tested. None of these standards explicitly allow the use of COTS. [Skram07] But there are ways to integrate those pieces of software safely. E.g. the supplier enforces a system architecture that is robust and reliable enough to fulfil the requirements and provides proofs given by commonly accepted fault analyses tools. [Skram07] suggests an architecture that uses diverse fault tolerant components with voters and redundant results. Another way could be to test the used components exhaustively, which may be really expensive and not feasible for huge component based systems.

3.4 Linux in Safety-Critical Environments

When examining the appropriateness of the application of Linux we have to first take a look at the characteristics of operating systems and the criteria they have to fulfil in order to be taken into consideration for a safety-critical environment.

3.4.1 Linux – A Brief History

Linux is a Unix-like operating system that has evolved and is further developed out of an Open-Source process for which it is one of the most prominent examples. The first Linux kernel has been released on September 17th 1991 for Intel x86 processors. It was enriched by standard utilities taken by the GNU project. Recently a broad range of so-called Linux distributions, which differ of Linux kernel distributions and available applications. They are available from a great number of different vendors and may be tailored for a wide variety of use. Linux systems are taken for use as servers, routers, embedded systems, desktop computers and even mobile phones.

Being a Unix-like system means that it behaves in a manner similar to a Unix system while not necessarily conforming to or being certified to any version of the Single UNIX Specification. [6] The term can include free software / open source operating systems inspired by Bell Labs' Unix or designed to emulate its features, commercial and proprietary work-alikes, and even versions based on the licensed UNIX source code (which may be deemed so "Unix-like" that they are certified to bear the "UNIX" trademark). There is no formal standard for defining the term, and some difference of opinion is possible as to whether a certain OS is "Unix-like" or not. [5] There exist a number of "Unix-like" operating systems as you can see in Figure 21.

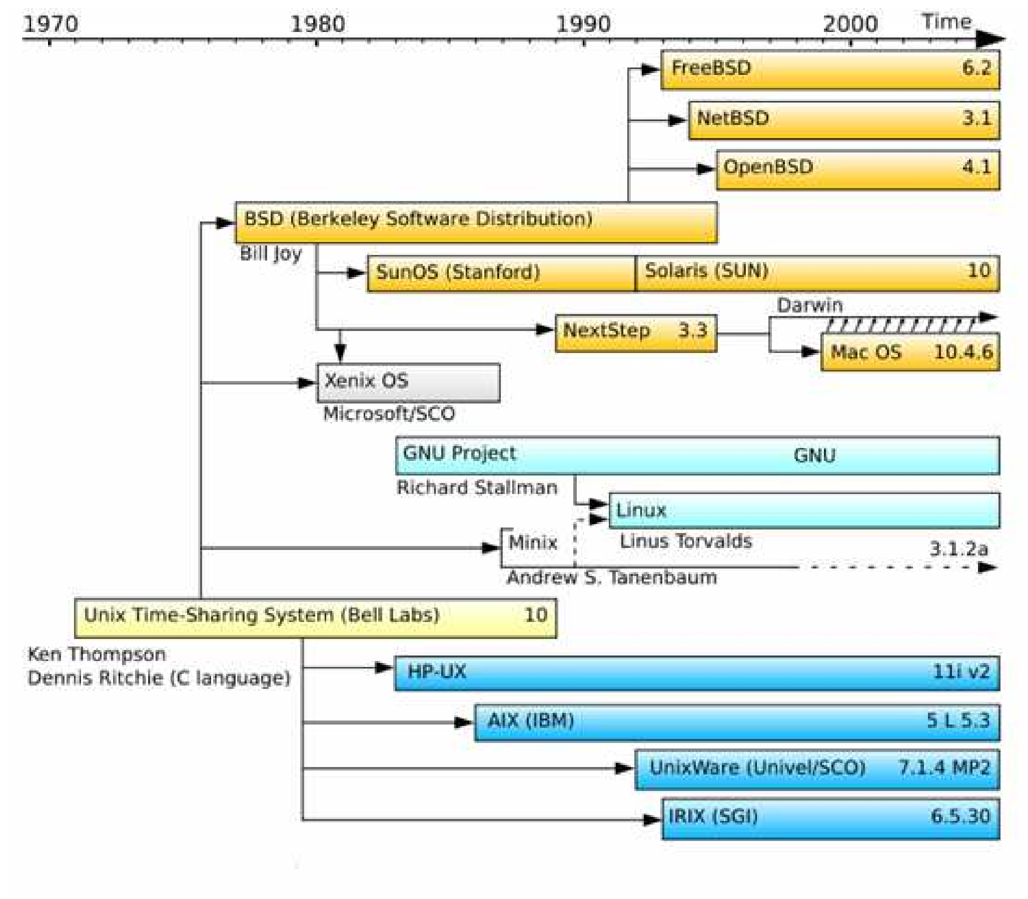


Figure 21: "Unix-like" operating systems pedigree [5].

The UNIX operating system has been developed in 1969 by a group of AT&T employees at Bell Labs. Prominent members of this group were Ken Thompson, Dennis Ritchie and Douglas McIlroy. The other Linux column, the GNU Project, started in 1984 and its goal was to create a POSIX-compatible operating system from entirely free software. In 1985, Richard Stallman created the Free Software Foundation and developed the GNU General Public License (GPLv1), in order to spread the software freely [3]. All other tools that are commonly required by a fully featured operating system such as programming libraries, editors, shells and GUIS were developed in the early 1990s and so nearly all required services were ready by that time. What was missing, were crucial parts like device drivers, daemons and – most important – the kernel. Linus Torvalds has said that if the GNU kernel had been available by that time, he would not have decided to write his own.

Andrew S. Tanenbaum released MINIX in 1987, a Unix-like system intended for academic use, for which the source code was available, but own changes and modifications and redistribution was restricted. Furthermore it was designed for 16-bit design and not well adapted for the increasingly popular Intel 368 architecture.

For these reasons above, Linus Torvalds, a Finnish second-semester computer science student, decided to write his own non-commercial replacement of MINIX when he was at the University of Helsinki. In August 25th 1991 Torvalds posted a message in the MINIX user group introducing his new project in order to gain feedback and proposals. Linux kernel 0.0.1 was released in September 1991 and put on the net. At this time neither Torvalds himself nor the rest of the MINIX community thought it would be a big deal for computing. But people downloaded the code, tested and tweaked it and returned their changes to him. Torvalds included the changes and released new versions in periods of some weeks. The community grew and grew and the system was released under GNU General Public License, thus ensuring that the source codes will be free for all to copy, study and change. The next step was the emerging of vendors that provided distributable versions to ease the installation process and provide support for different platforms. Clustering technology was invented and the first Linux driven supercomputer connecting 68 PCs with cables running 19 billion calculations per second was built in 1996 by researchers as Los Alamos National Laboratory.

Driven by the big and still emerging community Linux development process hasn't changed far from the early days. People are still writing tweaks, ports and patches and send them over to be implemented in new kernel versions. There are few guidelines and restrictions to code as long as it runs. This makes those systems hard and expensive to use and is malicious to predictability. A famous code comment in the scheduling part of the kernel and a posting show that circumstance:

"Dijkstra probably hates me", Linus Torvalds, in kernel, sched.c

"How should I know if it works? That's what beta testers are for. I only coded it."
Attributed to Linus Torvalds, somewhere in a posting

They show on the one hand that the development process may be not suitable in an environment where predictability and reliability are crucial. On the other hand they show a preference to proven-in-use.

3.4.2 Operating Systems in Safety-Related Environments

There are a number of practical problems with using an operating system in a safety related environment. There is no standardized way of specifying the overall safety related behaviour of software components overall. As noted in [HSE02], there is a major difference between an operating system and any other kind of pre-existing software component in that the operating system provides a layer between the application and the hardware. Failure of the services provided by the operating system to the application program will therefore inevitably result in application software failure. It states further that the operating system could have internal fault containment features and fault-tolerance techniques implemented as well. Therefore the application software cannot itself provide important defensive mechanisms. In some cases hardware takes the part of checking itself.

[HSE02] states that an operating system to be suitable for use in safety related systems, it must satisfy the following criteria:

- The *behaviour of the operating system must be known with sufficient exactness*, in all relevant domains of behaviour, to provide adequate confidence that hazardous behaviour of the safety related application does not arise because of a mismatch between the belief of the application designer and the true behaviour of the operating system.
- The *behaviour* of the operating system *must be appropriate for* the characteristics of the *safety related application* in all relevant domains of behaviour.
- The operating system must be *sufficiently reliable* to allow the safety integrity requirements of the application to be met (when taken together with other system features). In other words, the likelihood of failures of the operating system features and functions used by the application must be sufficiently low.

It is also important that while testing may be a useful tool to eliminate many failures caused by a mismatch of the designer's opinion of how an operating system behaves and its behaviour in reality it is not guaranteed that all such mismatches can be eradicated and they do not lead to failures in operation. [HSE02] also states that the higher the integrity requirements to a system are, the higher is the importance of criteria 1.

[HSE02] recites the domains of behaviour for a software system that are identified in general in a report called "Regulatory Objects for Software in Safety Related Air Traffic Services" by British Civil Aviation Authority:

- Functionality
- Timing and Performance
- Capacity
- Failure Behaviour (of the system itself/ connected systems/ user programs);
- Overload Tolerance
- Reliability (safety integrity)
- Accuracy

Another safety standard in the aerospace industry, ARINC 653, suggests an Application Programming Interface for the operating system to the application layer for safety related avionics systems, which provides a solid basis for assessment of the quality of an operating system for such systems. [HSE02] The ARINC 653 safety model is split up into two levels of classification – functions and services. [HSE02] states that what is required is a functional failure analysis of the operating system, the result of which can be plugged into the analysis of the failure characteristics of each application that relies on the functions of the operating system. A failure of the operating system may thus appear as a base event in the fault tree for a particular failure mode of an application. Thus it is necessary to determine a set of functions that the operating system must provide if an application is to provide the intended functionality, and then to undertake a failure analysis of these functions [HSE02].

The ARINC 653 API description gives six families of functions that are determined for safety related applications that have to be fulfilled by the operating system as shown in Table 2. This provides a good overview of the main tasks that have to be considered when building infrastructure to prevent operating systems from harming safety.

i)	*Provision of secure and timely data flow* to and from applications and I/O devices.
ii)	*Controlled access to processing facilities.* The access of applications to the underlying hardware processing resources must be managed so that, for example, any deadlines can be met.
iii)	*Provision of secure data storage and memory management.* The aim here is to secure memory storage from corruption of interference by other applications or the actions the operating system takes on their behalf.
iv)	*Provision of consistent execution state.* This concerns the consistency of data and is mostly concerned with the state of the system after initialisation.
v)	*Provision of health monitoring and failure management* covers partial and controlled failures of the system (operating system, application, hardware).
vi)	*General provision of computing resources.* This covers provision of any of the services of the operating system. A failure of this function would imply an uncontrolled failure of the operating system.

Table 2 – Families of safety functions to be provided by operating systems [HSE02].

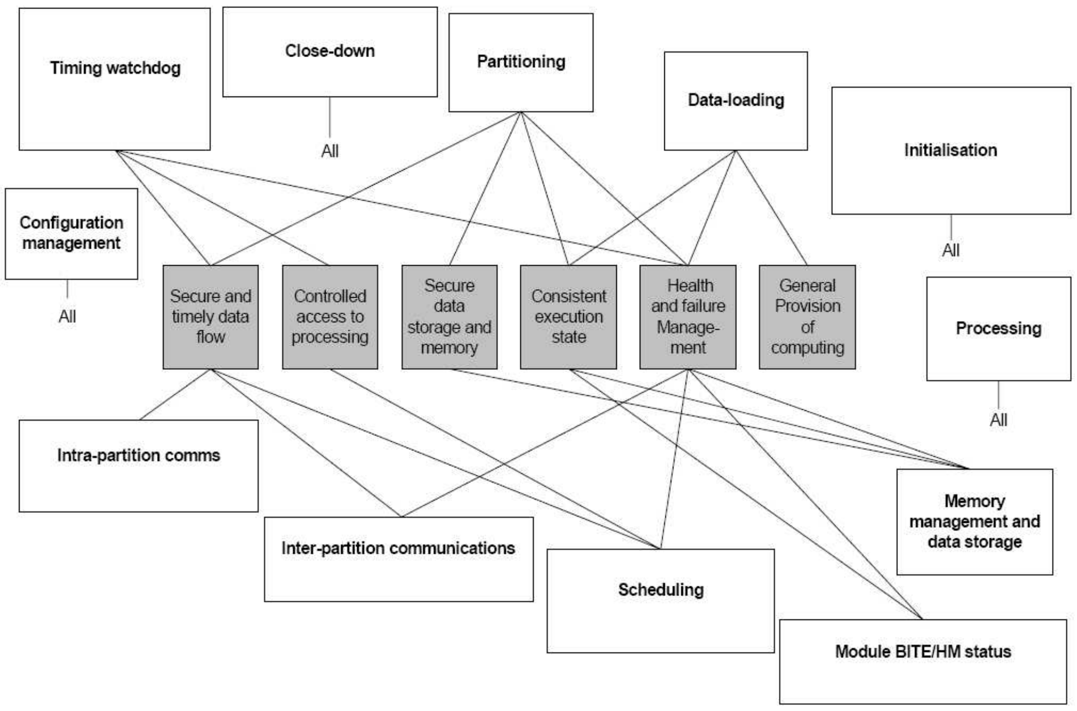

Timing watchdog
Close-down
All
Partitioning
Data-loading
Initialisation
All
Configuration management
All
Secure and timely data flow
Controlled access to processing
Secure data storage and memory
Consistent execution state
Health and failure Manage-ment
General Provision of computing
Processing
All
Intra-partition comms
Inter-partition communications
Scheduling
Memory management and data storage
Module BITE/HM status

Figure 22: Functions and related calls for ARINC 653 [HSE02].

Figure 22 shows how the links and services defined in the standard are linked. A third classification level is the suggested API that describes how services are invoked by the applications.

[HSE02] states that this classification scheme can be used to decide whether a given operating system is intrinsically suitable for use with a given safety related application, to compare the merits and disadvantages of a number of operating systems which are being considered for use in a safety related application and to facilitate a hazard analysis of the interaction between the application programs and the operating system, to ensure that no new hazards have been introduced by the use of the operating system, and where necessary to create derived requirements to mitigate operating system failures of weaknesses.

In order to make an assertion about the suitableness of an operating system to a safety related environment, we have 3 sources of evidence that we have to check against the described criteria. Those are field service experiments, testing and analysis.

3.4.3 Fulfilment of Safety Criteria provided by Linux

[HSE02] assessed standard Linux distributions and concludes that it would be broadly accepted for use in [IEC61508] SIL 1 and SIL 2 integrity environments. But this assertion is pretty much based on assumptions that the hardware is of suitable integrity as well as the applications based on it fulfil the safety requirements. It also states that it would certainly be feasible to certify Linux to SIL 3, but further evidence from testing and analysis must be provided.

A point very important to this work is that [HSE02] does not consider Linux as suitable for [IEC61508] SIL 4 environments as it would not be reasonably practicable to provide evidence that it meets the according safety requirements.

4 Overview

During the elaboration of the solution and the literature studies it turned out that a two-trail approach is the most suitable for the described problem. The Safety Supervision Layer consists of two parts. A library that wraps the usual unsafe functions and provides error detection capabilities for the functions under consideration, and a watchdog structure that periodically checks operating system functionality. As a pattern for the wrapping functions message passing libraries have been picked and a Safe Message Passing library is provided. The watchdog structure lacks of decisive advantages the direct error checking approach of the libraries provides. Hence this structure should primarily used check functionality the library cannot check directly because of their nature.

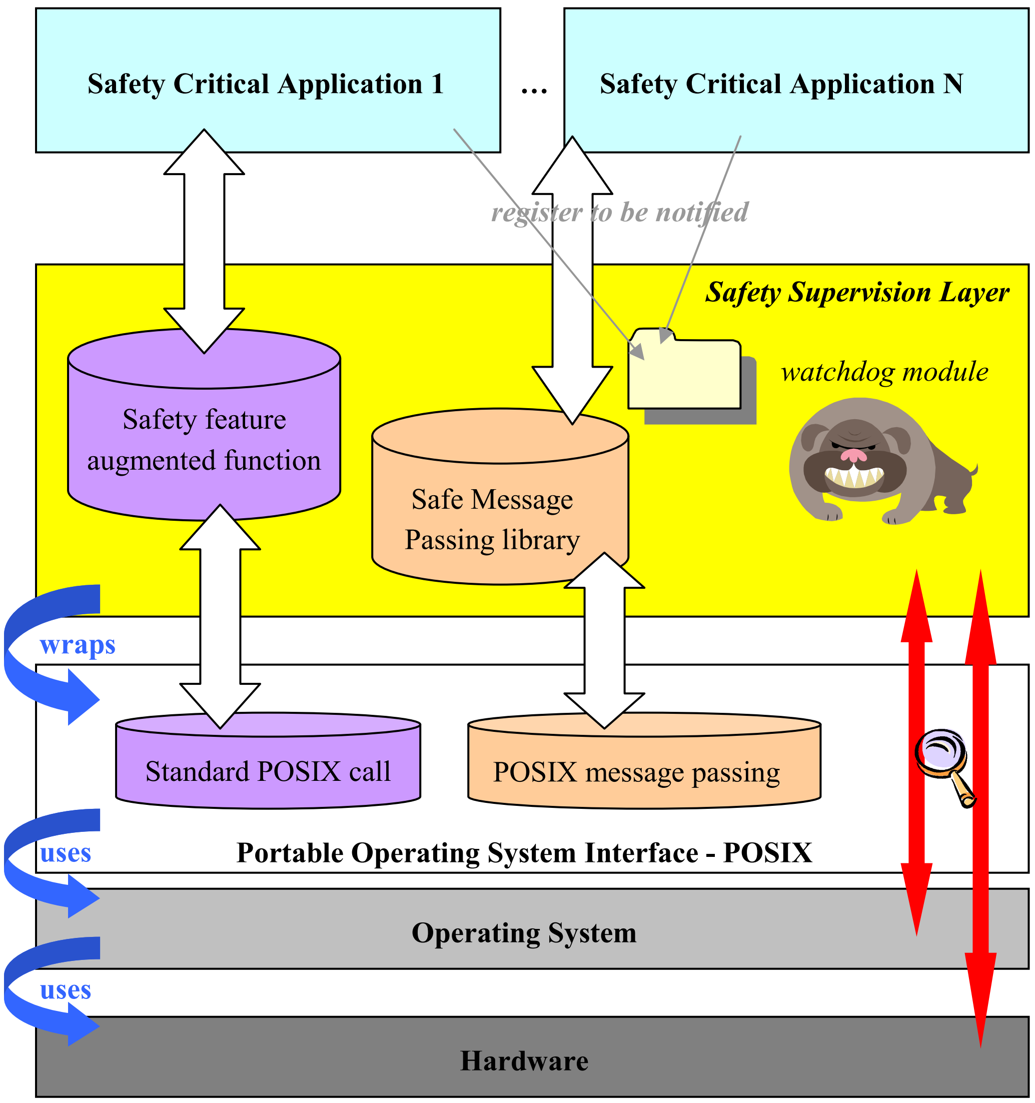
4.1 Idea

As the problem description shows the goal is to provide a facility that theoretically allows us to separate the application layer from the underlying infrastructure. Safe application developers shall finally be provided the possibility to rely on the functionality those lower layers provide in a way that they can be sure that the application is notified of any possible fault. Therefore guarantees are required that every fault situated in either operating system code or hardware is detected and passed over to some kind of error notification system.

Hence this leads us to some kind of dependable component view of operating systems and hardware, whereas at least the result may be seen as a safety feature wrapper around the POSIX layer. As our problem domain is a fail-safe one, there's the possibility to get to a safe state in which the system may be turned off without causing any harm to people or material. Most of the time this will be to just stop the train.

A two-trail infrastructure for the Safety Supervision Layer is suggested. It consists of a safe system call library and a safety watchdog system. Figure 23 shows things work together. One important characteristic is that we do not shut down our system immediately, but at first just notify on the one hand the system using usual signals and on the other hand the application using this library as most of the programs have their own specific needs and plan according to what to do when the system fails. Those actions may be notifying the operator via a certain interface, logging or even do some secondary action that may be needed to ensure system safety.

In order to provide system engineers the possibility to get to know why and when exactly a system was halted, the layer produces log files. Those log files are important for forensic analyses.



Safety Critical Application 1
…
Safety Critical Application N
register to be notified
Safety Supervision Layer
watchdog module
Safety feature augmented function
Safe Message Passing library
wraps
Standard POSIX call
POSIX message passing
uses
Portable Operating System Interface - POSIX
Operating System
uses
Hardware

Figure 23: Schematic representation of the Safety Supervision Layer.

4.1.1 Safe System Call Library

Weaknesses and lacks of common POSIX system call functionality are analysed and augmented by fault detection functionality. Applications use the new functionality exactly the same way as they use the common POSIX functions using preloading. The functions implement fault detection algorithms to get after faults in an end to end way and use the underlying POSIX functions themselves.

In this context libraries may deal with operating system functionality like message passing, process synchronization, timing functionality and a lot of other functions that may be utilized using POSIX. Applications using these libraries are notified immediately after an error is detected, as all results produced are checked. As stated above we need to prove that a given function covers all possible faults, so every library module has to be proven using common technologies like FTA and FMEA.

4.1.2 Safety Watchdog System

Often one may come over functionality that may not be checked immediately in their result. Constraints avoiding this may come from performance requirement or others that may prevent a fault to be detected in a single function's chain of execution. This is why we need some sort of watchdog. The watchdog is a user level demon that runs modular tests periodically. Those tests may check memory management, scheduling, timing and others.

If an application wants to profit from the watchdog functionality in order to be notified, it has to register at the watchdog's client registry. The demon will then inform subscribed applications to give them the chance to shut down gracefully, before it halts the system. The watchdog also writes its results to system log files to provide data for forensic analyses.

In some cases the watchdog may use the libraries described above and send some test data in order to check for erroneous functionality.

4.2 The Practical Side – An Example

Railway applications used in main lines as well as urban rails have strict and challenging requirements to dependability and safety. The TAS Platform is a vital computer platform, that is designed to meet these requirements [Kan04]. Figure 24 shows how this platform is structured. The main goal is to please the shared requirements of the different railway control systems according safety, dependability and real-time responsiveness. Logically the described Safety Supervision Layer has its place above the blue coloured POSIX layer.

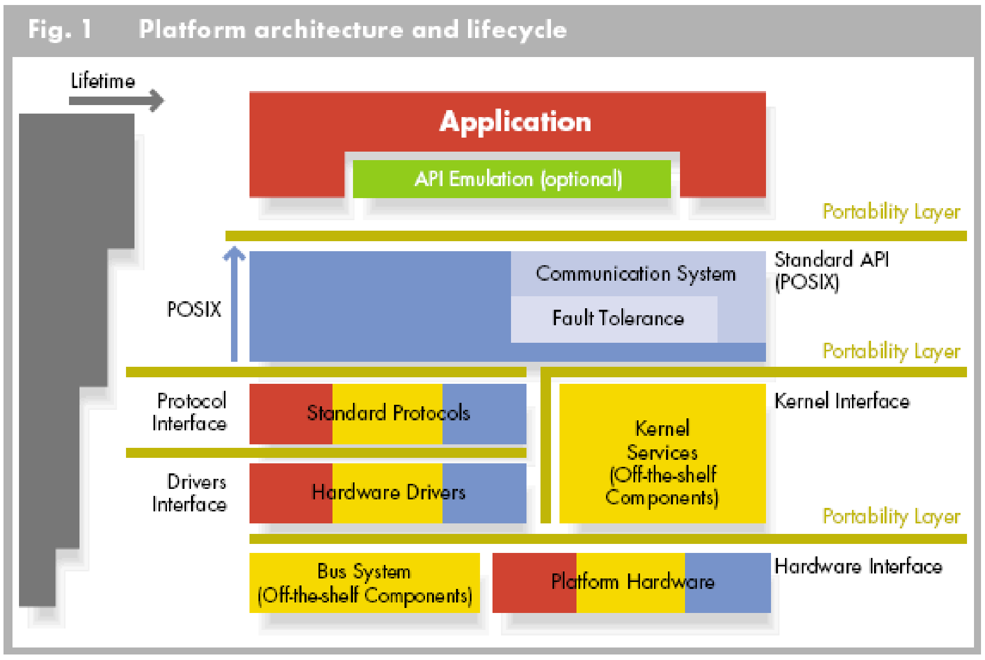


Figure 24: A safety platform for railway applications [Kan04].

4.3 Resulting Code, POSIX Compatibility

There two items my work resulted in. On is a concept of a watchdog structure taking care of the described faults. The other is a library to be preloaded that features safe message passing. POSIX compatibility is a requirement to the supervision layer as well as to the library that comes from the project partner and is supposed to guarantee portability.

The libraries are loaded and used as their original POSIX functions are, their names are those of the standard ones. The safe functions use the standard message passing functions via function preloading. In this work it turned out that a number of restrictions to be made in concerning the communication features, e.g. there is a restriction of 1:1 communication between two endpoints, which emerged mainly from the authentication requirements of [CEN50159-1].

5 Safe POSIX Message Passing

A major part of my thesis as well as a desired field of application of the Safety Supervision layer was the analysis of possible faults related to message passing using POSIX message queues. The implementation of a module to check this operating system feature led me to the implementation of a system library that raises alarms, if a fault is detected in the message passing functionality.

This side product turned out to be a library that may be used the same way as the standard POSIX message queuing procedures and hence may be preloaded to override the common unsafe mechanisms although it uses the traditional system calls at a lower level.

5.1 POSIX Message Passing

Currently the POSIX standard holds the following functions that are referred to as Message Passing functions. Those are the items I built wrappers, which can be preloaded, for.

5.1.1 Basic Workflow

The POSIX standard (IEEE Standard 1003.1-2001) defines inter process communication structures based on message queues. In literature those are known as POSIX message queues and are based on System V message queues, in which the processes can communicate with one another by means of messages. Each message is sent through an IPC message queue, where it remains until another process reads it. When the other process reads – or consumes - it, the message is deleted. A message is composed of a fixed-size header and a variable length body. Messages can be given a *message type*, which allows a consumer to selectively retrieve messages from a queue. The characteristic that the message is destroyed by the kernel leads to the fact that only one process can receive a given message.

To send a message, a process uses the `mq_send(...)` function, passing 3 parameters:

- The IPC identifier of the desired message queue

- The size of the message body
- The address of a User Mode buffer that contains the message type followed by the message body

To retrieve a message the `mq_receive(...)` function is used with the following parameters:

- The IPC identifier of the IPC message queue resource
- The pointer to a User Mode buffer to which the message type and body should be copied
- The size of the buffer
- A value that specifies the type of the message that shall be retrieved. If this value is 0, the first message in the queue is received. If it is positive, the first message of the specified type is returned.

To get to a Fault Analysis of the POSIX message passing functionality it is crucial to entirely understand how the system calls behave in usual and erroneous situations. Therefore I give a comprehensive description of the data structures and functions and the error notifications they may yield in the following paragraphs. The fault analysis chapter deals then with where there are possible failures that should be detected by the layer. Based on the results of those analyses, the end of this chapter describes possible solutions to the problems spotted.

5.1.2 Data Structures

In POSIX Message Passing there are generally two important data structures, which are `mqd_t` which holds the information about the message queue and `msg_msg` which basically represents a message:

Type	**Field**	**Description**
`struct list_head`	`m_list`	Pointers for message list
`long`	`m_type`	Message type
`int`	`m_ts`	Message text size
`struct msg_msgseg *`	`next`	Next portion of the message
`void *`	`security`	Pointer to a security data structure (used by SELinux)

Table 3: `mqd_t` **data structure.**

An instance of the `mqd_t` data structure is returned when the message queue is opened or optionally created. Table 4 describes the properties of a message queue.

Type	Field	**Description**
`long`	`mq_flags`	Holds the set flags (0 or O_NONBLOCK).
`long`	`mq_maxmsg`	Specifies the maximum number of messages in the queue.
`long`	`mq_msgsize`	Defines the maximum size of the messages in bytes.
`long`	`mq_curmsgs`	Gives the total number of messages currently in the queue.

Table 4: The `mq_attr` data type.

5.1.3 Functions

The following functions are subject of this work's analyses and proposals. They embody the standard message passing functionality.

```
mqd_t mq_open( const char *name, int  oflag ); /** open existing mq */
mqd_t mq_open( const char *name, int  oflag, // open possibly not
             mode_t mode, struct mq_attr ); // existing mq (O_CREAT)

mqd_t mq_close( mqd_t  mqdes ); // close a message queue

mqd_t mq_send( mqd_t mqdes, const char *msg_ptr,    // send message
            size_t msg_len, unsigned int msg_prio );

mqd_t mq_receive( mqd_t mqdes, char *msg_ptr,  // receive message
                   size_t msg_len, unsigned int *msg_prio );

mqd_t mq_getattr( mqd_t mqdes, struct mq_attr *attr );

mqd_t mq_setattr( mqd_t mqdes, const struct mq_attr *newattr,
               struct mq_attr *oldattr );

mqd_t mq_notify( mqd_t mqdes, const struct sigevent *notification );
```

5.2 Fault Analysis, Possible Errors

What I explained above is how the message passing functions are supposed to behave according to the POSIX standard. Notifying a process of errors such as run outs of memory or permission denials are part of this specification. Therefore those are not treated as faults and can be clearly expected and foreseen by a system developer. Hence they have to be handled at the application level. What cannot bee foreseen by the system engineer without costly verification and validation are faults situated in the implementation.

In order to characterize erroneous functionality one has to define the correct way a system or component is supposed to behave. Figure 25 shows the very basic set up for our environment. A producer shall produce messages. They shall be consumed by a series of consumer processes. Messages that are retrieved on the consumer side shall contain the same data that has been passed to the `mq_send` function. The messages the consumer processes receive shall arrive in the same order that they are intended, which means that they shall be received the older after the newer respecting their priority. Another important thing is that we have to guarantee that the processes only consume messages that are addressed to them and that the messages are really consumed by their intended process.

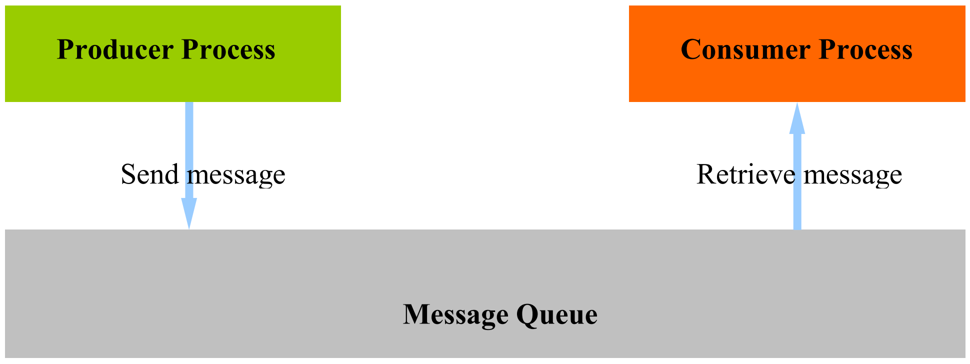


Figure 25: Basic message passing functionality.

The statements above only show some points that have to be elaborated. In the analyses below the functions are checked for possible points of failure that could harm this intended functionality. Suggested counter measures are written in italic characters and subsumed in the solution chapter.

The following Failure Modes and Effects Analysis (Table 5, Table 6) is based on the proposals and requirements of CENELEC 50159-1 and covers all the failure modes described in the standard. So this is why the following analysis can be considered complete.

5.2.1 Failure Modes and Effects Analysis

Failure Mode	Cause of Failure	Possible Effects	Possible Action to permit detection
1) Message is sent, but not consumed by the receiver.	- Receiver is down or blocking. - Receiver does not exist. - Message is lost in the communication channel. - Message is consumed by another process.	Intended Receiver does not get the message. Safety critical actions initiated by the message are not performed.	Create feedback infrastructure to enable the sender to check for correct retrieval.
2) Message is sent and consumed by the right receiver, but the contained data is wrong.	- Bit flips occurred somewhere in the communication channel. - Message queue memory integrity is violated by some other processes. - Message passing functionality yields wrong results for certain input parameters.	Data may not be read correctly by the receiver. Wrong actions may be initiated.	Implement checksum functionality to enable the receiver to independently check for violated data.
3) Message is consumed by an unintended receiver.	- The wrong process "steals" and consumes the message from the queue. - Defect message passing functions put the message on the wrong queue. - Race conditions, errors in synchronization mechanisms.	Intended receiver does not get the message. Important actions may not be undertaken. The unintended receiver may undertake hazardous actions.	Implement authentication infrastructure to provide the possibility to adjudicate properly whether the intended receiver did get the message. Introduce unambiguous identifiers for message consumers and feedback infrastructure.

Table 5: Message passing FMEA (1).

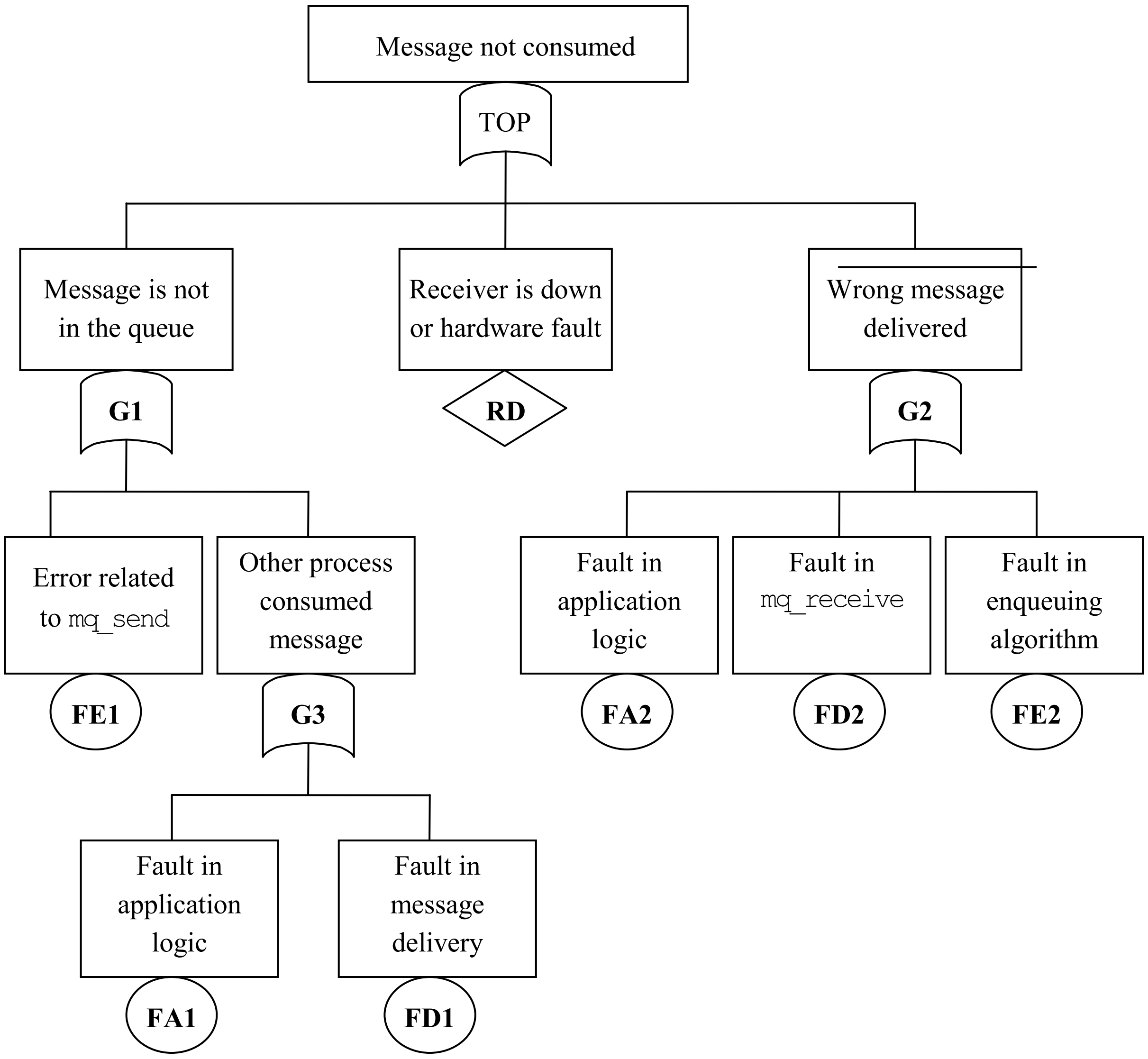
Failure Mode	Cause of Failure	Possible Effects	Possible Action to permit detection
4) Messages are not consumed in the right order (newer messages are consumed before older ones).	- The implementation of the queuing algorithm contains faults. - The implementation of the receiving algorithm contains faults. - Different delays at senders while putting messages on the queue - Incorrect scheduling.	Messages can cause actions on the receiver side. If those actions are performed in the wrong order, hazardous states may be reached.	Give messages sequence numbers and timestamps on the sender side. Checking those items the receiver can autonomously decide whether an error has occurred or not.
5) Messages are consumed more than once.	- Faults in the consummation algorithms may erroneously fail to delete the consumed messages from the queue. - Faults in file handling	If the information content of the message is not idempotent, inconsistencies may occur.	Sequence numbers are an easy way to detect these faults.
6) Same messages are put on the queue more than once.	- Faults in the queuing algorithm - Faults in the file handling	If the information content of the message is not idempotent, inconsistencies may occur.	Sequence numbers are an easy way to detect these faults.
7) Message passing infrastructure is constructed but no messages can be sent.	- Faults in the input checks or signalling.	This may lead to unnecessary error detection delays as detection not occurs until the first message is sent.	Send periodic test messages and receive test feedback.

Table 6: Message passing FMEA (2).

5.2.2 Fault Tree Analysis

The following chapters describe the spotted situations in terms of fault trees and provide qualitative analyses for them. I examine suitable counter measures for the described failure causes. Quantitative analyses are not to be done for the given case as it is nearly impossible to generate reliable numeric data for the reliability of existing software components. The events containing bold character shortcuts are described in the explanation text below each figure. Each node containing an index is explained in the text below, others are not explained further.

5.2.2.1 Message is sent, but not consumed by the receiver



Message not consumed
TOP
Message is not in the queue
Receiver is down or hardware fault
Wrong message delivered
G1
RD
G2
Error related to mq_send
Other process consumed message
Fault in application logic
Fault in mq_receive
Fault in enqueuing algorithm
FE1
G3
FA2
FD2
FE2
Fault in application logic
Fault in message delivery
FA1
FD1

Figure 26: FTA for case 1) Message is not consumed by the receiver.

- **RD** Receiver is down.
  The intended receiver process may be down or suspended. This may be critical in our environment, we suggest our receiver to be up. *Timeouts* and a *feedback infrastructure* will help detect this fault.
- **G1** Message is not in the queue
  - **FE1** Error related to `mq_send`
    Because of arbitrary reasons and faults in the `mq_send` code the function may fail either to put a message on the queue correctly or just fail to report an error. This may happen if `mq_send` is intrinsically intended to yield -1, but returns 0 showing success. *Feedback* provided by a *second way communications channel* will detect such malfunctions.
  - **G3** Other process consumed message
    - **FA1** Fault in application logic.
      The application logic may initiate message delivery to a wrong process in some cases. *Unique sender and receiver identification* as demanded by CENELEC-50159 will help to cope with some of such problems.
    - **FD1** Fault in message delivery.
      There may be faults in code executed by calls of `mq_receive`. The function may deliver the message to unintended processes. *Feedback* and *timeouts* at `mq_send` are a solution.
- **G2** Wrong message delivered.
  Generally the causing events can be dealt with *unique message IDs* and *sequence numbers*.
  - **FA2** Fault in application logic.
  - **FD2** Fault in `mq_receive`.
  - **FE2** Fault in enqueuing algorithm.
    Problems caused by code executed by `mq_send` may put the message on a different queue or on a wrong place in the queue.

5.2.2.2 Message is sent and consumed by the right receiver, but the contained data is wrong

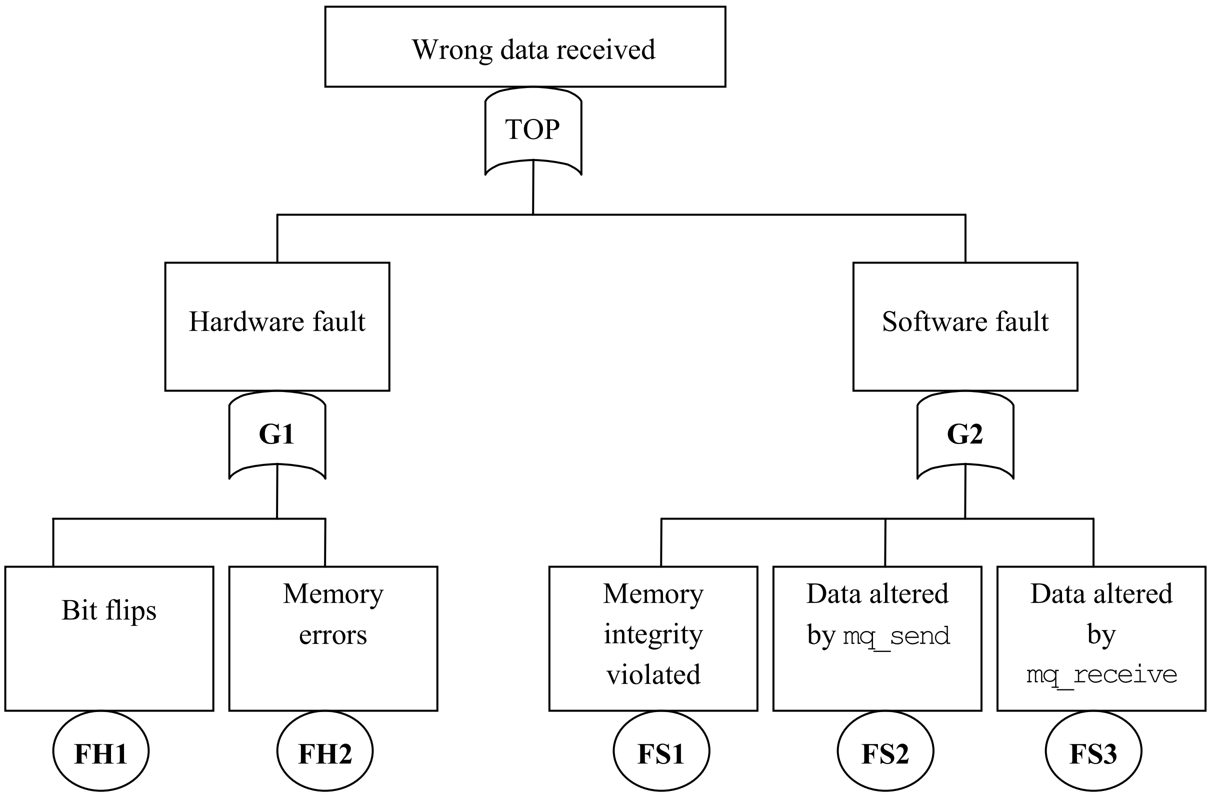


Figure 27: FTA for case 2) Wrong data received.

- **G1** Hardware fault.
    - **FH1** Bit flips.
      Bit flips are bit manipulations meaning that a bit is changed from 1 to 0 or from 0 to 1. This may happen because of environmental influences or defect hardware components.
    - **FH2** Memory errors.
      Defect memory components may store or yield malicious bit patterns, which leads to harmed data.
- **G2** Software fault.
    - **FS1** Memory integrity violated.
      Data integrity of the process's space is harmed by another process and therefore leads manipulates message payload.

- **FS2** Data altered by `mq_send`.
  Faults in code executed by the function itself may unintentionally change data payload. This is what should originally be prevented by the function wrapper.
- **FS3** Data altered by `mq_receive`.
  Faults in code executed by the function itself analogously to *FS2*.

Generally data errors may be handled with adding redundancy. In our case a *checksum* should be generated over the whole message payload by `mq_send` and appended to the sent message. The receiving function – `mq_receive` – should now check whether that appended *checksum* corresponds to the given data. Hence it can then independently detect harmed data. Checksums should be appended to feedback messages as well as another wall against unintended data manipulation.

5.2.2.3 Message is consumed by an unintended receiver

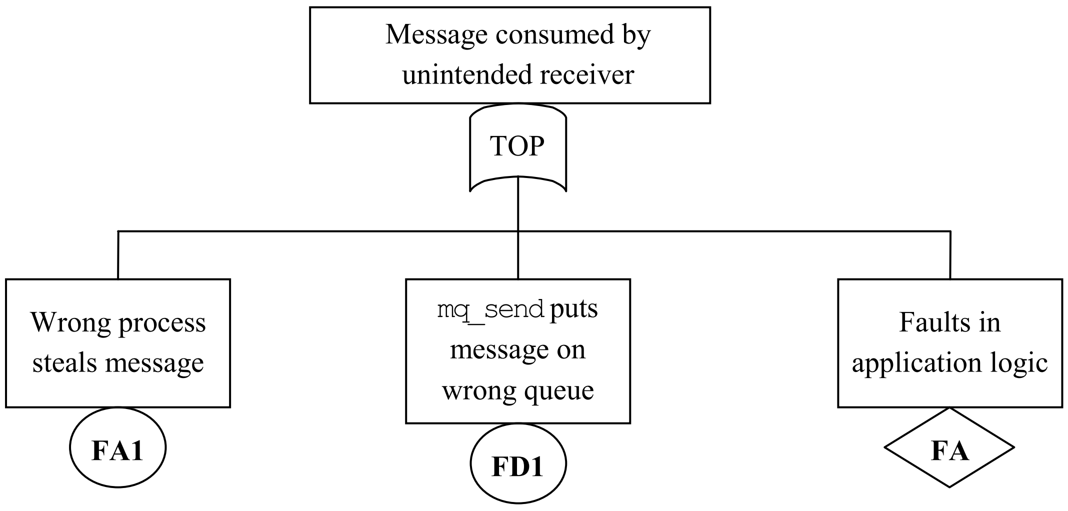

Message consumed by unintended receiver
TOP
Wrong process steals message
FA1
mq_send puts message on wrong queue
FD1
Faults in application logic
FA

Figure 28: FTA for case 3) Message consumed by unintended receiver.

- **FA1** Wrong process steals message.
  Such problems may appear due to problems in a system's file handling as well as ambiguous race conditions.
- **FD1** `mq_send` puts message on the wrong queue.
  The function itself may explicitly put the message on the wrong queue or address it to the wrong sender. Checksums explained above should therefore be implicitly calculated over the sender ID as well. The receiving process has to check

categorically check whether a given message is intended to it as an additional wall against malicious message reception.

- **FA** Faults in application logic.
  Faulty establishment of the system's communication infrastructure may also lead to unclear addressing.

To satisfy the requirements of CENELEC-50159 *unambiguous sender and receiver identification* has to be done. The introduction and processing of such IDs are an appropriate measure to deal with the faults described in this paragraph.

5.2.2.4 Messages are not consumed in the right order

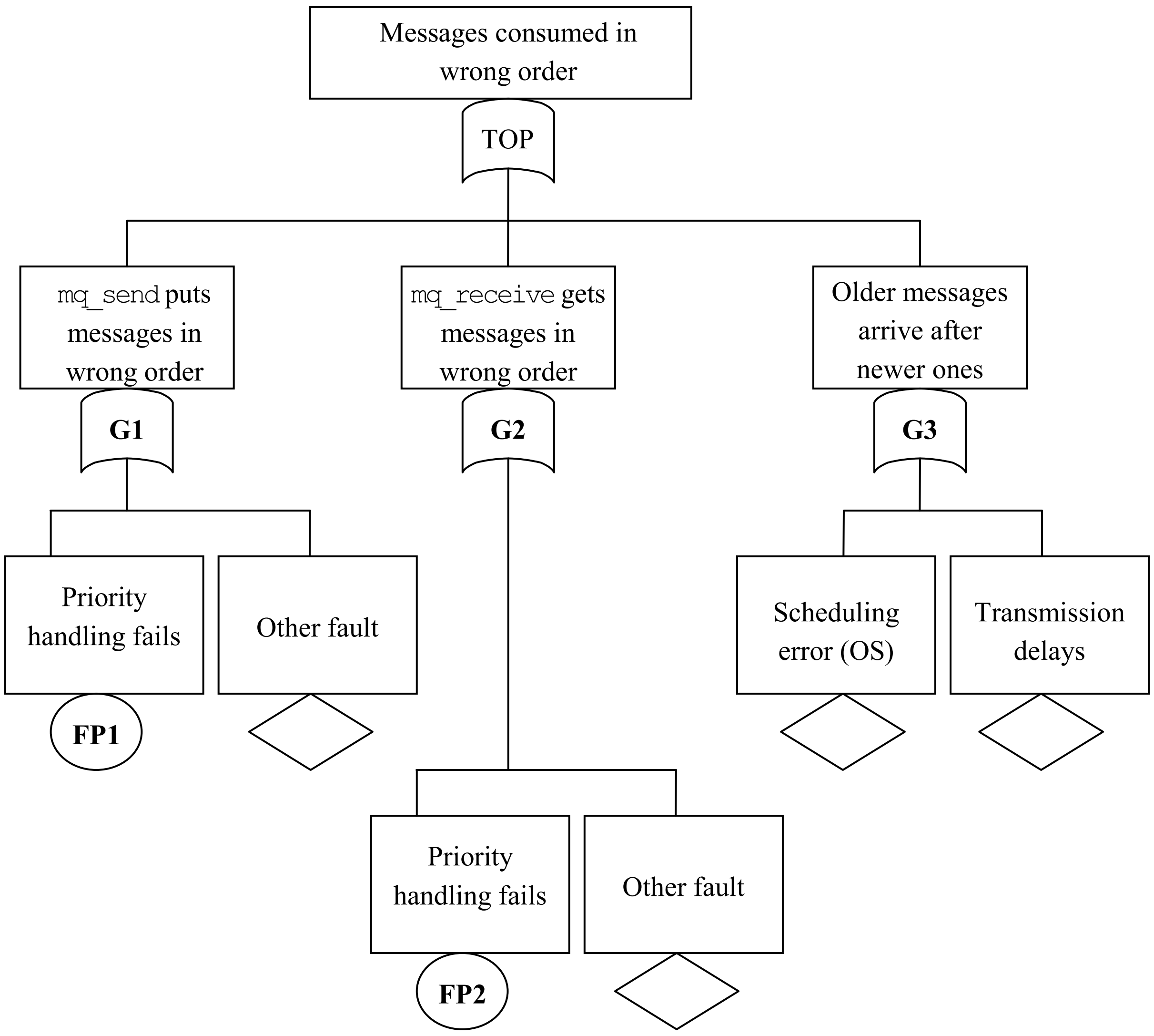

Messages consumed in wrong order
TOP
mq_send puts messages in wrong order
mq_receive gets messages in wrong order
Older messages arrive after newer ones
G1
G2
G3
Priority handling fails
Other fault
Scheduling error (OS)
Transmission delays
FP1
Priority handling fails
Other fault
FP2

Figure 29: FTA for case 4) Messages are not consumed in the right order.

- **G1** `mq_send` puts messages in wrong order.
  Faults in the function itself.
    - **FP1** Priority handling fails.
      I listed this one here as priority handling belongs to the most complex functionality of the message passing infrastructure. Complex functionality may contain more faults than simple.
    - Other faults
- **G2** `mq_receive` gets messages in wrong order.
    - **FP2** Priority handling fails. (see FP1)
    - Other faults
- **G3** Older messages arrive after newer ones.
    - Scheduling errors (OS)
      Errors done by the operating system's scheduling mechanisms may put execution blocks of the sending of newer messages before older ones. These faults are situated in the operating system domain and are currently not checked by the message passing functions. As we want to see everything below the layer as a grey channel, this has to be checked as well.
    - Transmission delays
      Irregularities in the transmission of the data shake the sequence message are arriving in. This is probably not handled property by current implementations.

Faults in this class may generically be handled by the introduction of *sequence numbers* for each message. As message passing knows several priorities for the messages those sequence numbers must be held for each priority given. Sequence numbers should be included in the checksum computation.

5.2.2.5 Messages are consumed more than once

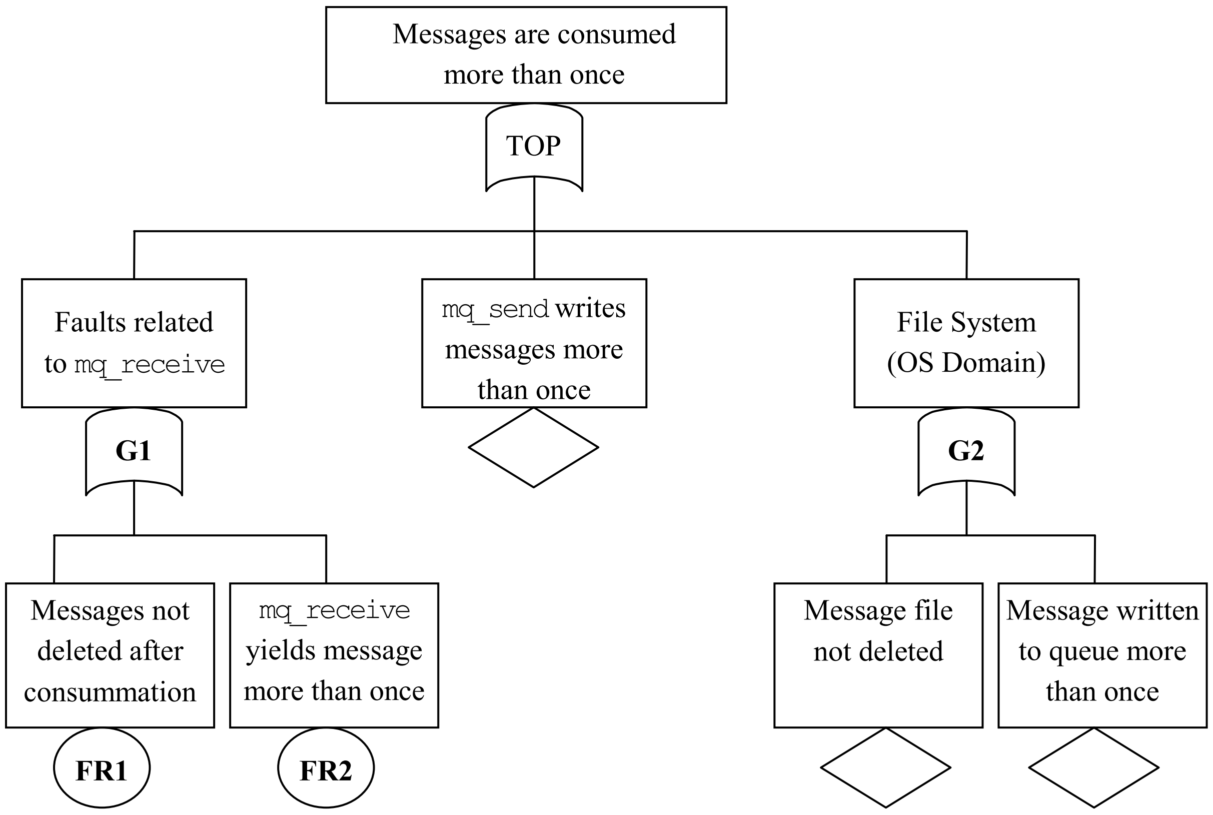

Messages are consumed more than once
TOP
Faults related to mq_receive
mq_send writes messages more than once
File System (OS Domain)
G1
G2
Messages not deleted after consummation
mq_receive yields message more than once
Message file not deleted
Message written to queue more than once
FR1
FR2

Figure 30: FTA for case 5) Messages are consumed more than once.

- **G1** Faults related to `mq_receive`.
    - **FR1** Messages not deleted after consummation.
      This may happen because of faulty behaviour of mq_receive related functions. The executed code may behave divergently to the specification in a way of failing to delete a consumed message.
    - **FR2** `mq_receive` yields messages more than once.
      The function returns a message that is already deleted from the queue more than once. This may happen because of buffer or variable failure.
- `mq_send` writes messages more than once.
  May also be caused by failures related to file system, as well as system call domain. Its effects are that a basically correct `mq_receive` function will yield messages intended to be consumed once more than once.

- **G2** File System (OS Domain).
  - Message file not deleted.
    The file system may fail to delete the message or alter the message queue file correctly without an error message to be passed to the service interface.
  - Message written to queue more than once.
    This error behaves analogously to FR2.

5.2.2.6 Same messages are put on the queue more than once

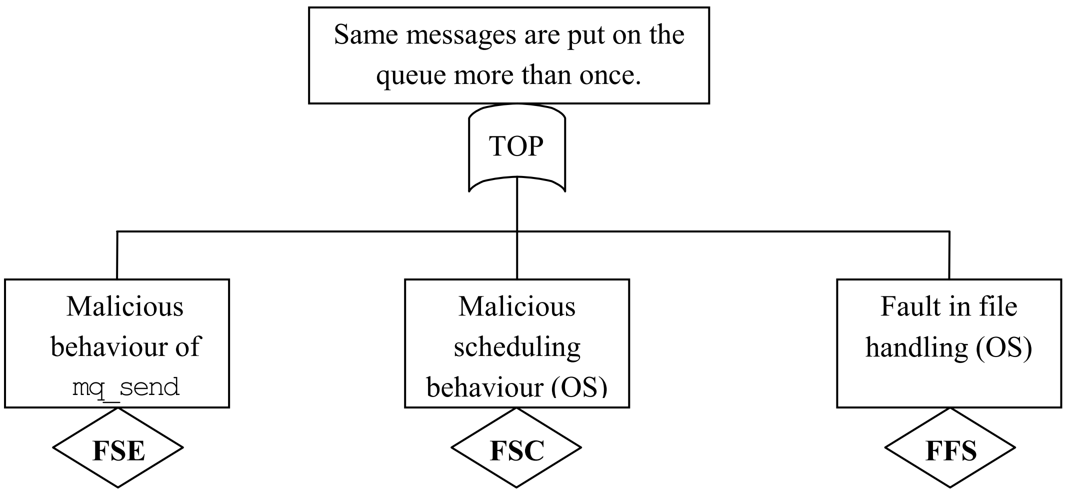
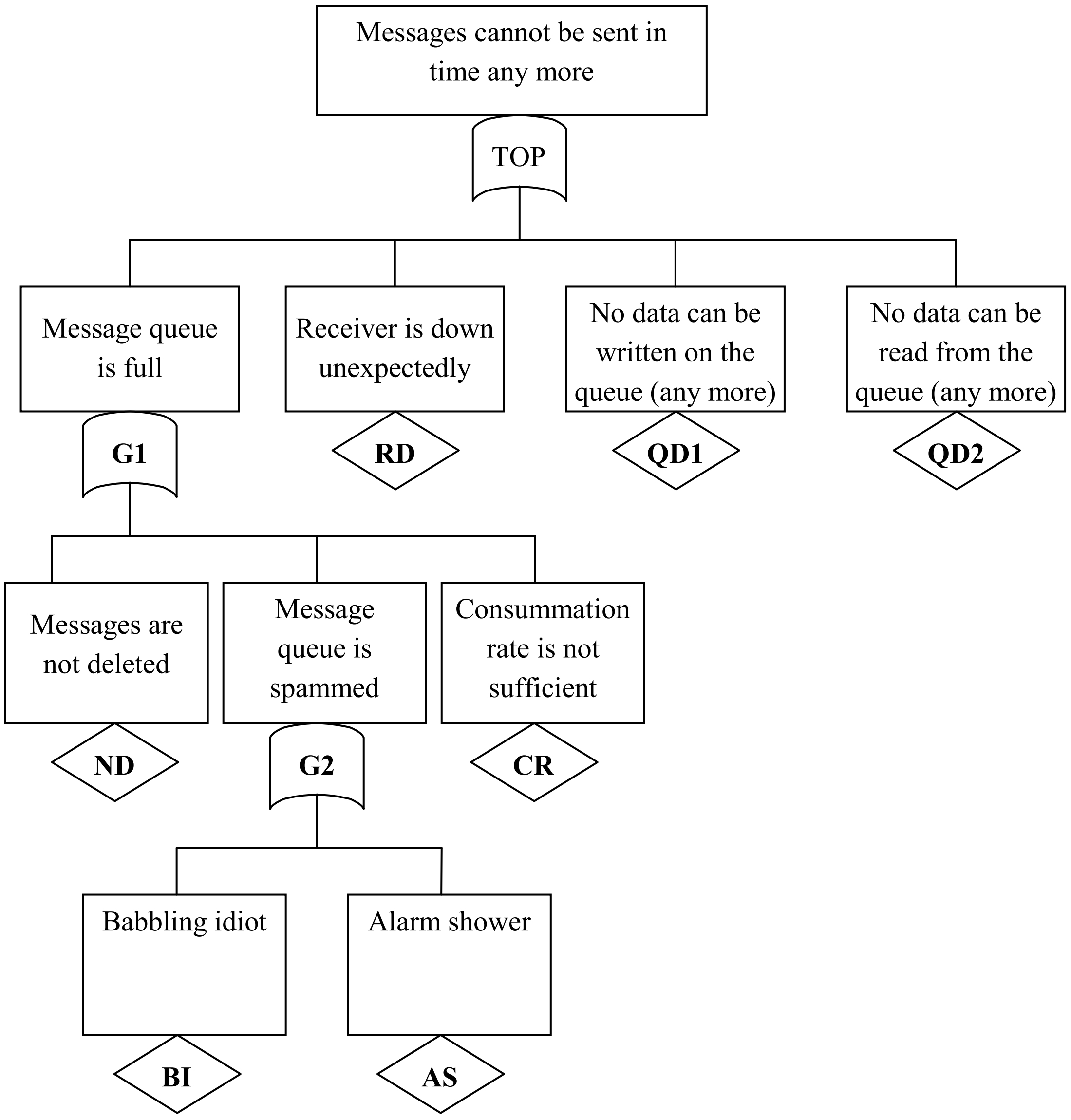

Same messages are put on the queue more than once.
TOP
Malicious behaviour of mq_send
Malicious scheduling behaviour (OS)
Fault in file handling (OS)
FSE
FSC
FFS

Figure 31: FTA for case 6) Same messages are put on the queue more than once.

- **FSE** Malicious behaviour of `mq_send`. Explained in case 5.
- **FSC** Malicious scheduling behaviour (OS).
  Process scheduling may execute the command blocks intended to write a message to the queue more than once.
- **FFS** Fault in file handling (OS).
  Works analogously to the file system errors in case 5, while in this case the file system alters the message queue data divergently from the specification.

5.2.2.7 Message passing infrastructure is established but no messages can be sent or received



Messages cannot be sent in time any more
TOP
Message queue is full
G1
Receiver is down unexpectedly
RD
No data can be written on the queue (any more)
QD1
No data can be read from the queue (any more)
QD2
Messages are not deleted
ND
Message queue is spammed
G2
Consummation rate is not sufficient
CR
Babbling idiot
BI
Alarm shower
AS

Figure 32: FTA for case 7).

- **G1** Message queue is full.
    - **ND** Messages are not deleted.
    - **G2** Message queue is spammed.
        - **BI** Babbling idiot.

        - **AS** Alarm shower.
    - **CR** Consummation rate is not sufficient.
- **RD** Receiver is down unexpectedly.
- **QD1** No data can be written on the queue (any more).
- **QD2** No data can be read from the queue any more.

5.3 Safe Message Passing Architecture

What the FMEA and the fault tree analysis yield is just what is defined and demanded in CENELEC 50159-1. The error for the data type is seen in as a data error according to those analyses. There is just one step that inhibits POSIX message passing from the applicability of the standard, as it defines methods for closed communication channels, which means that

- The communication media is known and specified.
- Only authenticated access is allowed.
- *There exist a maximum number of communication participants.*

The first two requirements can be fulfilled, as the message passing functionality is defined clearly by a precise specification and the standard prescribes the introduction of identifiers for environments where such don't exist. The problematic point is the third. Although there are a maximum number of message queues defined in the system, there are no mechanisms to define how many processes can write to and read from the queue.

So for the applicability of the standard restrictions have to be defined for that last point. As the project partner uses message queues primarily for 1-process to 1-process communication, the authentication mechanisms could be restricted to one-to-one communication. So through FTA analyses we come to exactly the same failure modes that are defined in the standard. This prescribes the following mandatory safety measures:

- Recognition of a sender error
- Recognition of a data type error
- Recognition of a data value error
- Recognition of outdated data or data that has not been received in time.
- Recognition of a drop of communication after a defined time.
- Assurance of the functional independence of the safe functions from the used layers of the unsafe communication channel.

Apart from that the project partner required the functions to recognize a break in the communication channel, before processes try to communicate.

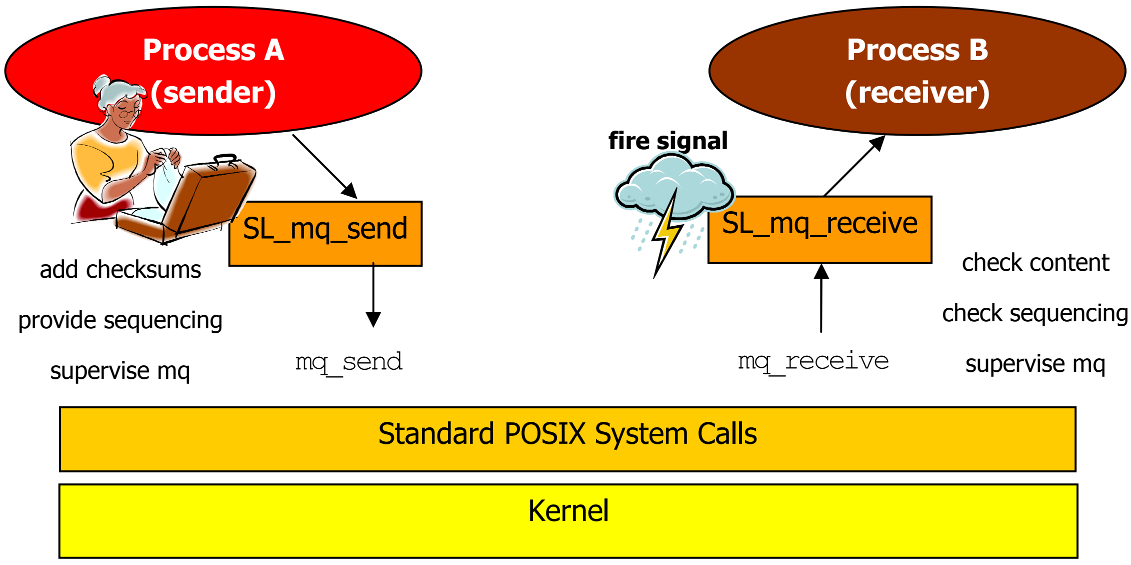


Figure 33: Approach to safe message passing.

Figure 33 gives a schematic representation of the approach chosen. Please note that in the following specification the names of the common POSIX functions stay as they are. The safe message passing library functions will be named with a prefix "`SL_`" to make the descriptions more understandable to the reader. In production this prefix should be removed and the functions be preloaded as described above.

5.3.1 Features

The analyses described above leads to the features the Safe Message Passing Library has to provide in order to comply to the standard. They are enumerated below.

5.3.1.1 Feedback Infrastructure

To be able to determine whether a message has not consumed by the sender or to decide whether a message has just not be consumed within defined time constraints, a communication channel back to the sender is needed. In that feedback messages receiver and sender identification, sequence numbers and checksums are included again.

5.3.1.2 Sender and Receiver identification

An authentication system is provided by the `mq_open` function. When the sender opens the message queue, it generates an identifier for itself. The same is done by the receiver, who sends a "Hello" message to the sender, just after the initialization.

5.3.1.3 Sequencing of message

To determine whether the messages arrive in the right order, the Safe Message Passing functions sequence each message. The first message gets the sequence number INT_MAX-1 to ensure that value overflow works properly. The sender increases its sequence counter after each send, the receiver checks the received message sequence number and increases its own after consummation.

5.3.1.4 Periodic supervision of the communication channel

Even if there's no planned communication test messages are sent through the channel to ensure that it is working. This was not a feature required by the standard but demanded by the project partner.

5.3.1.5 Checksum

A checksum is computed over the whole message payload containing the sender and receiver identifiers, the sequence numbers, timestamps and data type verifiers. This provides to a counter measure to common mode failures.

5.3.2 Basic Data/ Control Flow

As stated above certain restrictions have to be introduced to guarantee the applicability of CENELEC 50159 due to its restrictedness to closed communication systems. One of those is that the layer will only allow two communication partners for a logical queue. This circumstance implies two roles for the communication partners. One takes the part of the message queue creator, the other process takes part in that communication. There is an initialization process described below that will recognize if a third process tends to violate that restriction.

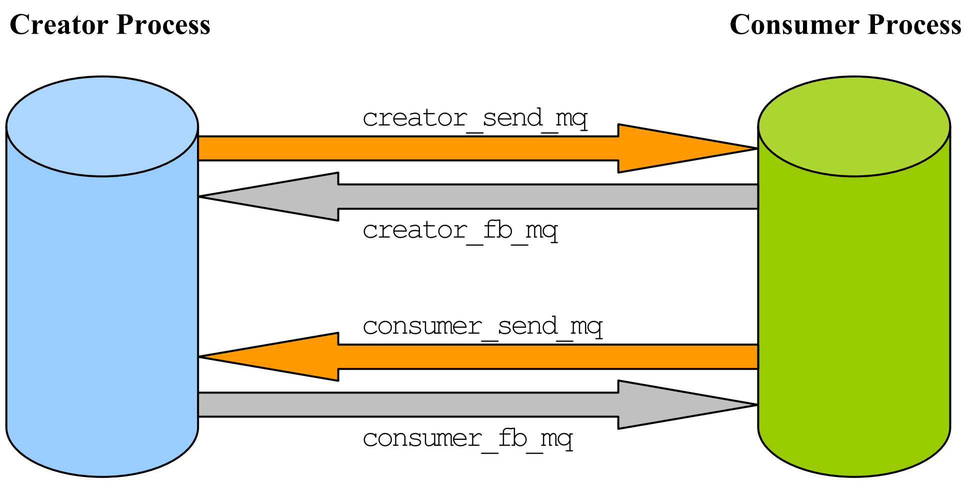


Figure 34: 4 raw message queues in the 1:1 communication.

Figure 34 shows the relationship between the creator and the sender. As we want to guarantee the respective sender that the respective receiver has or has not received a message properly, feedback infrastructure is introduced. To simplify the architecture one logical safe message queue now consists of 4 raw message queues who are explicitly opened in read only, respectively write only mode depending on which side of the communication they are opened. The respective `***_send` queues are message queues for the intended message payload, the others are control queues used for feedback. Figure 35 schematically describes how the data messages are structured. This architecture is suggested by [CEN50159-1].

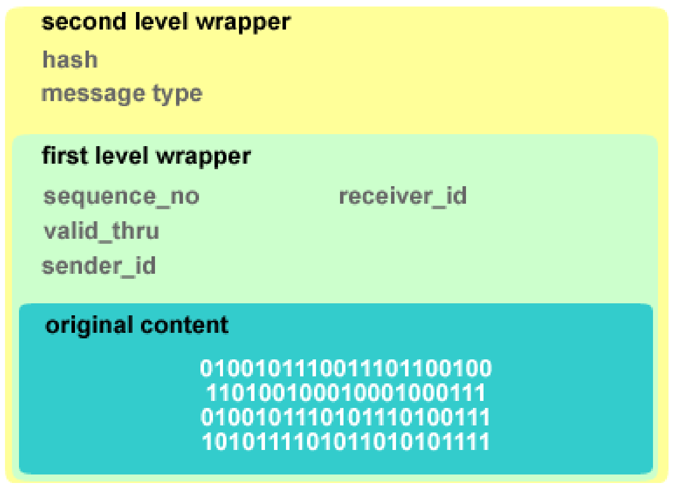


Figure 35: Basic message structure.

5.3.3 Data Structures

As the POSIX functions are preloaded and enriched with functionality more data is needed to describe the items used by the communication infrastructure. So the descriptors produced by the common functions have to be overwritten and enriched as well.

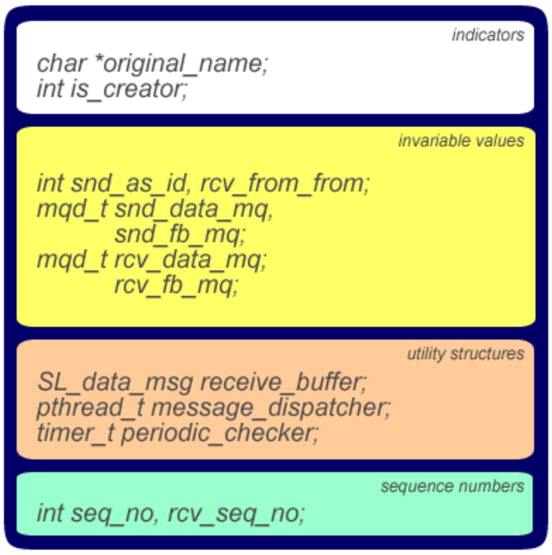


Figure 36: `SL_mqd_t` descriptor.

Figure 36 describes the new structure that is generated at `mq_open` and passed to the application. At `mq_send` and `mq_receive` this `struct` is passed analogously to the common POSIX functions. The white *indicators* field holds values that refer to the role and the logical name of the calling process. They are of no productive use but are useful for debugging and logging purposes. *Invariable values* are once set in the initialization process and are not allowed to change over the lifecycle of the process. Those are the sender and receiver identifiers and raw message queue descriptors. *Utility structures* are set up in the initialization phase and hold descriptors of needed helper functions specified below. *Sequence numbers* are of global character according one logical queue. Hence they got their place in the message descriptor. The semantics of the stated variables is further explained in the functional specification below.

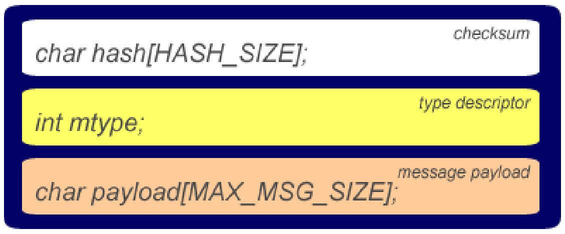


Figure 37: `SL_msg`**, basic wrapper for all messages.**

Figure 37 shows the overall wrapper for all types of messages that run over the layer. It captures two of the CEN50159-prescribed measures: the checksums and message type checking. This is the structure that is serialized and sent through the raw message queues. The lower level message structures are serialized and put to the *payload* buffer. The original type of the message is stored in the *mtype* field. Then a checksum generation algorithm is used to get a checksum value to be stored in the *hash* field. This wrapper is especially useful because we want to guarantee that the checksum is generated over all available data.

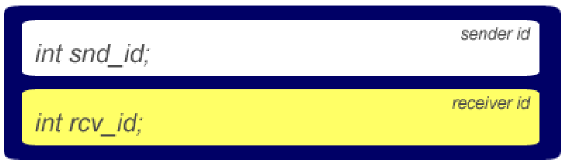


Figure 38: `SL_init_msg`**, initialization message.**

Messages of the type shown in Figure 38 are only sent/ received in the initialization phase while the one creator and the one consumer of the message queue perform a double handshake protocol to identify themselves. If an init message is received in the production phase this will be recognized as an error.

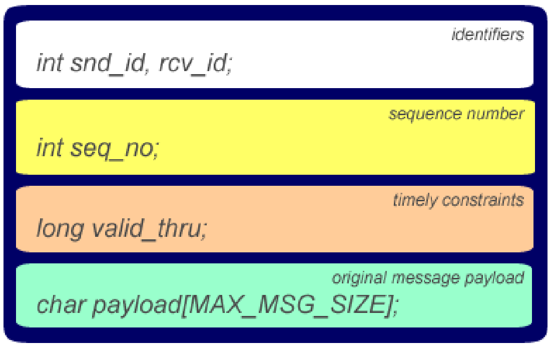


Figure 39: `SL_data_msg` **used for data and periodic checks.**

Figure 39 shows basically the message `struct` that is to be sent, when an explicit `SL_mq_send` is performed. The original message payload is serialized and put to the payload field. Then the other variables necessary are set like sender and receiver ids, the current sequence number and a descriptor, when a message has to be consumed and confirmed.

5.3.4 Sender and receiver identification

One topic that occurred during the conception of the Safe Message Passing Layer was how to implement sender and receiver identification as required by [CEN50159-1]. The problem was to define identifiers for the communicating processes. The solution chosen was to take the process' id as the sender id. In the initialization procedure this identifier is handled over to the receiving process. If the receiving process receives a message stamped with another sender identifier, this is recognized as an error.

This solution is justified by the following argumentation: [CEN50159-1], on which the Safe Message Passing Layer is based on, is restricted to closed communication channels, which means that:

- **Only authenticated access is allowed.** This is done by the operating system and is considered as given. Only processes that are allowed to take part in this communication may access the message queue. Authentication errors are part of another operating system functionality and should be tested by other wrappers or checks. This is what justifies the use of the process ids as identifiers.
- **There exist a maximum number of communication participants.** There are only 2 communication partners allowed for one logical message queue. This is defined by the 1:1 restriction. If another process tries to break that restriction and take part in the communication, this is recognized as an error.
- **The communication media is known and specified.** The raw POSIX message passing functionality is the communication media under consideration. This is specified properly by the POSIX specification.

The main class of faults that shall explicitly be detected by this approach are invalid application infrastructure setups. One exemplary case would be a process that is intended to communicate with a child process. Potentially the process forks another child process while the message queue descriptor is passed to that child. The child could now try to send messages to the other process through that queue. This is not intended as it may harm message sequencing and feedback functionality. As the handling of such setups is not intended by the specification, an error will be detected by the receiver because of diverging process identifier passed over to the message.

5.4 The preloaded POSIX Functions

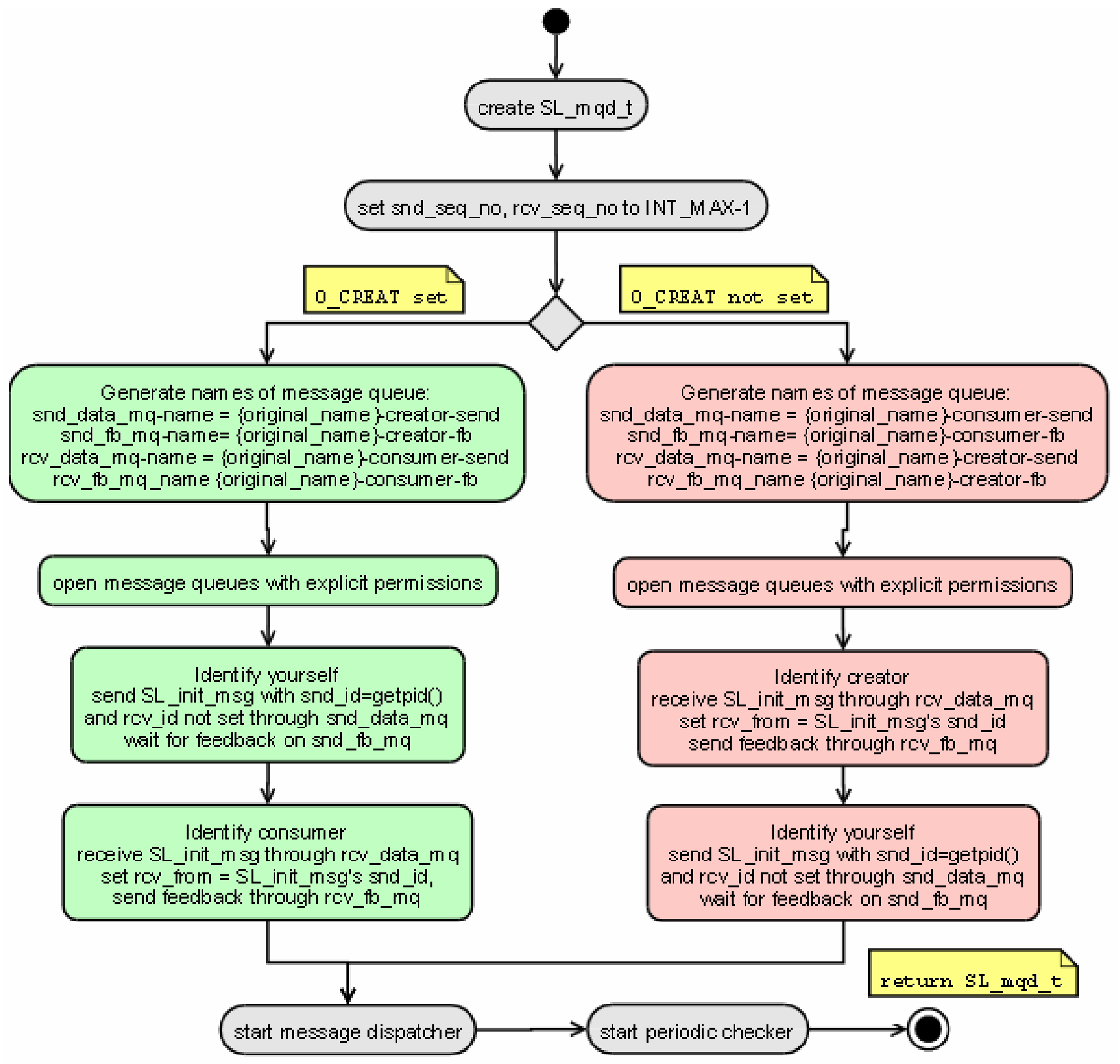
This chapter describes how the safety methods are implemented and the considerations and restrictive constraints made for their design. Restrictions are especially important as they are needed to consider Message Passing functionality as a closed communication system. One important issue is that real time constraints were not under consideration for the original Message Passing functions, but the goal is to allow the programmer that uses the safe functionality to embed it exactly the same way as the old, unsafe function. So the real time constraints are implemented as a global timeout value. Their valid through property is added to the message on the very beginning of a function and checked at the receiver as well as on the very end.

A very important constraint is that our message passing functions are set to be unidirectional. This means that only the process that opens a message queue with `O_CREAT` may send a message through this message queue and only the first process that opens it without O_CREAT may consume messages from it. The `O_EXCL` flag shall always be set implicitly to avoid confusions.

5.4.1 Opening of the Message Queue

```
SL_mqd_t SL_mq_open( const char *name, int  oflag ); // open as consumer
SL_mqd_t SL_mq_open( const char *name, int  oflag,
              mode_t mode, struct mq_attr ); // open as creator
```

As `SL_mq_open` is called on the sender and the receiver side there are two paths run through. On the creator side when the message queue is opened with `IPC_CREAT`, the function sets the sequence counter to `INT_MAX-1`. Then it generates its own sender ID randomly. After doing that it creates on the one hand the actual message queue that is used by this one process to send its own messages. It is opened just with the write permission. Then it creates another message queue with read only permission and starts the initialization phase. The mode and `mq_attr` attributes are ignored implicitly, as the setup of the raw message queues below the layer are opened in clearly defined write and read only modes depending on their role in the 1:1 communication.



create SL_mqd_t
set snd_seq_no, rcv_seq_no to INT_MAX-1
O_CREAT set
O_CREAT not set
Generate names of message queue: snd_data_mq-name = {original_name}-creator-send snd_fb_mq-name= {original_name}-creator-fb rcv_data_mq-name = {original_name}-consumer-send rcv_fb_mq_name {original_name}-consumer-fb
Generate names of message queue: snd_data_mq-name = {original_name}-consumer-send snd_fb_mq-name= {original_name}-consumer-fb rcv_data_mq-name = {original_name}-creator-send rcv_fb_mq_name {original_name}-creator-fb
open message queues with explicit permissions
open message queues with explicit permissions
Identify yourself send SL_init_msg with snd_id=getpid() and rcv_id not set through snd_data_mq wait for feedback on snd_fb_mq
Identify creator receive SL_init_msg through rcv_data_mq set rcv_from = SL_init_msg's snd_id send feedback through rcv_fb_mq
Identify consumer receive SL_init_msg through rcv_data_mq set rcv_from = SL_init_msg's snd_id, send feedback through rcv_fb_mq
Identify yourself send SL_init_msg with snd_id=getpid() and rcv_id not set through snd_data_mq wait for feedback on snd_fb_mq
return SL_mqd_t
start message dispatcher
start periodic checker

Figure 40: `SL_mq_open` **command flow.**

Figure 40 shows the commands that have to be performed in order to set up the basic infrastructure. At first a `SL_mqd_t` struct is created and the sequence numbers are set to the `INT_MAX-1` value. `INT_MAX` is defined in `limits.h` and specifies the highest value for the integer data type. As the overflow when the data maximum value for a data type is reached is a possible source of failure, it is advantageous to notice such a fault as soon as possible. This is why the sequence numbers are initialized to that value and e.g. not to 0 or 1.

Then the raw message queues are opened with modes depending on their role and process. For Example, if the logical name(the name specified by the application) of the queue is “`testmq`” the queue with the actual name “`testmq-creator-send`” is opened on the creator side as `snd_data_mq` with the flags `O_CREAT` and `O_EXCL` is opened with `O_WRONLY` permission. O_CREAT indicates that the message queue should be created if it does not exist. O_EXCL means, that that an error should be returned if the queue exists yet. This helps maintaining the 1:1 constraint. The O_WRONLY prevents messages from being received from that channel. On the consumer side "testmq_creator_send" is opened as rcv_data_mq in O_RDONLY mode. This task is performed by the establish_mqs(...) function. See below for further details.

After the opening of the raw channels the initialization phase starts performing a double handshake protocol. At first it is the creator's part to send an initialization message containing its process id as sender id through its snd_data_mq. The receiver listens on its rcv_data_mq for that message. It sets its rcv_from value according to the SL_init_msg's snd_id and sends feedback. This feedback is again consumed by the creator that sets its rcv_from to the value in the feedback message. Then the same process is performed vice versa checking the data that has been set before.

A structure called message dispatcher is started. This is a thread that listens on the queue and performs first checks, sends feedback and handles messages over to the intended receivers. Because of that structure 2 semaphores had to be introduced and are initialized before starting the dispatcher, one of which is for writing to, one for reading from the buffer. After that the periodic checker function realized as a POSIX timer is started as requested by the project partner.

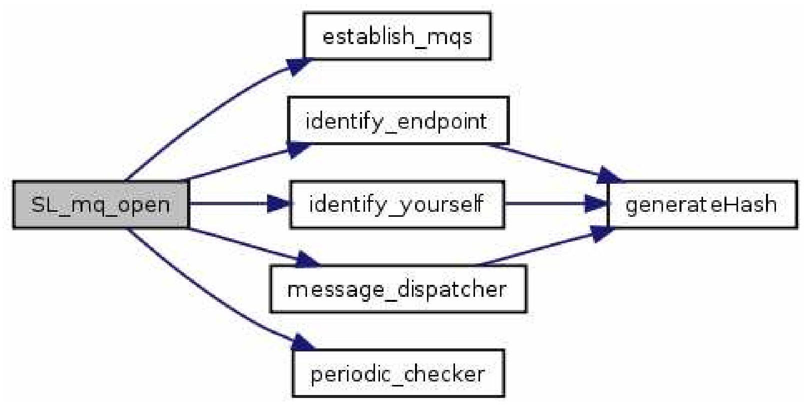


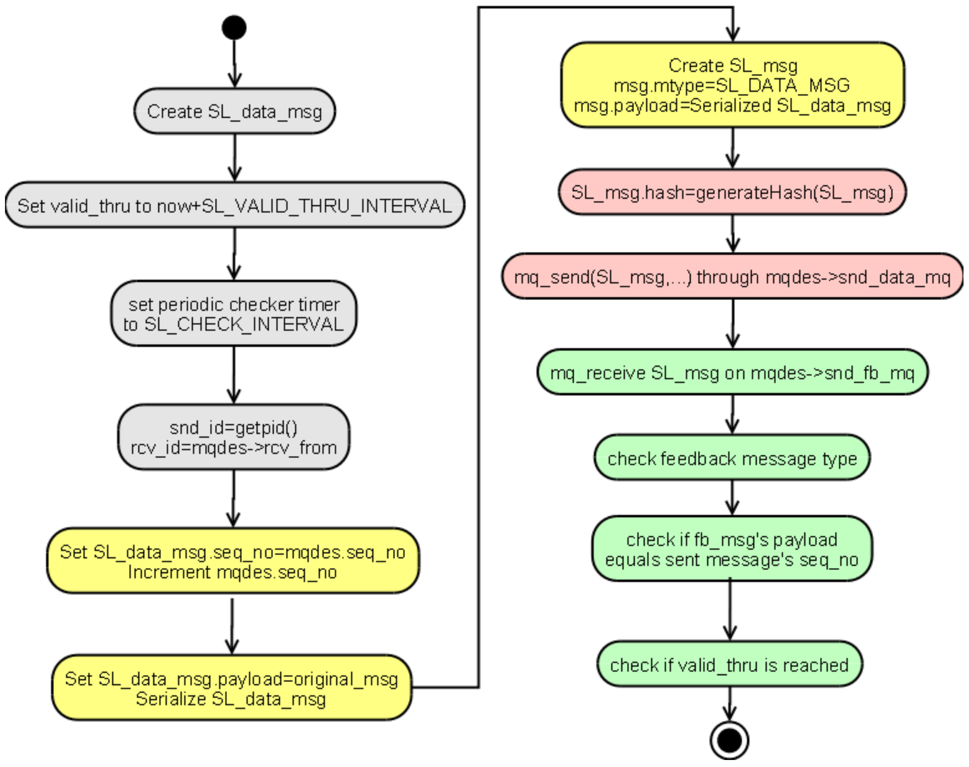
Figure 41: SL_mq_open **call graph.**

Figure 41 shows the crucial initialization steps performed by SL_mq_open and how the functions are linked. As you can see, the initialization task is quite complex and uses a bunch of helper functions. Those helper functions are contained in an own header file named smp50159_mq_open_helpers.h and are explained in detail below and in the Doxygen documentation. The identify_yourself and identify_endpoint functions are called in alternate order and use all of the four raw message queues in order to ensure that they are functioning.

5.4.2 Send Message

```
int SL_mq_send(SL_mqd_t *mqdes,
        const char *msg_ptr,
        size_t msg_len,
        unsigned int msg_prio)
```

Figure 42 shows how the designed sending method operates. As suggested above it puts all the needed check data on the message, puts it on the queue and waits for feedback. For the prototype the message priority parameter is ignored.



Create SL_data_msg
Set valid_thru to now+SL_VALID_THRU_INTERVAL
set periodic checker timer to SL_CHECK_INTERVAL
snd_id=getpid() rcv_id=mqdes->rcv_from
Set SL_data_msg.seq_no=mqdes.seq_no Increment mqdes.seq_no
Set SL_data_msg.payload=original_msg Serialize SL_data_msg
Create SL_msg msg.mtype=SL_DATA_MSG msg.payload=Serialized SL_data_msg
SL_msg.hash=generateHash(SL_msg)
mq_send(SL_msg,...) through mqdes->snd_data_mq
mq_receive SL_msg on mqdes->snd_fb_mq
check feedback message type
check if fb_msg's payload equals sent message's seq_no
check if valid_thru is reached

Figure 42: `SL_mq_send` **control flow.**

As shown the very first step after creating a `SL_data_msg struct` is to set the `valid_thru` field. This is an absolute timestamp that is checked. Then the periodic checker timer for this message queue is set back to the maximum timer interval. This is done in order to avoid overhead, because there is little sense in periodically checking the send and feedback queue in situations where there is enough traffic on the queues, as there is enough checking by sending messages anyways.

Then the function puts the intended receiver and process's sender id onto the message header. Then it stamps the current "send sequence number" on it and increases the `snd_seq_no` value in the message descriptor. After that the original message is copied to the SL_data_msg's payload field without checking. The first level wrapper is now set up completely.

Now the second level wrapper has to be set up. Hence a SL_msg struct is created and the first level wrapper converted to a char array is put as the SL_msg's payload field. Then the message type is set to SL_DATA_MSG and a message hash is generated and set in the message header. (see generateHash(...) below)

After that the message is sent through the raw snd_data_mq. As we need to know if the message was consumed properly, we need to wait for feedback. In the prototype this is done by a blocking mq_receive in the normal message control flow, but there are more options that may be considered in further versions of the layer. The communication on the raw queues is always realized as blocking, so realizing some sort of feedback listener as a POSIX thread would be a proper variant. The function would in this case just create a thread after sending, which listens for feedback. If there is something wrong with it, the thread could halt the system anyways.

If the feedback arrives the process will check if the sequence number, which is the feedback message's payload corresponds to the one sent. Then it checks whether valid_thru is reached. If there is something wrong, a signal is raised.

5.4.3 Receive Message

```
int SL_mq_receive(SL_mqd_t *mqdes,
         char *msg_ptr,
         size_t msg_len,
         unsigned int *msg_prio)
```

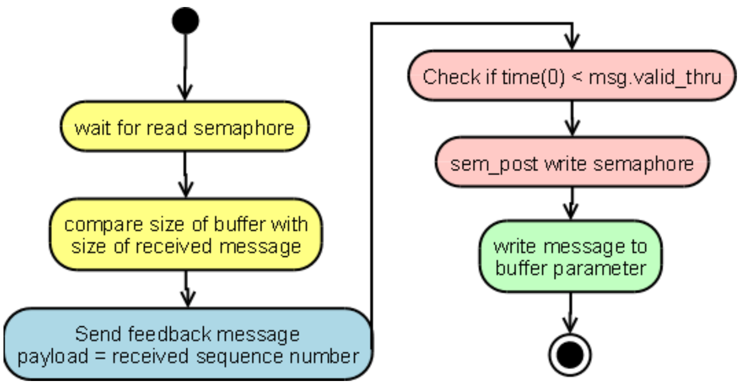

wait for read semaphore
compare size of buffer with size of received message
Send feedback message payload = received sequence number
Check if time(0) < msg.valid_thru
sem_post write semaphore
write message to buffer parameter

Figure 43: SL_mq_receive **command flow.**

Figure 43 shows how the explicit receiving algorithm is supposed to perform. At first the receiving process waits for a semaphore that logically allows it to read from the buffer, where the message dispatcher places a received data message. Then it compares the size of the message in the buffer with the size of the message received to avoid errors. Then it sends a feedback message through `mq_rcv_fb` containing `snd_id` set to `getpid()` and `rcv_id`. Then it checks, if the timely constraints are fulfilled and posts the semaphore that logically allows another thread to write to the buffer. After that the message dispatcher will get the next message from the queue. Then `SL_mq_receive` writes the message payload to the parameter handed over as buffer and returns the size of that parameter. If anything fails it either raises a signal or returns `-1`;

5.4.4 Closing the queue

By definition the communication channels shall – once opened – never be allowed to be closed. If this method is called, an error signal is raised. This is a requirement coming from the project partner who introduced that restriction.

5.5 Helper Functions

The following items describe the functionality of helper functions contained in diverse header files. For further descriptions and the code refer to the Doxygen documentation.

5.5.1 Establish the Message Queues

```
int establish_mqs(SL_mqd_t *mqdes,is_creator)
```

This function is the first helper in the initialization process. It generates the raw names of the four message queues. Then it opens two branches, one of which is called by the creator, the other one called by the consumer of the logical queue. The control flow is shown below in Figure 44.

The creator opens the queues with the flags `O_CREAT`, which means, that the queue shall be created if it does not already exist. Implicitly it adds the `O_EXCL` exclude flag, determining that the function shall return an error if the queue already exists. This is because of our 1:1 relationship constraint, stating that exactly one process is supposed to create the message queue and exactly one message queue is supposed to open the previously created structures. If later another process wants to take part in that communication, this violates the restriction, which is out of specification. Apart from that the raw queues are explicitly opened with permissions according to their role in the communication. The `snd_data_mq` and `rcv_fb_mq` are opened with write only permission, `rcv_data_mq` and `snd_fb_mq` are opened with read only permission to avoid unrecognized malfunctions later. This is not an error recognition strategy but merely a defensive programming approach.

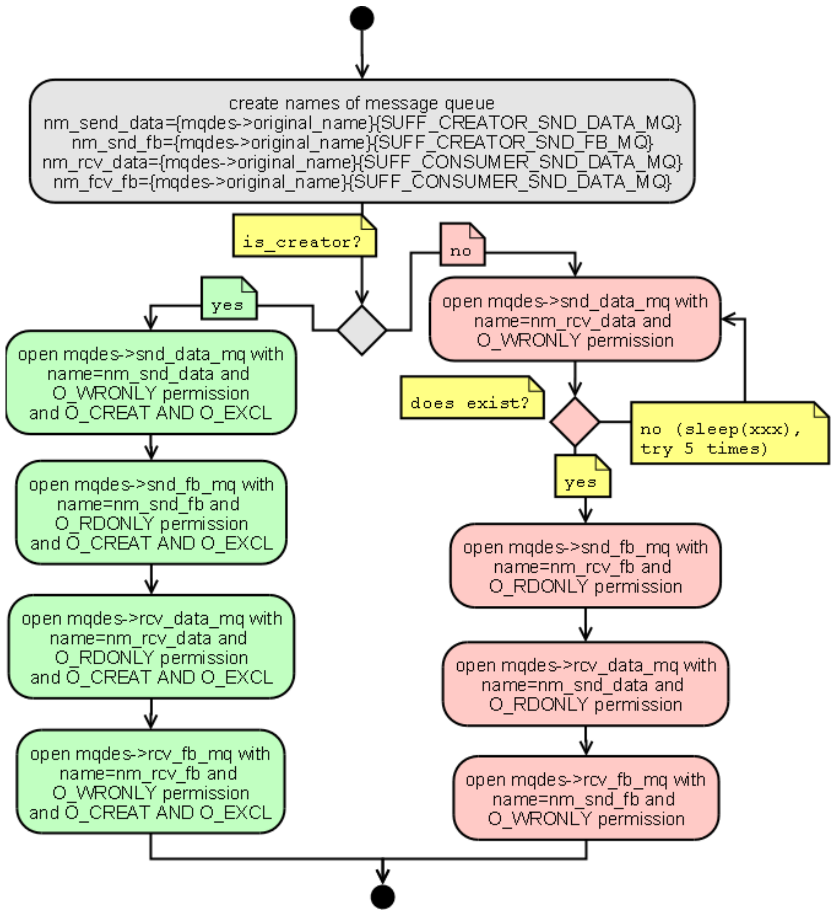

create names of message queue
nm_send_data={mqdes->original_name}{SUFF_CREATOR_SND_DATA_MQ}
nm_snd_fb={mqdes->original_name}{SUFF_CREATOR_SND_FB_MQ}
nm_rcv_data={mqdes->original_name}{SUFF_CONSUMER_SND_DATA_MQ}
nm_fcv_fb={mqdes->original_name}{SUFF_CONSUMER_SND_DATA_MQ}
is_creator?
yes
no
open mqdes->snd_data_mq with name=nm_snd_data and O_WRONLY permission and O_CREAT AND O_EXCL
open mqdes->snd_fb_mq with name=nm_snd_fb and O_RDONLY permission and O_CREAT AND O_EXCL
open mqdes->rcv_data_mq with name=nm_rcv_data and O_RDONLY permission and O_CREAT AND O_EXCL
open mqdes->rcv_fb_mq with name=nm_rcv_fb and O_WRONLY permission and O_CREAT AND O_EXCL
open mqdes->snd_data_mq with name=nm_rcv_data and O_WRONLY permission
does exist?
no (sleep(xxx), try 5 times)
yes
open mqdes->snd_fb_mq with name=nm_rcv_fb and O_RDONLY permission
open mqdes->rcv_data_mq with name=nm_snd_data and O_RDONLY permission
open mqdes->rcv_fb_mq with name=nm_snd_fb and O_WRONLY permission

Figure 44: Control flow `establish_mqs(SL_mqd_t *mqdes,is_creator).`

5.5.2 Generate the Checksum

```
char *generateHash(SL_msg msg)
```

This function is used to determine a checksum over a message that is to be sent over the channel. In this case, the message is serialized to a char array that is handled over to the crypt library's crypt function. A standard salt is defined globally. The checksum generation function may be changed easily to other variants. To give some short words the quality of checksum functions, the better the randomness of the checksum generator, the better quality of the checksum is, as it gets more and more unlikely that two similar data arrays map to the same checksum.

5.5.3 Sending Identification Data

```
int identify_yourself(SL_mqd_t *mqdes, int is_creator)
```

The identify yourself method also differs depending on what side of the channel it is executed. It basically creates a `SL_init_msg` and sets its `snd_id` to `getpid()`. If it is performed on the creator side, it is the first identifying function to be performed, so no intended receiver is known. On the consumer side it's the second to be called, so we already know which process is intended to be the receiver, so the `rcv_id` can be set to that value. The init messages are wrapped in an `SL_msg` wrapper and type and hash value are set. Then they're sent and the process performs a `mq_receive` on its `snd_fb_mq`. If the function is called on the creator side, the process does not know a priori, which process is the intended consumer. The feedback message contains the consumer's process id and so the creator can set its descriptor's `rcv_from` accordingly. If the function is performed on the consumer side, the process knows the id of the process that had identified itself before and just checks if those two values correspond. Otherwise it raises a signal.

5.5.4 Endpoint Identification

```
int identify_endpoint(SL_mqd_t *mqdes, int is_creator)
```

When executed on the consumer side it immediately listens on the `rcv_data_mq` for a `SL_init_msg`. When this message is received, it sets the message queue descriptor's `rcv_from` value to the `snd_id` of the `SL_init_msg`. Then it generates a feedback message putting its own process it as the message payload and sends it through `rcv_fb_mq`.

When called on the creator side it listens for a `SL_init_msg`. As the descriptor's `rcv_from` value is supposed to be set before by the `identify_yourself` message it checks the ids and sends feedback respectively to the results.

5.5.5 Periodically checking the Communication Infrastructure

This is the handler function for the timer that periodically sends a randomly generated message through the communication channel, if there is no intended traffic on it. If there is standard traffic, which basically means that there are `SL_mq_send()` calls, the timer value is set back in order to avoid overhead.

The mission of this timer handler is just to use the communication channel in an end-to-end way, it just performs `SL_mq_send()` calls to test if the channel is still up as intended. Hence every side of the queue checks its own `snd_data_mq` and `snd_fb_mq`, simultaneously verifying that the other side's message dispatcher is up.

The special nature of this construct is that it performs application invisible, uninitiated sends. Therefore there would be no explicit listener to consume that message. This is in fact why the message dispatcher, the receive buffer, the semaphores and the timer holder had to be implied to the system. They are needed because of the periodic checks and the need to compensate those uninitiated receives.

When a timer is run down, a signal is raised by the system. Hence it is not possible to pass the timer a contextual parameter, a workaround structure was imposed. Before the timer is initialized, a `struct` mapping the timer pointer to the message queue descriptor is put to a global linked list. The signal handler is handed over a signal descriptor containing a field, which indicates the timer id. Thus the timer handler gets the respective descriptor from the list.

5.5.6 The Message Dispatcher

```
void message_dispatcher(void *arg)
```

This process handles the delivery of all the incoming messages. It consumes the message, performs first checks and writes it to a receive buffer, where an `SL_mq_receive` call may get it from. If the message is of type `SL_FB_REQUEST_MSG`, which is sent by the periodic checker, it performs the required checks on it, immediately sends feedback and disposes the message. The reason why raw message consummation cannot be done directly via `mq_receive` is that there are different natures of the `periodic_checker` and explicit `SL_mq_sends` consummation. `SL_mq_sends` imply that there is a consumer waiting to process the sent message in a limited interval after the point in time the message is sent. Periodic checker messages do not expect another thread to be listening, but still the `SL_mq_send` function waits for that feedback.

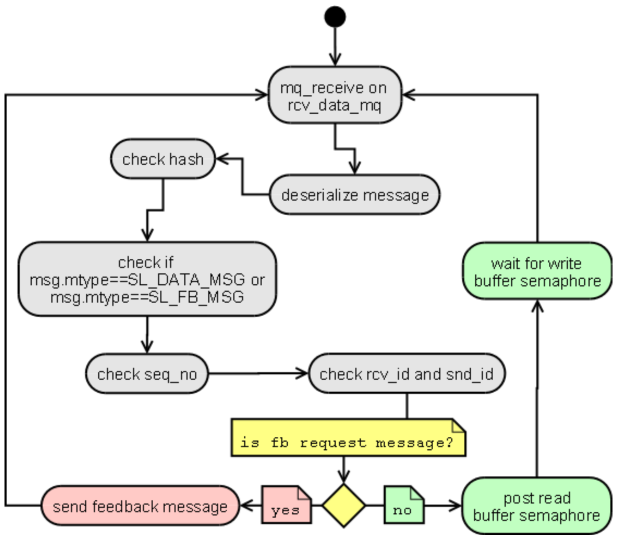

mq_receive on rcv_data_mq
check hash
deserialize message
check if msg.mtype==SL_DATA_MSG or msg.mtype==SL_FB_MSG
wait for write buffer semaphore
check seq_no
check rcv_id and snd_id
is fb request message?
send feedback message
yes
no
post read buffer semaphore

Figure 45: `message_dispatcher` **control flow.**

Figure 45 shows how the message dispatcher deals with the messages. It checks all items except for `valid_thru` and sends feedback, if the received message is a feedback request. If it is a normal data message, it posts the semaphore that is used to synchronize reading from buffer and waits for a semaphore that is used for writing to buffer. This listen-and-check task is performed in an infinite loop.

5.6 Test Environment

The Safe Message Passing layer was developed and tested under Ubuntu Linux 7.10 [12]. The library and test cases were written under Eclipse CDT [13] and tested using the built-in GNU debugger [14] plug-in. Safe Message Passing Layer's documentation has been generated by Doxygen [15].

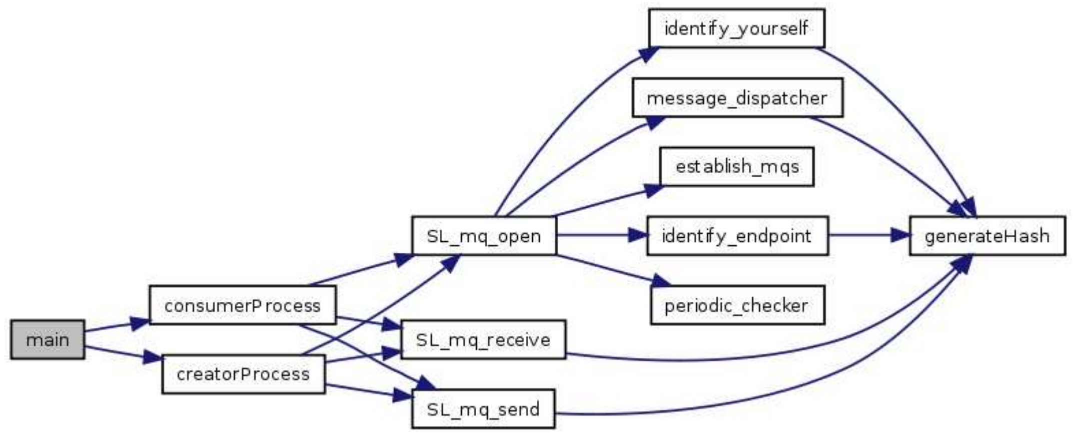
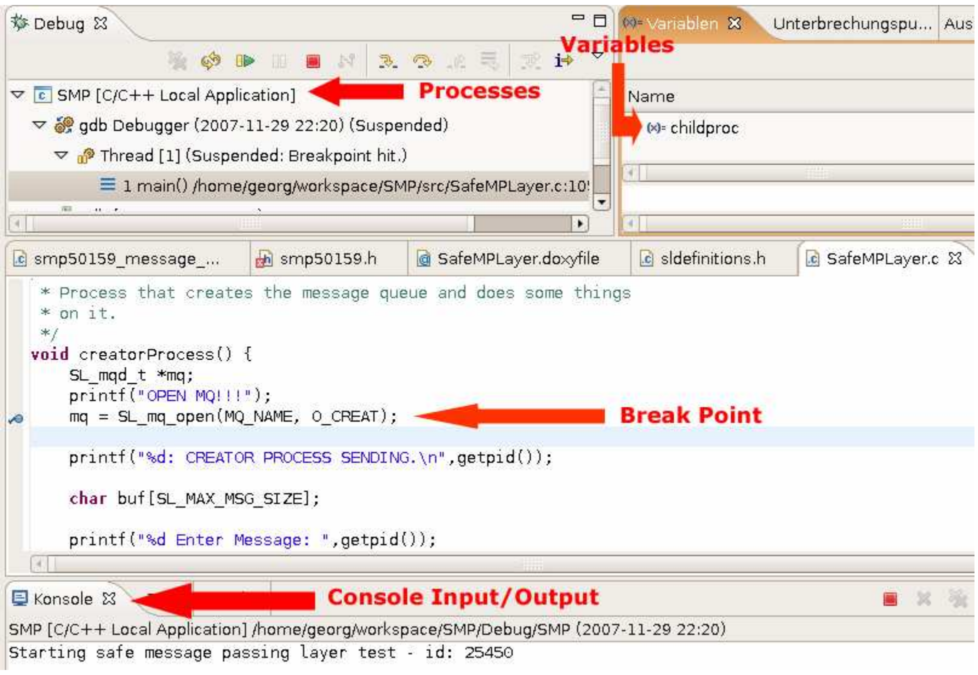


Figure 46: Linkage between the functions.

Figure 46 shows how the basic roles make use of the functions. The following test cases are based on the specification of the layer, but – as the verification is needed that the functions detect errors that seldom occur – fault injection is needed to show that it works as intended. The chosen way is to set GNU debugger breakpoints and alter messages to see if those changes are recognized as errors (as intended). Figure 47 shows the main view for graphical gdb interface. Processes and threads are shown, breakpoints can be set. On breakpoints the execution is halted and currently allocated variables and their current value are shown in the upper right. Those values can be modified. This is where that fault injection took place.



```
SMP [C/C++ Local Application]
  gdb Debugger (2007-11-29 22:20) (Suspended)
    Thread [1] (Suspended: Breakpoint hit.)
      1 main() /home/georg/workspace/SMP/src/SafeMPLayer.c:10
```

```
 * Process that creates the message queue and does some things
 * on it.
 */
void creatorProcess() {
    SL_mqd_t *mq;
    printf("OPEN MQ!!!");
    mq = SL_mq_open(MQ_NAME, O_CREAT);

    printf("%d: CREATOR PROCESS SENDING.\n",getpid());

    char buf[SL_MAX_MSG_SIZE];

    printf("%d Enter Message: ",getpid());
```

```
SMP [C/C++ Local Application] /home/georg/workspace/SMP/Debug/SMP (2007-11-29 22:20)
Starting safe message passing layer test - id: 25450
```

Figure 47: Eclipse CDT debug environment.

5.6.1 Single Client Case

A test case covering all the functional requirements was implemented. The test program just forks two processes one of which creates a safe message queue with a given name, the other opens that same logical queue as consumer.

Now the consumer listens for messages on that queue while on the creator side input is read from the command line interface. When the user enters a message and end of line is recognized, this message is sent. The consumer writes the received message to the console and listens for input from the command line itself. Now the creator listens for incoming messages and writes it to the console output. This is run in an infinite loop.

This case is a good choice to prove the basic functionality of the layer. In order to test the periodic checker simply no messages are sent intentionally. Faults can easily be injected by gdb. The breakpoints are set before the raw `mq_send` is performed. One can then either change the sequence numbers or ids before the data messages are serialized or change the hash or message payload or type of the `SL_init_msg` to test if errors are recognized.

5.6.2 Invalid Application Setup Case

This is a short test case simulating that an application forks processes that want to open the same logical queue. This would violate the 1:1 restriction, hence a signal has to be raised. This scenario may appear because of invalid application configuration.

5.6.3 Multiple Clients Case

The main application forks N processes that, the first of those processes opens one message queue for each of the others. The 1+m process then opens its queue as consumer. Then the other processes randomly wait to read from the standard input, take that input and send an according message to a server process 1. The server then sends back some data message. Figure 48 shows a schematic view of this test case.

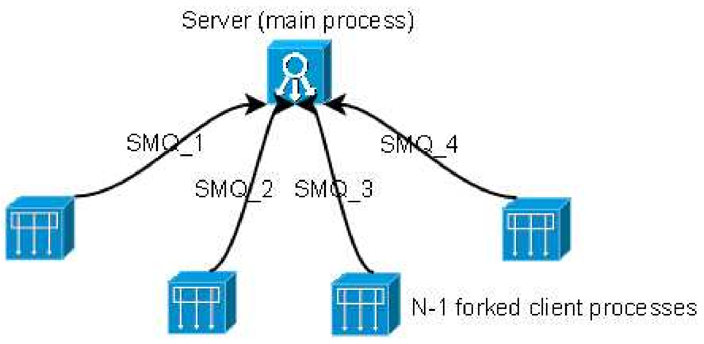


Figure 48: Multiple clients scenario.

5.6.4 Stress Test Case

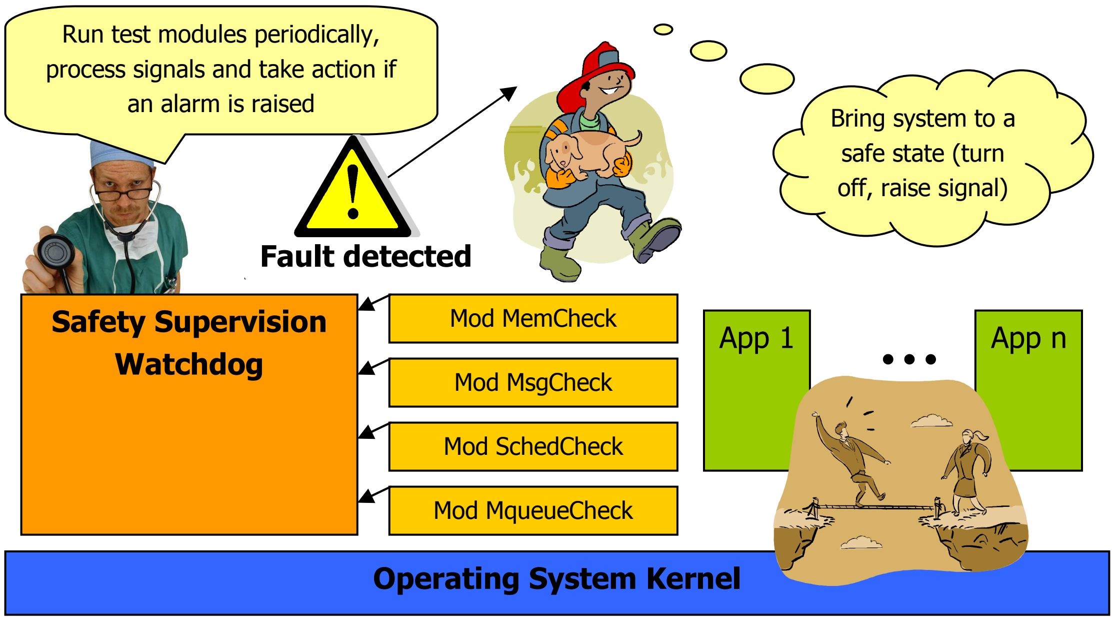
This test case is quite similar to the case cited in 5.6.2. The only difference is that this just produces and sends respectively receives message content randomly as fast as possible. The goal is to ensure that sequence number and timing implementation are sufficient. Results have been written to a file and analyzed.

6 Safety Supervision Watchdog

The safety library functions are an interesting way to detect errors exactly at the point in time they occur. They embody a direct approach that minimizes error detection latency. Nevertheless there are classes of errors that cannot appropriately be detected in such a direct way. For example when a process is about to starve because of malicious scheduling behaviour, this would be hard or impossible to detect. Hence the detection of such errors should be handed over to an outside structure. This is where the watchdog comes into play.

6.1 Architecture

The supervision watchdog part is a user level UNIX demon that periodically checks basic safety critical operating functionality. This functionality may be memory management, process memory integrity, scheduling, message passing. Modular extensions are supported.



Run test modules periodically, process signals and take action if an alarm is raised
Fault detected
Bring system to a safe state (turn off, raise signal)
Safety Supervision Watchdog
Mod MemCheck
Mod MsgCheck
Mod SchedCheck
Mod MqueueCheck
App 1
...
App n
Operating System Kernel

Figure 49: Concept of a periodically checking safety watchdog.

It is intended to substitute the part that the classical hardware online tests have to perform in current systems. There are some faults that may not be checked reasonably at the time when the functionality is used, so this is where we still need some kind of periodically running watchdog facility. The watch dog has not been implemented, this chapter describes in detail the part of the layer's mission such a construct is assigned, how it could be implemented and what kind of checks the daemon should perform. Figure 49: Concept of a periodically checking safety watchdog provides a schematic vision of how it is supposed to perform.

6.2 Basic Features

The watchdog basically checks functionality that can not at all or reasonably be checked in a direct way in the sense of the message passing layer. There are a lot of items for that current test scripts and programs have been written yet. A usual site to check when looking for a Linux system function test is the Linux Test Project [10]. There is the possibility to easily write wrappers for those modules.

6.2.1 Modular Extendability

The daemon should be kept extendable using common modular techniques. An interface should be defined specifying the basic actions to be taken. When starting up the daemon loads the modules specified by standard configuration files. Those files contain the name of the shared object specifying the test, the time when the test is to be started first and the interval to perform that test periodically.

The tests themselves contain function similar to common testing environments like JUnit methods to setup the test environment and cleanup methods specifying what to do when the system is intentionally shut down. Besides the three functions enumerated above, another handler is needed, which is a handler to deal with situation when the watchdog is shut down because an error has been detected. This is important for forensics and the circumstance that timely constraints mustn't be violated especially in actions that are intended to prevent hazardous situations. Below this function is referred to as panic function.

6.2.2 Logging and Debugging

Hence faults driving the watch dog to shut down the system it will be useful to have it log the tests it performed and when, whether the check was successful or not and a readable reason and state footprint of the point in time when the system was shut down. This helps the maintenance team to get after the fault.

Logs have to include entries for successful checks including the time when these tests were performed. When a fault is detected the module shall state as close as possible which kind of fault and where it was detected.

6.2.3 Notification

Safety-relevant applications may have their specific actions to be done when errors are recognized. As this is easy to implement when the checks are synchronously performed directly on a function call, the watchdog needs to implement a notification service those applications can register at. If the watchdog detects an error, it notifies the registered programs that can now take their own specific actions.

6.3 Modules

As explained above, the watchdog provides the possibility to write proprietary modules. As mentioned above, the Linux Test Project contains a lot of tested scripts to check system functionality. Those modules can be built out of existing tests, just a wrapper providing the capability to specify what to do in case of a detected error has to be built.

Currently LTP is a test application for which one can specify which tests to run and the its running the those checks once writing results to system logs. It neither request the split of the setup, cleanup, and panic functions, nor is it possible to run it periodically without using schedulers or wrappers.

On the other side there are a lot of scripts already written for this environment, which can easily be used for the layer with slight changes. So in order to prevent us from re-inventing the wheel, the watchdog should not be held as compatible as possible to the LTP.

6.3.1 Module Configurations

Each model can be configured flexibly in own configuration files. Three options are intended to be read by the layer. Those are:

- **Module name**. This is important for logging. The respective module calls a logging function provided by the layer. This logging function uses the given module name parameter to structure the logs. The module name should also be used to gain parameters from the configuration handler.
- **Starting time**. Indicates when after the start of the layer the check specified by the module should be performed for the first time.
- **Check interval**. Specifies the interval the module should be called from the first call on.

Apart from those, module specific checks should be supported. For example there should be provided the possibility to configure memory checks that they either perform a test on the whole memory or just sample parts. The module specific configuration feature is important because the certain tests are of completely different nature. E.g. memory checks significantly differ from network and scheduling tests.

6.3.2 Existing LTP Tests

[10] enumerates a lot of tests already written some of the issues suitable for the considered watchdog. Some of them are not suitable for use in this daemon, either because they are demanding too many resources or they are not relevant to safety issues.

Online tests that should be adopted are memory tests, disk input-output tests, task scheduling, process integrity tests and others. For example tests that should not be included are e.g. stress tests, because they could harm the system's performance.

Generally the Safe Message Passing Layer may easily be implemented as an own LTP test. The test cases described in the previous chapter just have to be configured and called by the layer. Hence the respective watchdog module just consists of code using the functions described by it. Please note that the restrictions to the 1:1 relation remains for the test. Only one to one message passing may be checked.

6.4 Limitations

The safety watchdog's nature allows the performance of periodic checks on every function intended. One special thing about this is that those tests are not performed on the results of explicitly called tasks. Thus they are not capable of detecting errors produced by functions currently called, but rather they are a best effort strategy to find faults before they are activated.

Nevertheless such a structure is required in a safety layer like the proposed, because there is functionality that may not reasonably be detected during action. For example some of them are process scheduling, certain timing issues or memory integrity. The layer can also be used for low-priority overall system checks, for example for memory management or file handling tests. Please not that the watchdog is not a direct way to find faults as they are not detecting an error when the function is called, but rather a heuristic approach to find faults before they are activated.

7 Conclusion and Outlook

The presented approach is a suitable method for avoiding wrong control actions by safety related applications based on faulty system calls by detecting their errors in a layer just below the application. The layer consists of two parts, the safety functions and the safety watchdog. The watchdog is needed because not all possible faults can be detected by the direct checks provided by the functions.

The layer is based on POSIX to ensure high portability. This is especially important because the structure itself is intended provide system developers to move their safety relevant applications to other operating systems and hardware without harming safety. It has to be noted the presented structure is not improving reliability and that its sole mission is to provide fault-detection and help to guarantee safety. Furthermore there are far more issues to be concerned when it comes to safety and fault-detection is just a small part in the puzzle that's only mission is to avoid hazardous situations initiated by unintended actions and calculations diverging from the specification.

The justification of the Safety Supervision Layer comes from the application of commonly used industry standards. To map the requirements described by the standards fault-analysis methods have to be applied on the according operating system functionality to check whether the standard is applicable. In this work CENELEC railway standards have been used.

The part realized in this work was a fault-detection wrapper for POSIX message passing functionality. The infrastructure used by the function wrappers is set up by the new `mq_open` function. Four raw message queues are used instead of one and they are opened with explicit permissions. The new `mq_send` function packs puts redundant check data on the message and the new `mq_receive` function checks if that data corresponds to the message payload. If an error is recognized a signal is raised or an error code is returned.

7.1 Approach

Applications rely on functions provided by operating systems. Many operating systems implement POSIX, the Portable Operating Interface. The POSIX layer uses operating system functions, which themselves rely on hardware.

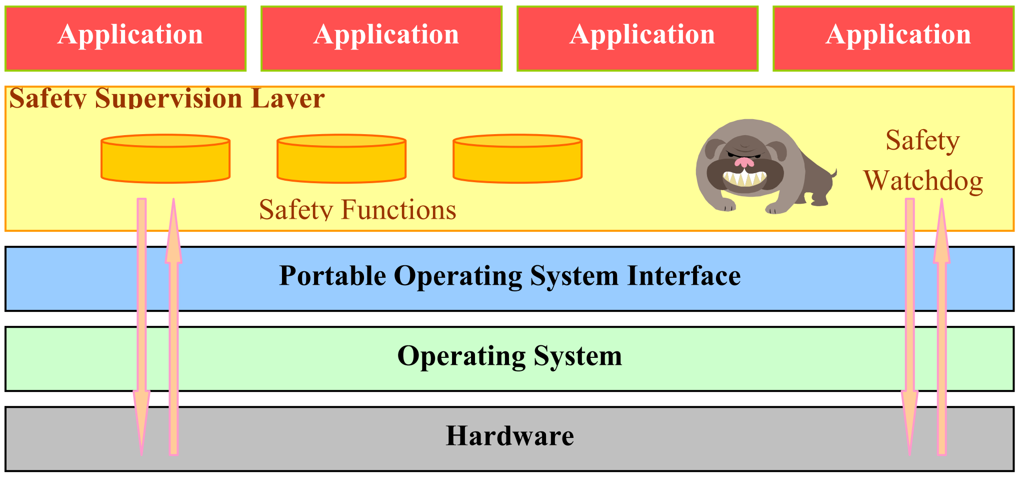


Figure 50: Solution schema.

Figure 50 shows a short schema of what is suggested in this work. The idea is to invent a layer between that POSIX and the application level. That Safety Supervision layer now uses the functions provided by the interface and checks if result for each system call is plausible. This is an end-to-end approach, redundant data is used to compare results with what is expected. There are faults that can not be detected directly, so a watchdog application is suggested to perform periodic checks.

7.1.1 Safe Message Passing

As an example for the safety functions a wrapper for POSIX message passing functionality is presented in this work. A logical queue now consists of four raw message queues every of which has got its own specific purpose. The safety requirements are taken from CENELEC 50159-1 [CEN50159-1] standard for safe messaging in closed communication systems. Based on the prescriptions in that standard the following features had to be implemented: sender and receiver identification, sequencing, checksums and periodic supervision of the communication channel. Wrappers provided for the usual functions `mq_open`, `mq_close`, `mq_send` and `mq_receive`.

`mq_open` establishes the basic messaging infrastructure, starts a `message_dispatcher` and initializes a `periodic_checker` timer function that checks the queues. `mq_send` takes the message intended to be sent and stamps redundant check data like sequence number and sender and receiver id and timing constraints on it and calculates a checksum over the message. The message together with the checksum is sent. `mq_receive` reads messages from the buffer the message dispatcher writes the received messages to. Synchronization is done using semaphores. `mq_receive` checks the received message and sends feedback. `mq_close` is always intended to raise a signal to indicate an error. Basically the Safe Message Passing Library provides 5 features:

- Checksums.
- Sender and receiver identification.
- Sequencing of the messages.
- Feedback infrastructure
- Periodic supervision of the communication channel.

Generally what is to be done when an error is recognized is to log the error and raise a signal. In the current version it's in the safety-relevant applications responsibility to take action. The Safe Message Passing layer has been tested using different scenarios.

7.1.2 Safety Watchdog

The Safety Watchdog is a framework for modular periodic tests. Those modules are intended to check specific system functions periodically in order to reveal faults before they can be activated. The watchdog itself is suggested to be a user level daemon that can be configured to run these tests, whilst actions to take and other functionality may be configured common configuration files. Beneath the modules it is intended to contain functions helpful for logging and maintenance.

The Watchdog remains a helping structure that solely is intended to check functionality that can not be tested directly by the Safe System Call Library. This is because the direct checking approach provided by the library provides decisive advantages compared periodic checks as explained in Chapter 7.2. The main disadvantage of the Watchdog is indeterminism and error detection latency (see Chapter 7.2).

Nevertheless there are advantages of such a structure situated in a layer right below the application. Good opportunities for saving costs and time-to-market are in sight, if it is possible to replace the hardware and system specific online tests by generic checks right above the operating system interface. The advantage of the end-to-end approach explained in Chapter 7.2 remains.

7.2 Conclusion

The elaborated structures are suitable to implement fault-detection in safety-critical systems. Generally they embody an approach to detect failures directly at the service interface, as the input is used to directly generate plausibility values for the generated output. This at least equals the current approach of validation and hardware tests. The Watchdog and Online Tests' goal is to detect faults before they are activated, which is a search for the needle in the haystack. The layer mainly checks for failures when its functions are executed, which yields systems that are more deterministic.

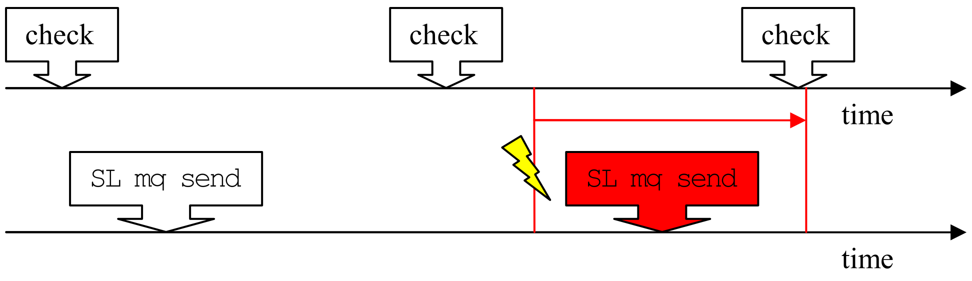


Figure 51: Layer functions are more deterministic.

Figure 51 shows that important difference between the online and the direct approach. Periodic checks just perform tests on certain points in time. The schema describes a fault, e.g. a permanent memory error emerging a short time after the last check. Hence in the first case the time span marked red may be dangerous as the error is not recognized. The line below shows the same for the Safety Supervision layer, where just little error detection latency is suffered.

An architecture based on a comparable structure to the proposed layer would yield some other advantages:

- **Portability**. In a mature version the layer will be a full substitute for tests that are currently written for specific operating systems and hardware. As is resides on top of POSIX it can be set up on top of this.
- **End-to-end testing**. The basis for the safety checks are the specifications of the functions the layer wraps. This means that concerns about interfaces between the lower level layers and their tests may be neglected, if the layer itself is written properly. Only checking the function's components' functionality just ensures that the components work properly, but not if the component works properly.

  Figure 52 visualizes the immense advantage of that approach. The schema describes a system intended to release cooling water into a vessel. This is initiated by a valve symbolized by the yellow triangle. The orange arrow symbolizes a component test checking if the valve is open or closed. The valve is erroneous, does not open, but indicates that it is opened. Such an error would then not be recognized. The green arrow shows a test checking the flow through the pipe, hence directly the effect or – in other words - what is intended by the specification. The Safe System Call Library maps this approach to Software Safety.

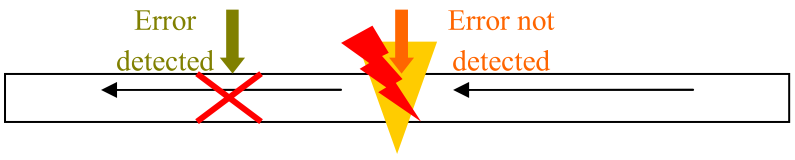


Figure 52 - End-to-end test advantage.

- **Direct result checking**. As shown above systems that use the Safety Supervision Layer do not suffer much error detection latency. This is only true for errors that are detected by the Safety functions, not for the functionality that is only supervised by the watchdog daemon.
- **Defensive system checking**. The watchdog function keeps on checking the system as a whole if requested. Therefore the periodic testing advantage, that faults may be detected before their activation, still remains.
- **Generic basis for operating system and hardware verification**. Apart from its mission in system operation the layer is a solid tool for benchmarking the layers below. System developers can take applications based on the layer and change the operating system and hardware below to see, on which it runs best.
- **Safe systems based on faulty infrastructure**. One initial target was the vision of just taking discount computers yielding errors and linking them to work together as a safe railway control system. As this would never be useful according system up-time and certification, it shows the desired direction. Good hardware is expensive, the price does not grow linearly, operating system verification is time and man-hour consuming as well.
- **Simpler application architectures**. The fact that systems using the layer can be assured that all the faults below are detected can extract complex fault detection and panic functions out of the application code. Developers can concentrate on the business logic, which helps them to prevent fault situated in the application layer. Nevertheless care has to be taken that the layer is seen as what it is, namely just the part in the puzzle that assures that all faults below the operating system are detected. The layer cannot detect faults in the layers above.

- **Fault Analyses for Operating Systems**. As described in the theoretical part on railway safety standards components not only consist of their code and hardware, but also on their documentation and specification, as well as their fault analysis. In a safety context their use can be justified with such analyses. If a wrapper containing a full fault analysis for an operating system below the POSIX layer is provided, the whole system may be considered as safely changeable.

An implementation of the suggested layer would certainly increase a product's time-to-market analogously to non-safety-critical component reuse. Software of Unknown Pedigree could potentially be used in considered railway system without harming safety. As the functions implemented by the layer influence a lot of common system calls, some disadvantages come along with it according a system's performance:

- **Restrictions to original functionality**. Because the wrapper functionality has to be justified by standards, the layer functions have to consider constraints introduced by them. As an example the Safe Message Passing Layer only supports 1:1 communication as explained in 5.
- **Structural overhead**. As one can clearly see in the message passing chapter process to process communication would be possible using one message queue. This has been changed to four to make use of several safety mechanisms and simplify the design. Overheads like this are often accepted as safety is more important that performance. Emerging from this structural overhead slower and more complex initialization mechanisms have to be taken into account as one can see in the SMP setup phase.
- **Communication overhead**. The exemplary Safe Message Passing layer shows the potential additional expenditure generated by the safety functions. To send a message with content "Hello", which is of size 5 bytes the message hash of 32 bytes, message type, sequence numbers and temporal constraints have to be added. For longer messages this is acceptable.
- **Calculation overhead**. Redundant data has to be generated to get plausibility criteria for the results of the functions to be checked. Besides that there are tests running permanently to check if system functionality is up.
- **No direct access to components**. Different from the shown online tests the watchdog does not directly check the system's inner hardware. This may be a problem for forensics searching for errors to increase uptime.

The reason why the fault detection functionality should not be implemented in the POSIX layer itself is that most systems are not safety critical and the performance loss described above cannot be accepted.

It has to be stated that such a safety supervision layer can not completely displace current verification and validation techniques. They are necessary to ensure system functionality and reliability. Nevertheless it can increase safety and be a valuable augmentation to current architectures and avoid indeterminism. Furthermore such *direct portable end-to-end operating system call checks* may save costs and time as they potentially replace non-portable mechanisms.

7.3 Outlook

This work considers itself as a starting point for the decomposition of safety-critical applications and the infrastructure they run on. The potential advantages of the possibility to change the underlying hardware and operating systems without harming safety are obvious. The following paragraphs propose improvements to the layer and fields of research that would bring benefits according the topic.

7.3.1 General Improvements to the Safety Supervision Layer

As described in Chapter 3.1, POSIX prescribes functions not only for message passing. Future versions of the layer should take concern of the other standards and implement wrappers for those other functions.

Functions suitable to be wrapped by the layer are for example file and directory operations, process creation and control, device drivers, shared memory, pipes, thread synchronization or floating point exceptions. The approach should be similar to the one in this work. First the desired functionality is selected, then there has to be researched what respective safety standards prescribe. Then suitable restrictions to the according functions have to be presented in order to match the standard's constraints. Fault Analysis methods have to be performed on the functions under consideration.

The resulting functions should overwrite existing POSIX functionality with the prescribed preloading technique to avoid programmers from unintentionally using the old ones. In the end the whole Portable Operating Interface should be wrapped by the layer.

7.3.2 Improvements to the Safe Message Passing Layer

The proposed Safe Message Passing Layer is a prototype that was never tested under operational conditions. Before its use in a productive environment certain improvements according some features and especially performance issues should be undertaken. One of them is the renaming of the functions to the name of the ones they wrap. This is described above as preloading, but has not been done in the current code to increase readability.

The first is that currently the layer just implements “blocking” send and receive. For architectural and fault detections concerns the layer is now intended to always perform raw `mq_send`s and `mq_receive`s in a blocking way. This means that the message dispatcher always waits until messages are in the queue. The `SL_mq_receive` is simulating the blocking case and waits until it gets the semaphore for reading from the buffer. Blocking `mq_send` means, that the sender waits, if the queue is full. The non-blocking case can be implemented with slight changes to the current functions. The `SL_mq_receive` would in this case just check if it gets the read semaphore immediately and return the value prescribed by the original `mq_open` specification if not. The non-blocking `SL_mq_send` would just check if the queue is currently full and perform analogously if not.

Another interesting issue is that currently there are 4 raw message queues used for the 1:1 communication. This is a structural overhead that simplifies fault detection. In productive versions of the layer one should look if it is possible to safely reduce that number and save resources. A clear log structure should be specified describing a format for timestamps, log messages, debug levels and file structures.

Currently the periodic checker timer handlers are performing the following way: When a logical queue is opened the message descriptors are written to an array in order to map the timer pointer to a message descriptor and back. Another way to hold these values are linked lists, which are slower in getting the mapped descriptors but less memory consuming. Apart from that the periodic checker initiated a series of changes in the basic conception that made the design much more complex. Those changes were not needed by other features. Hence it is desirable to find better ways to hold these structures.

7.3.3 The Watchdog

The watchdog should be implemented according to the specification given in that work. As stated above one should not try to reinvent the wheel but analyse and adopt the test cases shown by projects similar to [10]. The watchdog is a construct intended to replace hardware online tests. There should be research according how far this is possible. Another important point is that it should be evaluated if a structure like the watchdog is needed at all in later versions, as the approach is not deterministic as explained. Yet it is not clear if a direct fault detection approached could make the watchdog needless if the operating system interface is fully wrapped.

7.3.4 Future Fields of Research

During the elaboration of the Safety Supervision Layer a series of desirable fields for future research was recognized. The most interesting field of research is the question if it is possible to find a generic fault analysis for operating systems. As fault detection mechanisms are obviously based on fault analyses there would be the possibility to fully reveal ways to detect faults situated below the operating system's service level interface. Application developers could then be assured that all of the faults would be detected automatically and the fail-safe system could be shut down.

As the proposed layer solely checks for system call faults there would be the possibility to implement device and network component checks in that layer. This could be done by providing portable services that either wrap the device's functionality or perform periodic checks analogously to the proposed structure for the operating system.

Another interesting field is the question if there are ways analogously to the proposed to map the presented approach to support fail-operational systems. This would request more complex fault handling mechanisms and will require not only fault detection but also error correction techniques for operating systems.

References on scientific publications

[Aas07] Aas, Andreas. Use of COTS Software in Safety-Critical Systems, Norwegian University of Science and Technology, Essay in DT8100: Object-oriented systems, Spring 2007.

[Arl02] Arlat, Jean, Fabre, Jean-Charles, Rodríguez, Manuel, Salles, Frédéric. Dependability of COTS Microkernel-Based Systems. IEEE Transactions on Computers, Vol.51, No.2, February 2002.

[Avi01] Avižienis, Algirdas, Laprie, Jean-Claude, Randell, Brian. Fundamental Concepts of Dependability. Fundamental Concepts of Dependability, Research Report N01145, LAAS-CNRS, April 2001.

[CEN50159-1] CENELEC 50159-1. Standard for closed communication systems, 2007.

[DO-178B] DO-178B/ED-12B. Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc, 1992.

[EN50126] CENELEC EN50126. Railway applications – Specification and Demonstration of Reliability, Availability, Maintainability and 6060 Safety (RAMS), 1999.

[EN50128] CENELEC EN50128. Railway Applications - Communication, signalling and processing systems - Software for railway control and protection systems, 2001.

[EN50129] CENELEC EN50129. Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling, 2003.

[HSE01] HSE International Limited for UK Health and Safety Executive. HSE Report 336, ISBN: 0 7176 2010 7; Justifying the use of Software of Uncertain Pedigree (SOUP) in safety-related applications, 2001.

[HSE02] International Limited for UK Health and Safety Executive. Preliminary assessment of Linux for safety related systems, ISBN: 0717625383, August 14th, 2002.

[IEC61508] BS EN 61508. Functional Safety for Electrical/Electronic/Programmable Electronic Safety Related Systems, IEC, 2000.

[Kop06] Kopetz, Hermann. Slides for "real-time systems"-lecture 06, Chapter Fault-Tolerance Fundamentals, TU Wien, Real-Time Systems Group, 2006.

[Lev95] Leveson, Nancy. Safeware: System Safety and Computers. ISBN: 0-201-11972-2. Addison-Wesley,1995.

[Pul01] Pullum, Laura. Software fault tolerance techniques and implementation, Artech House computing library, ISBN 1-58053-137-7, 2001.

[Skram05] Skramstad, Torbjørn. Assessment of Safety Critical Systems with COTS Software and Software of uncertain pedigree, Proceedings of the First ERCIM Workshop on Software-Intensive Dependable Embedded Systems, Porto, Portugal, September 2005.

[Tam07] T. Tamandl, P. Preininger. Online Self Tests for Microcontrollers in Safety Related Systems, Proceedings of 2007 IEEE International Conference on Industrial Informatics INDIN07, pp. 137 – 142, 2007.

[Kan04] H. Kantz, N. Köning. TAS Control Platform: A Vital Computer Platform for Railway Applications, Alcatel España, 2004.

[Ye04a] Ye, Fan, Kelly, Tim. Criticality Analysis for COTS Software Components, In Proceedings of the 22nd International System Safety Conference (ISSC'04), Providence, Rhode Island, USA, System Safety Society, 2004.

[Ye04b] Ye, Fan, Kelly, Tim. COTS Product Selection for Safety-Critical Systems, In Proceedings of the 3rd International Conference on COTS-Based Software Systems (ICCBSS'04), R. Kazman and D. Port (Eds.), LNCS 2959, Redondo Beach, CA, USA, Springer, 2004.

[Yak99] Yakimovich, D., Bieman, J.M., & Basili V.R.. Software architecture classification for estimating the cost of COTS integration, Paper presented at the 21st international conference on Software engineering, Los Angeles, California, United States, 1999.

Internet references

[1] http://www.cenelec.org/Cenelec/About+CENELEC/default.htm, About CENELEC, retrieved on July 10th, 2007.

[2] http://standards.ieee.org/regauth/posix/, retrieved on July 5th, 2007.

[3] http://www.linuxjournal.com/article/9065, retrieved on August 1st, 2007.

[4] http://www.weibull.com/hotwire/issue21/hottopics21.htm, Reliability HotWire – The eMagazine for the Reliability Professional, retrieved on July 30th, 2007.

[5] http://en.wikipedia.org/wiki/Unix-like, retrieved on Aug 1st 2007.

[6] http://www.unix.org/what_is_unix/single_unix_specification.html, Single Unix Specification, retrieved on Aug 1st 2007.

[7] http://www.weibull.com/basics/fault-tree/index.htm , Fault Tree Analysis Basics, retrieved on Dec 9th 2007.

[8] http://www.ntnu.no/ross/srt/slides/fta.pdf, Fault Tree Analysis Introduction, retrieved on Dec 9th 2007.

[9] http://users.compaqnet.be/fmea/Links/Examples.htm, FMEA – Failure Modes and Effects Analysis, Examples, retrieved on Oct 15th 2007.

[10] http://ltp.sourceforge.net/, Linux Test Project, retrieved on Nov 11th 2007.

[11] http://ltp.sourceforge.net/tooltable.php, LTP Test Tool Matrix,
retrieved on Oct 22nd 2007.

[12] http://www.ubuntu.com, Ubuntu Linux, retrieved on Nov 2nd 2007.

[13] http://www.eclipse.org/cdt/, Eclipse C/C++ Development Tooling,
retrieved on Nov 2nd 2007.

[14] http://www.gnu.org/software/gdb/, The GNU Project Debugger,
retrieved on Nov 2nd 2007.

[15] http://www.stack.nl/~dimitri/doxygen/, Doxygen source code documentation generation tool, retrieved on Oct 20th 2007.

[16] http://lists.debian.org/debian-devel/1998/11/msg00222.html, Debian code size, retrieved on Nov 2nd 2007.

[17] http://people.debian.org/~jgb/debian-counting/counting-potatoes/, Counting Potatoes: The size of Debian 2.2, retrieved on Oct 9th 2007.

[18] http://libresoft.dat.escet.urjc.es/debian-counting/, Counting Debian, retrieved on Sept 10th 2007.

[19] http://www.engadget.com/2006/08/07/live-from-wwdc-2006-steve-jobs-keynote/, Steve Jobs keynote from WWDC, retrieved on Oct 24th 2007.

[20] http://www.knowing.net/PermaLink,guid,c4bdc793-bbcf-4fff-8167-3eb1f4f4ef99.aspx, *How Many Lines of Code in Windows?,* Knowing.NET, retrieved on Oct 18th 2007.

Appendix - CENELEC Member States

Figure 53 shows current CENELEC board member states (red) and affiliate countries adopting the standard.

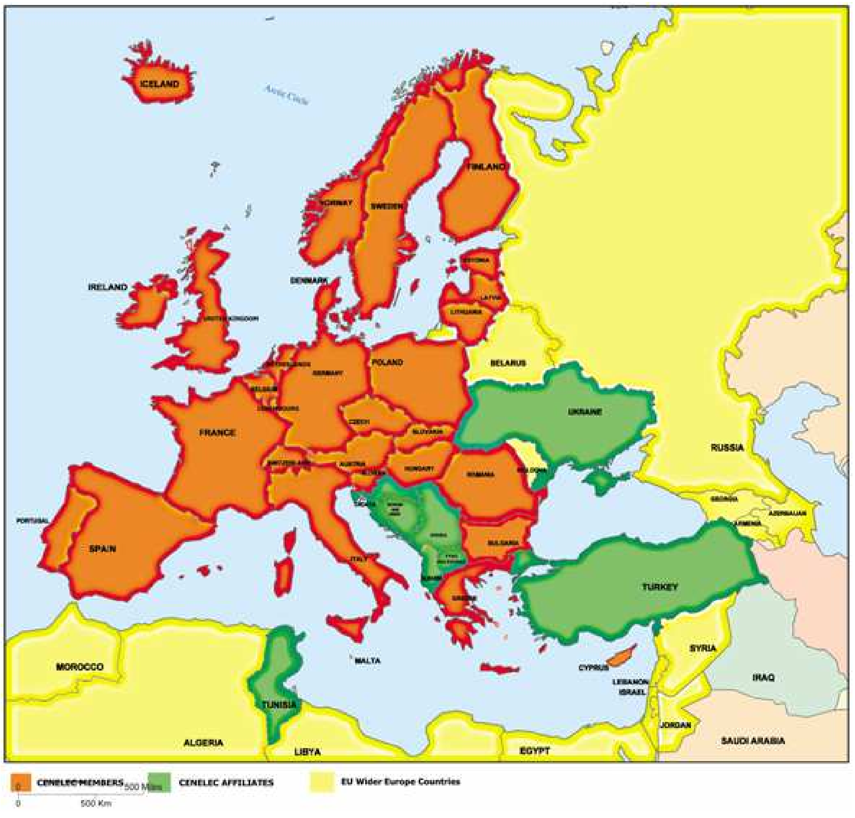


Figure 53: CENELEC member states and affiliates

Appendix - POSIX Message Passing

UNIX man pages provide specifications for the message passing functionality. Nevertheless some issues are not explained intuitively and suitable fort his work's context, hence they are explained below.

Data Structures

In POSIX Message Passing there are generally two important data structures, which are `mqd_t`, which holds the information about the message queue and `msg_msg` which basically represents a message. Table 7 shows the items of the `mqd_t` data structure.

Type	Field	Description
`struct list_head`	`m_list`	Pointers for message list
`long`	`m_type`	Message type
`int`	`m_ts`	Message text size
`struct msg_msgseg *`	`next`	Next portion of the message
`void *`	`security`	Pointer to a security data structure (used by SELinux)

Table 7: `mqd_t` data structure

An instance of the `mqd_t` data structure is returned when the message queue is opened or optionally created. Table 8 describes the properties of a message queue.

Type	Field	Description
`long`	`mq_flags`	Holds the set flags (`0` or `O_NONBLOCK`).
`long`	`mq_maxmsg`	Specifies the maximum number of messages in the queue.
`long`	`mq_msgsize`	Defines the maximum size of the messages in bytes.
`long`	`mq_curmsgs`	Gives the total number of messages currently in the queue.

Table 8: The mq_attr data type

Functions

The following functions describe the usual message passing functionality provided by the real-time libraries in `librt`. This description is basically taken from the according man pages and enriched by the author's own comments.

Opening Message Queue

```
mqd_t mq_open( const char *name, int  oflag );
mqd_t mq_open( const char *name, int  oflag,
               mode_t mode, struct mq_attr );
```

Connects to, and optionally creates, a named message queue. The queue is identified by a *name*. The operation is basically controlled by the `oflag` parameter, which holds exactly one of the following flags:

- **`O_RDONLY`**: Open the queue only for retrieving messages.
- **`O_WRONLY`**: Open the queue only to send messages.
- **`O_RDWR`**: Open the queue to send, as well as to receive messages.

The following flags can be ORed in the `oflag` argument:

- **`O_NONBLOCK`**: Open the queue in non-blocking mode. In situations where `mq_receive` of `mq_send` would block without this flag, these return immediately with the error `EAGAIN`.
- **`O_CREAT`**: Create the message queue if it does not already exist. The owner of the message queue is set to the effective user ID of the calling process. The group ID is set to the effective group ID of the calling process.
- **`O_EXCL`**: If `O_CREAT` was specified in `oflag`, and a queue with the given name already exists, then fail with the error `EEXIST`.

If **`O_CREAT`** is set, two more arguments must be passed. The mode parameter specifies the permissions to be placed on the new queue. These permission settings are masked against the process `umask`. The attr is set `NULL`, then the queue is created with implementation-defined default attributes.

There are several errors that can happen when opening the message queue:

- **`EACCESS`**: The queue exists, but the caller does not have permission to open in the mode that is specified
- **`EINVAL`**: Linux man pages give the following description to this error: `O_CREAT` was specified in `oflag`, and `attr` was not `NULL`, butt `attr->mq_maxmsg` or `attr->mq_msgsize` was invalid. Both of these fields must be greater than zero. In a

process that is unprivileged (does not have the CAP_SYS_RESOURCE capability), attr->mq_maxmsg must be less than or equal to the msg_max limit, and attr->mq_msgsize must be less than or equal to the msg_max limit. In addition, even in a privileged process, attr->maxmsg cannot exceed the HARD_MAX limit defined in limits.h.

- **EEXIST**: Both O_CREAT and O_EXCL were specified in oflag, but a queue with this name already exists.
- **EMFILE**: The process already has the maximum number of files and message queues open.
- **ENAMETOOLONG**: The name specified is too long.
- **ENFILE**: There is a limit on how many files and message queues can be opened the same time. If this limit is reached, this error is returned.
- **ENOENT**: This is returned, if the O_CREAT flag is not set and the message queue with the given name cannot be found.
- **ENOMEM**: Means that there is insufficient memory to open the queue.
- **ENOSPC**: This means that there is not enough space to create the message queue. This can probably occur, if the queues_max limit is encountered.

Closing Message Queue

```
mqd_t mq_close( mqd_t  mqdes );
```

If the calling process has attached a notification request to this message queue via mqdes, then this request is removed, and another process can now attach a notification request. This call may yield an EBADF error if the mqdes descriptor is invalid.

Unlinking a Message Queue

```
mqd_t mq_unlink( const char *name );
```

This call deletes the message queue defined by the name attribute *immediately*. The queue itself is destroyed once all other processes that have the queue open close their descriptors to it. If the operation is successful, mq_unlink returns 0, otherwise -1 setting errno to an error indicatior which may be EACCES, if the caller does not have permission to unlink the queue, ENAMETOOLONG, if the passed name was too long, or ENOENT, if there is no message queue with the given name.

Sending a Message

```
mqd_t mq_send( mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned int msg_prio );
```

This function adds the message referred by msg_ptr to the message queue defined by mqdes. If mqdes is invalid, the call returns the EBADF error. msg_len holds the size of the message payload specified by msg_ptr. This length has to be less than or equal than the maximal size of the message specified by the mq_msgsize attribute of the queue. If it's greater, the EMSGSIZE error is returned.

The msg_prio argument is a positive integer value that specifies the priority of the message. Messages are placed in the queue in a decreasing order of their priority, which means that higher priority messages are consumed earlier, while newer messages are appended after older messages of the same priority.

In certain situations it can be tried to put messages on a full queue, which usually means that there are more unconsumed messages in the queue than specified by its msg_maxmsg attribute. In this case two reactions are possible, depending on whether the queue's O_NONBLOCK flag is set or not. If it is not set, mq_send blocks until either it is allowed to write the message on the queue or it is interrupted by a signal handler. In the second case, an EINTR error is returned. If the flag is set, the call fails instead immediately returning the EAGAIN error.

```
mqd_t mq_timedsend( mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned int msg_prio,
        const struct timespec *abs_timeout);
```

The mq_timedsend functions behaves nearly the same way as the normal mq_send, except that it allows to send a timeout ceiling the time that the call blocks if the O_NONBLOCK flag is not set and mq_maxmsg is reached. The ceiling is an absolute timeout. If the message queue is full and the timeout is expired, the function returns immediately. If the timeout specified is invalid an EINVAL error is returned. If the call timed out before a message could be transferred, an ETIMEDOUT error is returned.

Receiving a Message

```
mqd_t mq_receive( mqd_t mqdes, char *msg_ptr,
                  size_t msg_len, unsigned int *msg_prio );
```

This call removes the oldest message with the highest priority from the queue and places it in the buffer space specified by msg_ptr. The paramenter msg_len passes the size of that buffer which has to be greater than the value specified in the mq_msgsize property of the queue, otherwise EMSGSIZE is indicated. If msg_prio is not null, the buffer is used to return the priority of the returned message.

mq_receive usually blocks until a message is received, unless the O_NONBLOCK flag of the message queue is specified. In this case the call fails immediately returning the error EAGAIN. There exists a derivate of mq_reveive named mq_timedreceive with an additional pointer of type const struct timespec that specifies that behaves analogously to mq_timedsend.

If the call is performed successfully, the functions return the number of bytes of the received message. If an error occurs, -1 is returned setting errno to indicate the error.

7.3.4.1 Getting Attributes

```
mqd_t mq_getattr( mqd_t mqdes, struct mq_attr *attr );
```

Writes an mq_attr data structure describing the message queue defined by mqdes in the buffer defined by attr. The call returns 0 if it is successful, respectively -1 on error. In the second case errno is set to EBANF, which means that mqdes is invalid.

7.3.4.2 Setting Attributes

```
mqd_t mq_setattr( mqd_t mqdes, const struct mq_attr *newattr,
                struct mq_attr *oldattr );
```

This function can only be used to change the value of mq_flags. The other values are ignored. If the oldattr parameter is not null, the old values, which would be given by mq_getattr(...) are passed to it. The call returns 0 if it is successful, respectively -1 on error. In the second case errno is set to EBANF, which means that mqdes is invalid, or EINVAL if newattr->flags contains set bits other than O_NONBLOCK.

7.3.4.3 Notifying

```
mqd_t mq_notify( mqd_t mqdes, const struct sigevent *notification );
```

This system call allows the calling process to register or unregister for the delivery of asynchronous notifications when new messages arrive on the empty message queue referred by mqdes.

The notification data structures and functionality was not part of my work, hence it is not further elaborated in this chapter.