VTT RESEARCH NOTES 2601

Timo Malm, Matti Vuori, Jari Rauhamäki, Timo Vepsäläinen, Johannes Koskinen, Jari Seppälä, Heikki Virtanen, Marita Hietikko & Mika Katara

Safety-critical software in machinery applications

VTT TIEDOTTEITA – RESEARCH NOTES 2601

Safety-critical software in machinery applications

Timo Malm, Matti Vuori, Jari Rauhamäki, Timo Vepsäläinen, Johannes Koskinen, Jari Seppälä, Heikki Virtanen, Marita Hietikko & Mika Katara

ISBN 978-951-38-7790-3 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0865 (URL: http://www.vtt.fi/publications/index.jsp)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 1000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 4374

VTT, Bergsmansvägen 5, PB 1000, 02044 VTT
tel. växel 020 722 111, fax 020 722 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O. Box 1000, FI-02044 VTT, Finland
phone internat. +358 20 722 111, fax +358 20 722 4374

Timo Malm, Matti Vuori, Jari Rauhamäki, Timo Vepsäläinen, Johannes Koskinen, Jari Seppälä, Heikki Virtanen, Marita Hietikko & Mika Katara. Safety-critical software in machinery applications. Espoo 2011. VTT Tiedotteita – Research Notes 2601. 111 p. + app. 10 p.

Keywords software, safety, safety-related, machinery, control system

Abstract

This report presents some important factors related to safety-critical software in machinery. The following subjects are considered in the text, bearing in mind the subject: the role of safety-critical software in machinery, statistics of software faults, requirements, safety and security principles, risk and hazard modelling, agile development, safety process patterns, safety-related architectures, verification and validation, phases of development and formal methods.

The general observation is that there are many methods for software design and it is difficult to choose the most relevant ones. The report shows some criteria for selecting methods and some aspects related to current topics. There are so many different safety-critical software applications in machinery that the research found the most interesting topics and then focused on them.

The statistics show that most defects arise during the requirements specification and architectural design phases of the lifecycle. This is before any coding. The statistics also show that the defect density is higher in large programs, i.e. the number of defects increases exponentially as the program size grows. It may therefore be better to separate safety-critical and standard code in order to keep the first one small. The separation of modules and keeping the connections between modules under control and narrow is recommended in order to have advantages in testing, understanding of the program, limited error spreading, program development etc. There are many kinds of self-diagnostic and monitoring functions that may be complex and increase the number of defects, but they increase safety and are needed in safety-critical code.

The standard IEC 61508-3, published in 2010, lays down many functional requirements for safety-critical programmable systems. However, there are also other standards related to functional safety and the safety of control systems.

This paper also considers some aspects related to the safety of agile methods. The standards show the requirements related to the phases of the V-model, but agile methods are not considered.

The functional safety of software is achieved through systematic (not intuitional) use of adequate methods in all phases of the programmable system lifecycle. Programmable systems contain hardware and software, both of which need to be considered in the validation process.

Preface

This publication was written to help machinery developers produce good safety-related software. Programmable systems and software in machines are evolving rapidly in many directions (e.g. towards large networks as well as small local systems) and we are therefore only able to give hints on specific important issues related to software safety. This document is constructed from separate articles related to specific topics. There are also short sections to complete and combine the topics.

The project was funded by Tekes (Finnish Funding Agency for Technology and Innovation) and realized by VTT and Tampere University of Technology (TUT). It was supported by a group of companies. The management and supporting group of the project consisted of Jari Räihä, Tekes; Juha Sipilä, Pekka Nykänen Pöyry/Tekes; Heikki Joensuu, Sami Salmi ABB; Matti Katajala, Safety Advisor; Kari Lehmusvaara, John Deere; Jouni Törnqvist, Bronto Skylift; Ari Lehtinen, Konecranes; Teemu Parkkinen, Sandvik; Tuula Mäntylä, Mika Maunumaa, Epec; Mika Karaila, Metso Automation; Matti Sundquist, Sundcon; Seppo Kuikka, TUT; Mika Katara, TUT; and Helena Kortelainen, Risto Tiusanen, VTT. Following researchers were working at the project and they had significant contribution in gathering material to the report: Jarmo Alanen, Kari Hakkarainen, Jere Jännes (VTT), Antti Jääskeläinen, Jani Paalijärvi and Mikko Salonen (TUT). Moreover, the fruitful discussions with project collaborators Prof. Shmuel Katz (Technion, Israel) and Tor Stålhane (NTNU, Norway) are gratefully acknowledged.

Tampere 12.10.2011

Authors

Contents

1. Introduction

Timo Malm and Marita Hietikko, VTT

1.1 The role of software in safety-critical systems

Companies have a growing need to prove safe functioning of software as software-based safety functions become more common. One reason for this trend is that often 'iron is replaced by brains' (robust systems vs monitored systems) because 'brains' are cheaper than 'iron'. In practice, a pipeline can be thinner if the pressure is well controlled or drive-by-wire functions can replace mechanical solutions. Many traditional safety functions, such as emergency stopping, are also realized with software-based systems.

One special feature of machinery software is that there are various systems with software. Safety-related software can be found in different parts of the system and often has to be isolated from standard software. Examples of programmable systems in machines include:

- safety programmable logic controllers (PLCs), safety buses, safety communication networks
- distributed systems with controllers, e.g. microcontrollers, PLCs, industrial PCs, field-programmable gate arrays (FPGA), application-specific integrated circuits (ASICS), etc.
- a PLC for control and supervision and a redundant hardwired system for specific functions like emergency stopping
- control systems with parameter programming, specific to automated machines, like machine tools, presses, paper machines, etc.
- machine control systems with an application software environment, specific to industrial robots and machine tools
- control systems for autonomous machines like robots with automated code generation from a photograph, CAD picture or automated guided vehicles (AGV) with automated tasks.

Safety-related software is increasingly found in safety PLCs, but in many cases safety depends on distributed systems and separate hardware. The variety of systems is quite large in cases in which risks are not high (SIL1 or PL a/b/c).

One typical safety function is stopping, and it is usually safe to stop a machine. The stopping function can be very dependable when proper redundancy is applied. There are some applications for which power must not be cut off, such as magnetic grippers, fans for removing flammable gas and controls for driving a moveable machine from a hazardous place. All in all, there are many kinds of safety functions, and there will be new safety functions as machines gain more autonomous functions and more responsibility is moved from humans to machines. Examples of fairly complex safety functions include stability of machines (e.g. mobile cranes and elevating work platforms), reduced speed (e.g. robots), restricted space (e.g. robots), anti-sway control (e.g. cranes), anti-collision systems of automated guided vehicles (e.g. harbour applications) and access control systems (e.g. automated mines).

Currently, the safety of a programmable control system is often validated according to functional safety standards: the IEC 61508 standard family [2010], IEC 62061 [2005] or SFS-EN ISO 13849-1 [2008]. The basic idea of the standards is to guide the designer to perform safety-related actions at each phase of the design, i.e. not to concentrate on laborious and often inefficient testing at the end of the development. The standard IEC 61508 [2010] presents more than 80 methods for programming and testing software. There are variations in the methods, and they can also be used to different extents. Cost-effective software and new developing methods such as open source software, COTS, re-used software and agile methods bring new challenges for safety.

1.2 Threats concerning software

Program faults can occur in many different phases of the program lifecycle. It is often difficult to say in exactly which phase the fault has occurred. The requirements could have been more specific or the coding should have solved the problem, and, finally, testing should have found the problem. It is possible to obtain a rough estimate of when the faults will occur however. Here, the considered software lifecycle phases are a requirements specification, design (architecture, module design), coding, documentation and modification. Testing and validation are not considered a source of faults. If they fail, a fault can be neglected, but new faults are not created.

Faults related to the requirements specification are usually missing requirements, but mistakes are made. The faults made in this early phase are difficult to find because, in the validation process, the validation tasks (for example, tests and reviews) are performed against the requirements specification.

Faults related to the design phase can be difficult and expensive to repair as the faults may require changes to the architecture, and a large part of the software may need to be rewritten.

Coding faults are the most common, but the programmer usually detects these faults quickly and repairs them. In the testing phase, the faults can also be revealed easily as the code is tested against the requirements. The faults that are not revealed are often those that occur seldom and in special situations.

The following figures are interpreted from Capers Jones's publication 'Software quality in 2008: A survey of the state of the art' [Jones 2008]. The research contains about 75 projects/month in 24 countries.

Figure 1 describes when faults are made in excellent (best 10%), average and poor code. The figure shows the absolute values of delivered faults (defect/function point). The average value of software defects in published programs is 0.75 defects/function point (d/fp). This is close to 5 defects per 1,000 source lines of code, but it depends on the code style [Rollo, 2010]. It is difficult to measure the number of defects and the values are therefore estimates.

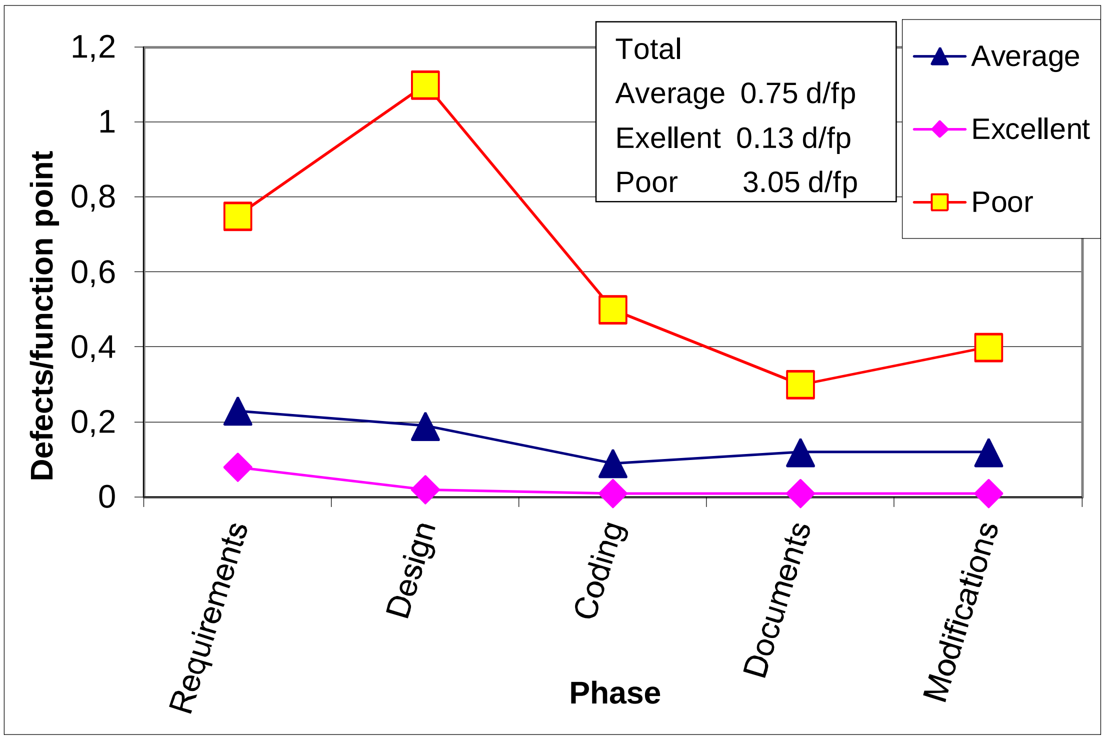


Figure 1. Defects per function point of excellent, average and poor software according to the origins of the faults [Jones 2008]

Figure 2 describes how faults are distributed between different phases of the software lifecycle. The dashed line shows the proportion of potential defects, which means all defects found during the development process and after delivery. The continuous line shows the proportion of defects found after delivery. The average removal efficiency is about 85% [Jones 2010], which means that there are more than six times more potential defects than delivered defects. In the figure, only the proportional values are shown, not the absolute values. If the value of the delivered defects is higher than of the potential defects, the defects appear worse than in the other phases of the development. The reason could be, for example, the difficulty of the phase, a lack of easy defects (revealing easy defects makes the numbers look better), resources, good tools or expertise. The reason for the proportional difference between the potential and delivered defects in each phase of the development can differ.

It has been shown that most of the faults in excellent software occur in the requirements specification phase. This fact has also been noticed by Nancy Leveson who mentions in her text that in aeronautic software (type: excellent software) almost all defects originate from the requirements specification [Leveson 2004]. This means that even if we apply many good methods to control the faults, we may still have faults related to the safety requirements specification. Faults in the safety requirements specification are often related to new technology or unknown risks, but there are also other types of faults and missing requirements. In average and poor software, more than half of the faults clearly originate from the requirements specification or architectural design phase. Many faults occur in the software architecture phase, especially in poor software. In excellent software, fewer faults originate in the coding phase than in average software. This can be interpreted as: if many good methods in the coding and testing phase are applied, the coding faults can be kept well under control. The figure shows that it is not possible to find all faults before delivery.

The size of the program affects the defect rates. The average defect removal efficiency of a small program is 95% and of a large program (>100000 fp) about 75%. The delivered defects per function point vary according to the size, from 0.09 to 2.39 [Jones 2010]. The number of defects grows exponentially with the size of the program.

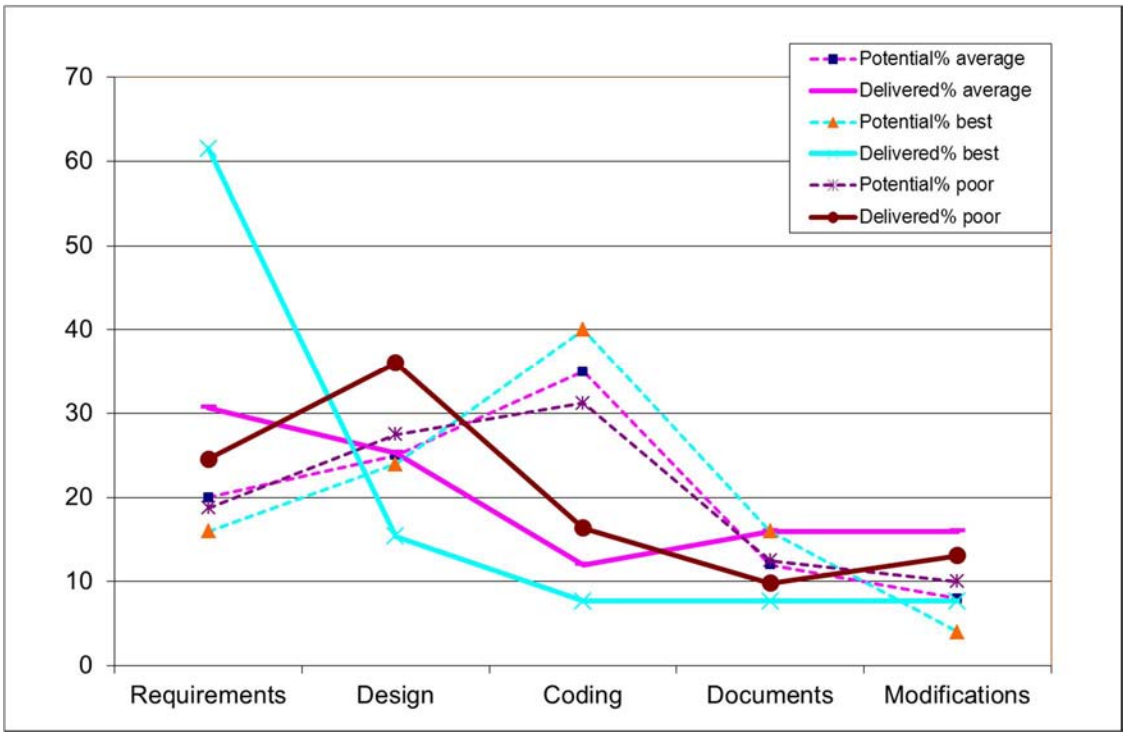


Figure 2. Percentage of excellent, average and poor software according to the origins of faults [Jones 2008].

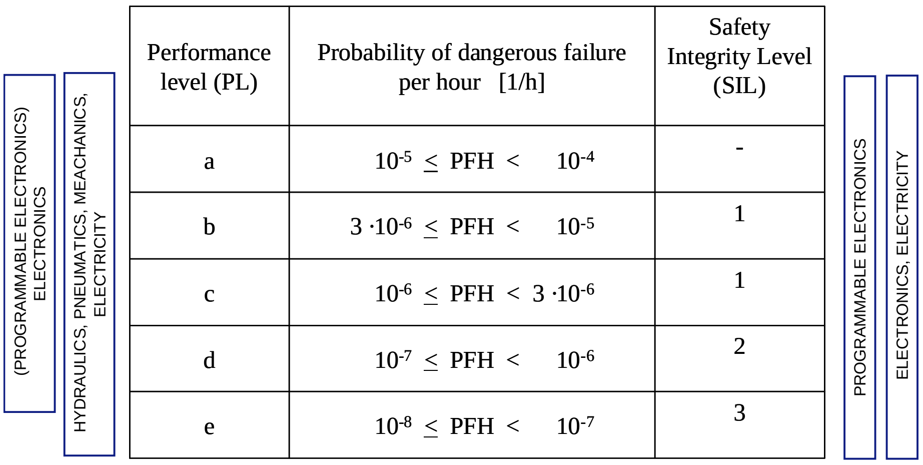
1.3 Requirements for safety-related software

There are two harmonized standards related to functional safety of control systems in machinery: SFS-EN ISO 13849-1 [2008] and EN 62061 [2005]. SFS-EN 62061 [2005] is related to programmable control systems and SFS-EN ISO 13849-1 is related to parts of control systems. It is possible to use one standard for one part of the system and the other standard for other parts of the system. SFS-EN ISO 13849-1 [2008] is simpler and has more precise requirements. The SFS-EN 62061 [2005] standard refers to the safety lifecycle model. Both standards have a quantitative approach (probability of dangerous failure), though software is mainly dealt with by qualitative means. Both standards also refer to IEC 61508 [2010], concerning when complex embedded software is considered.

SFS-EN 62061 [2005] is intended for use by machinery designers, control system manufacturers and integrators (e.g. logic manufacturers), and others involved in the specification, design and validation of a safety related electrical control system. It sets out an approach and requirements to achieve the necessary performance.

1.3.1 SFS-EN ISO 13849-1

EN ISO 13849-1 [2008] is a harmonized standard that describes safety issues related to pneumatics, hydraulics, electrical systems, electronic systems and programmable systems. Several ISO C-type machinery standards refer to EN ISO 13849-1. The safety systems are divided into performance levels (PL), which have their equivalents in safety integrity levels (SIL). Figure 4 shows the principle of how performance level requirements are derived from risks and from the required risk reduction. A risk graph is shown on the left side of Figure 4. It can be used to determine the required PL, marked with PL_r. The system is realized according to the requirements related to the specific performance level. The relation between PL and SIL is presented in Figure 3.


(PROGRAMMABLE ELECTRONICS) ELECTRONICS
HYDRAULICS, PNEUMATICS, MEACHANICS, ELECTRICITY
PROGRAMMABLE ELECTRONICS
ELECTRONICS, ELECTRICITY

Performance level (PL)	Probability of dangerous failure per hour [1/h]	Safety Integrity Level (SIL)
a	$10^{-5} \leq \text{PFH} < 10^{-4}$	-
b	$3 \cdot 10^{-6} \leq \text{PFH} < 10^{-5}$	1
c	$10^{-6} \leq \text{PFH} < 3 \cdot 10^{-6}$	1
d	$10^{-7} \leq \text{PFH} < 10^{-6}$	2
e	$10^{-8} \leq \text{PFH} < 10^{-7}$	3

Figure 3. Relations and applicability of SFS-EN ISO 13849-1 [2008] and IEC 62061 [2005].

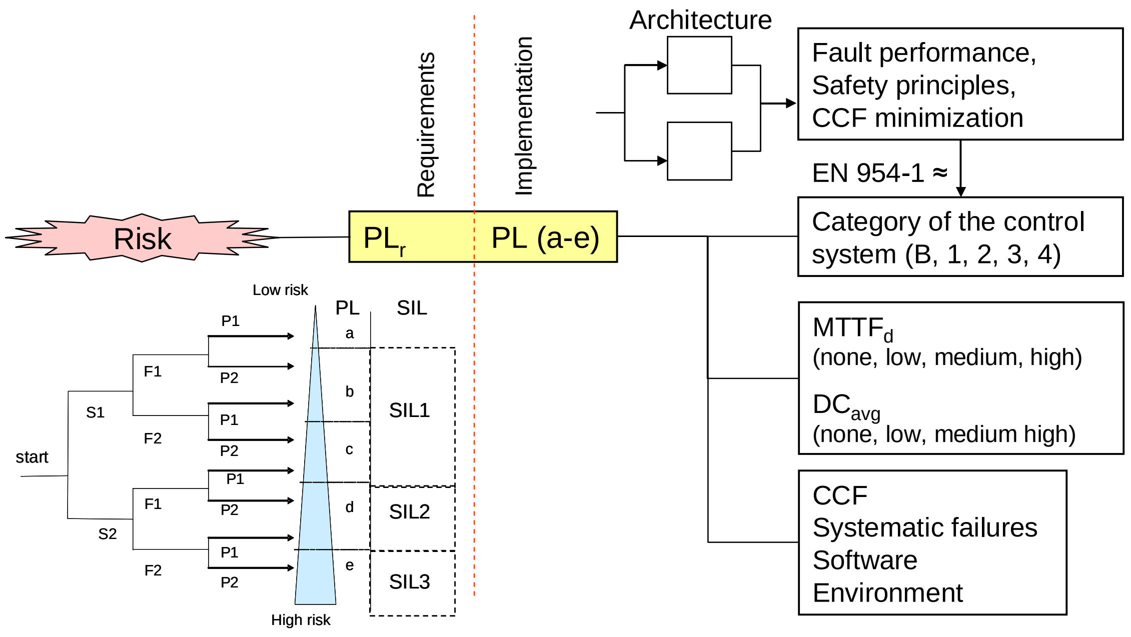

Architecture
Requirements
Implementation
Risk
PL_r
PL (a-e)
Fault performance, Safety principles, CCF minimization
EN 954-1 ≈
Category of the control system (B, 1, 2, 3, 4)
$MTTF_d$ (none, low, medium, high)
DC_{avg} (none, low, medium high)
CCF
Systematic failures
Software
Environment
Low risk
High risk
start
S1
S2
F1
F2
P1
P2
PL
SIL
a
b
c
d
e
SIL1
SIL2
SIL3

Figure 4. Requirements are derived from risks, and the necessary risk reduction and system are realized according to the requirements.

SFS-EN ISO 13849-1 [2008] sets requirements for both safety-related embedded software and safety-related application software. With regard to safety-related embedded software, the following basic issues are emphasized: verification and validation activities (V-model) as well as appropriate activities after modifications during the software safety lifecycle, documentation of specification and design, modular and structured design and coding, control of systematic failures, functional testing, and verification of correct implementation using software-based measures for control of random hardware failures. These are the basic measures to be applied to software (SW) components with PL_r levels from ‘a’ to ‘d’. Some additional requirements are given for PL_r levels from ‘c’ to ‘e’. The requirements relate to the project and quality management system, documentation, configuration management, structured specification, use of suitable programming languages and tools, modular and structured programming, separation into non-safety-related software, module size limits and definition of interfaces, use of design and coding standards, code review practices, extended functional testing, and impact analysis with activities after modifications. In addition to these, some requirements of IEC 61508-3 are applied to PL_r level ‘e’.

The same basic activities are applied to the safety-related application software. More detailed recommendations are given for different phases of the SW lifecycle however.

1.3.2 SFS-EN 62061

SFS-EN 62061 [2005] is derived from standard IEC 61508 [2005] to fit applications in the machinery area. It sets guidelines for different phases of the safety lifecycle and specifies requirements for the design and implementation of programmable safety-related control systems of machinery. This standard is applicable to all architectures and up to SIL3, whereas SFS-EN ISO 13849-1 is restricted to designated architectures: 1oo1 (one out of one) and 1oo2 (one out of two). The safety requirements specification comprises the functional safety requirements specification and safety integrity requirements specification. SFS-EN 62061 [2005] lays down requirements and a methodology to assign the required safety integrity level to each safety-related control function that is to be implemented by a safety-related electrical control system (SRECS). It also lays down requirements and methodology to enable the design of an SRECS that is appropriate for the assigned safety-related control function(s), integrate safety-related subsystems designed in accordance with SFS-EN ISO 13849 [2008] and validate the SRECS.

Section 6.10 of this standard sets requirements and measures to apply safety to software, starting from a software safety requirements specification and ending with a safety-related electrical control system integration, testing and installation phases. In the chapter discussing embedded software design and development, there is a reference to IEC 61508-3 [2010]. Annex C of SFS-EN 62061 [2005] provides a guide to embedded software design and development. The requirements relating to application software design and development are based on IEC 61508 [2010], but the requirements of different software lifecycle phases are also presented in SFS-EN 62061 [2005]. In addition to general requirements, it provides software configuration management and requirements for software architecture, support tools, user manual and application languages as well as application software design, code development, module testing and software integration testing.

1.3.3 IEC 61508

The IEC 61508 [Functional safety of electrical/electronic/programmable electronic safety-related systems. Parts 1-7, 2010] standard describes the safety lifecycle process and requirements related to each phase of the process. The standard is related to several application sectors (not only machinery) and is therefore quite broad. The standard is not harmonized and thus does not have the same status as SFS-EN ISO 13849-1 [2008] and SFS-EN 62061 [2005]. These harmonized standards refer to IEC 61508 [2010], however, especially with respect to embedded systems. A new version 2.0 of IEC 61508 was published in spring 2010.

IEC 61508-3 (Software requirements) [2010] describes the requirements for software, covering the different software lifecycle phases from requirements specification to the use and modification of software. Recommendations for the use of techniques and measures on different safety integrity levels (SIL) are given for each lifecycle phase in the annexes of IEC 61508-3 [2010]. Part 7 of IEC 61508 gives short descriptions for the techniques and measures mentioned in Part 3. More detailed descriptions of the techniques and methods can be found from the references in Part 7.

References

Chaos. 1995. The Standish Group Report. 8 p.

Hanmer, R. 2007. Patterns for Fault Tolerant Software, Wiley Publishing.

IEC 61508 ed2. Parts 1–7. 2010. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. International Electrotechnical Commission. 1000 p.

IEC 62061. 2005. Safety of machinery – Functional safety of safety-related electrical, electronic and programmable electronic control systems. International Electrotechnical Commission. 205 p.

Jones, C. 2008. Software quality in 2008: A survey of the state of the art. Software Productivity Research LLC. Available at: http://www.jasst.jp/archives/jasst08e/pdf/A1.pdf [achieved 3.2.2010]. 59 p.

Jones, C. 2010. Software quality in 2010. A survey of the state of the art. Software Productivity Research LLC. Available at: http://www.sqgne.org/presentations/2010-11/Jones-Nov-2010.pdf 31 p.

Leveson, N.G. 2004. The Role of Software in Spacecraft Accidents. Aeronautics and Astronautics Department Massachusetts Institute of Technology. Available at: http://sunnyday.mit.edu/papers/jsr.pdf [achieved 4.2.2010]. 27 p.

Rollo, A. 2006. Functional Size measurement and COCOMO – A synergistic approach. In Proc. of Software Measurement European Forum 2006. Software Measurement Services Ltd. pp. 259-267. Available at: http://www.compaid.com/caiinternet/ezine/rollo-cocomo.pdf [achieved 3.2.2010]. 11 p.

SFS-EN 62061. 2005. Safety of machinery – Functional safety of safety-related electrical, electric and programmable electronic control systems. Finnish Standards Association SFS. 198 p.

SFS-EN ISO 13849-1. 2008. Safety of machinery – Safety-related parts of control systems – Part 1: General principles for design. Finnish Standards Association SFS. 180 p.

2. Functional safety and security as elements of overall safety

Timo Vepsäläinen and Jari Seppälä, Tampere University of Technology, Department of Automation Science and Engineering

2.1 Introduction

Safety and security are essential aspects of critical control systems in any application domain. The development of both should be based on perceived risks and hazards, the identification of which is therefore an essential development activity. The main reason for this is the ability to target development work and efforts to the right parts and functions of the system so that the desired levels of safety and security can be achieved with minimal effort. In the project, the main focus of the second work package was on modelling safety-related systems and applications including the source information for their development as well as to support safety-critical system development with model-based techniques.

The idea of focusing on models in the development of systems and software applications has recently been the topic of numerous publications in several domains. Due to the interests and publications, there are also several acronyms related to the concept, such as Model-Based Development (MBD), Model-Based Systems Engineering (MBSE), Model-Driven Architecture (MDA) and Model-Driven Engineering (MDE). The idea of model-based development and related approaches is to use models as primary engineering artefacts during the development of applications instead of, for example, documents. The key promise of the approaches is the ability to automate part of the development work with model transformations that use models and specifications in the production of revised models and executables. In addition, formal or semi-formal modelling may enable the models to be analysed in order to reveal problems or error-prone aspects in design. From the point of view of safety standards, however, the application of model-based techniques may have been rare and difficult until recently. Nevertheless, the new, second edition of IEC 61508 [2010] states that automatic software generation may aid completeness, correctness and freedom from intrinsic design faults in architecture design. Consequently, the issue of how to develop safety-related systems with model-based techniques is both important and current.

In model-based development of safety-critical applications, the applications should thus be developed using models but they should also consider the requirements of safety standards. Consequently, our starting points in the project were safety and security standards and their general requirements, which are also discussed in more detail in the report of the second work package produced during the project. However, currently used modelling languages in software development, e.g. UML, hardly address all the information content and documentation needs related to requirements; the source information for application development, i.e. the risks and hazards related to the system in question; and the traceability between the development artefacts. There are reported approaches to cover structured presentations of hazards, requirements and traceability however, which could be integrated into model-based development. Parts of these techniques and notations will be discussed in Section 2.2. In Section 2.3, we do the same related to security and possible modelling notations and techniques that could contribute to achieving security. In Section 2.4, we present part of the collected modelling notations and how they can be integrated into the existing UML Automation Profile and used in transformation-assisted creation models of control systems and simulation models of controlled systems.

2.2 Safety

IEC 61508 [2010] is an international standard covering the aspects to be considered when electrical/electronic/programmable electronic systems (E/E/PES) are used to carry out safety functions. The standard consists of seven parts that focus on different aspects of the development of safety-related systems, including, for example, general and software requirements. The current version of the standard was published in 2010. IEC 61508 [2010] is of special importance as a functional safety standard for several reasons. Firstly, the standard was recently renewed. Consequently, for example, the list of recommended actions and techniques should be as modern as is applicable. Secondly, one of the purposes of IEC 61508 is to facilitate the development of industry-specific standards, which increases its importance. Finally, according to our interviews during the project, IEC 61508 may be the most difficult safety standard used as a basis for the development. This is because of the number of techniques to be used in the development as well as the difficulty of (parts of) them.

The overall safety lifecycle introduced by IEC 61508 [2010] Part 1 is presented in Figure 5. The overall safety lifecycle consists of 16 phases that cover the phases from concept and overall scope definition to decommissioning and disposal of the system. The realization phase of the E/E/PE safety-related systems (10th phase of the lifecycle model in the figure) can be further decomposed into smaller activities and phases. The phases of the software safety lifecycle include a software safety requirements specification in terms of safety function and safety integrity specifications, software safety validation planning, software design and development, PE integration (hardware/software), software operation as well as maintenance procedures, planning and software safety validation.

For the different phases of software development and design, the standard recommends several measures and techniques. In addition to techniques, the standard defines properties of systematic integrity, the achievement of which is promoted by the recommended measures and techniques. The properties for systematic integrity that the standard defines for the safety requirements specification phase include: 1) completeness with respect to the safety needs addressed by software, 2) correctness with respect to the safety needs addressed by software, 3) freedom from intrinsic specification faults, including freedom from ambiguity, 4) understandability of safety requirements, 5) freedom from adverse interference of non-safety functions with the safety needs to be addressed by software, and 6) the capability of providing a basis for verification and validation [IEC 61508 2010]. These properties were also used as a target for the modelling approach to be developed in the work package. In general, we try to integrate modelling notations from several languages, modelling profiles and standards in order to facilitate the development of safety-related software applications with model-based techniques.

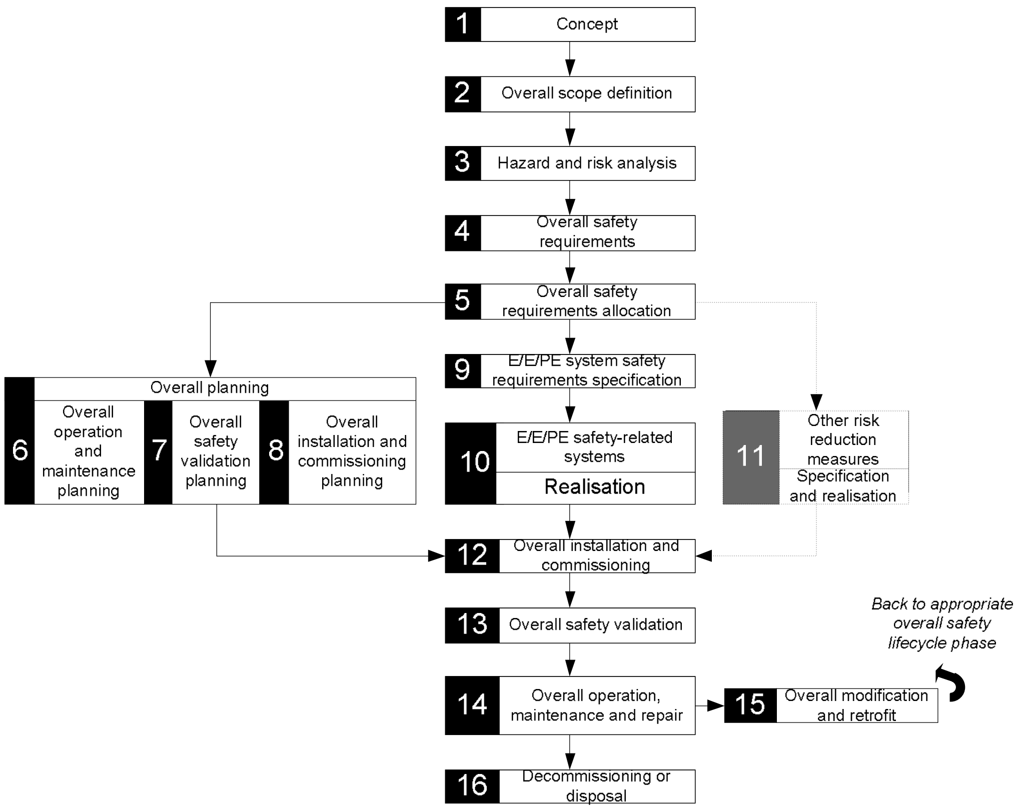

1 Concept
2 Overall scope definition
3 Hazard and risk analysis
4 Overall safety requirements
5 Overall safety requirements allocation
9 E/E/PE system safety requirements specification
Overall planning
6 Overall operation and maintenance planning
7 Overall safety validation planning
8 Overall installation and commissioning planning
10 E/E/PE safety-related systems
Realisation
11 Other risk reduction measures
Specification and realisation
12 Overall installation and commissioning
13 Overall safety validation
14 Overall operation, maintenance and repair
15 Overall modification and retrofit
Back to appropriate overall safety lifecycle phase
16 Decommissioning or disposal

Figure 5. Overall safety lifecycle of IEC 61508-1 (modified from IEC 61508 [2010]).

2.2.1 Risk and hazard modelling

Hazard and risk analysis constitute an essential part of the development of safety-critical systems. For example, in the overall lifecycle of the IEC 61508 [2010] standard, the risk and hazard analysis constitute the third lifecycle phase, the intention of which is to determine the hazards and hazardous events of the EUC and the EUC control system, the event sequences leading to the hazardous events and the EUC risks associated with the hazardous events [IEC 61508 2010].

In addition to using risks and hazards as a basis for the safety requirements specification, however, modelling risks can aid various purposes in the development of safety-related and safety-critical applications. According to OMG [OMG 2008a], the motivation of the risk metamodel and modelling could be the practical use of UML to support risk management in general and risk assessment in particular. In model-based risk assessment, UML models are used for three purposes: to describe the aim of the assessment on the right level of abstraction, to facilitate communication and interaction between different groups of stakeholders involved in the risk assessment and to document risk assessment results and the assumptions on which the results depend to support reuse and maintenance [OMG 2008a].

In addition, documenting hazards and risks associated with the hazards in the same models with the requirements and the design of the software could aid various other purposes. These purposes include, at least, traceability between the identified hazards and risks and the corresponding safety function and safety integrity requirements, understandability of the requirements and their mechanisms to reduce or remove risks and identification of the most critical functions. The developers of the system could always follow the requirements to the hazard model when in doubt and, the developers and implementers of safety functions could, for example, be instructed to always check the consistency between the artefacts before writing a single line of specification or code. By doing so, the consistency would be checked several times during the development by different people, including developers of safety requirements, implementers (programmers) and testers, to name a few.

Unfortunately, modelling of hazards and risks is currently not covered by many modelling languages or profiles used in software engineering. However, there are reported approaches to cover structured presentations of hazards such as the safety analysis profile by Douglass [Douglass 2009] and the approach of the UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (QoSFT) [OMG 2008a].

2.2.1.1 UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification

The UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms is a UML profile specified by OMG and intended to extend UML to cover the quality of service (QoS) and fault tolerance (FT) concepts. The extensions are defined in two general frameworks: the QoS modelling framework and the FT modelling framework. The QoS modelling framework describes the vocabulary to be used with high quality technologies and provides the ability to associate the requirements and properties with model elements in order to introduce extra-functional aspects in UML models. The framework for fault tolerance includes notations for model risk assessments, paying special attention to the description of hazards, risks and risk treatments. This framework also supports the description of fault-tolerant architectures based on object replications [OMG 2008a]. From the point of view of the project and the work package, the framework of interests is the fault tolerance framework. Figure 6 presents the submodels (packages) of the framework and their dependencies.

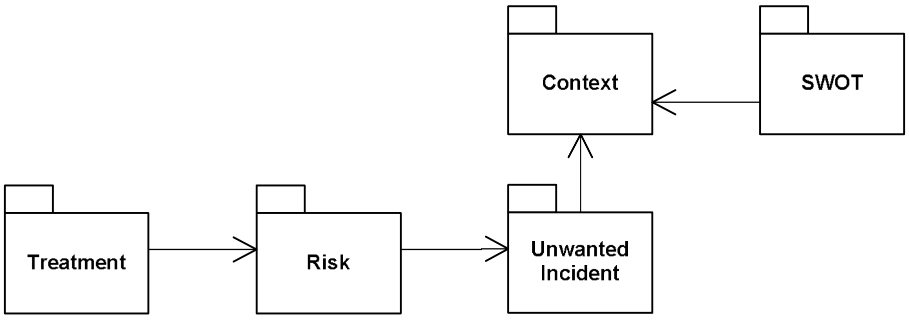


Figure 6. Submodels of the risk assessment framework.

In the profile, the main objective of the modelling may not be in the detailed specification of how the hazards may occur. Instead, it appears that the profile focuses more on factors determining the magnitudes of risks (likelihood, consequences), compromised assets, stakeholders and the treatment of risks. Treatment approaches include avoiding risk, reducing likelihood or consequences of risk, and retaining and transferring risks. The tracing of risks to requirements, however, is not covered by the profile.

To the authors, the aim of the risk assessment framework appears to be in documenting the risk analysis and possibly computer-aided analysis of the results of the assessment based on models completed with the concepts (stereotypes) of the framework. The aim of enabling computer-aided analysis of results justifies, for example, the detailed specification of values of risks, assets and treatments with the value definition concept. To the authors of the paper, however, the choice (approach) to focus on (possibly quantitatively) the analysis of results appears odd as there are dedicated tools and frameworks to aid risk analysis and assessment, whereas UML tools are, in general, more suitable for the development and construction of systems.

2.2.1.2 Safety analysis profile

According to Douglass [Douglass 2009], the objective of the safety analysis profile is to enable requirement capture and safety analysis with Telelogic® Rhapsody® in UML models. The use of UML can aid the development in a number of ways including providing design clarity, modelling architectural and low-level redundancy, creating safety-relevant views on requirements and design, and aiding safety analysis. UML can also be used for the specification of traceability between requirements and design. Views, on the other hand, can be supported with separate diagrams focusing on narrow aspects of the system under development with the underlying model still being consistent. One of the safety analysis approaches supported by the profile is the fault tree analysis (FTA) [Douglass 2009].

FTA is a commonly used graphical and analytic approach to the identification of risks and hazards associated with systems. In FTA, conditions and faults leading up to hazards can be analysed logically for cause-effect relations using commonly known logical operations such as AND, OR, XOR and NOT, and they can be also used in a quantitative way. Once the relations have been identified, safety measures can be identified and added to the analysis model. For example, the profile suggests constructing the safety functions so that, to arrive at hazardous conditions, the original faults would need to be ANDed with the faults of the safety measures [Douglass 2009]. In other words, to arrive at the hazardous condition, the safety function would also need to fail.

Figure 7 presents the core parts of the safety analysis profile metamodel defined by Douglass [2009]. The Hazard element in the metamodel represents a condition that will lead to an accident or loss and is usually a top terminal in an FTA diagram. Faults represent non-conformance of elements – physical or software – to specifications or expectations. The resulting conditions result from logical operations of faults and conditions. The required conditions represent conditions required for the faults to interact. LogicFlows connect the hazards, faults and conditions to logical operations. FaultSources and SafetyMeasures are normal UML elements that may be linked to the FTA model. FaultSources can manifest the fault or be the sources of the faults. SafetyMeasures represent the capability to detect, extenuate or mitigate faults. TraceToRequirement elements can be used to trace faults or hazards to Requirements. In addition to the elements, the profile defines several matrix and table presentations that are intended for summarizing faults, hazards and their associations with other model elements [Douglass 2009].

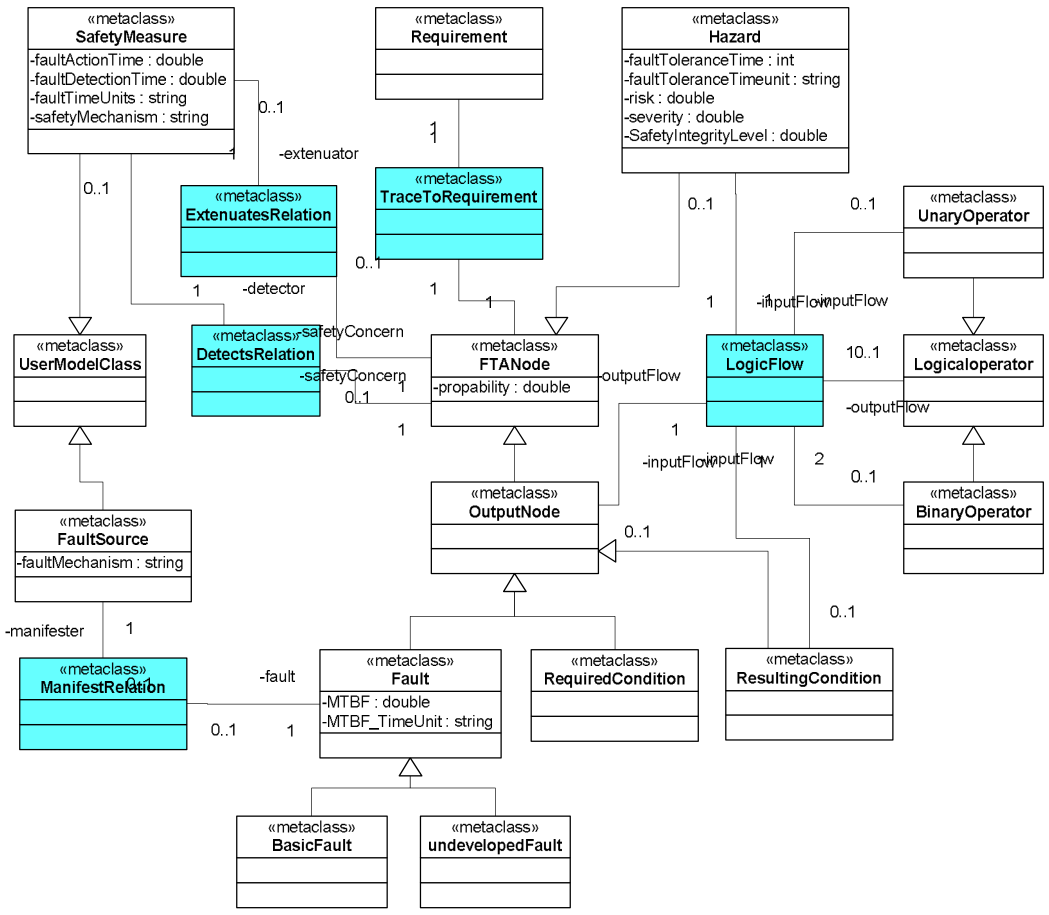

«metaclass» SafetyMeasure
-faultActionTime : double
-faultDetectionTime : double
-faultTimeUnits : string
-safetyMechanism : string
«metaclass» Requirement
«metaclass» Hazard
-faultToleranceTime : int
-faultToleranceTimeunit : string
-risk : double
-severity : double
-SafetyIntegrityLevel : double
«metaclass» ExtenuatesRelation
«metaclass» TraceToRequirement
«metaclass» UnaryOperator
«metaclass» UserModelClass
«metaclass» DetectsRelation
«metaclass» FTANode
-propability : double
«metaclass» LogicFlow
«metaclass» Logicaloperator
«metaclass» FaultSource
-faultMechanism : string
«metaclass» OutputNode
«metaclass» BinaryOperator
«metaclass» ManifestRelation
«metaclass» Fault
-MTBF : double
-MTBF_TimeUnit : string
«metaclass» RequiredCondition
«metaclass» ResultingCondition
«metaclass» BasicFault
«metaclass» undevelopedFault
-extenuator
-detector
-safetyConcern
-outputFlow
-inputFlow
-manifester
-fault
0..1
1
2
10..1

Figure 7. The core parts of the safety analysis profile metamodel (modified from Douglass [2009]).

The safety analysis profile thus provides several notable features. Firstly, the profile allows risks and hazards to be linked to requirements of safety functions (or other kinds of requirements) that can be specified within the same models. In conjunction with the requirements specification and modelling features of, for example, SysML, such models could include a great portion of the traceability information required by safety standards (such as IEC 61508 [2010]) related to development of safety-critical systems. Such information could also be exported from the models in the form of different kinds of tables and matrices to be included in the documentation for the systems. However, the profile lacks a treatment concept similar to the treatment concept of the QoSFT profile that could be used in the documentation of treatment ideas and options before the creation of actual requirements. The documentation of such ideas is also promoted by, for example, IEC 61508 [2010], which requires consideration to be given to the elimination of hazards during the risk and hazard analysis phase.

Secondly, the profile enables the use of fault tree analysis techniques within UML models. As such, structured information describing the fault mechanisms and their impact on possible occurrences of hazards can be included in the models. This kind of information could aid both the understandability of (safety-related) software requirements and their mechanisms to affect (hopefully remove or reduce) the risks.

For the documentation of hazards and risks, we suggest a combination of the modelling notations. FTA is a very analytical technique and enables quantitative analysis of hazards and can aid the discovery of means to handle the hazardous situations. To support traceability, it should be possible to trace hazards to the requirements. As well as tracing the hazards to the requirements, it would be beneficial to document the approach to handling the risks, similarly to the approach of the QoSFT profile. Our modelling approach related to hazards will be discussed in more detail in Section 2.4.

2.2.2 Requirements modelling

The requirements specification may be the most critical part of the development of complex systems. Requirements specifications should be formal enough to fulfil the requirements of standards yet be based on concepts familiar to developers. Currently used modelling languages in systems and software development hardly fulfil these properties however. For example, UML only defines use case concepts for stating (functional) requirements. SysML also defines a set of textual requirements specification concepts, though they can hardly be characterized as formal. In the domain of industrial control, there are also standards related to functional requirements, including IEC 62424 [2008] and IEC 61804 [2003]. These standards are, perhaps, more familiar to the developers in the domain, though they also enable a structured presentation of the required functionality and coupling to the instrumentation of the system. In a similar way to modelling the risks and hazards, we intended to extract usable notation and conventions from both for use in the model-based development of safety-critical applications.

2.2.2.1 IEC 62424: Specification for the representation of process control engineering requests in P&I diagrams and for data exchange between P&ID tools and PCE-CAE

The purpose of the IEC 62424 [2008] standard is to define a specification for the presentation of process control engineering (PCE) requests in piping and instrumentation diagrams (P&ID) and to enable data exchange between P&ID tools and control engineering tools in order to optimize the engineering process. Figure 8 presents the general graphical representation of PCE requests within piping and instrumentation diagrams. In addition to defining presentations and semantics of such PCE requests, the standard defines an XML-based format (CAEX – Computer Aided Engineering eXchange) for interchange of such information. The main purpose of the data exchange format is to enable transportation and synchronization of information between P&ID and process control engineering databases [IEC 62424 2008]. More details about the possible information content of PCE requests can be found from the report of the second work package.

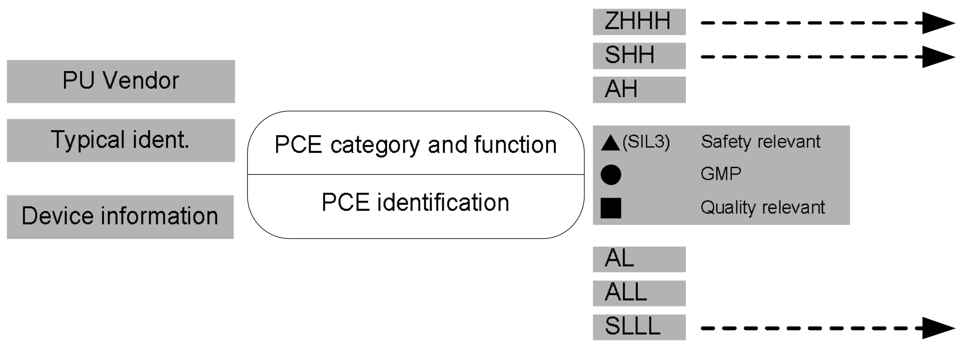


Figure 8. General representation of PCE requests in P&ID (modified from IEC 62424 [2008]).

Although the standard could be characterized as somewhat process-automation-oriented, it is one of the few international standards addressing specifications of functional requirements of automation applications and systems. In addition to defining the representation of PCE requests within P&I diagrams, the standard defines the data model and exchange format that, in principle, allow computer aided analysis of models. Nevertheless, the standard still lacks tool support, and the data model does not allow specification of, for example, detailed logic of interlockings or safety functions.

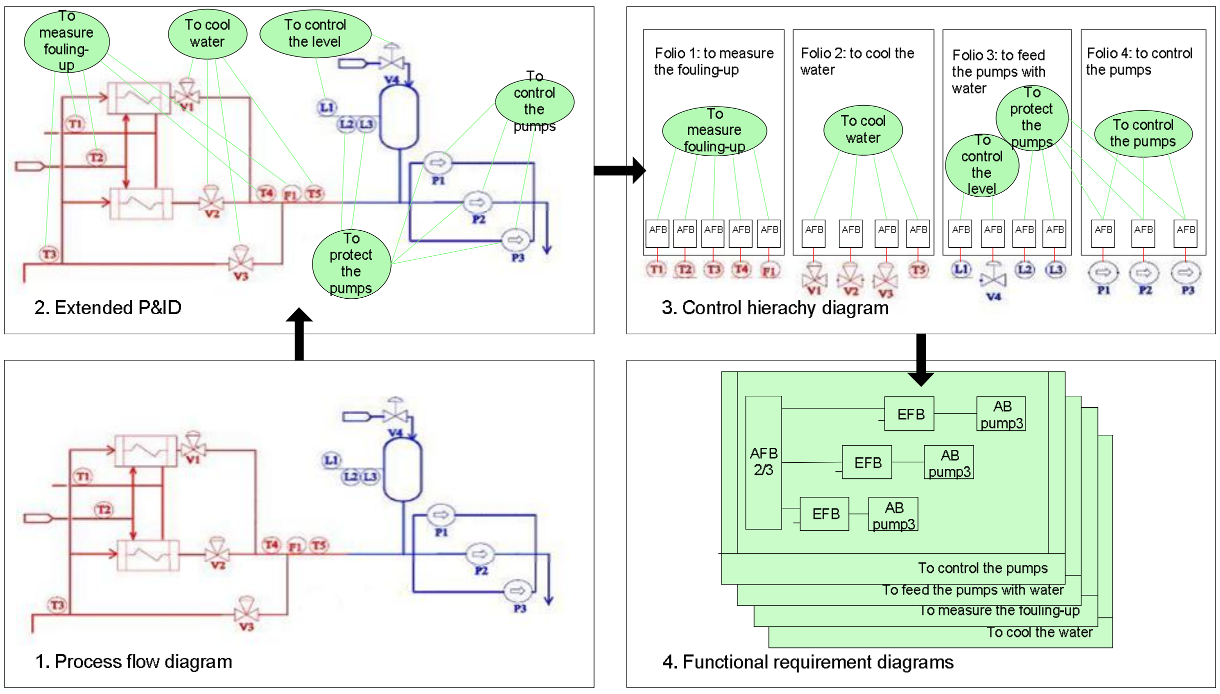
From the point of view of safety-critical software applications in machinery, the standard and the data model provide interesting capabilities. The standard allows specification of: 1) needs of safety functions ('safety function requirements') and their unique identifiers, 2) the instrumentation (actuators) that the safety functions are supposed to control or use to perform the safety functionality, 3) the instrumentation (sensors) that the safety logic uses as inputs, and 4) the required safety integrity levels of the safety functions. Another constitutive aspect of the standard is the tight integration of requirements related to the necessary functionality with process devices and instrumentation.

The standard also allows for the presentation of both safety-related and non-safety-related required functionality within the same models, which may, or may not, be desired depending on the application. Modelling both safety-related and non-safety-related functional requirements in the same models could reduce the probability of adverse interference of non-safety functions to safety functions, as both kinds of requirements would be visible to designers and developers at the same time. However, the data model of the standard lacks details and does not allow, for example, the specification of the actual safety functionalities that the safety functions are supposed to initiate in the systems, subsystems or actuators (e.g. stopping or starting motors or pumps, closing or opening valves, etc.). Moreover, the activating and disabling logic of the safety functions, i.e. the conditions activating and disabling the safety functions in the case of on-demand-mode safety functions, cannot be defined in detail.

2.2.2.2 IEC 61804: Function blocks for process control

Another international standard addressing the issue of the specification of functional requirements within the automation domain is IEC 61804 [2003], which originates from the power plant industrial sector. The specification aims to use IEC 61499 [??]-based function blocks (FB) during the complete lifecycle of the systems but from different points of views. As the standard states, the design starts from piping and instrumentation diagrams that give the requirements from the process and instrumentation view. The desired behaviour of the plant is presented in functional requirement diagrams (FRDs) using a (vendor) neutral FB language and without considering the detailed behaviour of the underlying devices. The constructs making up the FRDs are application blocks (AB) and elementary function blocks (EFB) that present the data and the algorithms, and they are used during the design phase. The FRDs can later be turned into detailed design via several designs, using the devices available with the interconnections and configurations of the devices [IEC 61804 2003].

According to the standard, FRDs are the specifications of the control functions required to control the process. The FRDs can also be obtained with studies that can be summarized as follows. Process flow diagrams are first used to identify the elementary process operations that are to be controlled. The (required) control functions of the elementary process operations and of the process are then defined in piping and instrumentation diagrams. The (required) control functions are structured as a set of folios and presented in control hierarchy diagrams (CHDs). To achieve the FRDs, the details of the control functions can then be specified for each folio [IEC 61804 2003]. Figure 9 presents the phases and simple examples of phase products.



To measure fouling-up
To cool water
To control the level
To control the pumps
To protect the pumps
2. Extended P&ID
Folio 1: to measure the fouling-up
Folio 2: to cool the water
Folio 3: to feed the pumps with water
Folio 4: to control the pumps
AFB
3. Control hierachy diagram
1. Process flow diagram
AFB 2/3
EFB
AB pump3
To control the pumps
To feed the pumps with water
To measure the fouling-up
To cool the water
4. Functional requirement diagrams

Figure 9. Lifecycle from process flow diagrams to functional requirement diagrams (modified from IEC 2003).

The similarities between the two referenced IEC standards (IEC 62424 [2008] and 61804[2003]) include the use of piping and instrumentation diagrams, and tight integration with the instrumentation of the process to be automated. Neither standard is intended to be used only to present safety function requirements. The standards can be used for that purpose, however, which is clear in the case of IEC

62424 [2008]. In the case of IEC 61804 [2003], safety and security are included in the list presenting present and probable future needs of control systems.

Compared with IEC 62424 [2008], IEC 61804 [2003] allows more detailed presentation of logic and algorithms related to the required functionality, including safety functions. For such specifications, the standard supports the use of function blocks that may vary from large application blocks to elementary blocks with one or more inputs and outputs. The blocks generally consist of logic and algebraic operations with which the conditions activating the functionality can be specified. Boolean and algebraic operations and specifications should also allow computerized processing of the requirements. Moreover, the semiformal methods that IEC 61508 [2010] suggests for the requirements specification phase include logic and function block diagrams that are used in IEC 61804 [2003].

The strengths of the standard also include familiarity for automation developers because of the function block (logic diagram)-based approach. However, the use of function blocks may restrict the approach to function-block-based PLC and DCS systems. Moreover, another downside of the approach is that it can be seen as specifying how the functionality should be implemented instead of only specifying what is required. Specifying only the required functionality without suggesting any kind of solution for the implementation, however, would probably be more difficult and the question would probably also depend of the FB libraries used during the specification. Moreover, the detailed specification of the logic already during the requirements specification phase may be a necessity in some cases.

We only suggest a few additions to the documentation of requirements in our modelling approach to the existing modelling concepts of UML AP. The additions are discussed in more detail in Section 2.4.

2.2.2.3 SysML

SysML is a general purpose modelling language specified by the Object Management Group [OMG 2008b] and intended for modelling systems that may include combinations of hardware, software, data, people, facilities and natural objects. SysML reuses part of the modelling concepts (called UML4SysML) and diagram types of UML and defines new diagram types to extend those of UML: parametric diagram, block definition diagram, internal block diagram and requirement diagram.

In SysML, a requirement specifies a capability or condition that should (must) be satisfied, a function that a system should perform or a performance condition that a system should achieve. In SysML, the requirements are text-based and thus more informal compared with, for example, requirement concepts presented in the earlier sub-chapters. Consequently, for example, computer-aided processing of requirements may be more difficult. In SysML, however, there are several concepts intended to support traceability between requirements, design elements and test cases.

In general, the traceability features of SysML allow the requirements to be linked to both the implementing model elements and test cases verifying the requirements. Compared with UML, these features are new – which is natural as UML does not contain elements corresponding directly to the requirements and test cases of SysML. In the context of the development of safety-related and safety-critical systems and applications, the simple trace relationships could be used to define at least the required backward and forward traceability between the requirements, architecture and design (provided that the architecture and detailed design of the system would be defined using UML/SysML).

2.3 Security

Whereas the functional safety of systems containing software parts has been the topic of several modelling-related standards and publications, finding similar approaches or papers about security proved difficult during the project. Several standards and publications address the topic of security and a number of development processes emphasize security but not that many relate to modelling. Consequently, during the project we also needed to focus on security standards. We were able to discover a modelling-based approach, however, that could improve the achievement of security goals and the understanding of the underlying risks.

We would also like to point out some observations about security and development of data-secure systems. Perhaps the best results for security could be achieved by integrating data security into the development process instead of assessing and correcting security risks in specified or even implemented systems, as has sometimes been suggested. The main reason for this is that it may not be possible to start from the risks if the system does not yet exist to be analysed. The starting point therefore already differs from the starting points of safety functions (systems) in which the system containing the risks already exists, at least in design documents. It may not be possible to handle security-related risks either without taking the risks into account in all the development phases and, most importantly, during the design and implementation of networked interfaces. However, potential risks that need to be addressed during the development should be noted during all development phases by, for example, marking ports (or similar elements in the design) that handle data over a public network.

Development techniques should also include information about data security. In the case of safety-related systems, for example, testing could also always include security testing. Similarly to SIL or PL levels, components and subsystems could be classified based on their data security levels so that for the whole system to achieve a certain level of security, all its subsystems would need to be at that or a higher level. In the future, such security levels could also be added to functional safety standards to complete the SIL and PL classifications because, in general, a compromise on security may affect the correct behaviour of the system, including safety.

2.3.1 Towards unification of safety and security

The purpose of IEC 61508 is to provide a unified approach that is rational and consistent for all safety lifecycle activities of functional safety systems. The standard covers all lifecycle phases from concepts through decommissioning. Security is an asset that the standard does not address, however, although compromising security could also lead to safety being endangered if, for example, a worm could block the behaviour of a subsystem of a safety-related system.

According to IEC 61508, the lifecycle phase, which follows the concept and scope definition phase, is the hazard and risk analysis phase, the aim of which is to determine (all) the hazards and hazardous events, and the sequences leading to hazardous events and risks associated with the hazards. Perhaps, part of the security-related hazards and risk could be specified already during this phase. For example, security-related risks could initially be searched for from the user requirements specifications, such as, remote control needs. This puts all the weight on carrying out risk analysis iteratively as it may not be possible to identify all the security-related hazards before designing the EUC control system. There can be alternative approaches to unifying the development of the functional safety and the data security however.

2.3.1.1 Security-related lifecycle phase(s)

As IEC 61508 states, the overall safety lifecycle should be used as a basis for claiming conformance to the standard, but a different overall safety lifecycle can be used to that given by the standard provided that the objectives and requirements of each clause of the standard are met [1]. Consequently, to address the data security concerns, a suitable lifecycle phase (or phases) related to the development of data-secure applications could be added to the lifecycle model of the standard. In order to apply systematic hazard and risk analysis techniques for identification of data-security-related hazards and risks, the phase would require, at least, design documents related to both the E/E/PE system and the control system and, consequently, could not be carried out earlier than simultaneously with the realization phase of the E/E/PES system. Figure 10 sketches the approach without identifying any activities or sub-phases required to assure data security.

However, the (unnamed) phases could include, for example, identification of the data-security-related hazards and risks and, if necessary, restructuration and modifications to the architectural and detailed design of the safety-related software. In any case, it is vital to realize that the risks have to be identified (found with risk and hazard analysis methods) irrespective of whether they are functional or security-related. The ways to eliminate or mitigate them may then be partially different. Security checks should also be included in all the lifecycle phases. Security affects the risk analysis (phase 3) in which questions like “can we be affected by a new and unknown Internet virus when using web-based device configuration?” It affects the realization (phase 9) when the use of cryptographic techniques can make the connections more reliable. It affects the maintenance (phase 14) when any change in networking environment will have an effect on the critical system. It affects the disposal (phase 16) when releasing devices to material recycling can expose passwords and usernames outside the company without erasing the systems properly.

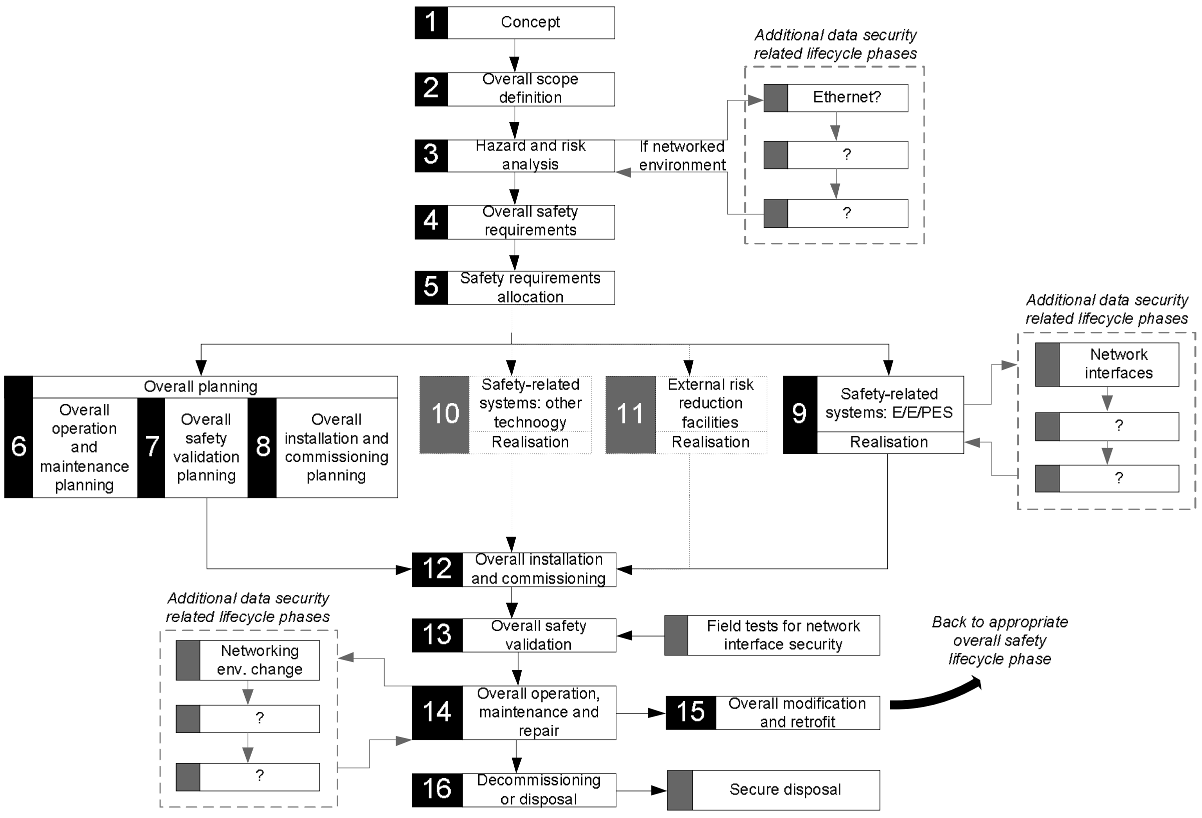

1 Concept
2 Overall scope definition
3 Hazard and risk analysis
4 Overall safety requirements
5 Safety requirements allocation
If networked environment
Additional data security related lifecycle phases
Ethernet?
?
?
Overall planning
6 Overall operation and maintenance planning
7 Overall safety validation planning
8 Overall installation and commissioning planning
10 Safety-related systems: other technoogy Realisation
11 External risk reduction facilities Realisation
9 Safety-related systems: E/E/PES Realisation
Network interfaces
12 Overall installation and commissioning
13 Overall safety validation
Field tests for network interface security
14 Overall operation, maintenance and repair
15 Overall modification and retrofit
Back to appropriate overall safety lifecycle phase
16 Decommissioning or disposal
Secure disposal
Networking env. change

Figure 10. Conceptual lifecycle model including additional lifecycle phases to assure data security. The security should be integrated into all the relevant phases.

Another approach to increasing data security may be to use an approach similar to functional safety. For example, security-related risks could be searched for from the user requirements specifications, such as remote control needs, and possibly handled with security-related functions. (They could be called security functions instead of safety functions.) Security-related functions could be used for, for example, authentication of users, authorization, and detection and correction of communication errors, to name a few. Such functions can hardly ensure the achievement of security without other activities however. There is a fine line between secure systems and usable secure systems that are useful for the intended audience. In the best security, people understand where the line is, but it is not always easy to implement development frameworks.

2.3.1.2 Addition of security checks

In addition to the sketched approach, the data security could be treated based on a ‘lightweight’ approach that would not require modifications to the lifecycle model. In this approach, security checks could be added to practically all the development activities and lifecycle phases of software applications. During these checks, it could be assessed whether the products (for example, specification) of the current development phase somehow compromise the security. If the security is compromised, corrections could be planned already during the session and implemented before continuing to the following development phase.

2.3.1.3 Secured for the networked environment – extending certifications

Safety-critical equipment is always certified. One way of increasing the security of the product as well as the security awareness of customers could be to extend, for example, the SIL3 requirements to SIL3+. SIL3+-certified products have been tested not only to physical component failure but also

- against the denial of service attacks, for example, *n* forged UDP packets/second
- against broken protocol attacks, for example, tested using all tests against TCP and UDP, test tool X, test set Y
- against broken web-based user interface design using most common web application problems, test tool X, test set Y.

The testing and certification trend can already be seen in industrial automation. There are companies that certify products using their own test sets.

2.3.1.4 Securing the product via design methods

Most security consultants provide a training solution to make products and systems more secure. Training and education are not enough however. Their main purpose is to educate programmers, users and designers in security-related problems. The only solution that works is to force programmers and designers to take security into account.

Current programming tools are good at hiding the network connections from the user, which is what they have aimed to do for the whole Internet era. Industrial automation and control systems are networked software products with different requirements for the network connection, depending on the transferred information. For safety systems, this means the inclusion of a reliability factor for all low-level network protocols.

Safety-related connections require the use of safety protocols, and various real-time class communications require the use of protocols and solutions that meet the necessary real-time capabilities. It is therefore not only that, for example, a safety protocol is used but also that the low-level transport protocols or the implementations that these protocols use are tested and have an estimated reliability factor. The real security problem usually lies in the fundamental network protocols, which can cripple the use of a higher level safety protocol.

One of the solutions to force the programmers and designers to notice and address these requirements would be to implement the ‘networked connection’ type of interface to common modelling and code generation tools of which UML is one example. The networked connection type would have subcategories depending on, for example, the type of network connection (shared with other traffic, dedicated link, wireless link). Wireless connection would have the rule ‘if cannot be sure secured connection is used then enforce encryption on interface’ and so forth. This approach can be viewed as implementing the secure best practices for networked communication to application programming interface connections used within the design tool.

2.4 Modelling approach of the Ohjelmaturva project

The development of safety-critical software applications in machinery requires a systematic approach in order to achieve and maintain the required functional safety. In general, the development lifecycle phases and activities, such as those proposed or required by IEC 61508, need information produced by a previous lifecycle phase or phases and produce information that can be used during later phases. Work package 2 of this project was primarily concerned with the design and specification (lifecycle phases 4 and 5 in IEC 61508 [2010]) of such software applications and the means of modelling that could be used to support the phases. Thus, our aim was to support the development of safety-critical and safety-related applications by enabling the presentation of information (modelling) that supports both the requirements specification and the development activities that follow from such applications. We focused on modelling risk and hazard information and requirements in more detail. In addition, we extended our previously defined and implemented capabilities to generate Modelica simulation models based on early design models in order to enable simulation-aided development of safety-related functions.

2.4.1 Risks and hazard modelling

The aim of the risk modelling package that was developed during the project was to enable the definition of both hazards and risks related to the EUC (Equipment Under Control), the EUC control system and the system performing the safety functions. The aim of the modelling concepts is to cover the risks and hazards related to both safety and security. Our main focus, however, was on safety. The profiles that were used as a basis for the definition of the modelling concepts include the quality of service and fault tolerance mechanisms profile, the safety analysis profile and the CORAS profile.

The central concept within the package is the Hazard concept, which defines the properties common to safety-related hazards as well as security-related hazards. These properties include the unique identifier of the hazard (ID), textual description, estimations of likelihood and consequences of the hazard, and the resulting estimation of risk. The enumeration literals for the likelihood and consequences have been extracted from IEC 61508 [2010] (qualitative); however, the risk values can also be defined independently of the values of the likelihoods and consequences. Thus, by nature, the concept is intended to cover definition (description) of both the unwanted incident and general properties of the risk related to the incident or event in question. The information content of Hazards may vary depending on the phase of development. For example, only an ID and a description for each hazard could be specified during the initial definition of the hazards. More detailed properties could follow the detailed risk and hazard analysis. In addition to the properties described, Hazards may reference Assets defined within AssetDefinitions. The intention of the AssetDefinitions is to enable definition of the enterprise assets on which the risks or hazards (especially security-related hazards) could have had an impact or that have been taken into account during the analysis.

The intention of the HazardRelation concept is to enable definition of relations between hazards, such as hazards initiating or including other hazards. The intention of the RiskTreatment concept is to allow definition of the approach to reduce or remove a risk. The properties of the concept include a general approach to treat the risk (such as avoiding it or reducing its consequences), a more specific description of the treatment approach and whether or not the treatment will be implemented as a safety function. The properties can be given values depending on the state of the project and the amount of information available. Figure 11 illustrates the concepts of the basic risk modelling package.

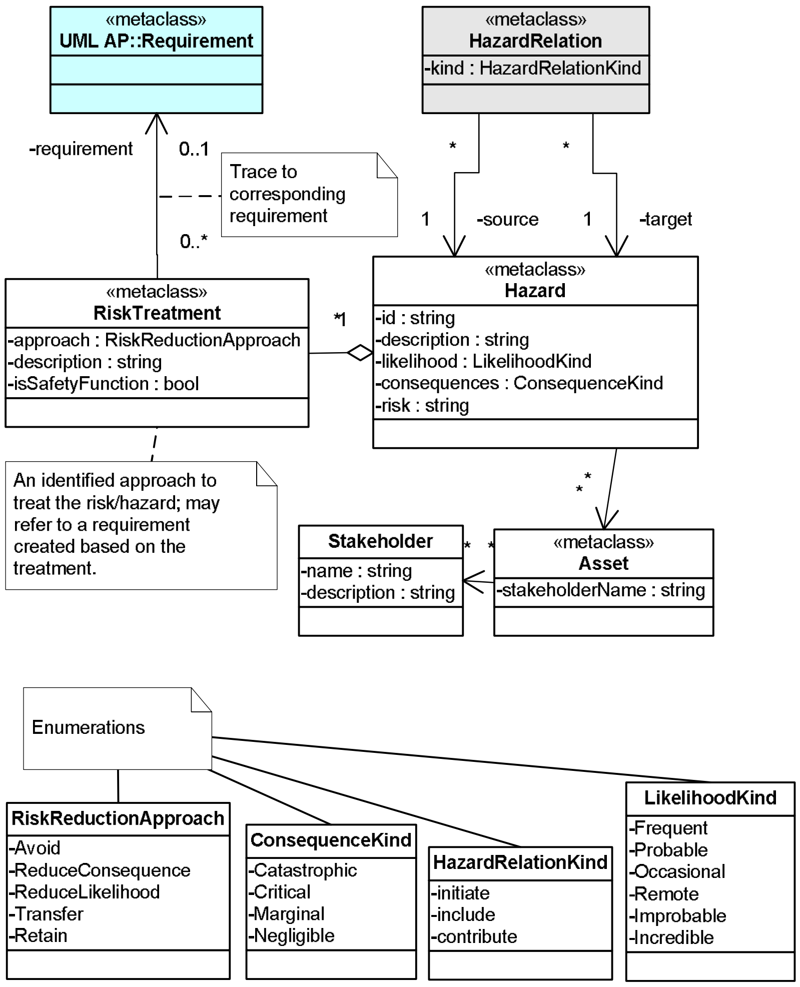


Figure 11. Basic risk modelling package.

The aim of the second sub-package of the risk modelling package, FTA extensions, is to enable definition of hazards and conditions leading to the occurrence of hazardous events and situations by enabling the use of fault tree analysis (FTA) models. The main justification for the use of FTA is that it enables highly analytical and structural specification of how the faults and (required) conditions may lead to hazards. This kind of structured descriptions of the occurrences of the hazards can then be used when ideating and specifying safety-related functions that may contribute to avoiding the hazard, making its occurrence less probable or its consequences less critical. Part of the modelling concepts to be presented have been extracted and re-structured from the safety analysis profile presented earlier.

The central concept within the package is the abstract HazardModelNode concept, the concrete instances of which include the Hazard, Fault, RequiredCondition and ResultingCondition concepts. Hazards, which may represent both safety and security hazards, are intended to be used as the topmost elements in the diagrams, and their attributes include an (unique) ID, description, estimates of likelihood and possible consequences and resulting risk. Faults represent faults in the system and contain a textual description of the fault. In the case of security-related and software-induced hazards, the faults may also refer to modelling elements that (may) cause the fault if such an element can be specified during the analysis. In any case, the description attribute can be used to describe the fault and other information related to its occurrence.

ResultingConditions and RequiredConditions, on the other hand, can be used to model conditions resulting from logical operations and conditions required for the faults to interact, respectively. In addition to the concepts presented above, the definition of fault tree analysis models requires logical operators, which are discussed in more detail in the report on the work package and connectors between faults, conditions, operators and hazards.

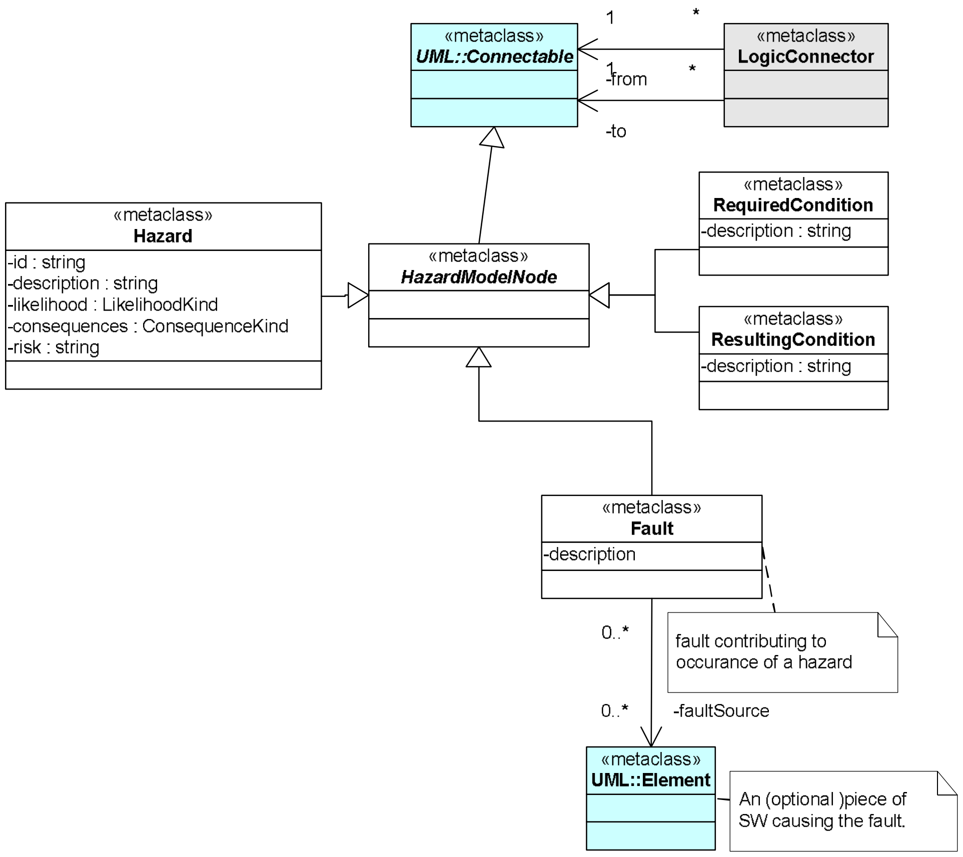


Figure 12. FTA extensions package.

2.4.2 Requirements modelling

Whereas the concepts defined within the risk modelling package could be used without other parts of the profile, the aim of the concepts of the requirements modelling package is to define additional concepts supporting the definition of safety-related requirements in conjunction with the existing requirement concepts of UML AP.

The aim of the safety actions package is to define concepts required for the definition of safety actions to which the safety action requirements and interlocking requirements may refer. The SafetyActionEnumeration is an enumeration-like concept consisting of enumeration-literal-like SafetyActionLiterals. The intention of the SafetyActionLiterals is to identify the purpose of the interface, such as stopping, locking, shutting down or starting a device. With such enumeration-like constructs, the collection of possible safety actions specific to certain pieces of instrumentation can be defined specifically for the needs of the current project. The SafetyActionInterface concept extends the UML AP RequirementInterface and thus acts as a port-like interface between the UML AP structured requirements related to the needs to interlock or control devices. Figure 12 presents the concepts and their relationships to existing UML AP concepts.

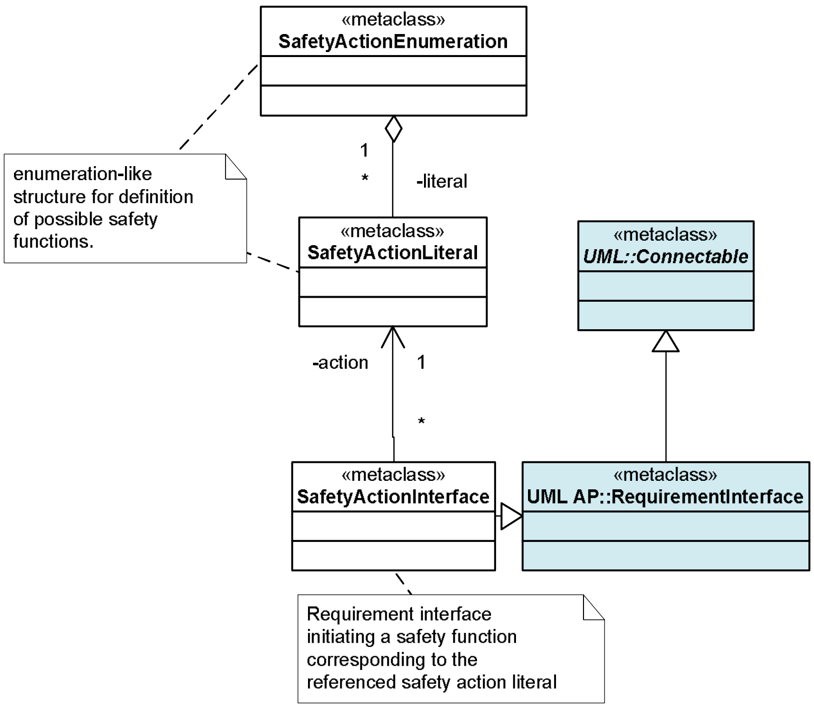

«metaclass»
SafetyActionEnumeration
1
*
-literal
enumeration-like structure for definition of possible safety functions.
«metaclass»
SafetyActionLiteral
«metaclass»
UML::Connectable
-action
1
*
«metaclass»
SafetyActionInterface
«metaclass»
UML AP::RequirementInterface
Requirement interface initiating a safety function corresponding to the referenced safety action literal

Figure 13. Safety actions package.

The modelling of safety functions and not-safety-critical interlocks, as it appears to the authors, requires the definition of at least two pieces of information: the conditions enabling the safety functions (in the case of on-demand-mode safety functions) and the actual activity and logic performed by the function. Whereas the aim of the safety actions package is to enable definition of the actions, the aim of the safety logic definition package is to enable definition of the activation and control logic of the safety functions. To enable such definitions, the approach of the package is to use logical operations such as AND and OR, which can be connected to interfaces of the safety function and interlocking requirements.

Structured requirements for UML AP, including the safety function and interlocking requirements, can already be connected to other structured requirements with, for example, data requirement interfaces and interlocking requirement interfaces presenting the need for data interchange between required functionalities and required interchange of interlocking signals, respectively. Consequently, the challenge of the specification of the activation logic can be seen as presenting the logic from the input interfaces of the safety (or interlock) function requirements to the output safety action interfaces. Operations required for performing such operations may include, for example, discretization of signals of any type to Boolean signals and logical operations similar to those defined in the (FTA) operations library. Figure 13 illustrates the contents of the safety logic definition package.

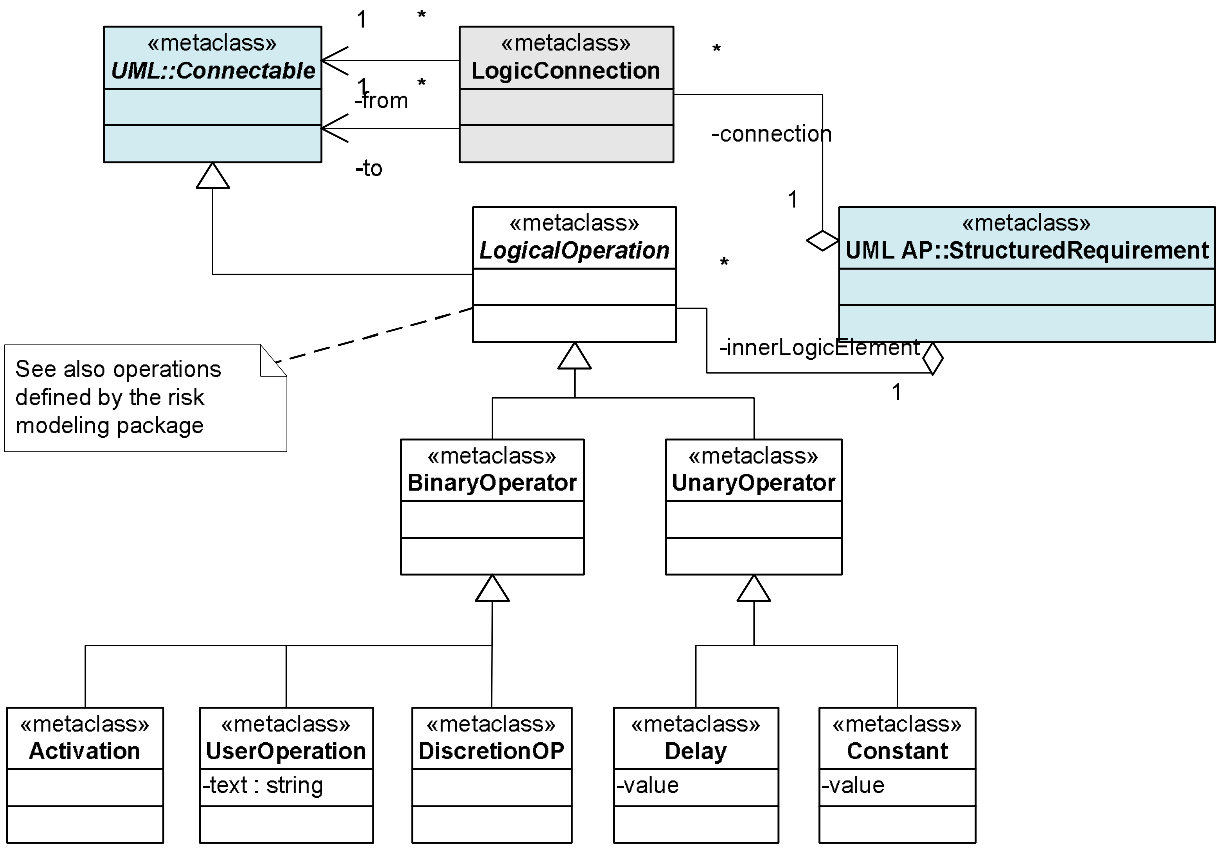


Figure 14. Safety logic definition package.

The safety-related refinement package defines two concrete and one abstract refinement type extending the existing RequirementRefinement concept. The first of the new concepts, SafetyRefinement, provides a common base for the concrete types that further extend it. The aim of the first concrete refinement type of the package, SILRefinement, is to enable the definition of the required SIL (Safety Integrity Level) for safety function requirements and other kinds of requirements presenting the need for safety-related functionality. The possible SIL levels defined by IEC 61508 [2010] are 1, 2, 3 and 4, but, in addition to these, the package allows for the specification of a SILRefinement with a level 0, indicating that the required safety integrity level is below 1. The aim of the second of the refinement types, TechnologyAllocation, is to allow for the allocation of early safety function requirements to the requirements for E/E/PE systems, systems of other technologies and external risk-reduction facilities.

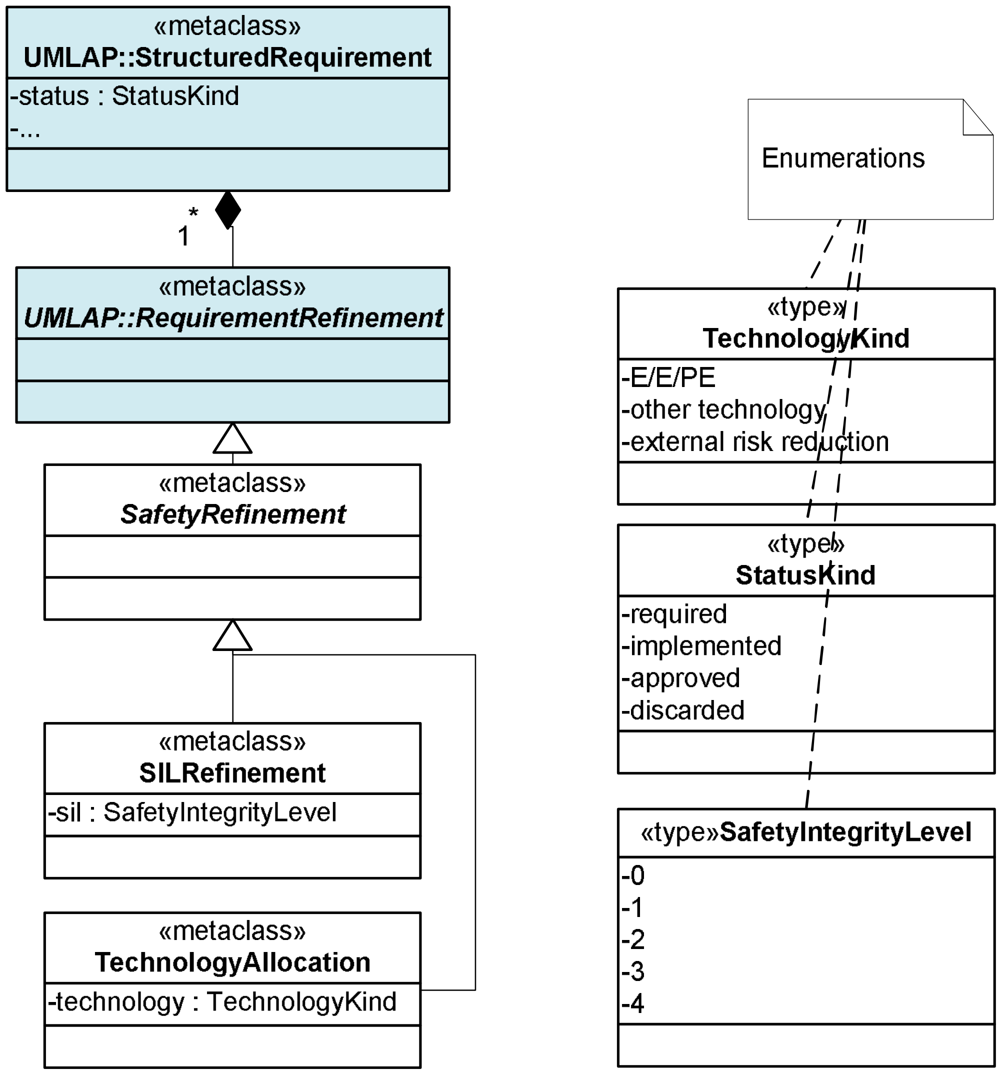


Figure 15. Safety-related refinement package.

2.4.3 Tool support for the approach

The UML AP Tool is a prototype level modelling tool that has been implemented on the Eclipse platform, and it is intended to provide the basic modelling functionality needed for the design and specification of the automation and control application's requirements, functionality and structure. The tool consists of plug-ins implementing the essential modelling concepts (metamodel) defined by the UML Automation Profile (for which this section has presented additions), a graphical editor plug-in, plug-ins implementing the diagram types defined by the profile, a plug-in contributing the properties of the new concepts (metaclasses) to the properties view of the Eclipse platform and a plug-in implementing the transformation-related extension interface of the tool. The tool has been implemented to extend the topcased modelling toolkit version 2.2.0, and it thus also allows for the use of modelling concepts and diagram types of UML and SysML within the models developed with the tool.

In order to enable further development and experimental use of the new modelling concepts, the modelling concepts presented in the earlier sections were implemented with the tool along with a new diagram type for use with the concepts of the risk modelling package. A modification to the internal block diagram [SysML] was also developed for the tool in order to allow use of the contents of the safety logic definitions package and FTA operations library. The new diagram type, related to the risk modelling package, is called *risk modelling diagram* and the diagram type that allows the logic definition is called the UML AP internal block diagram. Examples of the use of the diagram types are presented in the report on the work package in which the concepts are used in a simple modelling project.

In addition to the modelling concepts, a new export plug-in and additions to an existing export plug-in for generating ModelicaML models were also implemented during the project. The approach for generating simulation models has been presented in detail in Vepsäläinen and Kuikka (2011a). A detailed presentation of the safety-related modelling concepts is available in Vepsäläinen and Kuikka (2011b), which also presents more background information and justification for the modelling approach.

References

Ohjelmaturva publications:

Vepsäläinen, T. & Kuikka, S. 2011a. Simulation assisted, model-based development of safety related interlocks. In: 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications, July 29–31, 2011, Noorwijkerhout, the Netherlands.

Vepsäläinen, T. & Kuikka, S. 2011b. Towards model-based development of safety-related control applications. In: 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), September 5–9, 2011, Toulouse, France.

Other references:

ANSI/ISA-99.00.01-2007. 2007. Security for Industrial Automation and Control Systems, Part 1: Terminology, Concepts and Models. Approved October 2007.

Braber, F., Hogganvik, I., Lund, M.S., Stølen, K. & Vraalsen, F. 2007. Model-based security analysis in seven steps – a guided tour to the CORAS method. January 2007. Springer Netherlands. BT Technology Journal. Vol 25, No 1, pp 101–117.

Douglass, B. 2009. Analyze system safety using UML within the Telelogic Rhapsody environment. IBM Corporation.

EN 50159-2. 2001. Railway applications – Communication, signalling and processing systems Part 2: Safety related communication in open transmission systems. CENELEC.

IEC 61499.

IEC 61508-3. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems, Parts 1–7. International Electrotechnical Commission. Edition 2.0. International Electrotechnical Commission.

IEC 61804. 2003. Function blocks (FB) for process control – Part 1: Overview of the system. International Electrotechnical Commission.

IEC 62424. 2008. Specification for Representation of process control engineering requests in P&I diagrams and for data exchange between P&ID tools and PCE-CAE. International Electrotechnical Commission.

ISO 15408. 2009. Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and general model. Version 3.1. Revision 3, July 2009.

OMG 2008a. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification. Version 1.1.Object management Group.

OMG 2008b. OMG Systems Modeling Language (OMG SysML). Version 1.1. Object management Group.

3. Development process of safety-related software

3.1 Agile development of safety-critical software

Matti Vuori, Tampere University of Technology, Department of Software Systems

3.1.1 Introduction

There is a tendency for companies to transform their software and product development practices into a more incremental form using a special agile software development lifecycle and project management models. Incremental development processes have been used previously, but agile processes add new project planning, management and execution principles to them. The main value of agile processes still comes from controlled increments and releases, which they produce more often than previous models. Controlled releases should be especially helpful in obtaining feedback from the customer and for managing project risks. (Note: it is assumed that the reader of this report has some familiarity with the concepts of agile software development; if this is not the case, the Wikipedia article about agile development can be a good starting point to learn about the topic [Wikipedia. Agile software development.]).

In his book, Larman [2004] lists the following key motivations for iterative development:

- Iterative development is lower risk; the waterfall is higher risk
- Early risk mitigation and discovery
- Accommodates and provokes early change; consistent with new product development.
- Manageable complexity.
- Confidence and satisfaction from early, repeated success.
- Early partial product.
- Relevant process tracking; better predictability.
- Higher quality; less defects.
- Final product better matches true client desires.
- Early and regular process improvement.
- Communication and engagement required.
- IKIWISI required [IKIWISI = I'll Know It When I See It].

These are all benefits that companies seek when starting to use agile methods. Modern agile processes promise even more benefits compared with previous iterative/incremental models, especially the following:

- Shorter time to the first releases and more frequent releases
- Reduced amount of process and project documentation but better communication and thus a smoother process
- Increased customer participation.

The most important benefits depend on the type of development: mainly whether it is a customer-oriented development of tailored systems or a mass-market-oriented development. Sometimes the approaches can be combined if the products are developed for a small key clientele and the offering is targeted at the mass markets. The approaches and needs are different however.

In the development of tailored systems, the customer's essential needs that can benefit from agile are early release and understanding of the system through using it, regular releases at a sensible pace so that the new system can be learned and all the necessary adaptations can be made in time, and making changes to plans during the process in a flexible way. In mass-market product development, needs may be based more on the manufacturer's desire to control risks and respond fast to competition and emerging market needs.

The goal of reaching these targets influences the way companies approach the development process. Some companies may start the agile process with a very vague idea of a concept, some may see it predominantly as a way of making software production more controlled, and some may simply aim for a row of productive solutions and new releases to customers. Release orientation and release readiness are in fact seen as a key element of agile approaches.

Agile development has received criticism however. Moser et al. [2007] wrote: "Although agile processes and practices are gaining more importance in the software industry there is limited solid empirical evidence of their effectiveness." Coplien [2011] looks back on the development of agile processes and notes that, "However, as with most trademark-able labels, manifestos, and other documented ideals, the reality of the trumpeted practice often missed the mark. 'Agile' became a label for a wide collection of otherwise unrelated practices, a collecting point that empowered people to justify their favourite practice." Kruchten [2011] documents topics that the agile community is not really willing to tackle for a variety of reasons:

1. Commercial interests censoring failure; 2. Pretending agile is not a business; 3. Failure to dampen negative behaviour; 4. Context and Contextual applicability (of practices); 5. Context gets in the way of dogma; 6. Hypocrisy; 7. Politic; 8. Anarchism; 9. Elitism; 10. Agile Alliance; 11. Certification (the 'zombie elephant'); 12. Abdicating responsibility for product success (to others, e.g., product owners); 13. Business value; 14. Managers and management are bad; 15. Culture; 16. Role of architecture and design; 17. Self-organizing team; 18. Scaling naïveté (e.g., scrum of scrums); 19. Technical debt; and 20. Effective ways of discovering info without writing source code.

This being the situation, the agile approach and agile practices need to be chosen very carefully in a company, and an experienced organization needs to rely on its own engineering sense to decide on its processes.

Private and public organizations have been replacing their waterfall models or even incremental models with various agile project models and their corresponding practices for some time. This has been going on in all kinds of development domains: simple as well as complex systems. Agile has been used mostly in small projects; its application in large projects is still a research issue [see, for example, Rohunen et al. 2010]. A company named VersionOne has published annual surveys on the adaptation of agile development [5th Annual State of Agile Development Survey 2010], the reports of which contain plenty of detailed information. There are still some development contexts in which there is insufficient understanding of the way agile processes can be used so that they bring benefits and do not endanger the quality of the operation and products, and the product business or customers' operations.

The development of safety-critical systems is one such area. It should be noted that it is not a heterogeneous area but consists of many different development cultures defined by, for example:

- Type of product and system – from medical devices to machines and nuclear power plants
- The role of software in the system – is it mainly a software-based system or is software still only in a restricted role and the product is perceived as, for example, a mechanical device
- The size and scope of the system – small personal devices clearly require a very different approach to large plant level systems
- The risk level of the system – factory machines have a very different risk level to nuclear power plants.

Thus, there are many variables but no generic answers, and we should not copy the practices blindly from another field but try to understand the context and see what possibilities agile approaches may bring and how they should be applied. Which parts of current practices could they replace and how should they be supplemented? Are there 'obvious' agile practices to implement, and which agile practices should definitely not be used in the given context?

3.1.2 An analysis of the applicability of agile methods in safety-critical development

In the Ohjelmaturva project, we conducted an analysis of the applicability of agile methods to safety-critical development. For a full report, see Vuori [2011].

The study used the following main methods: 1) identification of agile principles, process and activity elements, and practices that are key issues from the viewpoint of developing safety-critical software systems, 2) analysis of the identified elements: what possible risks may there be in applying them, and how should they be supplemented, modified or avoided and by which means, 3) mapping of already identified required or otherwise essential tasks of development, quality and safety assurance into a generic agile development framework, and 4) synthesis of key issues for guidance in process tailoring and development in companies.

The safety-criticality of the context of the study was moderate. We assessed mostly the safety integrity levels SIL levels 1, 2 and 3 [see Wikipedia. Safety Integrity Levels] in this analysis. The methods for assigning SIL levels to a system are based on hazard/safety/risk analysis at system level and can be found in standards IEC 61508-5 [2010] and SFS-EN 62061 [2005]. The standard series IEC 61508 [2010] was selected as a reference for this analysis because it is the most important basic safety standard for developing safety-critical software for machines and various automation systems. IEC provides a Frequently Asked Questions site for the standard series that explains the standard's approach and application nicely [IEC 61508 FAQ 2011]. IEC 61508 [2010] is also considered quite challenging, so if the agile processes can be used with it, it should be easier with many other standards. In 2010, the standard series was renewed, and this analysis provides a well-timed opportunity to address some of its changes and their impacts on the development process and tasks.

3.1.3 Anatomy of agile development

When we consider agile development, we need to have an understanding of what it consists of. For the purposes of this study, agile software development consists of the following elements:

- Principles
- Project model
- Software development lifecycle
- Software engineering methods, techniques and practices.

The principles consist of values, development principles, policies and thinking patterns of individuals and occupational groups and stakeholders. The project model is the concept of project planning and execution. A big part of the project's execution consists of developing software using a specific software development lifecycle, which in turn uses various software engineering methods, techniques and practices, some of which may be developed specially for agile development, and some that will have a more generic origin.

By now, basic agile processes are understood to have 'lost' some respected software engineering practices, and we need to be careful to analyse that these will not remain lost in the safety-critical context. Coplien and Bjørnvig [2010] outline these practices as:

- Architecture
- Handling dependencies between requirements
- Foundations for usability
- Documentation
- Common sense, thinking and caring.

There have been some methodological approaches to bring some of the missing elements into place (for example, Lean, which is analysed later in this report), but in the safety-critical context, the analytical elements have deep roots, which are likely to be able to address the deficiencies of agile when applied properly. In all the cases, the basic agile processes found in textbooks need to be supplemented with any practices that a given context or development situation requires. This is one of the things that we are trying to do here – to assess how common agile processes need to be modified in order to be appropriate in safety-critical development.

An agile development process is thus usually never ‘fully agile’. The so-called hybrid processes combine agile practices with traditional ways of doing things. Kennaley [2010] has analysed software development processes in a historical context and presents outlines for the next phase of software development, which again combines the best parts of various development paradigms.

In addition to the aforementioned elements, agile development is not just a project execution paradigm or a type of engineering but a culture, which means that when we are ‘going agile’ we need to consider organization culture issues, psychology and dynamics as well as process issues, but those are not in the scope of this paper, except for an analysis of the agile values.

Agile and Lean

Lean is an approach that is often associated with the context of agile development. Lean was originally called ‘Lean production’, but today ‘Lean’ only represents a vision of a good, efficient way of action. It has been applied mostly in manufacturing industries, though it has recently started to come into the software development world [see, for example, Poppendieck 2007] to complement agile development. We therefore included an analysis of Lean principles in our study.

Read more from the report

In order to keep this report short and readable, we will not go into the details on the analysis, which can be found in the report ‘Agile Development of Safety-Critical Software’ [Vuori 2011].

3.1.4 Guidance on changing the process to make it more agile

Based on the study, we offer the following guidance to companies that aim to develop their processes to make them more agile.

3.1.4.1 Prerequisites

Shared sense of a need to change

Agile is not a goal as such but a means to meet some business or other goals.

For any change in an organization, there should be a clear, shared sense of urgency to change processes as well as a feeling that the current way of action is no longer enough. Without this, the probability of success is not high enough – there needs to be a want to change. A requirement for this can be the formulation of a business case for process development: what benefits would a different release practice bring. Everyone should feel that there are shared problems that the agile development model would solve.

Solid process understanding and vision

There should be an understanding of the software and product development processes – agile and others – that should help with the understanding of one basic question: do we really aim for maximum agility or a balanced approach? This cannot be repeated too often: the goal is not to be agile but to obtain the best results by using processes that best fit the goals in this particular environment. (As orientation material on balancing agile and plan-driven methods, we recommend Barry Boehm's and Richard Turner's [2003] book 'Balancing Agility and Discipline – A Guide for the Perplexed'.)

Professionalism in action

The traditional approach to process change is that all the pieces are not in place, and an attempt to change a process may result in chaos and failure. Before transformation, the organization therefore needs to be able to fulfil, for example, all the requirements of IEC 61508 [2010] or other applicable standards without problems. This provides a baseline for process development and makes it possible to see any process problems clearly.

The quality of current activities should be high and the execution of the current process professional. If this is not the case, the change should probably not be attempted.

Experts and collaboration

Expertise on agile is needed in the transformation to guide the rest of the people in the transformation process. Process people or consultants are needed who can lead the transformation. Here, it should be noted that agile consultants do not necessarily understand all the process requirements of safety-critical development and the leading principles of safety and risk management. Even world-class agile experts may have limited understanding of testing, verification and validation. All the necessary areas of expertise need to be presented in the process.

3.1.4.2 Ten generic guidelines for process development

The following are seen as important guiding principles for the transformation of traditional process models into more agile:

- Respect your own engineering skills and experiences in process development.
- Do not follow fashion. Seek proven practices and analyse their suitability to your environment and culture.
- Understand that not everything needs to be agile and that 'more agile' does not mean 'better'.
- Remember that the more safety-critical the system and its development, the more control is required for the process and that will usually mean less agility.
- No single paradigm is sufficient. Being just agile or just plan-driven is not enough. Some areas may require less agility than others, for specific process needs and for balance.
- Much of any company's current activity is tacit in nature and not formulated or documented. Thus, you may be blind to your current success factors. External consultants are often needed to see what is essential in your activities.
- Safety-critical development is all about managing risks. In the same way, you need to manage the risks of making process changes.
- 'Find the style that is best for you' is the message in all development. If you act like everyone else, how can you be the best in your branch?
- Process developments are never just mechanistic process issues. Agility needs to fit into the corporate culture, which is a hard thing to change.

- Co-operation and collaboration between all stakeholders is one key issue of Agile. The same applies to process development.

3.1.4.3 Risk analysis of the transformation

A risk analysis should be carried out for the process change in order to identify potential pitfalls. This author has previously devised a risk map to present risk areas to assess. It is based on the idea that a transformation of project practices is very much a cultural change. Pure processes are easy to change, but in software development it is a question of the way people work together.

Elements of the software process – practices, processes, techniques and tools – can be thought of as forming a toolbox of items from which we are free to choose. Therefore, there as many moments when we need to make a decision: shall we use that particular practice or not? Boehm and Turner [2003] present a rule that is particularly well suited to safety-critical development:

> *Is it riskier for me to apply (more of) this process component or to refrain from applying it?*

3.1.4.4 Step-by-step strategy

Basic elements that provide value for any process model

One strategy is to first implement some enabling practices and other process elements that bring benefits to any development process. These include:

- More thorough unit testing and code quality assurance so that there is a solid base for changing the product.
- Continuous integration so that there is always a working product at hand to assess how it behaves and to learn from it.
- Test automation. Automating as many of the tasks as possible that are required for efficient verification and validation, yet supporting many test paradigms (including exploratory testing) for diversity and test effectiveness.
- Agile process automation – automated reporting tools and information systems that can easily be tailored to changing situations in a project.
- Automated tracing. When a change to a requirement, design or implementation is being planned, or has been made, tools are needed to see immediately what it affects and what else needs to be addressed. Good information systems provide this functionality.

Make rules clear

Agile requires clear rules to be successful – freedom is always best within constraints. The following need to be defined:

- Who owns the products, the product business and safety?
- Who owns the development process?

- What are the principles of product development – is it customer-led or innovation-led? Both may require different processes and participation.
- How do we weight the stakeholders' opinions? Is product development a democratic process or is some party clearly leading it?
- How open do we really want to be towards the customers?
- How much do we really wish to engage subcontractors?

Communication and collaboration

As agile is about communication and people working together, constantly learning, a proper environment needs to be created for that to happen.

- More self-reflection is needed in the process. All participants need to look into the product and process frequently to see what, in the way of action, needs to be improved. This calls for frequent lessons learned / retrospect meetings in the process.
- Team-building and improved team leadership. Teams need to able to work as teams and learn that; learning to lead teams in more collaborative ways takes time.
- Improved communications tools. Anything that helps people to communicate better yet provides a peaceful environment for those tasks and for persons who will benefit from it.
- Improved communication between distributed sites (including any subcontractor sites). No party should be left in a secondary role in communication.

Make time-consuming phases efficient

Validation and, especially, external certification, can be time-consuming and thus difficult to implement in the agile process. The processes therefore need to be developed so that validation and certification are as efficient as possible.

- Creation of external processes outside the incremental main development process.
- Clear understanding of the times at which the validation-related tasks need to be carried out.
- Making a very clear distinction between safety-critical and other requirements, and SIL levels, so that the elements requiring validation or re-validation can be identified exactly.
- Efficient information systems so that any paperwork or routines or finding and checking of development records do not take time.
- Flexible arrangements for validation. If external validators are used, the situation can be reassessed – could the validation be carried out internally?

Provide control

Create such process features that help in controlling and bringing possible problems to light as soon as possible.

- Split requirements into smaller items, aiming for a similar size to aid planning and estimation.
- Improve meetings – make them more frequent, get all people to participate; develop meeting cultures.
- Create dashboards and information systems that visualize the project's status to everyone.

Split the process

Any development process can be split into more independent iterations. Essential factors for these:

- A chain of events from planning what to do to evaluating what has been achieved.
- Aiming for a demonstrative whole at the end of each increment.
- True planning in the beginning of an increment.
- Using real product development practices in defining requirements. That is, the input should be something that provides value: new use case, user story or similar but NOT a technical specification or a change request. Change requests belong in the domain of maintenance not in the domain of a new product development lifecycle.

Increase the number of releases

Increasing the number and frequency of releases can be gradual, and in safety-critical development it should probably be so.

- Practise making release plans that are rougher and based on value and risk, not just technology and functions.
- The increment-based lifecycle can be exercised during implementation – select features to implement and test during an increment and in the second phase, and transfer more design work inside the increments. At the same time, the practices of architecture design need to be developed so that they can also evolve sufficiently.
- Development of release processes and practices so that all steps leading to the customer's use of the product are as efficient as possible yet do not compromise safety in any way.
- When the processes are in good shape, releases can be added in a controllable manner if there is benefit to be gained.

Add any other practices that bring value

There are and will be many good practices in the agile culture. The idea is not to implement all of them, just those that really bring value. For example, while pair programming – an important old agile practice – seems to have lost its appeal slightly recently, it could be a tool to transfer practical knowledge and experience from older developers to the new developer generation.

Nevertheless, any mature software development should not restrict its views to just one paradigm. The current agile culture did not just emerge: it is a result of process evolution, of combining ideas and practices into a new whole. In the same way, processes in organizations should evolve by making changes that provide quality or performance improvements to projects. Agile has been called a dumping ground for practices, and companies cannot afford that.

Therefore, for new process development ideas, many relevant areas should be looked into, not just agility. The others include:

- New project development
- Software science
- Safety analysis and safety and risk management
- Quality management
- Testing
- Information management
- Communication
- Process control.

Continuous improvement

Continuous improvement of the process. Keep improving the process and meeting any problems one at a time.

The agile processes that are applied need to be developed further, just like any processes. Processes are only optimal in a given context, and learning changes that – not to mention external influences from technology, changing standards and so on.

Continuous improvement should have an important role in any organization.

3.1.5 Conclusions

There are many reasons why agile development could be especially beneficial to safety-critical development:

- Customers need time to understand all safety repercussions, and early releases allow that.
- Communicating design and safety information is easier, as agile emphasizes verbal communication, not just written documentation.
- If a new technology should prove unreliable, the situation can be detected early and changes made.
- Risk analysis has a gradually evolving scope (the concept), which should improve its quality.
- Safety assessments made in small increments can be more focused, delivering better results.
- A rhythmic process of integration – at all levels, from code to customer production systems – should reduce many kinds of problems.
- Even if the increments are not aimed at production, their behaviour can be well tested, assessed and understood in simulation. Any corrective measures can be based on practical observations from executing the system, not just the analysis.
- Agile can provide a good learning experience for all involved, if applied properly.

Agile development may have some potential obstacles, however, including the high requirements for verification and validation, safety assessments and similar tasks that are required to approve a safety-critical system to market. Traditionally, the agile processes have been used mostly in non-safety-critical situations and the outlook on processes has been very different to what the safety-critical development requires:

- Priority on delivery <> priority for safety and reliability
- Focus on customer 'wants' <> focus on customer 'needs' identified by careful analysis
- Technical critical point implementation <> Technical critical point validation
- Communication based on informality <> communication needs objective information on use and safety needs.

The obstacles can be overcome with a good strategy and professionalism, however, and in many cases agile development can become a very fruitful approach to developing safety-critical software.

References

Ohjelmaturva publications:

Paalijärvi, J. 2010. Development of Safety-Critical Software using Agile Methods. Master of Science Thesis, Tampere University of Technology, May 2010. 54 pages.

Paalijärvi, J., Katara, M., Karaila, M. & Parkkinen, T. 2010. Agile development of safety-critical software for machinery: A view on the change management in IEC-61508-3. In: Proceedings of the 6th International Conference on Safety of Industrial Automated Systems SIAS 2010, Tampere, Finland, 14–15 June, 2010.

Vuori, M. 2011. Agile Development of Safety-Critical Software. Tampere University of Technology. Department of Software Systems. Report 14. 95 p. Available at: http://urn.fi/URN:NBN:fi:tty-2011061414702 (This publication contains the full report of the study presented in this chapter.)

Vuori, M., Virtanen, H., Koskinen, J. & Katara, M. 2011. Safety Process Patterns In the Context of IEC 61508-3. Tampere University of Technology. Department of Software Systems. Report 15. 128 p. Available at: http://urn.fi/URN:NBN:fi:tty-2011061414701

Other references:

5th Annual State of Agile Development Survey. 2010. Final summary report. November 7, 2010. Version One. 33p. Available at: http://www.infoq.com/news/2010/08/state-of-agile-survey

Boehm, B. & Turner, R. 2003. Balancing Agility and Discipline – A Guide for the Perplexed. Addison–Wesley. 266 p.

Coplien, J. 2011. Agile 10 years on. InfoQ. This article is part of the Agile Manifesto 10th Anniversary series that is being published on InfoQ. [Referenced 11/03/2011] Available at: http://www.infoq.com/articles/agile-10-years-on

Coplien, J.O. & Bjørnvig, G. 2010. Lean Architecture: for Agile Software Development. Wiley. 376 p.

IEC 61508 FAQ. Edition 2.0. 2011. International Electrotechnical Commission. [Referenced 23/03/2011] Available at: http://www.iec.ch/functionalsafety/faq-ed2/

IEC 61508-5 ed2.0. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 5: Examples of methods for the determination of safety integrity levels. International Electrotechnical Commission. 97 p.

Kennaley, M. 2010. SDLC 3.0. Beyond a Tacit Understanding of Agile. Towards the next generation of Software Engineering. Fourth Medium Press. 280 p.

Kruchten, P. 2011. The Elephants in the Agile Room. Philippe Kruchten's Weblog. Available at: http://pkruchten.wordpress.com/2011/02/13/the-elephants-in-the-agile-room/

Larman, C. Agile and Iterative Development. A Manager's Guide. Addison-Wesley. 342 p.

Manifesto for Agile Software Development. 2001. [Referenced 01/03/2011] Available at: http://www.agilemanifesto.org/

Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A. & Succi, G. 2008. A case study on the impact of refactoring on quality and productivity in an agile team. Lecture Notes in Computer Science 5082: Balancing Agility and Formalism in Software Engineering. Springer, pp. 252–266. Second IFIP TC 2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2007. Poznan, Poland, 10–12 Oct. 2007. doi: 10.1007/978-3-540-85279-7_20

Poppendieck, M. & T. 2007. Implementing Lean Software Development: From Concept to Cash. Addison-Wesley. 276 p.

Rohunen, A., Rodriguez, P., Kuvaja, P., Krzanik, L. & Markkula, J. 2010. Approaches to Agile Adoption in Large Settings: A Comparison of the Results from a Literature Analysis and an Industrial Inventory. In: Ali Babar, M. Vierimaa, M. & Oivo, M. (Eds). Product-Focused Software Process Improvement, LNCS 6156, pp. 77–91. Springer-Verlag Berlin Heidelberg.

SFS-EN 62061. 2005. Safety of machinery — Functional safety of safety-related electrical, electric and programmable electronic control systems. Finnish Standards Association SFS. 198 p.

Wikipedia. Agile software development. [Referenced 08/05/2011] Available at: http://en.wikipedia.org/wiki/Agile_software_development

Wikipedia. Safety Integrity Levels. Available at: http://en.wikipedia.org/wiki/Safety_Integrity_Level

3.2 Safety process patterns in the context of IEC 61508-3

Matti Vuori, Heikki Virtanen, Johannes Koskinen & Mika Katara, Tampere University of Technology, Department of Software Systems

3.2.1 General

Standards can be difficult to comprehend and implement in practice. This is due to many factors, such as the generic nature of standards in using concepts and vocabulary of any particular context and also the specific nature of the standards, which makes them refer to and acknowledge only the issues that they have been authorized to tackle – the idea being that there are other standards for other issues. Safety-related standards can thus be difficult to grasp and the IEC 61508 [2010] series is no exception. While one expert in a company may have the time and capability to understand fully the standard, it needs to be communicated to others so that it is practised in projects and other day-to-day activities. Some external help is clearly required. Training is one route, but even that needs more understandable descriptions to communicate the issues.

A process pattern is a concept that aims to present important aspects of an activity with a modular expression that can become familiar to personnel. In fact, the pattern descriptions highly resemble the description used in many companies, such as:

- Process description cards used as instructions
- Templates of use cases used in software development.

In the Ohjelmaturva project, we have therefore researched the use of safety process patterns to help in the use the IEC 61508 standard series (2nd edition) [2010] and, especially, its third part [IEC 61508-3 2nd ed. 2010], which concerns software development.

The main publication of the research is the report 'Safety Process Patterns in the Context of IEC-61508-3' [Vuori et al. 2011], the main contents of which we present in this chapter.

The report presents: a) some ideas behind the patterns aimed at giving guidance to future pattern developers, and b) a preliminary pattern collection. The patterns presented in this report do not form a complete collection of all the necessary patterns, nor do they cover all the aspects of the standards but present a view of the standards that, in our opinion, does not conflict with the standards and can greatly aid in their understanding and use. Note that the collection has not yet received proper evaluation, and it is mostly intended to be a prototype and a starting point for describing the standard-related issues with patterns in a particular context. Note that this chapter mostly addresses issues of traditional V-model-based development. For an analysis of the way the requirements of the standards could be fulfilled in an agile development process, see the chapter presenting that topic and Vuori [2011].

3.2.2 Some background on patterns in software development

Patterns are recurring structures or relationships between elements. The concept is used to try to understand and share the understanding of complex phenomena both in human actions and in technological systems. They are developed by examining existing or described activity and detecting the pattern by analysis. Patterns use a concise ‘pattern’ language that describes the defining elements of the pattern in a generic form. The elements include name, context, solution, resulting context and other information. Patterns have been developed for many purposes since the 1990s:

- Organizational patterns have been developed to make organizational structures and behaviour visible also in software development organizations, including agile development [Wikipedia. Organizational patterns.].
- Software design patterns have an advanced understanding of software design and architectures [Wikipedia. Category: Software design patterns.].
- Use cases are a very important use of the pattern philosophy [Wikipedia. Use case.]. A use case is a recurring element in every software development project – many are identified and presented in a standard way.
- Process patterns capture, among other things, software development issues ([Ambler 2011] has built a nice website around them).
- Project pattern research has included studies of global software development projects [Välimäki et al. 2009].
- Communication and knowledge sharing in the context of software engineering (see Vesiluoma [2009] whose dissertation also contains plenty of information about patterns).

Thus, patterns have evolved into a proven tool to understand a domain’s activity and issues and to externalize and share knowledge. Patterns use a concise presentation consisting of a short description of key elements. This same principle has been used in many organizations as a template for process instructions, giving the instruction a generic, standardized form and short length (optimally one page) and thus better understandability than traditional, longer instructions. Thus, patterns have great potential to be used in companies’ processes.

3.2.3 Purpose of safety process pattern collection

The goal is to help organizations improve product development and make safer systems as, with the help of the patterns, they could: 1) understand the standards’ requirements better, and 2) understand how the requirements show in practical software development work. The patterns can be used for many purposes:

- Description of development processes and safety lifecycle processes in an understandable way.
- Frequently Asked Questions – they should provide answers to common needs of understanding how a process or task should be carried out.
- Presentation of generic workflows to be used as design and tailoring of workflows in companies and in evaluating current practices.
- Explaining informal appendices to the official standards.
- A basis for an organization’s understanding of its own activities and a description of those, to enable communication, training and development of practices.

It should be noted that the patterns are just models and should *not* contain all the details of the safety standards but an overview of them and links to the relevant parts of the standards. If the patterns aimed for completeness, organizations would trust them too much and neglect understanding of the actual standards.

3.2.4 Structure and contents of a safety process pattern and the pattern collection

The patterns use a standardized structure presented in Table 1.

Table 1. Structure of a safety process pattern.

Element	Description
Name	A unique name for the pattern. Describes what the pattern is about.
Context	The context describes the initial situation in which the pattern is thought to be applied (i.e. where the steps should be performed). It may also have preconditions or requirements that have to be fulfilled before the pattern can be applied. Such a requirement could include, among other things, a reference to some other pattern.
Problem	The patterns try to solve a problem, which is described in this section. In our patterns, the problems are usually given in the form 'How to...'.
Forces	The forces section presents the reasons for applying the pattern. The forces do not discuss the solution, but after defining the forces, the solution is usually more or less obvious and easy to adopt.
Solution	The solution defines the steps that should be followed to solve the problem. The steps cover the requirements defined in the standard. For example, if the standard requires documentation to be written, the solution will have a step corresponding to that requirement. The solution should cover the forces defined earlier in the forces section. There is usually a picture – a diagram, picture of a process flow, a mind map or similar.
Resulting Context	The resulting context is the new context that is achieved after the pattern has been applied. It describes what has been achieved by applying the pattern.
Related Patterns	References to other patterns that are related to this pattern: often patterns whose execution precedes this pattern or starts after this pattern; patterns that are similar to this one; or patterns that are more detailed implementations of a more abstract pattern.
Standard References	References to the clauses in the IEC 61508 standard series that explain the elements of the pattern or give requirements for its execution.
Authors	Who has written the pattern.
Status	During development, acceptance status. During the rest of the pattern lifecycle, update information.
Notes	Freeform notes and links to more information, for example, Wikipedia pages.
Tags	Classification tags. Any number of tags that help classify the pattern.

The elements, fields, 'standard references', 'authors', 'notes' and 'tags' are not usually found in patterns in literature, but extensions are important in the practical long-term use of the patterns. The number of elements should be kept as small as possible however. Adding more elements will at some point make the patterns worse, not better. The information regarding applicability will therefore be included in the 'notes' field, as necessary.

The patterns can belong to the following main types:

- Generic organizational patterns – generic principles and practices in an organization in projects and in its overall activities and processes.
- Generic process and product control patterns. Patterns that are repeated during a project many times and are implemented during the development lifecycle or the safety lifecycle many times. These are process issues that provide a solid context for development.
- Development approaches and technologies. These include technology choices, etc. that are applied in the context of the control patterns and during the lifecycle and process patterns.
- Patterns for individual phases of the safety lifecycle or the software development lifecycle. These may be carried out only once in a project.

3.2.5 Patterns included in the collection

The following patterns are included in the collection presented in this report and fully described in Vuori et al. [2011]. The subheadings starting from 'Software Safety Requirements Specification' correspond to chapters of IEC 61508-3 [2nd ed. 2010].

Table 2. Patterns introduced in this collection.

Name	Problem
Generic organizational patterns	
Multiple Viewpoints	How can the viewpoints that project participants represent in the project be identified?
Understand Cultures in Co-operation	How can we co-operate in a multi-cultural project so that cultural differences are managed so they do not endanger safety?
Assign Roles and Responsibilities	How can we select project participants and assign roles and responsibilities so that the use of expertise and the required independence are in optimal balance?
Diversity in Team Practices	How can the principle of diversity be applied to all tasks?
Competence Management	How can we ensure that each member has the required competence?
Continuous Communication	How can we obtain rapid input from everyone in a way that does not slow progress but makes the project proceed more efficiently? How can we get busy professionals to participate in the process?
Transparency of Action and Information	How can we know what is actually being done and whether there are any problems?
Anti-pattern: Information Hiding	We know that we have problems and are ashamed of it. How can we hide the situation until a miracle happens and the problem is solved? How can we suppress information so that it will not be leaked to competitors or the media?
Generic process and product control patterns	
Phase Workflow	How is a process phase carried out while satisfying safety lifecycle process requirements?
Verification of a Work Product	How can we verify that a work product meets its requirements?
Split and Manage Details	When a work product, like a module or a specification, is being developed, how can we ensure that work can be carried on in parallel, so that we can address issues independently, verify each small task and see the state of the whole?

3. Development process of safety-related software

Name	Problem
Single Development Task Control Workflow	How can we have a simple, generic workflow that allows tracking of the completion of single tasks and progress of a set of tasks?
Acceptance of Phases and Tasks	How are work products accepted in the development process so that they can be used safely?
Configuration Management	How can we know which elements, configuration and version the assessed software system consists of and what has changed compared with a baseline?
Forward Tracing	How can we link activities and development items so that we can find out, at any time, what actions are planned and have been carried out regarding the items? How can we follow the chain from any requirement to its verification and testing in an easy way? If some aspect of the work product changes, how can we know what items in the following process phases are invalidated due to that change?
Backward Tracing	How can we know what requirements, plans or instructions our work is based on and must thus be verified against?
Suspect and Prohibit	How can practical suspicion be supported and converted into practical action? How can we know what – if anything – should be changed based on our suspicion?
Escalation of Issues	How can the issue be raised and handled at an appropriate level in the project or line organization?
Use of Checklists	How can we remember all the issues that need to be checked? How can we be sure that others remember all the issues that need to be checked?
Continuous Improvement	How can ways of action be improved so that future projects are more efficient and have fewer problems?
Development approaches and technologies	
Flow Between Design Levels and Tests	While using a controlled approach, how can a constant flow of testing ideas be supported?
Selection of Methods / Techniques	How can methods and techniques be selected so that the decision leads to ways of action that result in a safe system, fulfil the requirements of the IEC 61508 standard series and are justified before the project begins as well as afterwards?
Use of Formal Methods	How are formal methods introduced into an organization?
Selection of Support Tools and Development Languages	How is a set of support tools and languages selected that fulfils safety requirements and can be proven to produce reliable results?
Software Safety Requirements Specification	
Software Safety Requirements Specification	How are safety requirements specified for the software system?
Software Design & Development	
Software Development	How is a software system created that fulfils the specified requirements with respect to the required safety integrity level?
Software Architecture Design	How is software architecture created that fulfils the specified requirements with respect to the required safety integrity level?
Software Architecture Verification	How can we ensure that the software architecture design adequately fulfils the software safety requirements specification?
Technical Diversity	How is a system created so that no more than one element of it fails due to a disturbance? Or, how can common cause failures be avoided?
Formal Methods Aided Design and Verification of Joint Behaviour	How should the parts (components) defined in the architecture behave in order to obtain the expected behaviour of the total (sub)system?

Name	Problem
Software System Design – general	How can the software be designed so that it can be implemented, verified and validated?
Software System Design Verification	How can we ensure that there are no incompatibilities between the software system design specification and the software architecture design?
Generic Glue	How can we be sure that the given requirements for the next phase are well defined, i.e. complete and do not contain any contradiction?
Detailed Module Design	How can an individual software module be developed so that it can be implemented safely and reliably?
Glue Design and Implementation	How can we be sure that the design is correct, detailed enough, and implemented with reasonable effort and without extra design decisions?
Coding	How can reliable and safe program code be created that is easy to modify when the need arises and also easy to audit – for the purpose and for safety and security?
Analytic Design and Code Quality Assessment	How can reliable and safe program code created that is easy to modify when the need arises and also easy to audit – for the purpose and for safety and security?
Verification Testing	How are programs and their components verified at all abstraction levels?
Module Testing and Simulation	How can the module be tested to verify that it meets the requirements?
Module Integration Testing	How is the collection of modules tested in the architecture to verify that the system meets the functional requirements?
PE Integration Testing	How is the integrated system tested so that its functioning and functional safety can be verified?
Regression Testing	A change in software can lead to problems in other parts of the system. To identify those effects, regression testing is used.
Model-Based Testing	Designing test cases that cover the specification and are easy to maintain when the artefacts are modified requires much effort when it is done manually.
Software Aspects of System Safety Validation	
Software Validation Planning	How can a plan for validating the safety-related software aspects of system safety be developed?
Software Validation	How can we ensure that the integrated system complies with the software safety requirements specification at the required safety integrity level?
Configuration Auditing	How can we assess how the system differs from the last validated baseline?
Software Modification	
Software Modification Planning	How can the modification of the software be planned so that the modification activities can be performed safely and the resulting product be fully understood and validated?
Software Modification	How can we ensure that the required software systematic capability is sustained when the validated software is modified?
Impact Analysis	How can we best assess how a proposed change impacts the system?
Functional Safety Assessment	
Functional Safety Assessment	How can the functional safety of software be assessed analytically?
Failure Analysis	How can we analyse software errors and system failures in order to prevent them reoccurring? How can we understand how the system handles failures and whether it does it properly? How can we understand how failures propagate through the system?

Software Operation & Maintenance Procedures	
Writing of the Safety Manual	How can we communicate our knowledge of safe use to the users?

3.2.6 Examples

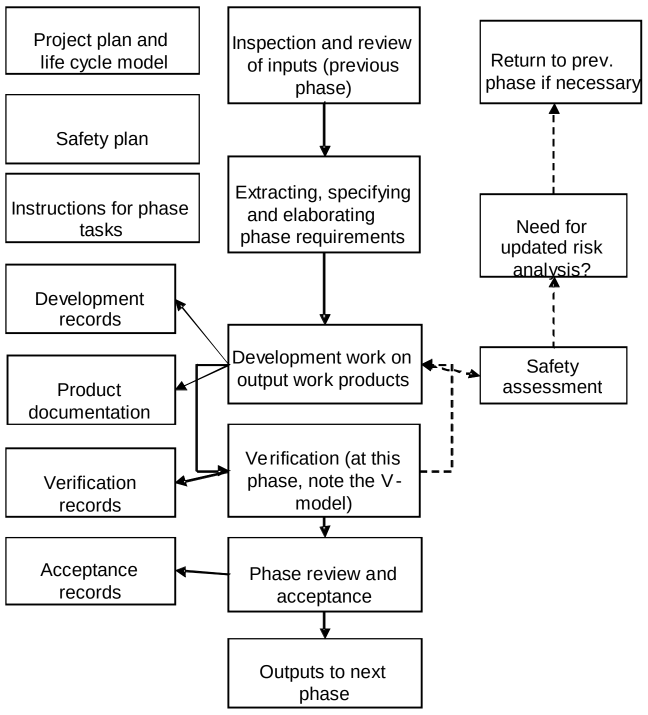
The following three examples of pattern types are included in the collection:

- Phase Workflow is an example of a process workflow pattern.
- Assign Roles and Responsibilities presents an approach to a situation and contains a mind map.
- Software Validation Planning. This is a pattern that combines a workflow mindset and a safety-conscious understanding of issues and has many references in the IEC 61508 series [2010].

3.2.6.1 Phase Workflow

A phase is an important building block of any software development lifecycle and is quite strictly influenced by the IEC 61508 [2010] requirements. It is therefore a natural application for process patterns.

As the name implies, the pattern contains a process workflow that is shown in a simplified form suitable for explaining to various interest groups in training, auditing and other situations.

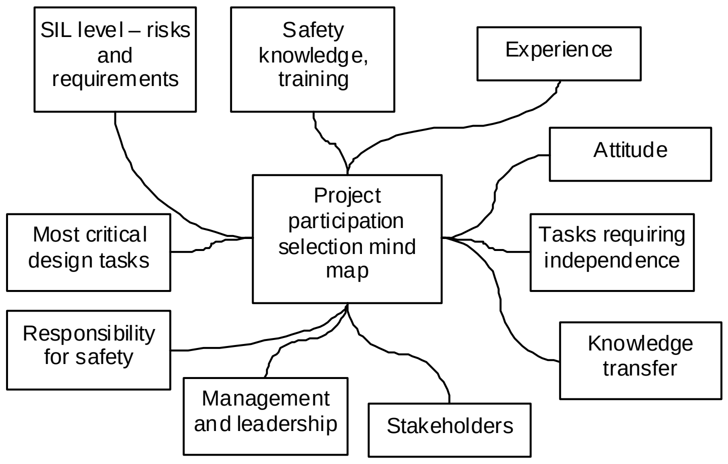
Name	Phase Workflow
Context	A development phase is started after a previous one has been completed.
Problem	How is a process phase carried out, satisfying safety lifecycle process requirements?
Forces	Each phase needs to implement the safety management principles and tasks that the IEC 61508 [2nd ed.] series requires, as they are seen to be critical to the process to produce a safe system.
Solution	A generic work flow:  Project plan and life cycle model Safety plan Instructions for phase tasks Development records Product documentation Verification records Acceptance records Inspection and review of inputs (previous phase) Extracting, specifying and elaborating phase requirements Development work on output work products Verification (at this phase, note the V-model) Phase review and acceptance Outputs to next phase Return to prev. phase if necessary Need for updated risk analysis? Safety assessment Critical elements of the process: • Inputs need to be inspected and reviewed. By inspection we mean a thorough analysis of, for example, the requirements, and by review we mean reaching a consensus on inputs that are flawless, for the purpose of the phase. • Guidance is provided by project-level plans and task-specific instructions. Adherence to plans is checked in reviews. • All safety-related tasks are documented by records. • During the work, mostly analytic verification is carried out, but testing will take place later – note the V-model as a framework. • For most work products, safety is assessed and the work product corrected, as required. • There is always a feedback loop to the previous process phases. • As development produces new information about the use of the product, it needs to be assessed whether hazard or risk analyses need to be updated. This may necessitate updating of many requirements. • All items under work and work products are configuration controlled. This includes documentation. • Before transferring outputs to the next phase they need to be reviewed and accepted. This internal acceptance must not be confused with validation.

Result-ing Con-text	A successfully carried out process phase providing solid output for the next phase and next develop-ment tasks.
Related Patterns	Verification of a Work Product Acceptance of Phases and Tasks Configuration Management
Standard Refer-ences	IEC 61508-1 (2nd ed.) presents the generic process requirements IEC 61508-3 (2nd ed.) explains how this process is implemented in software development tasks
Authors	Matti Vuori
Status	Version 29/04/2011
Notes	
Tags	workflow, phase, process

3.2.6.2 Assign Roles and Responsibilities

The assignment of roles and responsibilities does not take place often in a project, but during a company's lifecycle it obviously happens tens or hundreds of times. It is also a critical task to understand and carry out, as achieving good safety requires good competencies. This pattern does not contain a workflow, however, but a mind map, describing the criteria and thinking to be applied to the assignment situation.

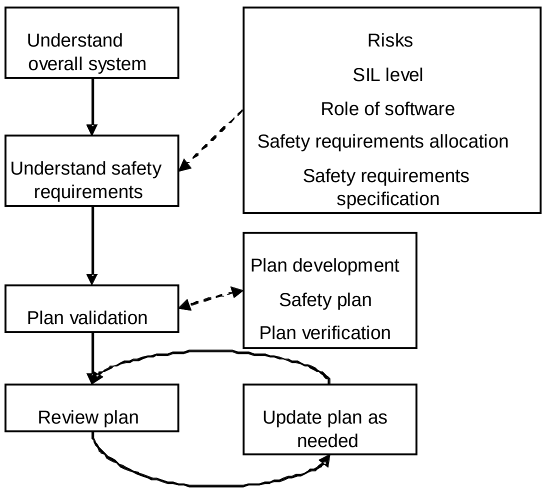
Name	Assign Roles and Responsibilities
Context	A safety-critical development project is being planned. Participants, roles and responsibilities need to be assigned to individuals.
Problem	How can we select project participants and assign roles and responsibilities so that the use of expertise and required independence are in optimal balance?
Forces	When a project starts, a requirement for safety-critical development is an explicit assignment of responsibilities and thus also roles.
Solution	Key points in this process: • Understand the mandatory requirements for participants based on the SIL level and other project requirements.. • Identify the need for independence in verification and validation and select individuals for those tasks. They cannot have a role in development tasks. • Define who is responsible for safety and accepts the project's products. • The challenging parts of the project require experience and skills. Assign the most capa-ble people to the challenging tasks. • At high SIL levels, safety attitudes have great importance in team selection. • Support organizational learning by combining various levels of expertise. • Consider knowledge transfer with other units and subcontractors when selecting team members. • Consider employing external experts for added competence even when independence is not a requirement. • If external validation (perhaps leading to certification) is required, plan a good way to in-clude that the party is in the process from early on.

	• While all participants should have generic safety-related knowledge and skills, project-based training should always be considered. The definition of roles and responsibilities does not mean that the development needs to be bureaucratic, it just means that we know that someone will concentrate, especially on the issues and on who can help others to do their tasks better.  SIL level – risks and requirements Safety knowledge, training Experience Attitude Project participation selection mind map Most critical design tasks Tasks requiring independence Responsibility for safety Knowledge transfer Management and leadership Stakeholders
Resulting Context	A project organization is formed in which everyone understands what is expected of him/her and what he/she can expect from others. A good starting point for flexible collaboration.
Related Patterns	Multiple Viewpoints
Standard References	IEC 61508-1 (2nd ed.), subclause 6.2.13 describes competence requirements for project personnel IEC 61508-1 (2nd ed.), subclause 6.2.13 describes criteria for appropriateness of competence
Authors	Matti Vuori
Status	Version 2011-04-29
Notes	Wikipedia article Project governance describes project roles related issues http://en.wikipedia.org/wiki/Project_governance
Tags	project, planning, organization, roles

3.2.6.3 Software Validation Planning

Once again, this is something that happens only once in a project but requires careful thinking so that the projects can be effective and efficient. This is a pattern that combines a workflow mindset and a safety-conscious understanding of issues, and it has many references in the IEC 61508 series [2010].

Name	Software Validation Planning
Context	Software safety requirements specification has been finalized.
Problem	How is a plan developed to validate the safety-related software aspects of system safety?
Forces	Validation is a task that needs to be a planned activity so that the plans can be assessed to meet the requirements of IEC 61508 and the validation can then be compared with the plan to see that it has been carried out properly. The validation of software is also done in the context

	of the overall system.
Solution	The main process: • Understand the overall context and the system and the software's role in it. • Understand the validation requirements based on the project's SIL level. • Make a clear distinction in all plans between the validation of the safety requirements and the validation of other product requirements. • Decide on the parties who do the validation, considering the required independence (for example, independent company unit, external validator) and need for certification. • Create an overall safety plan that ensures that the development and safety assurance process is sufficient. • Plan all verification steps so that they ensure that the validation will proceed smoothly. • Plan the validation, leaving sufficient calendar time for its activities. This is usually in a form similar to a project plan. Do this in close collaboration with the party that will be doing the validation. • Consider in the plans that the validation process may not pass the first time and thus changes may need to be made and validation repeated. • Plan some co-ordinated collaboration with the party doing the validation so that the development process can be guided in a positive direction (yet maintaining independence of the validator). • Review the plan with all stakeholders and ensure that everyone understands the criticality of the validation – without it the product cannot be taken into use.  Understand overall system Risks SIL level Role of software Safety requirements allocation Safety requirements specification Understand safety requirements Plan development Safety plan Plan verification Plan validation Review plan Update plan as needed
Resulting Context	A planned validation process that can be executed when the product is ready for validation.
Related Patterns	Software Validation
Standard References	IEC 61508-1 (2nd ed.), Clause 7.14 describes the safety validation requirements. IEC 61508-3 (2nd ed.), Clause 7.7 defines the process for system validation. IEC 61508-3 (2nd ed.), Table A.7 presents recommended techniques for software aspects and properties of system safety validation at different SIL levels. IEC 61508-3 (2nd ed.), Table C.7 describes the strictness of various ways of application of the software aspects and properties of system safety validation.

Authors	Matti Vuori
Status	Version 2011-04-29
Notes	While in the ‘ideal world’ the validation plan should be based on stable requirements, things change and evolve and thus the validation plan needs to be updated as well during the development process. See Wikipedia article Verification and Validation http://en.wikipedia.org/wiki/Verification_and_validation
Tags	validation, software system, software aspects, overall system

References

Ohjelmaturva publications:

Process Patterns for Safety Critical Software. [Referenced 04/05/2011] Available at http://sites.google.com/site/safetypatterns (This is a TUT developed site for collecting safety process patterns based on IEC 61508.)

Vuori, M. 2011. Agile Development of Safety-Critical Software. Tampere University of Technology. Department of Software Systems. Report 14. 95 p. Available at: http://urn.fi/URN:NBN:fi:tty-2011061414702 (This publication contains the full report of the study presented in this chapter.)

Vuori, M., Virtanen, H., Koskinen, J. & Katara, M. 2011. Safety Process Patterns In the Context of IEC 61508-3. Tampere University of Technology. Department of Software Systems. Report 15. 128 p. Available at: http://urn.fi/URN:NBN:fi:tty-2011061414701 (This publication contains the full description of the pattern collection.)

Other references:

Ambler, S.W. 2011. The Process Patterns Resource Page. [Referenced 04/05/2011] Available at: http://www.ambysoft.com/processPatternsPage.html

IEC 61508 ed2. 2010. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. International Electrotechnical Commission. 1000 p.

IEC 61508-1 ed2.0. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 1: General requirements. International Eletrotechnical Commission. 127 p.

IEC 61508-3 ed2.0. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements. International Electrotechnical Commission. 234 p.

Välimäki, A., Kääriäinen, J. & Koskimies, K. 2009. Global Software Development Patterns for Project Management. Proceedings of EuroSPI 2009, CCIS 42, pp. 137–148.

Vesiluoma, S. 2009. Understanding and Supporting Knowledge Sharing in Software Engineering. Tampere University of Technology, Publication 843. 158 p. Available at: http://URN.fi/URN:NBN:fi:tty-2011061414701

Wikipedia. Category: Software design patterns. [Referenced 04/05/2011] Available at: http://en.wikipedia.org/wiki/Category:Software_design_patterns

Wikipedia. Organizational patterns. [Referenced 04/05/2011] Available at: http://en.wikipedia.org/wiki/Organizational_patterns

Wikipedia. Process patterns. [Referenced 04/05/2011] Available at: http://en.wikipedia.org/wiki/Process_patterns

Wikipedia. Project governance. [Referenced 29/04/2011]. Available at: http://en.wikipedia.org/wiki/Project_governance

Wikipedia. Use case. [Referenced 04/05/2011] Available at: http://en.wikipedia.org/wiki/Use_case

Wikipedia. Verification and Validation [Referenced 29/04/2011]. Available at: http://en.wikipedia.org/wiki/Verification_and_validation

4. Phases of software development

Section 4 describes the most important phases of software development. Some phases have been given more attention here because they had more focus during the project, and some topics, which are important with respect to software safety, are considered in more detail here. The structure of this section resembles the V-model (Figure 16). Each phase is considered as a separate item and (therefore) the text is also applicable to other design models than the V-model.

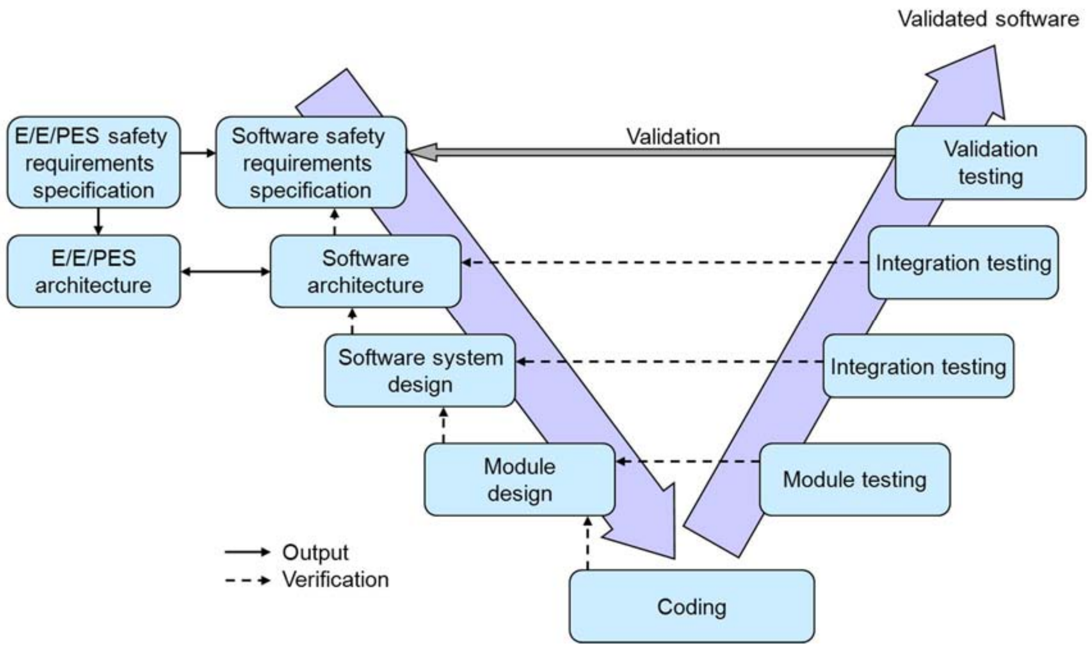


Figure 16. V-model [IEC 61508-3 2010].

4.1 Safety requirements specification and safety principles

Timo Malm, VTT

The safety requirements specification is a very critical phase of software development. A large proportion of the defects is made during this phase, and these defects are the most difficult to reveal. Since the safety requirements specification is drawn up at the beginning of the software development, modifications can be expensive if a defect is found at the end of the development process.

The standards present requirements related to the safety requirements specification and its development. All of the standards refer to risk assessment, though there are also other sources of safety requirements. As so many defects are related to the requirements specification phase of software development, it is important to gather information about requirements and risks from many different sources. The standards related to functional safety (IEC 61508 [2010], SFS-EN 62061 [2005] and SFS-EN ISO 13849-1 [2008]) claim that the following two requirements must be specified:

- Functional requirements specification. This is related to, among other things, performance criteria (e.g. response time), interfaces and descriptions of functions.
- Safety integrity requirements specification. The required safety integrity level (SIL) or performance level (PL) must be defined for each safety function.

The requirements of standards IEC 61508-3 [2010], SFS-EN 62061 [2005] and SFS-EN ISO 13849-1 [2008] resemble each other, but IEC 61508-3 [2010] specifies the requirements in detail. It also presents several lists of requirements, which need to be specified if applicable.

4.1.1 Sources of requirements

The standards related to functional safety concentrate on requirements that originate from risk assessment (according to the safety lifecycle). The identification of hazards is not well described, and each standard has its own method for the classification of hazards. The items to be defined are severity, exposure time, possibility of avoiding occurrence and probability of unwanted occurrence. The safety requirements in all standards originate from known risks that are evaluated, and the related requirements are documented in standards. Although all safety requirements could be concluded from risks, it is good to use other sources too. For example, the component supplier may know the specific risks that are formulated for the requirements. Figure 17 shows sources for the safety requirements specification.

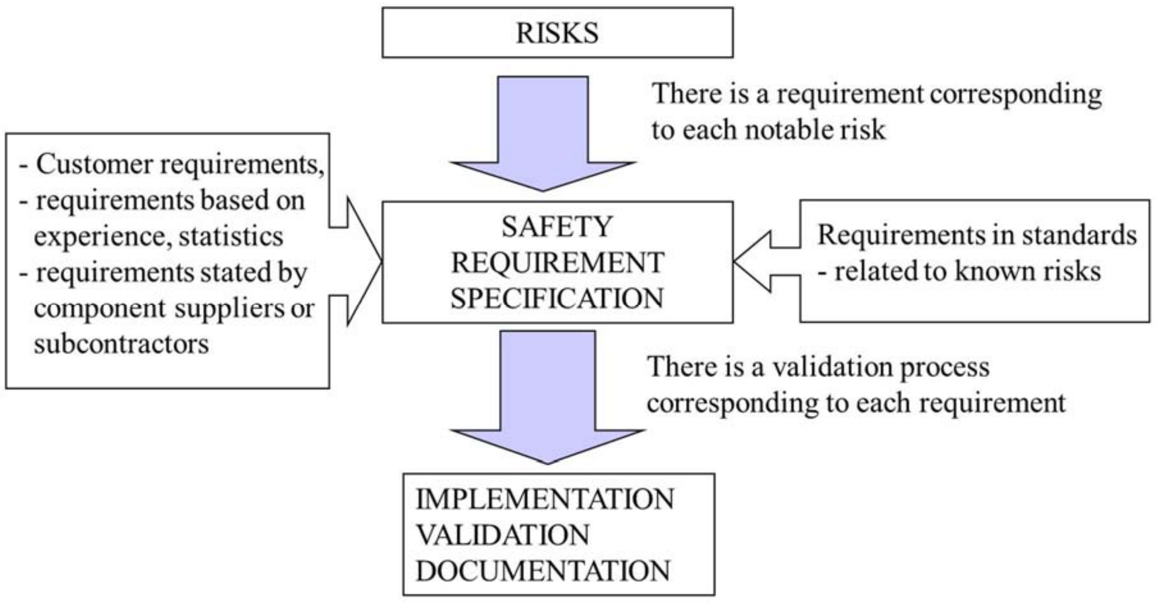

RISKS
There is a requirement corresponding to each notable risk
- Customer requirements,
- requirements based on experience, statistics
- requirements stated by component suppliers or subcontractors
SAFETY REQUIREMENT SPECIFICATION
Requirements in standards
- related to known risks
There is a validation process corresponding to each requirement
IMPLEMENTATION
VALIDATION
DOCUMENTATION

Figure 17. Sources of requirements.

The NASA Guidebook [NASA Software Safety Guidebook 2004] mentions as sources of requirements: system requirements (specification), safety and security standards, hazard and risk analyses, system constraints (hardware and environmental), customer input and software safety best practices (generic and company requirements, etc.). With respect to risk analyses, in addition to PHA, the NASA Guidebook [2004] recommends the use of both top-down and bottom-up analysis methods (as mentioned in Section 3.3) for identifying software safety requirements.

According to the IEEE Guide for Developing System Requirements Specification [IEEE 1233-1996], the following techniques can be used to identify requirements: structured workshops, brainstorming sessions, interviews, surveys/questionnaires, observation of work patterns, observation of systems' organizational and political environments, technical documentation review, market analysis, competitive system assessment, reverse engineering, benchmarking processes and systems, simulations and prototyping.

References

IEC 61508 ed2. 2010. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. International Electrotechnical Commission. 1000 p.

IEC 61508-3. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems Part 3: Software requirements. International Electrotechnical Commission. 234 p.

IEEE 1233-1996. IEEE Guide for Developing Requirements Specifications. New York, the Institute of Electrical and Electronics Engineers. 24 p.

IEEE 830-1998. IEEE Recommended Practice for Software Requirements Specifications. New York, the Institute of Electrical and Electronics Engineers. 32 p.

NASA Software Safety Guidebook. 2004. NASA Technical Standard NASA –GB-8719.13. 388 p.

SFS-EN 62061. 2005. Safety of machinery – Functional safety of safety-related electrical, electric and programmable electronic control systems. Finnish Standards Association SFS. 198 p.

SFS-EN ISO 13849-1. 2008. Safety of machinery – Safety-related parts of control systems – Part 1: General principles for design. Finnish Standards Association SFS. 180 p.

4.2 Safety-related architectures of machinery control systems and software

Jari Rauhamäki, Tampere University of Technology, Department of Automation Science and Engineering
Timo Malm, VTT

At the time when EN 954-1 [EN 954-1 1996] was a major safety-related standard, the classification of safety-critical systems was obtained almost entirely based on the architecture of the system. Other matters have been added to safety-related standards succeeding the EN 954 [1996]. The safety levels were managed with architectures of predefined categories. Though these categories did not actually consider software, possible software was assessed in a similar way to categories such as hardware. Multiple processors typically provided natural diversity. It was later realized that two processors, and then even a single processor, might be sufficient. Although software cannot really substitute redundant hardware, it provides opportunities to decrease hardware so that redundant structures are obtained without installing a vast number of redundant processors. Program faults cannot be eliminated by architecture alone however. Architectural solutions can just increase or decrease the number of program faults and the software capability to tolerate and process them.

Program faults can occur in many different phases of the program lifecycle. Quite often, it is difficult to say in exactly which phase the fault has been made. The requirements could have been more specific or coding should have solved the problem. Some research has been carried out, but not much that estimates the phase when faults are introduced. It is possible, however, to gain a rough estimate when faults occur. Here, the considered software lifecycle phases are the requirements specification, design (architecture, module design), coding and testing.

There are several different definitions of software architecture. Here are two definitions [Haikala & Märijärvi 2006]:

- The structure of the components of a program/system, their interrelationships, and the principles and guidelines governing their design and evolution over time.
- The structure of the system. Architecture can be divided into hierarchical parts, which communicate via their interface. One part can be class, component, module or subsystem.

Architecture is often related to a textual or graphical view of the system. The views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers and project managers. The so-called 4+1 architectural view model contains the following views: physical, logical, development and process view [Wikipedia. 4+1 Architectural View Model.]. There are also specific languages that are used to describe the architecture, such as UML, SA/SD and Architecture Description Language. This text considers safety issues and, here, physical or development views can describe safety aspects adequately.

The architecture of the software affects the safety performance of the system. With a specific architecture, it is possible to make good or bad as well as safe and unsafe code. Architectural decisions can urge the designer to avoid specific errors that are often related to architectural properties. For example, in distributed systems, the isolation of functions is easier to realize than in single processor systems, as software is executed on separated processing units and there is thus no need to arrange scheduling to the same extend as in a single processing unit. In some cases, it may even be almost impossible to prove adequate safety without a specific type of architecture. Such 'difficult-to-prove' factors include, for example, adequate isolation and error detection.

One argument is that if members of the programming team cannot understand or it would take too long to understand, very sophisticated features would be useless. Important factors are simplicity, straightforwardness, modularity, locality, abstraction before programming and architectural style, i.e. implementation philosophy [Haikala & Märijärvi 2006].

4.2.1 Defensive techniques related to architectural structures

Techniques are often related to specific architectures, but they can be applied to almost any. For example, a modular approach is often needed in order to define the structure of the program. The techniques required in a project can be defined in the coding rules. For safety purposes, the applied techniques can be more important than the architectural style. Such techniques (gathered from IEC 61508-3 ed2.0 2010) are:

- **Clear structure** (modularity, structural programming, object-oriented-programming). The program is divided into smaller subprograms, which are easier to understand. The coupling between the modules is as thin as possible (defects do not spread easily to other modules).
- **Simplicity** and predictability of behaviour.
- **Fault containment regions**. These are created to prevent propagation of software faults. This can be related to firewalls preventing fault propagation from non-critical software to safety-critical components, one redundant software unit to another or one safety-critical component to another. Hardware firewalls are more reliable, but there are also software firewalls. [NASA 2004]
- **Stateless design**. No transaction is influenced by earlier transactions; specific input always results in the same associated output.
- **Avoiding difficult structures** (e.g. recursion, dynamic objects). Difficult structures can be defined in coding rules (see Table B1 in IEC 61508-3 ed2.0 [2010]).
- **Defensive programming**. E.g. variable plausibility, range and type checking.
- **Diverse redundancy** or diverse monitoring techniques. In diverse redundancy, a program is designed and implemented N times in different ways and the results are compared.

- **Failure assertion programming**. Checking and reporting the pre- and post-conditions of the program before executing it.
- **Recovery** techniques (backwards recovery, re-try fault recovery). If a fault is detected, the system is reset to an earlier proper internal state.
- **Graceful degradation**. This technique gives high priorities to the most critical functions.
- **Error detecting codes**. Codes can detect error in data. They are applied to, for example, data communication and storage when it is possible that the data may change (a large amount of disturbance). Typical codes are parity bit, CRC code, etc.

4.2.2 Architectures in SFS-EN ISO 13849-1

The standard SFS-EN ISO 13849-1 Safety of machinery – Safety-related parts of control systems [SFS-EN ISO 13849-1 2008] is used in the machinery domain. The standard considers machinery application from a safety point of view and provides requirements on the hardware and software aspects of safety-critical functionalities. From a software architecture view, the standard defines architectural frameworks that are recommended for use in safety-related machinery applications. The standard provides a designated architecture that is used as a foundation for more detailed architectures. On the other hand, the architecture can be constructed from several serial designated architectures. The standard does not support, e.g., several parallel channels (MooN systems) and such systems should be considered according to IEC 62061 [2005] or the architectural model should be fitted to a designated architecture. The reason for this is that SFS-EN ISO 13849-1 [2008] only gives the equations for quantitative calculations to one or two channel systems. The architecture of the system is related to categories, and it is the most important factor in determining the performance level (PL) of a safety function.

There are of course many requirements in addition to the architectural ones. Most of the requirements are related to a specific performance level. For example, category 2, PL=b, software differs from category 2, PL=d, software although the basic architecture can be similar. The reason is that there are more requirements for PL=d software. Most of the requirements are related to the procedure and tools for developing software, and the actual source code can therefore be similar. The difference is similar between SIL1 and SIL2 software.

The generic software architecture given in SFS-EN ISO 13848-1 [2008] is depicted in Figure 18. The architecture is a very high-level of abstraction and states very little about the system. However, the most important aspect is shown. That is, the software should have separated input and output modules from the processing module. In practice, this could mean that there are sensor and actuator abstractors in the software. The rationale behind this approach is the simplification of and focus on a core problem. The processing model can focus on implementing the required logic of the actual safety function. The input modules are responsible for providing the processing module with a simple interface to obtain measurement and other input values. Thus, the processing module can focus on the actual processing and be clearer and simpler. The rationale of the output module is similar to that of the input module, but it provides a nice interface to send out control values etc. rather than receive input values. All the details of the communication and other non-processing-related matters between other devices are hidden in the abstracting modules.

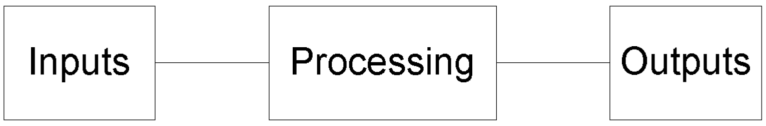


Figure 18. Generic software architecture by SFS-EN ISO 13849-1 [2008].

By using the generic architecture presented in Figure 18, practically any kind of control software can be implemented. There may be an arbitrary number of input, processing and output units forming the actual software structure, yet the main architecture complies with the standard. The generic architecture does not introduce requirements, e.g. considering the redundancy or number of channels used. These aspects are covered in architecture categories defined in SFS-EN ISO 13849-1 [2008]. Categories are logical system architectures that are used on various PLs. The architecture category presents a logical architecture of the system that serves as a foundation for system developers. The connection between software architecture and logical system architecture defined by the category lies in the interconnections of redundant and/or testing system modules within the category architecture. In practice, the generic software architecture is embedded in the logic part(s) in the category architectures (see 4.2.2.1-4.2.2.3).

4.2.2.1 Category B and 1 architecture

Categories B and 1 are the simplest architecture categories provided by SFS-EN ISO 13849-1 [2008]. Categories B and 1 are similar in structure, but category 1 requires the use of ‘well-tried components’, which is not required by category B. Programmable systems are typically not related to category 1. The maximum achievable PL for category B is PL b whereas for category 1, the maximum is PL c. Neither of the categories is well suited to safety-critical machinery applications, as the PL c application is equivalent to the SIL 1 application. The category 1 device is often a simple (over-dimensioned) push button, a limit switch or valve, which can be difficult to replace with a more complex device without changing functionality (duplication, monitoring). SIL 1/2 and PL b/c/d are typical requirements for safety-critical machinery applications.

Category B and 1 system architecture is illustrated in Figure 19. Input devices are connected to single logic, which controls the outputs. The architecture is a single channel architecture, and a fault in any of the system modules thus results in a loss of safety function. There are no redundancy requirements.

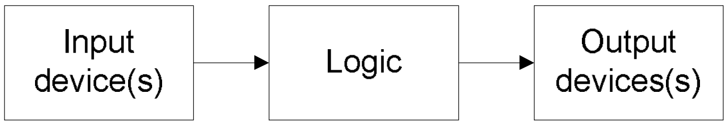


Figure 19. Category B and 1 system architecture.

As there are no requirements that consider the redundancy or testing functionality of the software architecture for category B, the system implementation is quite straight forward. The software architecture is simply embedded in the logic block of the system. This is illustrated in Figure 20.

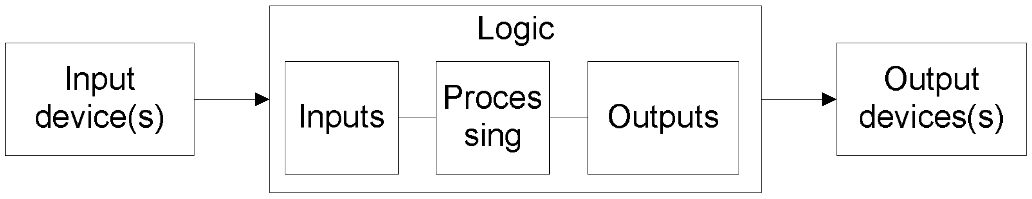


Figure 20. Software architecture in a category B system.

4.2.2.2 Category 2 architecture

The category 2 architecture is a step up from the category B/1 architecture. The main difference between category B/1 and category 2 is the testing facility introduced in category 2 architecture. The purpose of the testing functionality is to ensure that the safety function and its components are operating correctly. If a malfunction is detected, corrective measures are taken before the worst case when a dangerous failure occurs (e.g. safety function failure on demand). If the final switching element of the category 2 system fails, a dangerous failure is possible as there is no duplication (redundancy). Some redundancy may be needed if the testing functionality is implemented as recommended by the standard.

The category 2 system architecture is illustrated in Figure 21. The architecture includes a single channel safety function, presented in the upper row of the figure. The main safety function is similar to the category B architecture. The architecture also defines a testing channel. The purpose of the testing channel is to test the main safety-function-related modules periodically and for certain events (such as start-ups). Testing equipment/functionality can be implemented as part of the safety function or separated from it. This provides the possibility of running the testing functionality on the same device as the main safety functionality. It should be noted that the testing functionality needs to be able to cover all the devices and modules related to the safety function. If any module is left without testing ability, the category 2 architecture cannot be used. In such cases, category 3/4 architecture needs to be chosen if category B/1 is insufficient for the application.

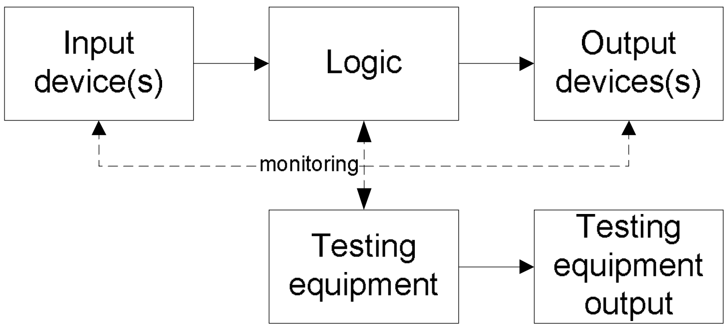


Figure 21. Category 2 system architecture [SFS-EN ISO 13849-1 2008].

The generic software architecture solution for category 2 system software architecture is provided in Figure 22. The approach does not address all the aspects but provides a possible guideline for more detailed architecture consideration. The foundation of the architecture is similar to the category B/1 software architecture. The testing functionality has been added to the architecture as required however. In this approach, the testing functionality is deployed on the same processing resource as the main safety function logic. The testing is performed through test interfaces (TestIFs). The processing module also includes a monitor that is used to monitor the testing module. The monitoring is heartbeat-based, but other approaches are also applicable. This is not the most dependable approach, but it allows additional processing resource to be left out of the system (if the approach given in the following chapter is not used).

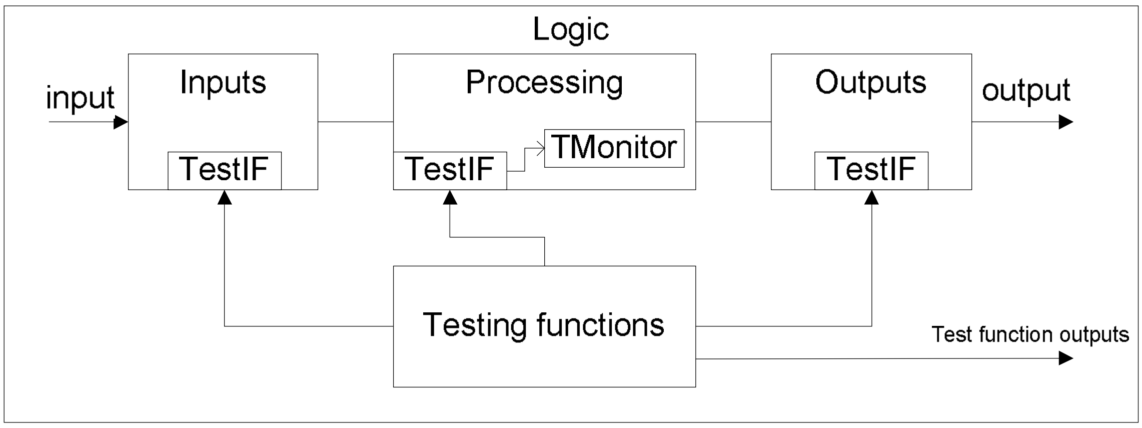


Figure 22. General software architecture for a category 2 system.

Although category 2 requires high MTTF values to achieve PL d, it is still a tempting option in the machinery domain. By investing in good hardware (with sufficiently high MTTF values), the absolute number of devices required in the safety function can be low compared with the architectures required by categories 3 and 4 (see 4.2.2.3). Practically[1], these architectures require the use of redundant devices and thus the device number increases. As the testing functionality can be deployed on the same processor as the main safety functionality, only testing equipment outputs (if any are needed) are added to the category 2 architecture when compared with the category B/1 architecture. A decrease in the device number ultimately reduces the weight and space requirements of the safety functionality hardware. The overall cost may also decrease, especially if the number of manufactured machinery units is relatively large. This compensates for a slightly greater software development cost compared with a pure single channel system.

[1] It needs to be ensured that no single fault can cause the loss of a safety function. If there is no redundancy in the devices, this requirement is hard to fulfil.

4.2.2.3 Category 3 and 4 architecture

The category 3 and 4 architectures are the highest and most complex system architectures introduced by SFS-EN ISO 13849-1 [2008]. The architecture consists of two redundant channels that monitor each other (to some extent). This effectively makes the architecture a 1oo2 architecture. That is, as the description in the standard requires, a single fault in any part of the safety-related system must not cause loss of the safety function. When implemented adequately (considering, e.g., common-case failures and software diversity), the dual channel system is safer than a single channel system (cf. category B/1 architecture 4.2.2.1).

The category 3 and 4 architectures are illustrated in Figure 23. The architectures are a dual channel system effectively consisting of one of two independent single channels that monitor each other. According to SFS-EN ISO 13849-1 [2008] Section 4.6.2 and IEC 61508-2 [2010] Section 7.4.3, the SIL (and PL) in one channel of the 1oo2 systems can be claimed to be one level higher than if the channels had adequate independence and diversity. This means that, for example, in a 1oo2 SIL2 system (adequate independence and diversity), single channel software (and hardware) needs to fulfil the SIL1 requirements. If the software is not diverse it needs to fulfil the SIL2 requirements.

The level of diagnostics (diagnostic coverage=DC) is at least 60% in categories 2 and 3 whereas in category 4, the DC value is better than 99%. If a category 2 or 3 system has better diagnostic coverage than 90%, the system may reach a higher performance level if the $MTTF_d$ (Mean Time To Dangerous Failure) value is also good. The diagnostic coverage value represents how well all parts of the system are monitored. The monitoring scheme can be chosen by the developer, in which case the best-suited approach can be taken. Monitoring of the channels should be independent because, if there are dependencies, the probability of safety function loss due to a single fault increases.

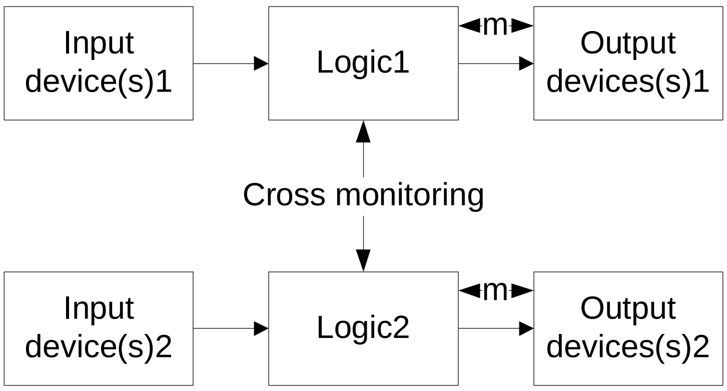


Figure 23. Category 3 and 4 system architecture [SFS-EN ISO 13849-1 2008].

In category 3 and 4 software architecture, the diversity aspect needs to be considered. As mentioned already, a single fault may not lead to the loss of a safety function, and this also applies to software. If software is replicated for both channels, a software fault can (but does not necessarily) cause a loss of safety function. This implies that some diversity is needed in the implementation of category 3 and 4 software. The issues related to the selection of the kind of diversity and the depth of diversity that is required or needed is not discussed in this section. The software architecture for a category 3 and 4 system is illustrated in Figure 24. The software architecture consists of two separated logic units. Physically, the two logic units can be located on the same safety PLC (redundancy is inside the PLC) or in different units. The main logic is separated from the monitoring functionality. The CrossMonitor module receives data from another channel and compares it with the output of its main logic output. If the monitor detects large enough (or any) deviation between outputs, it notifies the main logic module, which can then act accordingly.

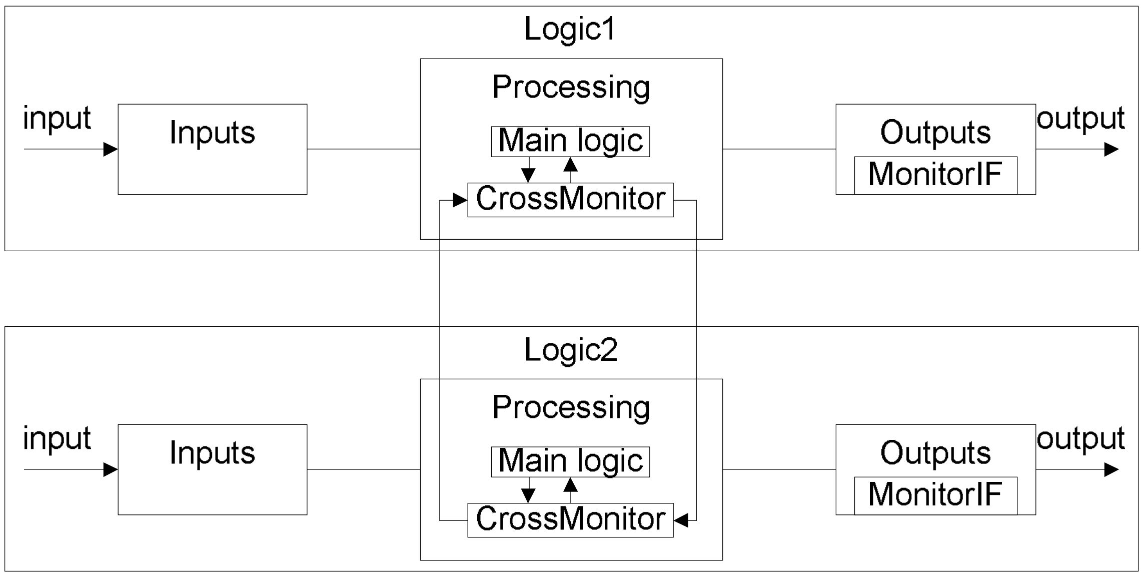


Figure 24. Category 3 and 4 software architecture.

4.2.3 Software architectures that promote fault tolerance

The following sections present and evaluate concrete software architectures that promote fault tolerance. Most of the architectures implement the principles introduced in Section 3. It should be noted that the architectures only represent one possibility for implementing the underlying principle in the form of software architecture.

4.2.3.1 Recovery blocks

The recovery block architecture uses sanity check, re-try, redundancy and diversity principles. The main idea of the architecture is to employ diverse components and the re-try principle with a degrading service level to increase the success rate of carrying out the task implemented with the recovery blocks structure. That is, the primary component provides the best level of service, and the level of service degrades if alternative components are needed. Due to the nature of the architecture, the recovery blocks (depending on the diversity of the alternatives) provide means to tackle temporal and design (including specification and implementation) faults. Temporal faults may decrease as time passes when alternative components are executed. Faults in specifications and implementation are tackled with diverse components that ought to be diverse from the specification to provide means to tackle the specification-related faults.

The process flow of the recovery blocks is illustrated in Figure 25. First, a checkpoint must be established to which the block state can be reverted in the case of a fault. The first alternative component (of N available ones) is executed. If the execution succeeds without exception, the output is evaluated against the acceptance test(s) (such as value range and CRC). If the acceptance test fails or the exception signal is generated during the execution of the alternative, the state is restored using the checkpoint created in the beginning. After restoration, the next alternative is executed. The preceding acceptance and exception procedures apply. The cycle is continued until the accepted output is produced or all alternatives have been tried. If no accepted output is obtained, a failure exception is produced. The recovery process must be executed within a predefined time slot. The time slot in the machinery sector is often quite short and the recovery is therefore often related to the initial values. The exception must be handled by the component requesting service from the recovery block.

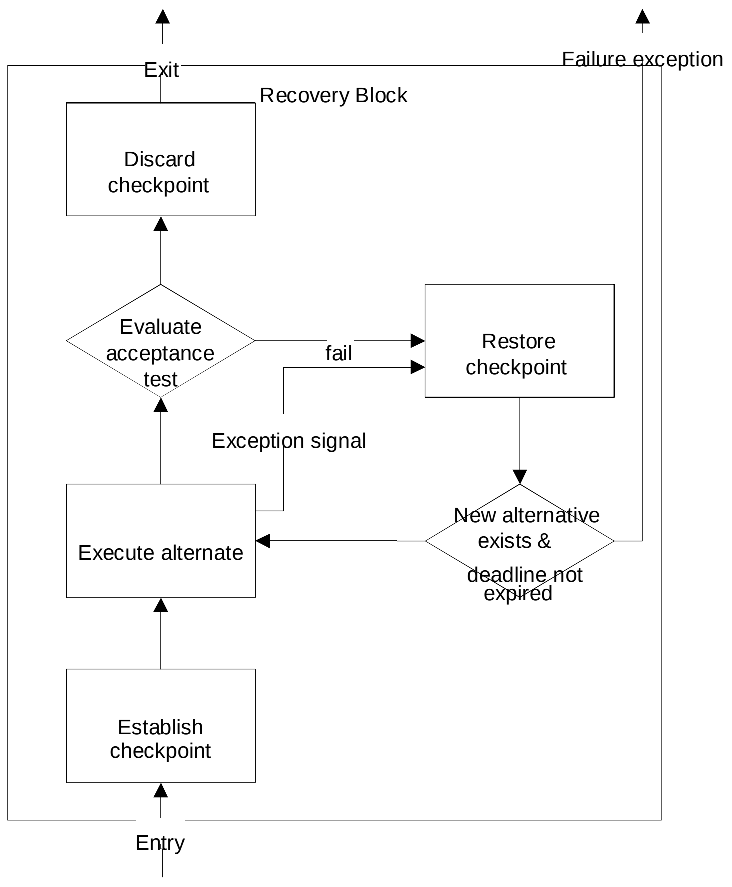


Figure 25. Recovery blocks [Tyrell 1996].

4.2.3.2 Monitored output

In the monitored output[2] architecture, a safety function component is used alongside the main functionality to override its output if a dangerous state is detected. The safety function is always able to override the main functionality output by using, e.g., a switch or other component. The architecture enables advanced control functionality, as the main control functionality is not necessarily developed as safety-critical software. Instead, the safety function logic is developed as safety-critical software and is responsible for safety. Naturally, the main functionality can be implemented so that it tries to prevent a trip of safety functionality. The basic layout of the architecture is presented in Figure 26.

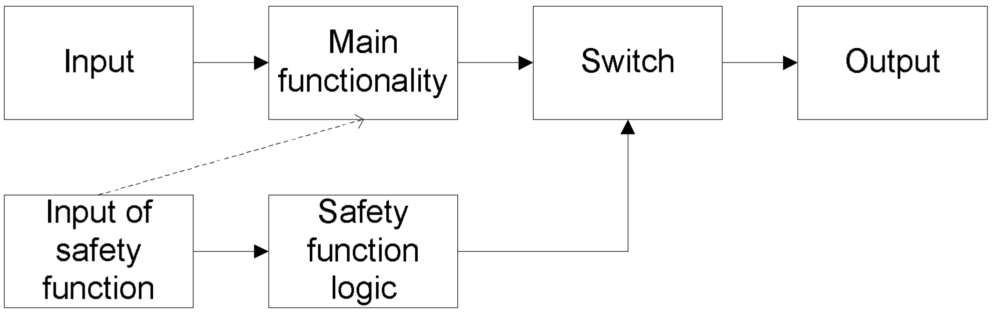


Figure 26. Monitored output architecture.

The architecture enables full separation of the safety-related functionality from the main functionality. The main functionality does not need to provide any safety functionality such as operations (though it may include such operations). The safety function needs no information from the main functionality. The safety and main functionalities may share inputs, but the safety functionality must also use dependable safety-related inputs.

Though the main functionality is not responsible for safety-related decisions, it is possible to implement safety-promoting functions into the main functionality if needed/desired. These functionalities/restrictions try to keep the system on target in the operating region and thus prevent safety functionalities from activating. These functions may restrict, for example, output of the functionality so that dangerous outputs (of the main functionality) are avoided. For instance, tank temperature control may limit the heating power so that the tank temperature does not rise too high. The main responsibility of safety is on the safety function. It ensures that in all cases, the system remains in a safe state of operation.

[2] Monitored output is not a widely accepted name for this kind of architectures, but it is used to describe the architecture in this report.

4.2.3.3 MooN architectures

This subsection discusses MooN (M out of N) voting architectures (majority voting [IEC 61508-7 2010]). M indicates how many software components require a safety function from N available ones to perform the safety function[3]. The basic principle of voting is to process the same calculations by multiple logical units and decide the final output according to the results.

The architectures discussed in this subsection are considered to be deployed on a single CPU and thus the N components participating in the voting have to be. N identical software components on the same CPU only provide fault tolerance against pathological situations such as memory (RAM/ROM) faults on certain blocks of memory. Although diverse software modules executed on the same CPU do not offer complete protection from faulty hardware, they provide fault tolerance against software faults as the probability that all the N diverse components include the same fault mitigates as the value of N grows. To attain diverse software components, the components ought to be developed by separate teams using different designs (including architecture and detailed design), programming languages and compilers.

The voting architectures that can be packed inside the safety function component are presented in Figure 27.

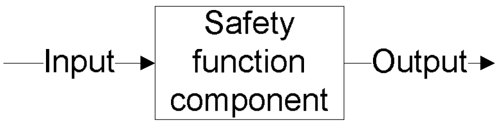


Figure 27. Safety function logic component.

4.2.3.3.1 1oo1

One out of one (1oo1) is the simplest safety architecture available. The 1oo1 architecture does not include redundancy or diversity as only a single decision channel is available, as depicted in Figure 28. As the architecture does not implement redundant channels, the architecture includes no fault tolerance. Whenever any of the modules included in the architecture fail, the function is not able to carry out the safety function it is designed to carry out. When implemented alongside the main functionality, however, the architecture provides the promoted safety compared with the situation in which no safety function is used.

[3] This means[3] that M modules out of N available ones must agree on triggering a safety function before it is actually trigged. The majority voter can also be more complex and may include specific rules, e.g. for discarding votes.

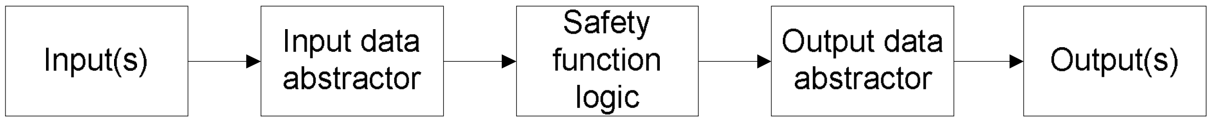


Figure 28. 1oo1 architecture.

4.2.3.3.2 1oo2

The one out of two (1oo2) architecture is the simplest voting architecture that actually uses voting in the decision-making process. The architecture uses two channels, each of which is capable of carrying out the same task, e.g. detecting over-pressure in a steam tank and acting accordingly. The architecture is presented in Figure 29. The architecture uses redundancy and diversity principles. There are two redundant safety function calculation channels. If one of the channels fails, the other channel is still able to produce the correct output. Channels are diverse to degrade the probability of failure of both channels simultaneously. In 1oo2 architecture, both channels (the input and logic chains related to safety function logics 1 and 2) are or can be executed on the same CPU. The minimum requirement is to have diverse safety function logics. As logics are dependent on the input data, it is recommended to also have diverse inputs and input data abstractions as this reduces the probability of simultaneous input data failures (neglecting critical hardware failure).

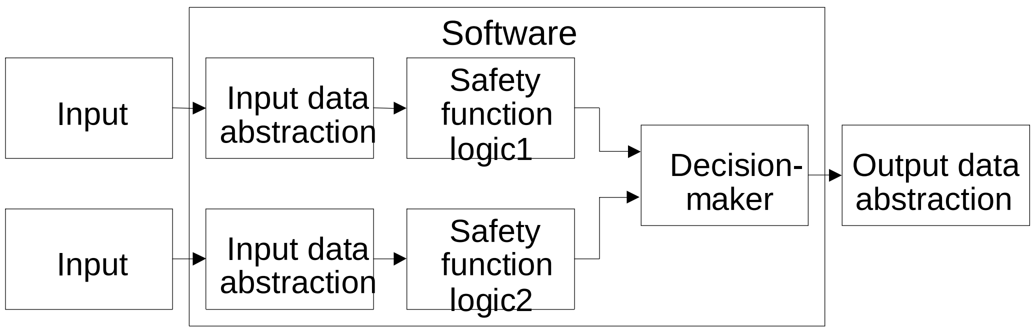


Figure 29. 1oo2 architecture.

The 1oo2 architecture promotes safety at the cost of availability. The principle is to take the safest approach of the two alternatives. That is, if either of the channels proposes activation of the safety function, it is activated. The decision-maker is constructed so that the safest action is always taken[4].

[4] Two analogies from electronics can be used in this context. Depending on the applications, the 1oo2 architecture may resemble either a connection in series or parallel of two switches in series with load. The safe approach is not to let current to the load in the equivalent circuit in series connection. That is, if either switch is open, no current flows to the load and the system is in a safe state. In contrast, in some applications, the safe thing to do is to let current to the load. For example, consider a cooling pump at a nuclear plant. In such a case, the equivalent connection would be two parallel switches in series with load (cooling water pumps). Either way, the failure in one of the channels cannot disable the correct operation of the safety function.

Due to this principle, the 1oo2 architecture is not best suited to situations in which the system owns no safe state, for example, certain parts of a flying airplane. Instead, suitable applications include, for example, machinery applications in which an operation is halted or disabled to prevent injury or machine damage. Such an application could be, for example, the halting of an assembly robot if an obstacle (human or other object) is detected inside the working area.

4.2.3.3.3 2oo2

The 2oo2 architecture is an availability-orientated version of the 1oo2 architecture (see 4.2.3.3.2). In the 2oo2 architecture, both channels need to agree before the safety function is activated. Due to this principle, the 2oo2 architecture is a non-safety-promoting architecture. If either channel fails and does not claim any need for safety function activation, the safety function will not be activated, regardless of the output of the other channel. Thus, 2oo2 architecture should not be used in safety-critical applications.

4.2.3.3.4 2oo3

The 2oo3 architecture is a (majority) voting architecture that provides improved availability over the 1oo2 architecture and improved reliability over the 2oo2 architecture (with a suitable decision scheme). The architecture also enables detection of the faulty channel. The architecture is depicted in Figure 30. The architecture employs three channels that are connected to one decision-maker unit. Each channel calculates output, and the output is forwarded to the decision-maker unit. If the channels are diverse, the architecture provides coverage against design and implementation faults (not including common faults). If the channels are identical, protection is only obtained against certain types of random faults (see 4.2.3.3).

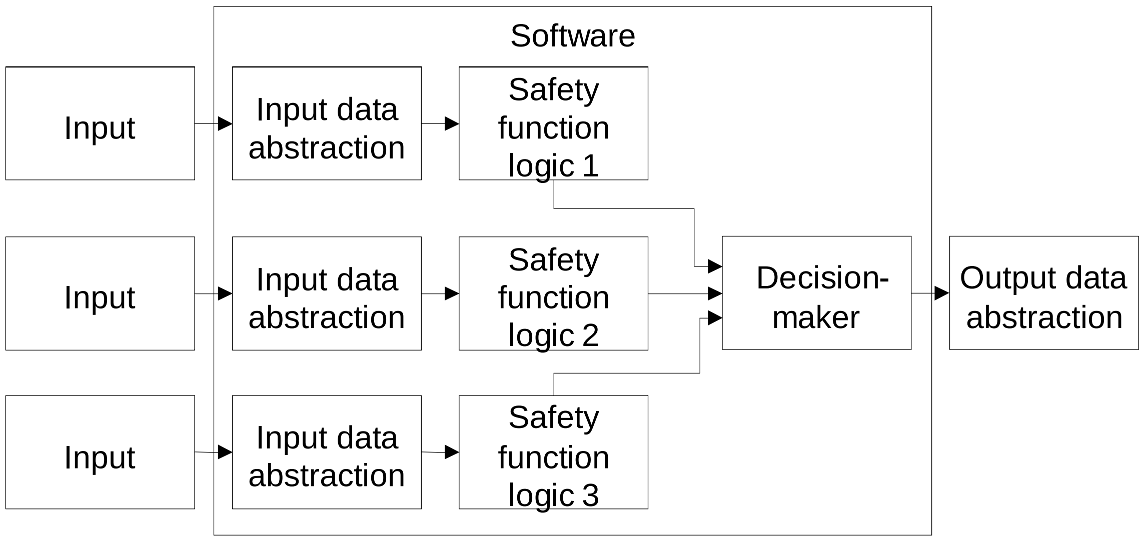


Figure 30. 2oo3 architecture.

The module that produces the final output value is the decision-maker unit. As the architecture employs three channels, the decision-maker unit may follow various schemes to obtain the final output value. A suitable scheme depends on the application and functionality of the system. Possible schemes include:

- unanimous agreement
- majority agreement
- average value[5]
- median value.

As with the 1oo2 architecture, the 2oo3 architecture can also be used without a dedicated decision-maker unit. Instead, each safety function logic unit may control an output unit (i.e. actuator) directly. Such a control scheme is used in, for example, airplanes. The control surfaces are operated with three hydraulic actuators (cylinders) from which each is capable of controlling the surface independently. That is, if one actuator is controlled in the positive direction and two others in the negative direction, the outcome is movement in the negative direction.

4.2.3.4 Idealized fault-tolerant architectural component

The idealized fault-tolerant architectural element (iFTE) is a fault-tolerance approach proposed by Lemos, Guerra and Rubira [de Lemos et al. 2006][6]. The approach relies on (precise) exception handling to achieve fault tolerance. That is, exceptional situations are acknowledged and handled in a controlled way to avoid major system failures. As usual, exceptions are primarily handled locally by the component that produces the exception. However, the architecture also allows (requires) for the definition of the way the components of the system should co-operate to handle faults that the producer component is unable to handle locally. Exceptions are typically seen as a fault detection or, more precisely, a fault/failure information method. In this architectural scope, the exceptions play a more major role as they define how the fault handling responsibility is distributed between the software components as well as within a single software component. The approach promotes fault containment in particular. Faults are primarily handled near their source. Support for system-wide fault handling also exists as the elements may co-operate to handle faults.

The main architectural property of the solution is to divide each architectural element into two distinct parts from an operational point of view. Each element of the (fault-tolerant) software includes subcomponents responsible for normal operations and a subcomponent handling abnormal situations.

[5] In most cases, the average value is not good, as computing failures may lead to huge numbers that dominate the average value. Another hazardous average value case may arise when the correct direction is right or left but not straight.

[6] The article provides a more in-depth introduction to the architecture and its principles.

The central concept of the architecture is the idealized fault-tolerant element (iFTE) that encapsulates the normal and abnormal operation of the element. The architecture of generic iFTE is depicted in Figure 31. The iFTEs comes in two variants: idealized fault-tolerant components (iFTComponent) and idealized fault-tolerant connectors (iFTConnector). The former represents implemented functionality or service, for example, motor control. The latter represents a resolver used to provide compatible interfaces between collaborative iFTComponents.

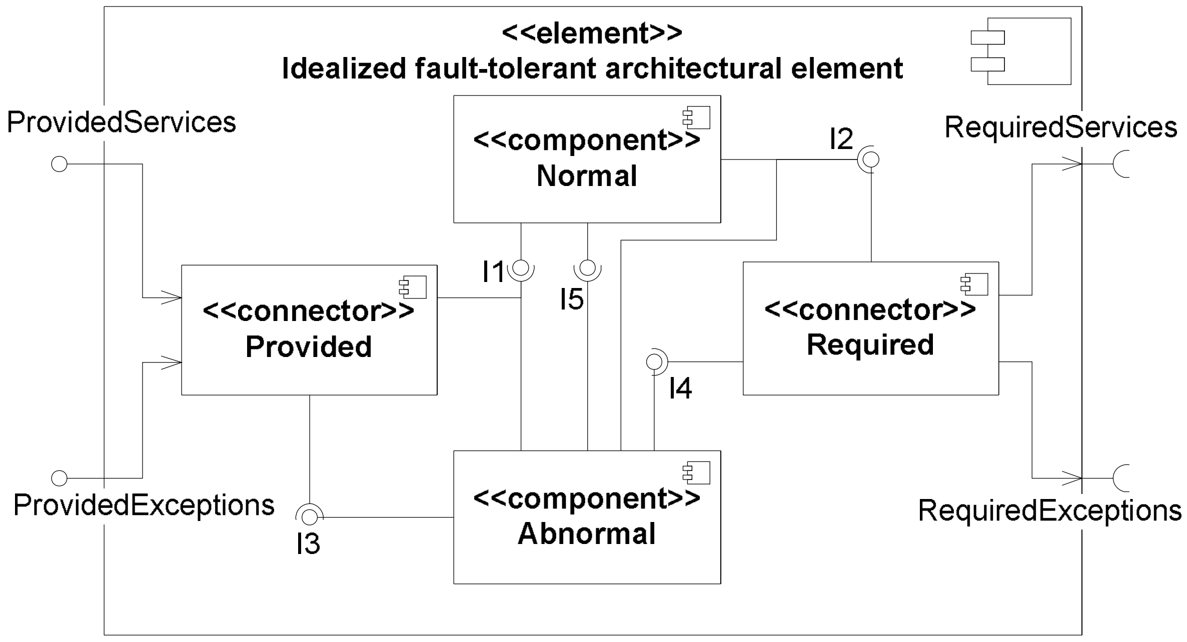


Figure 31. Internal structure of an idealized fault-tolerant architectural element; reproduced from de Lemos et al. [2006].

4.2.3.5 Graceful degradation

There are several definitions for graceful degradation, which can be roughly divided into two main groups. According to IEC 61508-3 [2010], the aim of graceful degradation is 'to maintain the more critical system functions available, despite failures, by dropping the less critical functions.' In a more detailed (practical) description, the standard states that critical functionality of controller A needs to be deployed on controller B (less critical), if controller A is not capable of carrying out the (safety-critical) functionalities of its responsibility. Other sources typically define graceful degradation as a more or less lightweight technique that offers a reduced level of service in the presence of faults.

4.2.3.5.1 IEC 61508 [2010] interpretation of graceful degradation

Graceful degradation as described by the IEC 61508-3 [2010] is illustrated in Figure 32. The main principle is to deploy critical functionalities on controllers that execute less critical functions in case the controller executing the critical function malfunctions. In Figure 32, the following deployment chain is given. A hardware failure occurs in the hydraulics controller (*primary node*) and thus the controller is incapable of functioning and carrying out its tasks. The controller executes a safety-critical function and needs to stay operational. As the controller malfunctions, it is not able to send periodical heartbeat messages indicating that it is fully operational. The logger controller (*spare node*) monitors the heartbeat messages and notices that these messages are no longer arriving and concludes that the hydraulics controller is not operational. The logger controller notifies other controllers of the system (e.g. to obtain a safe state). The logger controller then disables its non-critical main functionality (logging of error situations) and loads the safety-critical function formerly executed by the hydraulics controller. When the safety-critical functionality is operational, the controller informs the other controllers of this and some level of service may be offered again. It should be noted that the example supposes that both controllers have access to sensors and the actuators needed to carry out the safety-critical functionality. If these devices are accessible through the system bus, the redeployment is relatively easy.

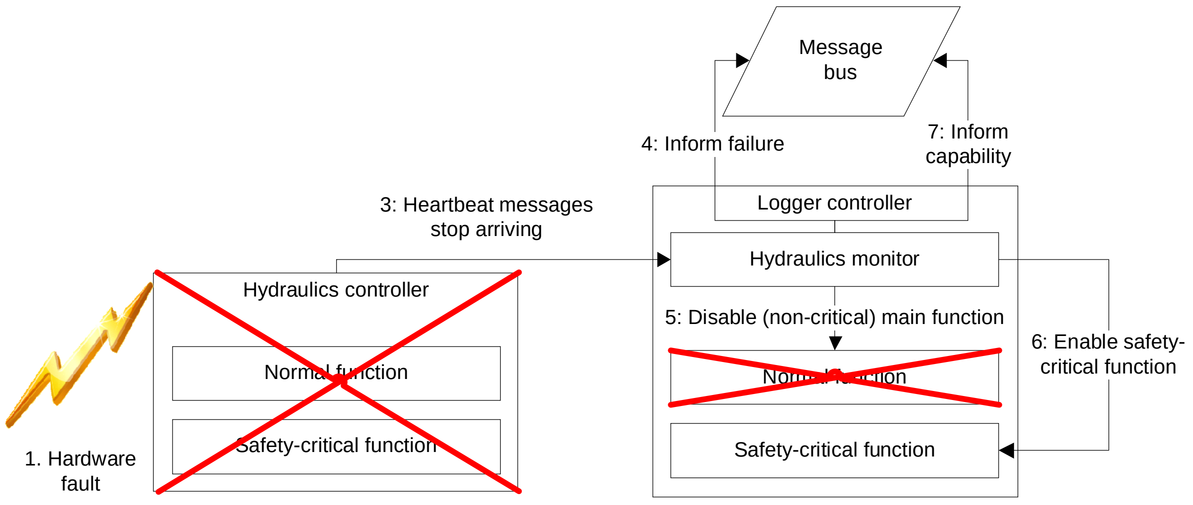


Figure 32. Graceful degradation according to IEC 61508 [2010].

The graceful degradation described by IEC 61508 [2010] can be considered a heavyweight approach to obtain safe and dependable software. It adds considerably to the complexity of the software and introduces issues of both software and system architecture. However, the technique offers the possibility of using fewer nodes in the system, as there is no need for every node that executes a critical function to be duplicated, as one node may (if applicable) serve as a spare for multiple nodes. In IEC 61508-3 [2010] on the lower SILs (SIL 1-2), graceful degradation is an alternative architectural measure to re-try a fault recovery mechanism. The re-try mechanism can be considered an easier approach in many situations and is thus more recommendable. On higher SILs, however, graceful degradation is highly recommended, whereas the re-try mechanism has no effect according to IEC 61508-3 [2010]. Thus, on higher SILs, graceful degradation is, in practice, an obligatory method unless cogent reasoning not to use it is given. Such situations could be, for example, missing support from the coding language or hardware reasons. In some cases, the overwhelming increase in complexity may also be accepted as a reason to leave out a highly recommended method.

4.2.3.5.2 Other approaches to graceful degradation

As mentioned, graceful degradation can also be a considerably more lightweight approach than the one presented by IEC 61508 [2010]. The main idea of this kind of approach is to keep the original functionality, as far as possible, by reducing the level of service of the functionality. That is, the (safety) function implementing graceful degradation is also able to provide some level of service in the presence of expected problems. For example, consider the situation described in Figure 33 in which a measurement value composer uses three sensors (each connected to a separate connection) to obtain an accurate measurement value by calculating the average over individual sensor values. Graceful degradation in this setup could be, for example, tolerance of failure sensors. If and when a sensor fails, the measurement value composer should not fail but provide the value user with a measurement value, though the accuracy would decrease (only two usable values instead of three). In general, this approach could be described so that the architecture (or the implementation of it) should not fail (e.g. wait infinitely for a failed sensor) but provide some level of service (e.g. send worst case value) that could be used further in the software. A service provider should of course inform the service requester about the state it is operating in, i.e. inform about the level of the service indicator alongside the provided service.

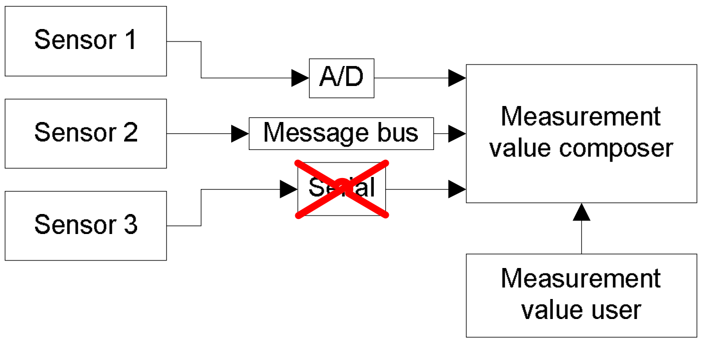


Figure 33. Simple graceful degradation set-up.

4.2.4 Architectural techniques for non-interference between software elements

In this section, architectural techniques and principles to achieve non-interference between software elements are introduced. The techniques and principles are gathered mainly from the safety-related standards IEC 61508 (IEC 61508-3 [2010]) and SFS-EN ISO 13849 [2008], though other methods and techniques are also introduced. In practice, non-interference is achieved by providing sufficient independence for the software elements. Software architecture plays a major role when independence issues are considered. The architecture must consider the issues and provide suitable solutions to achieve sufficient non-interference between SCS and NSCS elements.

Independence of software elements can be achieved ensuring spatial and temporal independence between software elements as suggested by IEC 61508-3 [2010] (in informative Appendix F). Here, the IEC 61508 [2010] assumes that the software elements are executed on single processors (and that they also share other hardware). Better isolation is, of course, achieved by applying different processors, computers and even power supplies, but these means are not considered here.

Some isolation means have already been mentioned in Section 4.2.1: fault containment regions, failure assertion programming and defensive programming. Methods such as glueware and wrapping can also be used for encapsulating a subprogram. The main purpose of glueware is to make the interface more convenient and to make it fit with the main program. It is possible, however, to add isolation features as well there. The methods, which are mentioned in this paragraph, consider mainly errors related to data. There are also other types of errors (see 4.2.4.1) such as temporal errors. They are considered in Section 4.2.4.4.

4.2.4.1 Potential sources of interference between software elements

There are numerous sources that may cause unwanted software element interference. IEC 61508-3 [2010] lists (non-exhaustive) such sources that need to be considered when independence between the software elements is aimed for. Table 3 introduces a set of sources causing interference between software elements and provides a short description of each source.

Table 3: Sources of software element interference [IEC 61508-3 2010].

Interference source	**Description**
Shared use of random access memory	Two or more software elements use a shared memory location to communicate with each other. If none of these elements is SCS, there should be adequate methods to protect NSCS elements from corrupting the data. From a more generic view, when two or more software elements use the same memory unit, there is a risk of memory corruption, either through software error or malicious usage. Software A may overwrite software B's memory location if the memory locations are not protected.
Shared use of peripheral devices	Peripheral devices such as actuators, permanent storage disks, A/D-converters, etc. enable interference between software elements. The main interference form is reservation of the device. A software element may reserve a resource (device) so other elements cannot use it during this time. Obviously, if an NSCS element reserves a resource it cannot be used by SCS elements without pre-emptive functionality. Another interference method is related to the control of shared peripheral devices. A peripheral interface may include some public functions or data that may be usable by any software element. Thus, NSCS may be able to interfere with SCS elements through the interface of the shared device. This issue should be considered in the system architecture, which should deny interfering use of problematic device interface data or functions.
Shared use of processor time (two or more software elements are executed on a single CPU)	Whenever two or more software elements are executed on a single CPU (which is assumed here to be a single core) the processor time must be shared between the elements. This is typically handled by the operating system, which executes the elements in separated processes. It must be ensured that, e.g., a fault, deadlock, crash or pending request in one element does not block or delay processing of the other element. Various techniques can be used. Time-triggered architecture enables a periodical context switch (change of executed process/element), which ensures predefined time slices for each process. Event-driven architecture is similar but the trigger for the context switch comes from a non-time based event, e.g. message received from the system bus. Note that the cyclic execution architecture is problematic because, if the execution becomes blocked, the following element also becomes blocked or delayed.

Interference source	Description
Communication between elements to achieve the overall design	In some cases, a number of elements need to communicate to achieve the desired safety functionality. Communication between two or more software elements is always problematic from a safety point of view. Possible failures include, for example, passing bad data (delayed, corrupted or wrong variable), crashing during data transfer, communication synchronization and confirmation success. A communication protocol needs to be decided. Communication through a non-deterministic message bus can also add interference. The message bus can be seen as a hardware peripheral but also a communication channel that is needed in successful communication. A badly designed messaging system may cause interference to elements not driven through messages on a single CPU.
Consequent failures	A failure in one element may cause a failure in another element. Typical instances of failures causing consequent failures are overflows, divide by zero, and null and bad pointers. Unhandled exceptions are typical reasons to crash an element, which may also lead to a crash of other elements if they are not sufficiently isolated.

4.2.4.2 Spatial independence

Spatial independence as defined by IEC 61508 is “the data used by a one element shall not be changed by another element. In particular, it shall not be changed by a non-safety related element.” [IEC 61508-3 2010]. That is, each element must be the only entity that is able to change the data managed by it. The software architecture should honour the principle of this independence. The suggested approaches to achieve spatial independence are [IEC 61508-3 2010]:

- Hardware memory protection
- Operating system supporting processes with its own virtual memory supported by hardware memory protection
- Rigorous design and code analysis to demonstrate sufficient independence
- Software protection to ensure integrity of higher integrity element data from modifications of lower integrity level elements
- (Denying a data pass from lower integrity elements to a higher integrity element if the integrity of the data cannot be confirmed)
- (Suitable data passing methods)
- (Consideration of permanent storage devices in spatial partitioning).

4.2.4.3 Controlled data passing methods

The methods and techniques discussed here relate to denying a data pass from lower integrity elements to a higher integrity element if the integrity of the data cannot be confirmed and the suitable data passing methods mentioned are in the list above. When considering safety functions, the highest possible SIL or PL can be defined by looking at the lowest SIL (or systematic capability) or PL of the subsystems to which the safety signal applies. A non-safety signal can therefore drop the SIL or PL of a safety function dramatically. The architecture and the way the system is divided into subsystems affect the SIL or PL determination. The basic guideline for the architecture of the SCS is to deny data passing from lower integrity elements, i.e. non-safety-critical software elements. The IEC 61508 [2010] fortunately provides a way round this restriction. If the integrity of the data passed can be confirmed, the pass can be made. It depends on the situation and the kind of integrity checks that can or should be made. In some cases, elements (especially safety-critical ones) need to communicate and pass data to each other. If such a need exists, the software element should only modify the data of other software elements through a dedicated operation provided by SCS in its interface. Such an operation could, for instance, be a member function call or some sort of message pipe. This approach should be preferred over direct[7] modification, e.g. done through a shared/global variable. In every case in which data are passed from a lower integrity element to a higher integrity element, the higher integrity element must also ensure the integrity of the passed data. This is relatively easy to implement in a data passing mechanism relying on a dedicated method.

In Figure 34, a more recommendable approach is illustrated. In this approach, SCS_A uses a dedicated interface method to pass data to SCS_B. In this case, SCS_A cannot change the data without SCS_B noticing it (notice that myData is a private member variable). SCS_B can perform various checks to confirm data integrity before the value is saved. If the data have to be passed, this is the recommended way to do it. It is the architecture's responsibility to define the methods used to pass data between elements. For instance, the architecture may deny the use of shared and global variables and force the use of data hiding and integrity checks, which are obtained in Figure 34.

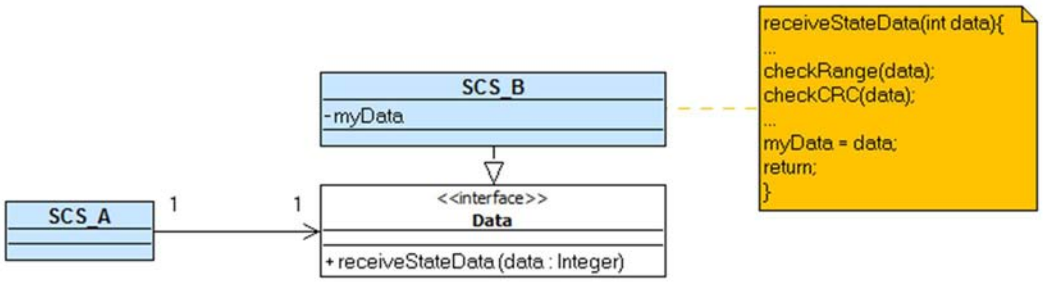


Figure 34. Recommended approach to passing data between SCS elements.

The preceding examples considered direct data passing between software elements. Data can also be passed through some kind of message mediator structure. In this solution, a mediator is established between the message passer and receiver. In some cases, such an approach is supported by the operating system or platform on which the application is run.

[7] Here, direct data change means direct access to a variable such as public member data of object orientated language. Such data can be altered by any element that has access to the public interface of the provider element.

4.2.4.3.1 Low level isolation methods

The hardware and operating system on which the SCS is run can also provide techniques to achieve spatial independence between software elements. These techniques are *hardware memory protection, operating system supporting processes with own virtual memory supported by hardware memory protection,* and *software protection to ensure integrity of higher integrity element data from modifications of lower integrity level elements*.

Hardware memory protection is a hardware-based solution to prevent access by software elements to certain memory locations. As stated by Haikala and Järvinen [2004], hardware memory protection is an automatic side effect of the use of page tables (or page registers) in the memory management unit (of a processor). Page registers (or tables) can also include additional information concerning rights to use certain memory pages, e.g. read, write and executions rights, and mark whether data are code, an OS section or normal user data, etc. [Haikala & Järvinen 2004]. These techniques are (typically) embedded in modern processors' hardware and are foundations of virtual memory used in modern operating systems. As hardware memory protection is established by hardware, it cannot be achieved with any solution on software architecture. Software architecture may require hardware memory protection to be available, however, if other spatial independence methods are based on it.

4.2.4.3.2 Other methods

The other methods to achieve spatial independence between software elements include *rigorous design and code analysis to demonstrate sufficient independence.* These methods are not architecture-related matters. The architecture of the software also defined the principles and guidelines according to which the software is developed however. Thus, the architecture may require the use of rigorous design methods and sufficient code analysis. For instance, each software module must be inspected for possible references in other memory areas (e.g. table overflows).

4.2.4.4 Temporal independence

Temporal independence is defined by IEC 61508 as: "one element shall not cause another element to function incorrectly by taking too high a share of the available processor execution time, or by blocking execution of the other element by locking a shared resource of some kind" [IEC 61508-3 2010]. That is, it ensures that each software element is provided with sufficient execution time and that no element can block another element's execution. Most of the methods and techniques to achieve temporal independence are related to scheduling, as this is one of them most important matters when the timings of the executed processes are considered. There are a very limited number of other methods to achieve temporal independence. The IEC 61508 suggests the following methods to achieve temporal independence between software elements [IEC 61508-3 2010]:

- Deterministic scheduling
- Strict priority scheduling
- Time fences between processes
- Starvation prevention of software elements

- (Rigour usage of peripheral devices).

4.2.4.4.1 Scheduling aspects

As mentioned above, scheduling is a fundamental aspect of temporal independence. The scheduling scheme must ensure that each process has a sufficient time slice in which the process is able to perform its tasks. In addition, it must be ensured that real-time requirements are met, i.e. tasks are finished in the dedicated time windows. The IEC 61508-3 [2010] suggests three different scheduling architectures: cyclic, time-triggered and strict priority. The two first are strictly deterministic, and the latter is dynamic (to some extent).

4.2.4.4.1.1 Cyclic scheduling

One of the possible deterministic scheduling schemes is cyclic scheduling. In the cyclic scheduling algorithm, the elements/processes executed on a single CPU are provided with predefined time slices in a predefined order. The time slices form a cycle that is repeated infinitely, thus providing each element with a predefined processing time during the cycle. The pros of the scheme are guaranteed time slice (no starvation). The con is that the scheme typically wastes resources if the time slices are not designed properly. The time slices must also be designed so that they are applicable in worst-case situations. The IEC 61508-3 [2010] suggests validating the timing requirements, demonstrating them statically. This ensures each slice is provided with the slack needed to handle worst-case situations. The scheme is suitable when there are well-defined processes and their execution times (also worst case) are well known, and the worst-case execution times are sufficiently close to the normal execution time. If there are many random events, processes also need to wait (e.g. hard drive file open), and processing time is wasted as the process is actively waiting.

An example of cyclic scheduling is illustrated in Figure 35. In the illustration, three SCS elements are executed on a single processor. Notice that a context switch and other additional mandatory operations are not considered. The cycle is run at 1000 ms intervals. The cycle is highly deterministic as each process has a fixed execution time in the cycle. The SCS2 has a considerably larger time slice. This may be due to high computational demand or a long worst-case execution time.

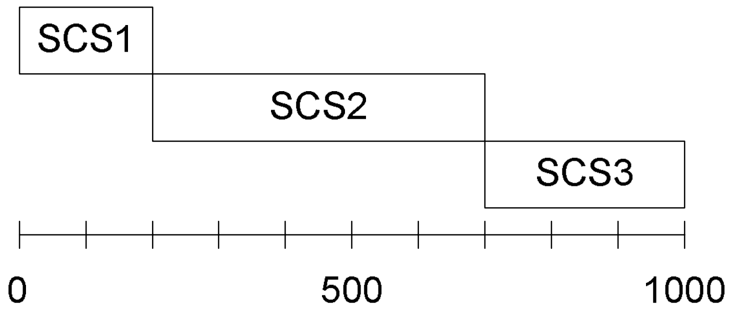


Figure 35. Example of a cyclic scheduling scheme.

The suitability of cyclic scheduling is somewhat questionable in machinery applications in which the system bus is used to enable communication between nodes. Such a system needs to be able to react to messages (events) received through a bus. The scheme is best suited to situations in which there is no need to react to random events.

4.2.4.4.1.2 Time-triggered scheduling

An alternative deterministic scheduling scheme to cyclic scheduling is time-triggered architecture. Time-triggered scheduling is a deterministic scheduling algorithm[8], but it has some improvements on cyclic scheduling. The software elements that need deterministic execution are triggered into execution at predefined points of time. Triggering is done periodically, e.g. every 500 ms. To this point, time-triggered scheduling seems like cyclic scheduling. The triggered process may call itself to wait any time it finishes however. In this situation, the scheduler takes the background process back to execution. Figure 36 illustrates very simple time-trigger scheduling. In the scheduling, an SCS process is invoked periodically with 500 ms intervals. The SCS may take up to 200 ms of time if necessary. On the first trigger, the SCS finishes before 200 ms have passed. Thus, the process goes to wait and the NSCS process is executed.

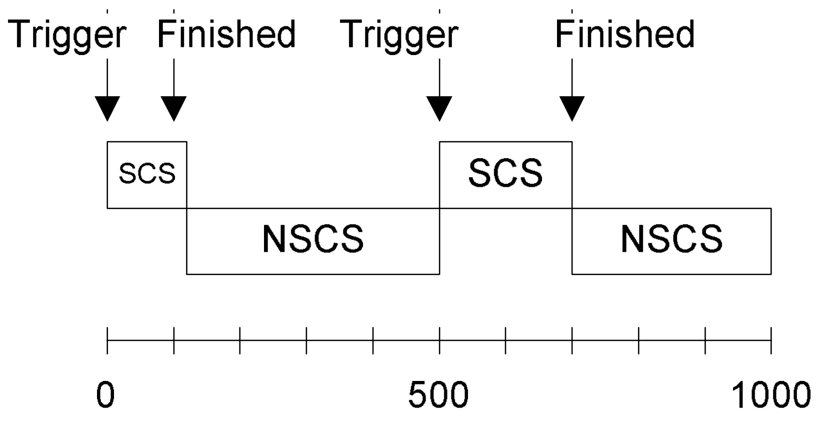


Figure 36. Time-triggered scheduling.

4.2.4.5 Complete hardware-based independence of elements

Ultimate independence is achieved when software elements are executed on independent processors, each element is provided with dedicated hardware (including sensors, CPU, memory, etc.) and no element communicates with another element in any way. This enables hardware-based spatial and

[8] The determinism only applies fully for the processes that are evoked with time triggers. The background process has spare time between time-triggered processes.

temporal independence. Complete separation of SCS and NSCS requires the deployment of SCS elements on hardware dedicated completely to safety-critical applications and ensures that SCS and NSCS elements do not depend on each other in any way. The hardware separation means hardware that includes, e.g. processing units (CPU, memory, etc.), sensors, actuators and communication hardware. However, it is not always possible, especially in mobile machinery applications (due to weight, space, cost or other issues), to provide SCS with dedicated hardware. In mass-produced items, it is also more cost-effective to rely on software-based solutions as duplication of software is cheaper than duplication of hardware.

On the other hand, for systems that are not mass-produced, such as special machines and plants, complete hardware-based independence may be a more cost-effective approach than integrating NSCS and SCS onto the same hardware. The development cost of software for non-mass-produced products is considerable, and providing SCS with dedicated hardware may well be cheaper as the development of the software becomes easier.

References

de Lemos, R., Guerra, P.A. de C. & Rubira, C.M.F. 2006. A Fault-Tolerant Architectural Approach for Dependable Systems. IEEE Softw., March, Vol. 23, No. 2, pp. 80-87. ISSN 0740-7459. doi: 10.1109/MS.2006.35. http://dl.acm.org/citation.cfm?id=1128592.1128713

EN 954-1. 1996. Safety of machinery. Safety-related parts of control systems. Part 1: General principles for design. European Committee for Standardization.

Haikala, I. & Järvinen, H. 2004. Käyttöjärjestelmät. 2nd ed. Talentum. 246 p.

Haikala, I. & Märijärvi, J. 2006. Ohjelmistotuotanto. Talentum. ISBN 952-14-0850-2. 440 p.

IEC 61508-3 ed2.0. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems Part 3: Software requirements. International Electrotechnical Commission. 234 p.

IEC 61508-3 FDIS. 2009. Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Requirements for software. 112 p.

IEC 61508-7 ed2.0. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 7: Overview of techniques and measures. 296 s.

IEC 62061. 2005. Safety of machinery – Functional safety of safety-related electrical, electronic and programmable electronic control systems. International Electrotechnical Commission. 205 p.

NASA Software Safety Guidebook. 2004. NASA Technical Standard NASA –GB-8719.13. 388 p.

SFS-EN ISO 13849-1. 2008. Safety of machinery – Safety-related parts of control systems – Part 1: General principles for design. 180 p.

Tyrrell, A.M. 1996. Recovery blocks and algorithm-based fault tolerance. EUROMICRO 96. ‘Beyond 2000: Hardware and Software Design Strategies’. Proceedings of the 22nd EUROMICRO

Conference. September 1996, Prague, Czech Republic. Pp. 292–299. DOI= 10.1109/EURMIC.1996.546394, http://dx.doi.org/10.1109/EURMIC.1996.546394

Wikipedia. 4+1 Architectural View Model. [Referenced 1.3.2011.] Available at: http://en.wikipedia.org/wiki/4%2B1_Architectural_View_Model

4.3 Coding - Coding standards

Timo Malm, VTT

The actual executable software is created during the coding phase of software development. Alternatively code can be (partially) generated from high level models. This is an important phase of the development and there are many kinds of supporting tools. Software can also be bought (COTS) or obtained (open source) from external sources. Although the coding phase is essential, only a small proportion of delivered defects originate from the coding phase. Most of the software defects are related to earlier phases of design (see Section 1.2). The programmer can usually remove most of the coding-phase-related defects before delivery.

Before the coding phase, the design and coding rules (standards) must be defined. The design and coding rules comprise the design and development methods to be followed as well as the coding rules (set of requirements including prohibitions and recommendations related to coding style). IEC 61508-3 and ISO 13849-1 give some rules, which should or shall be added to the coding rules. The coding rules depend on many aspects, such as the programming language, requirement criticality, programmable system and programming environment. Examples of coding rules can be found from, among other sources:

- IEC 61508-7 [2010], section C2.6.2
- SFS-EN ISO 13849-1 [2008], section J4
- Misra C [Misra C Publications]
- Programming in C++, Rules and Recommendations [Henricson & Nyquist 1992]

References

IEC 61508-3 ed2.0. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems Part 3: Software requirements. International Electrotechnical Commission. 234 p.

IEC 61508-7 ed2.0. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 7: Overview of techniques and measures. International Electrotechnical Commission. 296 s.

Henricson, M. & Nyquist, E. 1992. Programming in C++, Rules and Recommendations. Accessed 2.8.2011. http://www.ktverkko.fi/~msmakela/software/Ellemtel-rules-mm.html

Misra C publications. http://www.misra.org.uk/Publications/tabid/57/Default.aspx#label-ac-gmg.

SFS-EN ISO 13849-1. 2008. Safety of machinery – Safety-related parts of control systems – Part 1: General principles for design. 180 p.

4.4 Verification and Validation

4.4.1 Reflections on principles of good testing in the context of safety-critical development

Matti Vuori, Tampere University of Technology, Department of Software Systems

4.4.1.1 General

In this section, we discuss some generally accepted principles of good testing and briefly analyse how they relate to and are perhaps amplified in the context of developing safety-critical software. We think that this is important, as most test engineers and test managers gain their education and training in a non-critical setting. Some adjustment may therefore be required to their approach to the testing.

The understanding of principles of good testing has evolved fast during the last decades and is not well known outside the software testing community. This chapter gives us an opportunity to present some approaches to testing thinking to various audiences.

4.4.1.2 An overall software development culture provides a context for testing

First a word about the general software development culture: many companies that produce, for example, machines do not have a long history of software development. The company cultures, product development practices and processes have been developed for and evolved in the context of producing heavy, physical machines built of steel. The inclusion of sometimes complex software systems that include not only simple machine controls but also advanced user interfaces, a distributed system for configuring work tasks and logistics, etc. is a new phenomenon that is continuing to grow in importance. Due to the newness, however, it may still not have the status in the companies that it deserves, which reflects in the design and implementation as well as the testing and other quality assurance. Figure 37 presents some of the factors that affect testing.

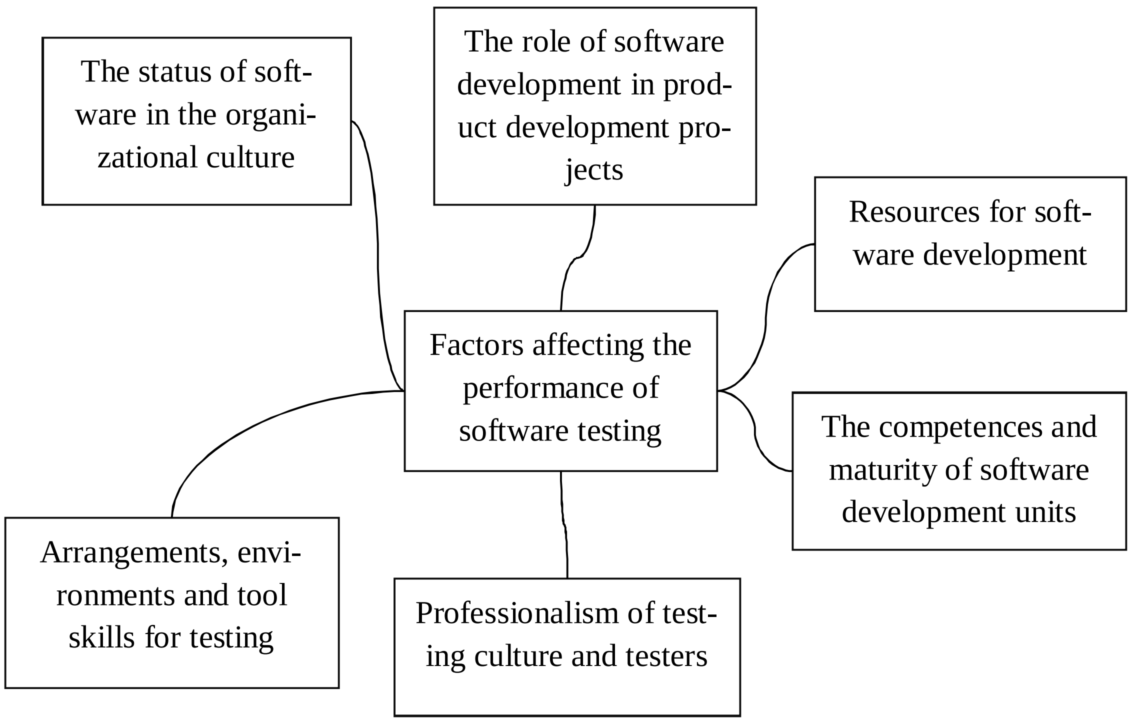

The status of software in the organizational culture
The role of software development in product development projects
Resources for software development
Factors affecting the performance of software testing
The competences and maturity of software development units
Arrangements, environments and tool skills for testing
Professionalism of testing culture and testers

Figure 37. Some factors affecting the performance of testing

Quality policy and safety policy

Good quality and safety cultures are requirements of good testing. A critical issue for both is the top management's true values, which greatly influence the resources for testing, the development of testing and the possible compromises in the pressures of the projects. A quality policy is a written statement of the way the top management sees the concept of quality, the importance of good quality and with which principles the company aims to achieve it. The policy is a requirement of quality management system standards. Such a policy should also be present for safety and risk management.

Process maturity

Safety-critical development is by nature a somewhat process-heavy and systematic activity. This is important because the process requires co-operation between many parties and needs to be very auditable. All this also applies to testing within the process. Where the development process is guided by safety standards and quality management system standards (like the ISO 9000 series [Wikipedia. ISO 9001.]), the testing practices can also benefit from some external guidance, such as the various testing maturity models, for example, TMMi [The Test Maturity Model Integrated 2011] and TPI [Test Process Improvement 2009].

The syllabi of the global certification system ISTQB [2011] also present an approach to testing that, in many ways, is quite compatible with safety-critical development. Their application does not mean that they guide the process development, however, which should be a needs-based integrated activity, but they present elements of testing that should be looked into.

4.4.1.3 Some principles of good testing

Understanding that testing can never be complete and that systems are never flawless

Testing can never prove that there are no errors in software and, in fact, it should be assumed and accepted that there are always errors in software. That is why redundancy is needed – even if one system fails due to an error, another can still perform the task. We need to understand that no amount of verification and validation will prove that the system is flawless.

Negative testing emphasized

The importance of negative tests needs to be emphasized. Positive tests verify that the system does what it is intended to do, but negative tests verify that it handles all disturbances correctly. As systems become larger and more complicated, this will become increasingly important.

Expecting the unexpected

Good testers realize that their understanding of the system is never complete and that every system is full of surprises. The tester must therefore have an open mind on all issues. Exploratory testing is therefore recommended to complement the systematic testing. In exploratory testing, the tester observes the system's behaviour and lets the observations guide the testing. One form of this testing is to try to break the system – perform harsh, unexpected actions and see how the system responds (in the manner of pulling the plug).

Sources of test conditions and test cases

There need to be various sources for the test conditions and test cases as shown in Figure 38. It is a generally accepted fact that no requirements specification can be complete or sufficient – if it were, it would be too large to be usable and require too many resources to create and maintain it. Thus, many of the sources of test conditions and test cases come from other sources and are often informal. Companies have experiences of developing similar systems, and testers know what kinds of issues need to be tested. Analyses during the development bring many factors to light. The creation of good tests is a continuous experience, and restricting the test to just any set of formal requirements would be a serious error.

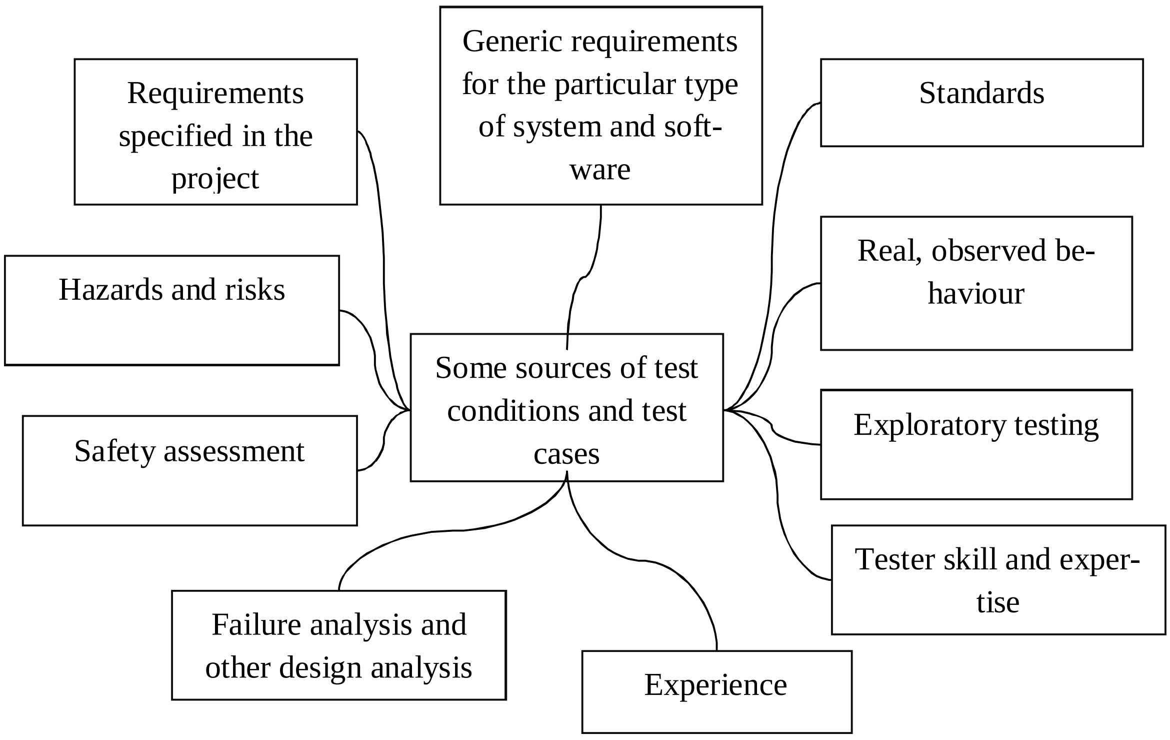

Generic requirements for the particular type of system and software
Requirements specified in the project
Standards
Hazards and risks
Real, observed behaviour
Some sources of test conditions and test cases
Safety assessment
Exploratory testing
Tester skill and expertise
Failure analysis and other design analysis
Experience

Figure 38. Sources of test conditions and test cases are not restricted to requirements.

Testing is started early in the development and carried on continuously

If testing is left late in the development process, any problems found will be hard to correct and will cause delays to the development. Even though validation should be based on carefully controlled and frozen configuration, it does not mean that testing should not begin as soon as there is something testable. This will provide early detection or errors and a good possibility of learning about the system's behaviour. For the purpose of learning, testing should not be left just to the testers, as designers need to be able to study through testing how the designs really work. While the testing may and should have a clear definition of the roles and responsibilities, collaboration is needed in all tasks.

Enabling early testing means that there should be facilities for performing testing before the actual integration of the software into the target hardware and the overall system. Here, emulators and simulators play a key role and all participants should have those at their disposal, including subcontractors.

Testing should be an ongoing activity and be carried out practically continuously at all test levels (module / unit testing, testing at all integration levels, testing of the overall system, testing of all architectures). Independently of the development process used, there should be a goal of running tests at module level, all integration levels and overall system level, practically continuously or at least in many test rounds of varying maturity.

Risk-based testing

Risk-based testing emphasizes more thorough testing of the system functions that carry the greatest risk. This approach is an obvious choice for safety-critical systems.

Testing should start with those functions that contribute most to safety. The assignment of risk levels to requirements is needed for this – a general SIL number is not sufficient.

Testing should begin with the highest risk items and then move to items that have a lower risk level. This does of course imply that the high risk items should be designed and implemented before others so they can be tested first.

Testing is important at all abstraction and integration levels

While the validation of software is mostly carried out at the overall system level, all test levels are very important in the creation of a safe and robust system.

Realistic test automation

Test automation can be important in making testing more efficient. One should never rely too much on test automation however. It always needs to be combined with good manual testing. Test automation is at its best in basic regression testing in which, after any changes, it can be verified that there are no adverse effects from the changes to the rest of the system.

Diversity in test practices and methods

Good testing consists of a rich collection of different approaches and methods. In the field of hazard and reliability analysis it has long been noted that one single method and approach can only produce so much information, and multiple approaches are therefore needed. The same applies to testing. Many approaches should be practised and there should be wariness of limiting them to any ‘school of thought’. Systematic testing, agile testing, test automation and manual testing, model-based testing and other forms can find a place in the same project. It may even be advisable to ensure that the testers have a varied educational and testing course background!

Configuration management

Configuration management is essential in any testing, as tests do not have much relevance if it is not known of what the system under test really consists. This is especially important during validation. All activities should be traceable back to the configuration and requirements. This can be a tedious task but can be helped with good, modern information systems.

The role of hardware configuration is obviously important. In traditional software testing, the hardware may only have a role as a test environment, but, in, for example, machinery applications, it is an integral part of the system under test, and it is this whole that is verified already during emulator or simulator testing. Thus, when, in traditional software testing, any changes in configuration can be made in order to extract failures, in this context it does more than that – it changes the system under test. This is something to be very careful about.

Testing is only one means of assessing the system

In many contexts, testing can have too large a role in determining the quality and acceptability of a system. Analytic methods need to be used in assessing the system, including reviews, failure and reliability analyses, architecture assessment, usability assessment, etc. Many of these are required by the safety standards, and it should be understood that they all form the whole of the project’s quality and safety management and are not separate things but closely connected and need to be used together more. For example, failure analysis should not just be a check of the system, but a task that produces new knowledge for testing.

Testing does not improve the system – it just produces information

This should be obvious, but, perhaps, unfortunately, it is not and no amount of testing will make a system better. It only improves the knowledge of the system and allows us to make better decisions about it. We therefore need to look into the timing of testing and how it produces the information so that the right decisions can be made.

Testers need to have the required competence

Testing is an area of expertise in which, for example, programmers are not sufficiently skilled based on just their education. It is a discrete area of expertise with its own professionals. For any safety-critical task, it is important that all participants receive sufficient training. For test engineers, this not only requires traditional tester training and quality training but also training in safety issues, hazard and risk analysis and reliability engineering – so they can use those activities and participate fruitfully in them. Even after training, however, a tester will require experience to be independently able to produce trustworthy testing. This implies that companies should have experienced testers in their projects and pay attention to transferring the experiences to new tester generations.

Suitable independence

Testers and the testing process need to have a level of independence from development so that there are no adverse psychological influences on the test designs or interpretation of test results. This is an important issue already in verification testing and obviously critical to validation testing.

...and co-operation and collaboration between designers and testers

Independence should not mean a lack of co-operation. It should be clear that developers and testers need to co-operate in all the development process phases and work in close collaboration in some of them. For example, failure analysis is a task in which all input from varying sources is critical.

Assuring testability

Testability is an important issue and concerns all system elements from architecture and the selection of technologies to implementation. Anything that is specified, designed or planned should be testable by manual or automated means – preferably both. Testability of system under development should be assured in the early stages of development. Besides design guidelines and testability consideration in the specifications, a special testability review is often recommended.

More information

The Ohjelmaturva publication ‘Safety Process Patterns In the Context of IEC 61508-3’ [Vuori et al. 2011] includes many testing-related patterns. The publication ‘Agile Development of Safety-Critical

Software' [Vuori 2011] contains a discussion on testing when agile software development methods are used. For a discussion on model-based testing, see Chapter 4.4.1.3 of this report.

References

Ohjelmaturva publications:

Vuori, M. 2011. Agile Development of Safety-Critical Software. Tampere University of Technology. Department of Software Systems. Report 14. 95 p. Available at: http://urn.fi/URN:NBN:fi:tty-2011061414702

Vuori, M., Virtanen, H., Koskinen, J. & Katara, M. 2011. Safety Process Patterns in the Context of IEC 61508-3. Tampere University of Technology. Department of Software Systems. Report 15. 128 p. Available at: http://urn.fi/URN:NBN:fi:tty-2011061414701

Other references:

ISTQB. 2011. International Software Testing Qualifications Board. [Referenced 30/05/2011]. http://istqb.org/display/ISTQB/Home

Test Process Improvement. 2009. A Step-by-step guide for improving your test process. Available at: http://www.sogeti.ie/Documents/Publications/Sogeti_Testing_Test_Process_Improvement_TPI_Step_by_step_guide_150609.pdf

Wikipedia. ISO 9001. [Referenced 30/05/2011]. Available at: http://en.wikipedia.org/wiki/ISO_9000

The Test Maturity Model Integrated (TMM*i*). 2011. TMM*i* Foundation. [Referenced 30/05/2011] Available at: http://www.tmmifoundation.org/html/tmmiref.html

4.4.2 Model-based testing

Heikki Virtanen, Tampere University of Technology, Department of Software Systems

4.4.2.1 Motivation

Testing is and will continue to be the most important method of verifying and validating systems, as there is no practical and applicable alternative. System behaviour has to be examined in an environment that is at least as hostile as the real production environment and at the level of the real implementation. This kind of set-up has so many details that the system cannot be checked thoroughly with formal methods, which would be the most obvious option for testing in a safety-critical setting. Formal methods can help in testing a system however.

Testing is usually based on test cases of some kind. A test case defines inputs and some kind of preconditions, operations to be performed and expected outputs, as well as post-conditions. The execution of the test case means that, first, a system under test (SUT) and its environment are brought to the state that meets the precondition, then defined operations are performed and the end state of the SUT and the environment are checked against the expected results and defined post-condition.

A large number of test cases are needed at several levels of testing. The V-model of testing [IEC 61508-3 2010, Fig. 6], for example, gives at least the following testing levels and sets of test cases:

- Test cases based on the requirements for validation and system level verification
- Test cases based on the architecture design for integration testing
- Test cases based on the detailed design and code for module testing.

In practice, the software development process is not as straightforward as the V-model suggests. There are many more goals of testing, and test cases are discovered all the way through the process, which is usually iterative in any practical project. A single test case is also very small and narrow at its logical size. In order to cover a practical set of features, many test cases are needed. For these reasons, the management and maintenance of the test cases are laborious, and in the product line environment they can turn out to be a real nightmare.

Another consequence of the small logical size of a test case is that, in practice, it is impossible to test reactive and concurrent systems adequately with manually maintained test cases. For example, if a system has a single shared resource, the number of required test cases for testing the shared access to that resource is comparable to the product of the number of test cases required to test the components using the resource in isolation.

4.4.2.2 Solution

One possible solution to the maintenance problem of test cases is to generate tests automatically based on a special *model*. In many cases, the features to be tested can be presented in a very compact form in a formal and machine-readable model. This kind of approach is mostly used to test the dynamic behaviour of the system, as the dynamic behaviour is easily and intuitively expressed with computationally well-behaved formalisms, most often with *state machines* of some kind.

There are two different kinds of models for model-based testing. System models are like abstract implementations or design models and describe the expected behaviour of the SUT from the internal viewpoint. Other kinds of models, confusingly called test models, have an external viewpoint of the behaviour of the SUT and describe what actions the environment can take to stimulate the SUT and the way the SUT is expected to respond. These approaches have no significant differences from the point of view of expressiveness [Malik et al. 2010].

Test modelling resembles manual exploratory testing when test models are used. A real SUT and/or documentation, whichever is available, is/are examined and simulated manually, and observations concerning the behaviour of the SUT and its environment are specified in the model. When done this way, test modelling is a powerful tool for inspecting the requirements and results of analysis, and most of the errors are found during modelling.

Model-based testing can be used off-line, which means that after test case generation and selection, test cases are used as in traditional testing, but there is a more powerful alternative, *online testing*. In online testing there is real-time interaction between the test generation of heuristics and the SUT, and the test run can be arbitrarily long. Another benefit is that a non-deterministic SUT can be tested far more easily, as unexpected, but otherwise legal, response guides test generation in an appropriate direction instead of yielding an inconclusive result. In some sense, online testing resembles a game for which the purpose of the heuristics is to win the SUT by finding a situation in which the response to an event does not match what was specified.

In a safety context, the online model-based testing opens attractive possibilities. For example, with an appropriate test model and testing environment, the test automation can drive and examine an SUT and its environment simultaneously and some of the risk-based testing can be performed automatically. More information on online testing in general can be found in Jääskeläinen et al. [2009].

4.4.2.3 Impediments and open issues

The workflow of model-based testing is very different compared with the workflow of traditional testing, and specialized tools and expertise are required. These factors have led to some obstacles when deployed in industrial environments [Janicki et. al 2011]. Most of these issues are related to training, changes in tasks, the capabilities of tools, etc., which are not specific to developing safety-related software.

Some issues, such as those related to test metrics, are more influential in a safety context than in ordinary software development. The quality of online model-based testing cannot be assessed with traditional metrics alone and there are no compatible, well-established metrics for it yet. Fortunately, the measurement of code coverage is independent of the testing method.

Models for testing have the same drawbacks as other formal descriptions. They can and will have errors that are hard to find solely by means of inspection. This impediment can be solved mostly by testing models and running tests early and often. Test-generation heuristics find structural errors. Other issues found are examined and taken care of by modifying the model or the system being developed, or both.

References

IEC 61508-3 Ed2.0. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems, Part 3, International Electrotechnical Commission. 234 p.

Janicki, M., Katara, M. & Pääkkönen, T. 2011. Obstacles and opportunities in deploying model-based GUI testing of mobile software: A survey. Software Testing, Verification and Reliability. To appear.

Jääskeläinen, A., Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Takala, T. & Virtanen, H. 2009. Automatic GUI test generation for smartphone applications – an evaluation, in 'Proceedings of the Software Engineering in Practice track of the 31st International Conference on Software Engineering (ICSE 2009)', IEEE Computer Society, Los Alamitos, CA, USA, pp. 112–122 (companion volume).

Malik, Q.A., Jääskeläinen, A., Virtanen, H., Katara, M., Abbors, F., Truscan, D. & Lilius, J. 2010. Model-based testing using system vs. test models – what is the difference?. In: 'Proceedings of the 17th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2010) (poster session)', IEEE Computer Society, Los Alamitos, CA, USA, pp. 291–299.

4. Phases of software development

Royce, W.W. 1987. Managing the development of large software systems: concepts and techniques. In: Proceedings of the 9th International Conference on Software Engineering, Monterey, CA. Pp. 328-338. IEEE Computer Society.

5. Models and methods for safety-related software production / Technical issues and related non-functional factors

Figure 39 shows the waterfall model tasks that need to be considered. The model shows tasks that are described in many standards, but there are also other aspects, such as selecting requirements and issues related to COTS. In the figure, the right side shows some dependencies on information sources.

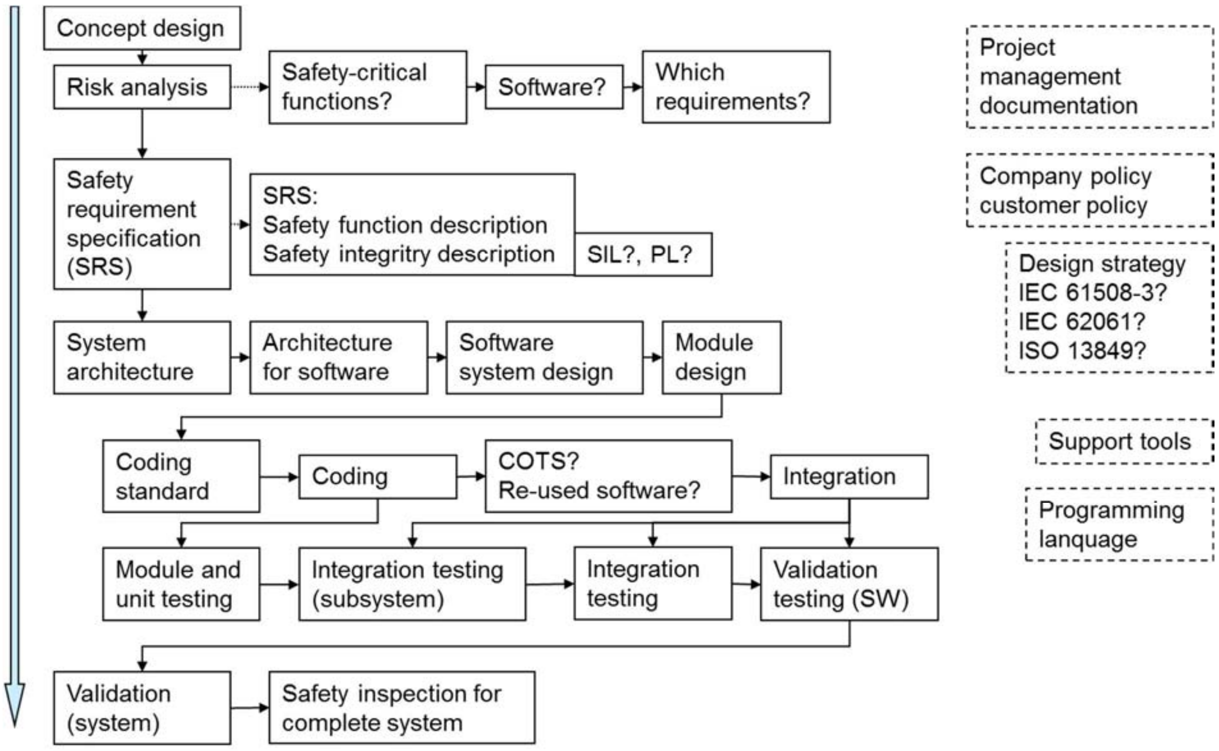

Concept design
Risk analysis
Safety-critical functions?
Software?
Which requirements?
Safety requirement specification (SRS)
SRS: Safety function description Safety integritry description
SIL?, PL?
System architecture
Architecture for software
Software system design
Module design
Coding standard
Coding
COTS? Re-used software?
Integration
Module and unit testing
Integration testing (subsystem)
Integration testing
Validation testing (SW)
Validation (system)
Safety inspection for complete system
Project management documentation
Company policy customer policy
Design strategy IEC 61508-3? IEC 62061? ISO 13849?
Support tools
Programming lanquage

Figure 39. Design process of safety-related software and aspects that a designer needs to consider.

5.1 Criteria for choosing methods

There are many methods for the different phases of software development. It is not feasible to apply all the methods that have been found; the applicable methods have to be selected. Some methods are mandatory and some are optional, but the designer should bear in mind the origin of each requirement.

Table 4 shows aspects that should be considered when choosing methods for software development or V&V. The list is concluded by checking standards and software defect statistics, and some ideas have arisen from project workshops.

Table 4. Criteria for selecting methods for the SW lifecycle.

Question	Comments
Is the method highly recommended for specific SIL/PL?	If the IEC 61508-3 standard is applied, then all highly recommended methods should be applied. If they are not, then some other methods should be used to cover the required properties. Good reasoning for the change of method is required. The same applies to SFS-EN ISO 13849-1 and IEC 62061 requirements.
Is the method recommended for specific SIL/PL? Is the method the best choice?	At least one of similar properties possessing the recommended methods should usually be applied.
Is the method quick and easy to use? Is the value cost/bug good?	The value cost/bug is usually better for methods applied in the early phase of the design. It must not be the only criteria.
Efficiency. Is the method efficient? Is the proportion of detected bugs good?	Good efficiency is better, but it is not the only factor to be considered. Efficiency is better when a specific property is considered for the first time.
Can the method reveal defects that are not considered before well enough? Does the method cover risks that are not considered sufficiently in previous analysis?	Does the method cover and reveal any new weaknesses of the program? If it is related to the same weaknesses as the previously chosen method, then it should be considered which method is required (one/both).
Are all fault types, properties and parts of the system covered?	Are there specific fault types or weaknesses that could be covered better with the new method than with the chosen methods?
Are all phases of the development process covered? Does the method cover a part of the process that is lacking?	All phases of the development process should include protective measures.
Does the method support the already chosen methods?	Can the results, tools or features of previously chosen methods be applied with the new method?
Credibility. Is the method well known? Do the customers respect the method? Do other parties respect the method (notified body, authorities)?	The methods mentioned in the standards have good credibility. The companies' own methods often have low credibility, though they may be good. New methods should be well documented.
Experience. Is there experience of the method? Tools available?	It is efficient to apply a method that is familiar to the users. It takes time to learn a new method. It is good to have tools to apply the method.
Does the method comply with the complexity and extent of the project?	The bigger the project, the more and better the methods needed. In big projects, quality is an important factor.
Resources. Resources available, cost of resources.	The resources and time should be in accordance with

Question	Comments
Time. Amount of time to the next milestone.	the work needed to fulfil the requirements.
Future. Is the future of the method promising?	Will the method and the tools be developed? Will they be compatible with other systems?
Company policy. Do the licences comply with company policy? Does the quality meet company policy? Do the security aspects comply with company policy?	Is there anything special or new related to the method or tools? The aspects can be positive or negative?

5.2 Formal methods in the safety context

Heikki Virtanen & Mika Katara, Tampere University of Technology, Department of Software Systems

5.2.1 General

Formal methods are rigorous with mathematically solid notations and analysis methods that cover a wide range of different techniques for specifying, designing, verifying and validating systems. Formal methods are used in contexts in which faults are unacceptably expensive or in which the state of the art methods, like informal deduction and testing, are not sufficient due the essential characteristics of the systems designed. Examples of these include complex integrated circuits, such as microprocessors, and concurrent systems, such as data transmission protocols. Even if these fields are not directly related to machinery, the methods and tools developed are applicable because formal methods are based on the theory of computation and are mostly independent of the application domain.

Formal methods have various benefits over other techniques because the syntax and semantics of the formal descriptions are unambiguous. When working with natural language, pseudo code, semi-formal graphical notations, etc., the meaning of the descriptions is always somewhat vague and incomplete. With formal methods, any incompleteness and errors are found earlier, many even in the requirement capture and specification phases.

The benefit was one of the motivations for an experiment we conducted [Jääskeläinen et al. 2011]. We verified a simple two-out-of-three voting system with modern verification tools called CBMC and EBMC (www.cprover.org). As input to the experiment, we drew up an informal specification and proposed a solution written in pseudo code. Since the inputs to CBMC and EBMC are C language and Verilog code, in which assertions are used for verification, we manually created such artefacts first (the Verilog code was produced by a translator from VHDL code). After that, the tools were used to verify that none of the inserted assertions could ever be false.

These kinds of automated tools can easily prove that code or a formal model is correct with respect to the formal specification (i.e. in this case assertions), but the specification can still be incorrect. It is unacceptably easy to write assertions that do not test the right thing. We used error seeding to verify assertions, but it would be better if there was a formal specification and rigorous transformation from the specification to assertions.

Separating the assertions from the production code is another relevant issue. In order to reuse existing assertions, when production code has to be modified, the assertions and the production code may not be coupled too tightly. When working with Verilog, the issue is solved quite easily as the assertions constitute a separate block. Furthermore, there are special techniques such as literate programming or aspect-oriented programming, with which the code can be combined from separate fragments. However, tool support is needed and as such may not always be available.

5.2.2 Application of formal methods

New editions of the safety standards emphasize the use of formal methods. They are highly recommended at high SIL levels and not using them has to be justified. One reason for not using them is that very few, if any, proven tools are available. This issue does not make formal methods, nor the skills required to use formal methods, unusable in safety contexts. For instance, formal methods and related tools can support and enhance traditional practices, and even the formal way of thinking has an influence on everyday working habits and helps to construct better quality products.

For example, a skilled person with knowledge of predicate logic and set theory can go through the program verification steps in his/her mind while coding or reviewing the code. This way, many annoying mistakes, like off-by-one errors or endless loops, are avoided or at least found straightaway without the tedious debug and repair cycles. Similar benefits can be achieved in analysis and design phases by using mathematical and semantically unambiguous formulas in descriptions.

The use of formal methods embedded in the traditional software process is one natural option to initiate their use. Another option is to purchase formal modelling or verification as a service, if the required investments are considered too expensive to compare limited needs. Experiences from successful pilot projects can be used as training material and in projects to come.

5.2.3 Open issues

The excursion into the world of formal methods left many open questions and led to attractive ideas. Intel, for example, has replaced a significant part of pre-silicon testing by formal verification [Kaivola et al. 2009]. To what extent is it possible to replace descriptions and techniques with the formal alternatives that are available today and how can they be used to break the workflow of the common V-model in order to obtain more valuable feedback earlier? Such early and automatic feedback is one of the mandatory prerequisites of agile software development, as discussed earlier in this report.

It is very difficult to construct a good product without good requirements, and safety standards therefore emphasize the capture of safety requirements. To express and check those requirements, there is an interesting notation called Safecharts [Dammag & Nissanke 1999]. With the hierarchical representation of the state and joint actions, Safechart seems to be closely related to the specification technique called DisCo [Kurki-Suonio 2005]. In turn, DisCo provides a means to create specifications incrementally in a rigorous way. If the hypothesis about the relation between DisCo and Safecharts is correct, it could move the safety requirement capture and inspection to the next and more rigorous level.

References

Ohjelmaturva publications:

Jääskeläinen, A., Katara, M., Katz, S. & Virtanen, H. 2011. Verification of safety-critical systems: A case study report on using bounded model checking. In the Proceedings of the 6th International Workshop on Systems Software Verification (SSV 2011), Nijmegen, the Netherlands, Aug, 2011.

Other references:

Dammag, H. & Nissanke, N. 1999. Safecharts for specifying and designing safety critical systems. In: Symposium on Reliable Distributed Systems, pp. 78–87.

Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodova, A., Taylor, C., Frolov, V., Reeber, E. & Naik, A. 2009. Replacing testing with formal verification in Intel Core processor execution engine validation. In: CAV 2009, LNCS 5643. Springer.

Kurki-Suonio, R. 2005. A Practical Theory of Reactive Systems: Incremental Modeling of Dynamic Behaviors. Springer-Verlag. 418 p.

6. Discussion

Timo Malm, VTT, and Jari Rauhamäki, Tampere University of Technology, Department of Automation Science and Engineering

Role of safety-critical software

The role of safety-critical software has increased in recent years. Hardware solutions in safety-critical applications are often replaced by software-based solutions as safety functions become more complex. This has been made possible by the emergence of suitable software-based safety systems and a change in the industry mindset. The latter circumstance has been crucial, as technology can only be applied when the mindset supports it. One of the reasons to change a paradigm must be the possibilities enabled by software-based safety systems. Software-based safety systems enable a more flexible and thus optimized way of implementing safety-related functionalities. In order to apply software-based safety systems, however, the software needs to be developed to meet the requirements of the safety-related system.

Faults and statistics

Faults can be introduced into software at every phase of its lifespan from the initial idea to maintenance. Studies have shown that the requirement and design phase of the development produces most of the defects of delivered software (see Section 1.2). This goes against the common association that the coding phase is the phase in which most defects are introduced, though this association is half true. The coding phase produces many defects, but these are mostly identified by testing and then corrected. Defects in the requirements and (architectural) design phases, however, are not as well identified and thus remain in the delivered software.

Only catastrophes or bigger hazards caused by software are well reported, and this may skew the statistics of reported defects. Famous catastrophes are typically related to large and complex programs, which (should) have been well validated. It cannot be seen from the statistics which phase of development is most critical for small programs. We only have general statistics, which give average reliability values. It appears that the problems are less significant in small programs.

The defect types depend very much on the systems. Machines such as robots, power presses and machine tools have programs that need to be changed weekly or even daily, and the software can be safety-critical. The wrong speed or programmed tool (e.g. the specific size of a grinding tool) for a machine can cause the tool to break, and the protective guards are not dimensioned for extreme speeds. Such a fatal accident happened in Finland in 2006 [TOT 1/2006].

Reliability – safety

There is a difference between excellent, reliable software and safe software, but the statistics shown in this paper only relate to reliability. According to the definitions, reliability is related to how well a system can fulfil the requirements specification, whereas safety is freedom from accidents. A system can be reliable if it fulfils a flawed requirements specification, but a flaw in a specification affects design and may make a system unsafe. Safety systems are quite often designed to be fail-safe. This means that if a system fails, it does not cause any danger. A fail-safe system is often less reliable than a standard system as it contains more components than a similar standard system. The difference between safe and ordinary software can be seen in field buses. Safety buses have longer checksums than the corresponding ordinary field buses (e.g. Profisafe 32 bits and Profibus 16 bits) [Alanen et. al. 2004]. Safety buses detect more faults, and the return back to normal communication is also more complex in safety buses (e.g. manual acknowledgement). To the user, the safety bus may appear less reliable as the defects in the standard bus are seldom severe and the user can observe more defects in the safety bus. Another example of conflicting safety and reliability requirements relates to military applications. A gun that cannot fire is safe but unreliable. Architecture can also have an effect on safety and reliability. A duplicated 2oo2 system is more reliable than a 1oo1 (single channel system) or 1oo2 system, but it is less safe. In software systems, similar duplicated software does not have much effect on safety or reliability, but diverse software has an effect on both. In some cases, there is even a contradiction between safety and reliability requirements. There is also a difference between software and hardware reliability functions. Software reliability increases in the long run whereas hardware reliability decreases. The more the software is applied without failures, the more reliable it is.

Requirements

Safety-critical software is typically developed to comply with a certain safety-related standard. In fact, the whole field of safety-critical software development (and the machinery domain in general) is standardized in the sense that laws and regulations typically demand compliance with a certain standard. The standards applied to safety-critical software systems include, for instance, IEC 61508-3 [2010], SFS-EN ISO 13849 [2008], SFS-EN 62061 [2005] and the ISO 25119 family (for agricultural and forestry machinery).

The safety-related standards consider the development process, techniques and methods of safety-critical software development. The IEC 61508-3 [2010] standard was specially analysed during the project, as process patterns were identified from the standard. The patterns describe the mindset and guidelines for the way a certain assignment could be carried to ensure compliance with the requirements of the standard.

Development process

The development of safety-critical software has traditionally resembled the V-model process. The V-model development process is suggested by many safety-related standards, e.g. IEC 61508 [2010]. The model defines clear phases, their outcomes and the testing procedures carried out in the development process. Although the V-model is suitable in safety-critical software development, it is not the only option. Nowadays, agile development methods are increasingly popular and there is interest in applying these methods to safety-critical as well as non-safety-critical software development. One point is that in the machinery sector, the code of the old machine usually already exists, and creating the code for the new machine is an iterative process. In the Ohjelmaturva project, the suitability of agile methods for the development of safety-critical software was studied. Agile methods have many potential benefits over traditional software development processes relating to, e.g., gradual development, constant assessment of risks and emphasis on verbal communication among the development team. Potential obstacles were also identified however. As the agile methods are typically productivity-orientated, there is a risk of, for example, neglecting the safety and reliability needs and focusing on product delivery. Nevertheless, the outcome of the analysis shows that agile methods also have potential in the safety-critical domain as long as their strengths are harnessed to serve safety and reliability causes.

<u>Architecture in duplicated systems</u>

The IEC 61508-2 and SFS-EN ISO 13849-1 offer the designer a new possibility in redundant systems of determining the required SIL or PL of a single channel. In duplicated systems, the required systematic capability (related to SIL or PL of subsystems) of one channel can be one step lower than the system capability. For example, if the required SIL is 2, the architecture of the system may consist of two parallel and independent SIL1 subsystems. The software can be SIL1 when it is different in the channels and other dependencies are considered (see the following citation and Figure 40).

> *IEC 61508-2: 7.4.3.2 For an element of systematic capability SC N (N=1, 2, 3), where a systematic fault of that element does not cause a failure of the specified safety function but does so only in combination with a second systematic fault of another element of systematic capability SC N, the systematic capability of the combination of the two elements can be treated as having a systematic capability of SC (N + 1) providing that sufficient independence exists between the two elements (see 7.4.3.4).[IEC 61508-ed2 2010]*

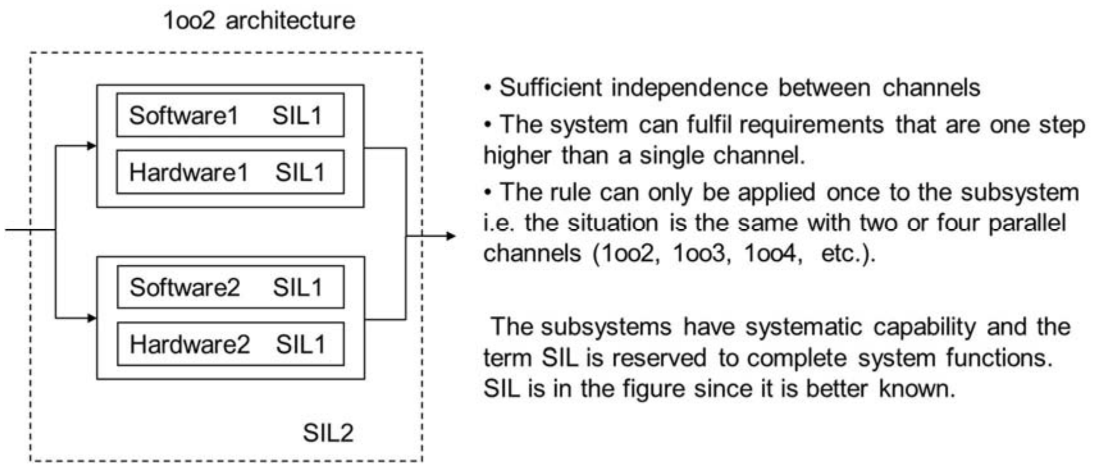

1oo2 architecture
Software1 SIL1
Hardware1 SIL1
Software2 SIL1
Hardware2 SIL1
SIL2
• Sufficient independence between channels
• The system can fulfil requirements that are one step higher than a single channel.
• The rule can only be applied once to the subsystem i.e. the situation is the same with two or four parallel channels (1oo2, 1oo3, 1oo4, etc.).
The subsystems have systematic capability and the term SIL is reserved to complete system functions. SIL is in the figure since it is better known.

Figure 40. Systematic capability of a duplicated system.

Future research

Technology related to software is developing rapidly, and it is therefore challenging to create safety-critical software. New tools are and will be available for the development of such software, for example, certified modules, multicore processors in safety applications, automated safety code generation from models, etc. Currently, safety-critical software is often created with certified tools, but more tools will become available in the future.

New sophisticated safety functions for machinery are and will be needed in the near future, and they may require more complex software. Such safety functions could include, for example: steering, stability, limping mode, sensor fusion, self-diagnostics, versatile monitoring of the performance of functions, situation awareness, collision prediction, component lifecycle prediction, returning from an exceptional situation to a normal run, safe stand still, safe force/speed/position control of machine/tool/workpiece, automated mode changes, safety functions operating after an accident to minimize the damage, autonomic machines, machine fleet control, etc.

During the project, some research topics were found to be very promising for achieving results. They were verification and validation, architectures, model-based safety, software components and software reuse.

Small is beautiful

As Figure 41 shows, there are more defects in large programs than in small programs, and the number of defects increases exponentially as the program size grows. Fortunately, the safety-critical code in machinery is usually relatively small and is often run on safety PLCs. The situation is changing, however, since automated functions become more complex and quicker, manual actions are too slow and the need for complex safety software increases. Many advantages of small programs can be achieved by using modular design and by keeping the connections between modules well under control and narrow.

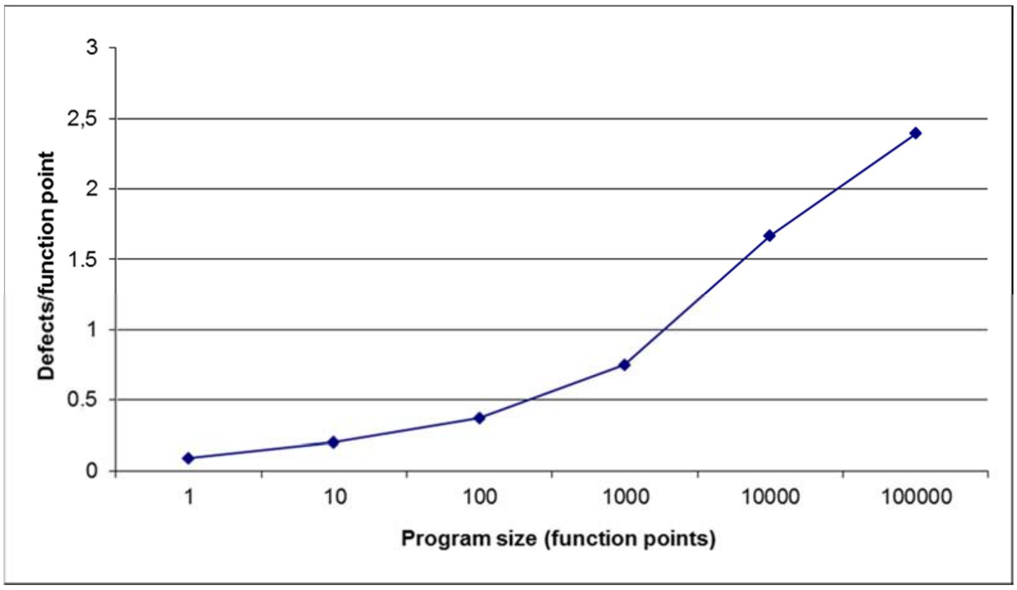


Figure 41. The defect density is higher in large programs than in small programs.

Leonardo Da Vinci: *“Simplicity is the ultimate sophistication.”*

References

Alanen J., Hietikko M. & Malm T. 2004. Safety of Digital Communications in Machines. VTT Research Notes 2265. http://www.vtt.fi/inf/pdf/tiedotteet/2004/T2265.pdf. 93 p + 1 app.

IEC 61508 ed2. 2010. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. International Electrotechnical Commission. 187 p.

IEC 61508-3 ed2.0. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements. International Electrotechnical Commission. 234 p.

IEC 62061. 2005. Safety of machinery - Functional safety of safety-related electrical, electronic and programmable electronic control systems. International Electrotechnical Commission. 205 p.

SFS-EN ISO 13849-1. 2008. Safety of machinery — Safety-related parts of control systems – Part 1: General principles for design. Finnish Standards Association SFS 180 p.

TOT 1/06. 2006. A piece of a grindstone flew towards the head of a machinist [In Finnish]. Federation of Accident Insurance Institutions. http://82.118.214.253:8080/tottipublic/totcasepublic.view?action=caseReport&unid=9

Appendix A: Terminology

The terminology of software safety is not clear as the terms come from different sources (IEC and ISO handbooks, electricity, communication, software development and occupational safety) and the meanings are different. SFS-EN 61508 [SFS-EN 61508-4 2010] has relatively broad expressions in its terminology, and many definitions therefore fit that standard. For example, the terms *fault* and *failure* have a long tradition in different branches and the definitions fit the applications. In electrical systems, a failure leads to a fault, but in communication systems it is vice versa.

Agile methods: "Agile methods generally promote a disciplined project management process that encourages frequent inspection and adaptation, a leadership philosophy that encourages teamwork, self-organization and accountability, a set of engineering best practices that allow for rapid delivery of high-quality software, and a business approach that aligns development with customer needs and company goals". [Wikipedia. Agile software development.]

Application software: Part of the software of a programmable electronic system that specifies the functions that perform a task related to the EUC rather than the functioning of, and services provided by the programmable device itself. [SFS-EN 61508-4 2010]

A (software) bug is the common term used to describe an error, flaw, mistake, failure, or fault in a computer program or system that produces an incorrect or unexpected result, or causes it to behave in unintended ways. [Wikipedia. Software bug.]

Commercial Off The Shelf (COTS): Generally available hardware or software component to be moved from one system to another. [Stålhane et al. 2000]

Common cause failure: Failure, which is the result of one or more events, causing coincident failures of two or more separate channels in a multiple channel system, leading to system failure. [SFS-EN 61508-4 2010]

Debug: The process of locating and eliminating errors that have been shown, directly or by inference, to exist in software. [Leguan 130 2011]

Defect: A failure of computer software to meet requirements. It is often considered as a software bug. [Wikipedia. Defect.]

Dependability: The dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable. [Avizienis et al. 2004]

Alternative definition: Trustworthiness of a computer system such that reliance can justifiably be placed in the service it delivers. The service delivered by a system is its behaviour, as its users perceive it; a user is another system (human or physical), which interacts with the former. [Koskimies & Mikkonen 2005]

Appendix A: Terminology

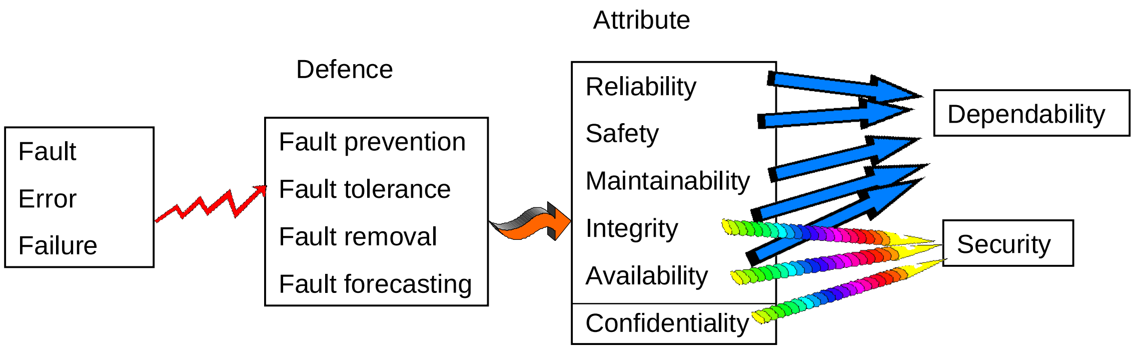


Figure 42. Some ways how to reach dependability. To reach security, also confidentiality must be considered.

Availability: readiness for correct service;
Reliability: continuity of correct service;
Safety: absence of catastrophic consequences
Integrity: absence of improper system alterations;
Maintainability: ability to undergo modifications, and repairs.
Prevention of development faults is a process, which includes development methodologies, both for software (e.g., information hiding, modularization, use of strongly-typed programming languages) and hardware (e.g., design rules).
Fault tolerance is carried out via error detection and system recovery.
Fault removal during the development phase of a system life-cycle consists of three steps: verification, diagnosis, correction.
Fault forecasting is conducted by performing an evaluation of the system behaviour with respect to fault.

Design pattern: They are proven solutions to recurring problems in software engineering. Design patterns are formally documented ideas that are used to prevent reinvention. Patterns provide proven solutions to specific problems where the solution is usually not obvious. The best patterns generate a solution to a problem indirectly. Good patterns do more than just identify a solution; they also explain why the solution is needed. [Leguan 130 2011]

Diversity: Different means of performing a required function. Example Diversity may be achieved by different physical methods or different design approaches. [SFS-EN 61508-4 2010]

E/E/PE: Electrical/electronic/programmable electronic. [SFS-EN 61508-4 2010]

E/E/PES: Electrical/electronic/programmable electronic system. [SFS-EN 61508-4 2010]

Embedded software: Software, supplied by the manufacturer, that is part of the safety related electrical control system (SRECS) and that is not normally accessible for modification. Firmware and system software are examples of embedded software. [Leguan 130 2010]

Error: Part of a system state that is liable to lead to a subsequent failure. It is a manifestation of a fault in the system. [Koskimies & Mikkonen 2005]

Alternative definition: Discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition. [Wikianswers 2010]

Error Handling: An implementation mechanism or design technique by which software faults are detected, isolated and recovered to allow for correct runtime program execution. [Haikala & Järvinen 2004]

Equipment under control (EUC): Equipment, apparatus, machinery or plant used for manufacturing, process, transportation, medical or other activities. [SFS-EN 61508-4 2010]

Fault, error and failure: The basic concepts for dependable computer-based systems are fault, error and failure. Since long they are defined and established in research on fault-tolerant computer systems. In certain respects the terminology differs from the standards for software engineering and for reliability. This is quite natural as the prime interest when studying dependable systems is the handling of defects in a computer-based system.

A *fault* is an impairment that exists in the system or in the usage of a system. A fault can be a design defect, an illegal input or a hardware failure. Normally a fault is dormant and if never activated it will never affect the behaviour of the system. Users can perceive a system as perfectly reliable if the faults never are activated and the system always behaves as expected and specified. If a fault is activated it will cause an *error* in the system, which means that the status of the system deviates from the designer's intention. If this erroneous state affects the external behaviour the system fails in giving service according to specification and we have a *failure*.

Faults are always hidden. Only errors can be detected as they in other engineering disciplines can be quantified. Once an error is detected we can:

- confine it, so the damage will not spread,
- diagnose it, so we know what measures to take to and
- treat it, so we can restore the system to its normal state.
- If we do not succeed in error detection and recovery, the error may propagate over the system boundary and cause a failure.

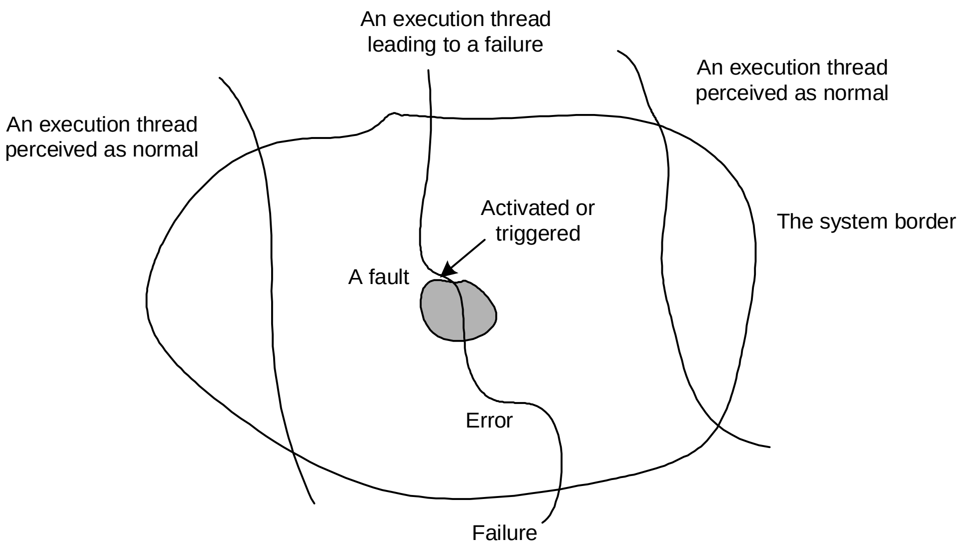

An execution thread leading to a failure
An execution thread perceived as normal
An execution thread perceived as normal
The system border
Activated or triggered
A fault
Error
Failure

Figure 43. The fundamental fault – error – failure chain. [Malm et al. 2007].

The notion of fault, error and failure is recursive. A failure in a component of a system is a fault on the next higher level. A failure in an integrated circuit may cause an output signal to be stuck at zero, which is a fault in circuit board. A programmer's failure in writing correct source code for a program results in a fault in the running program. [Avizienis 2004]

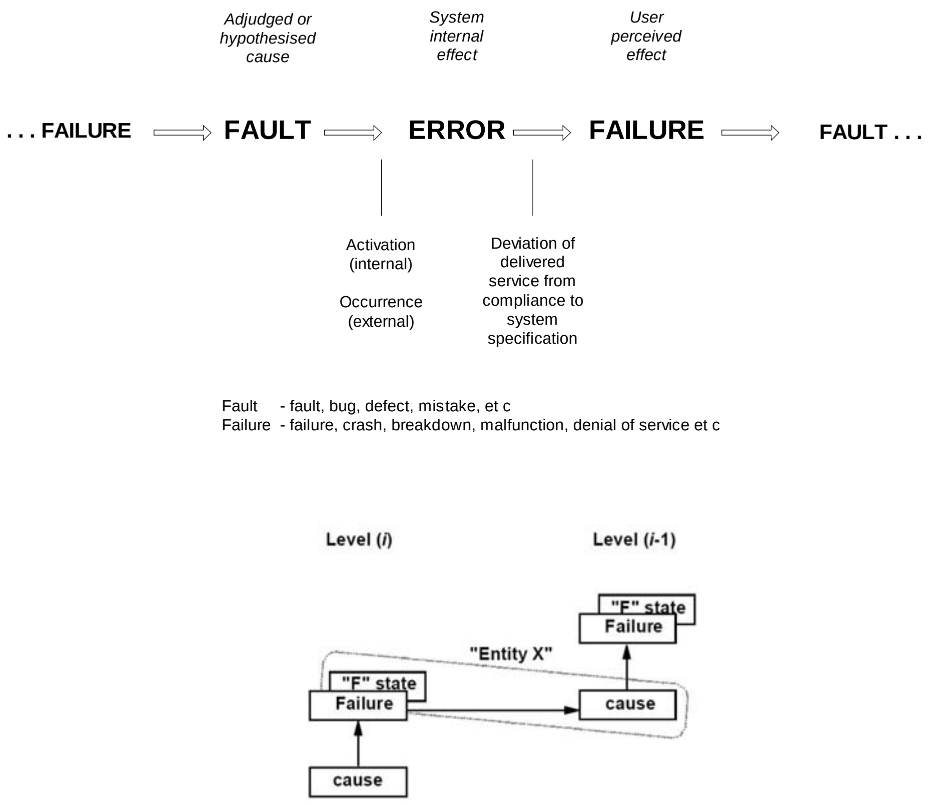

Adjudged or hypothesised cause
System internal effect
User perceived effect
. . . FAILURE
FAULT
ERROR
FAILURE
FAULT . . .
Activation (internal)
Occurrence (external)
Deviation of delivered service from compliance to system specification
Fault - fault, bug, defect, mistake, et c
Failure - failure, crash, breakdown, malfunction, denial of service et c
Level (i)
Level (i-1)
"F" state
Failure
"Entity X"
"F" state
Failure
cause
cause

Figure 44. The hierarchical relationship between failure and fault [below SFS-EN 61508-4 2010].

Alternative definitions used in hardware technology:

Fault: state of an item characterized by the inability to perform a required function, excluding the inability during preventive maintenance or other planned actions, or due to lack of external resources. A fault is often the result of a failure of the item itself, but may exist without prior failure.

Failure: termination of the ability of an item to perform a required function. After a failure, the item has a fault. “Failure” is an event, as distinguished from “fault”, which is a state. The concept as defined does not apply to items consisting of software only. [IEC 60050-191 ed1.0 1990]. [SFS-EN ISO 12100:en 2010]

Fault-tolerance: Ability of a functional unit to continue to perform a required function in the presence of faults or errors. [SFS-EN 61508-4 1990]

Firmware: Computer programs and data loaded in a class of memory that cannot be dynamically modified by the computer during processing (e.g. ROM). [Leguan 130 2011]

Full variability language (FVL): Type of language that provides the capability of implementing a wide variety of functions and applications. Examples: C++, Assembler. [SFS-EN ISO 13489-1 2008]

Functional safety: Part of the overall safety relating to the EUC and the EUC control system which depends on the correct functioning of the E/E/PE safety-related systems, other technology safety-related systems and external risk reduction facilities. [SFS-EN 61508-4 1990]

Hazard: A hazard is an undesirable condition that has the potential to cause or contribute to an accident. [Kopetz 1977]

Lean software development: Lean development could be summarized by seven principles: Eliminate waste, Amplify learning, Decide as late as possible, Deliver as fast as possible, Empower the team, Build integrity in, See the whole. [Wikipedia. Lean software development.]

Limited variability language (LVL): Type of language that provides the capability of combining predefined, application-specific library functions to implement the safety requirements specifications. Typical examples of LVL (ladder logic, function block diagram) are given in IEC 61131-3. A typical example of a system using LVL: PLC. [SFS-EN 13849-1 2008]

Maintainability (of a machine): Ability of machine to be maintained in a state which enables it to fulfil its function under conditions of intended use, or restored into such a state, the necessary actions (maintenance) being carried out according to specified practices and using specified means. [SFS-EN ISO 12100:en 2010]

Performance Level (PL). Discrete level used to specify the ability of safety-related parts of control systems to perform a safety function under foreseeable conditions. [SFS-EN 13849-1 2008]

Real-time system: A real-time computer system is a computer system in which the correctness of the system behaviour depends not only on the logical results of the computations, but also on the physical time when the results are produced. [Kopetz 1997]

Reliability: Dependability with respect to the continuity of service. Measure of continuous correct service delivery. Measure of the time to failure. [Koskimies & Mikkonen2005]

Rigour of technique: R1/R2/R3 criteria is useful for guidance purposes to make an informal link between the increasing level of rigour of the R1 to R3 progression and an increased confidence in the correctness of the software. [IEC 61508-3 ed. 2.0 2010]

Risk: A risk is combination of the probability of occurrence of harm and the severity of that harm. [SFS-EN ISO 12100:en 2010]

Safety: Freedom from unacceptable risk. [SFS-EN 61508-4 2010]

Safety case: A safety case is a combination of a sound set of arguments supported by analytical and experimental evidence substantiating the safety of a given system. [Kopetz 1997]

Safety-critical system: A system where a failure can cause damage on persons, property or the environment. In the reference [Kopetz 1997] this is synonymous with hard real-time computer system.

Safety function: Function to be implemented by an E/E/PE safety-related system or other risk reduction measures, that is intended to achieve or maintain a safe state for the EUC, in respect of a specific hazardous event. [SFS-EN 61508-4 2010]

Function of a machine whose failure can result as an immediate increase of the risk(s). [de Lemos et al. 2006]

Safety integrity: The probability of a safety-related system satisfactorily performing the required safety functions under all the stated conditions within a stated period of time. [SFS-EN 61508-4 2010]

Safety integrity level (SIL): Discrete level (one out of a possible four) for specifying the safety integrity requirements of the safety functions to be allocated to the E/E/PE safety-related systems, where safety integrity level 4 has the highest level of safety integrity and safety integrity level 1 has the lowest. [SFS-EN 61508-4 2010]

Safety lifecycle: Necessary activities involved in the implementation of safety-related systems, occurring during a period of time that starts at the concept phase of a project and finishes when all the safety-related systems are no longer available for use. [SFS-EN 61508-4 2010]

Safety-related system: A system that:

- implements the required safety functions necessary to achieve a safe state for the equipment under control, EUC, or to maintain a safe state for the EUC; and
- is intended to achieve, on its own or with other safety-related systems, the necessary level of safety integrity for the implementation of the required safety functions. [SFS-EN 61508-4 2010].

Safety-related control function (SRCF): control function with a specified integrity level that is intended to maintain the safe condition of the machine or prevent an immediate increase of the risk(s). [de Lemos et al. 2006]

Security: Dependability with respect to the prevention of unauthorized access and/or handling of information. [Koskimies & Mikkonen 2005]

Software isolation: Software isolation is method that separates COTS software from other software. The method can be based on software or hardware. [Stålhane et al. 2000]

SRASW: Safety-related application software. [SFS-EN ISO 13849-1 2008]

SRECS: Safety-related electrical control system. [de Lemos et al. 2006]

SRESW: Safety-related embedded software. [SFS-EN ISO 13849-1 2008]

SRP/CS: Safety-related part of a control system. [SFS-EN ISO 13849-1 2008]

SRS: Safety requirements specification. [de Lemos et al. 2006]

Trap Door: A trap door is a link to another part of a program which is unknown to the program developer and which may introduce an extra risk. [Malm et al 2007]

Testing: The act of subjecting to experimental test in order to determine how well something works. [WordNet Search - 3.1.2011].

Unit testing: A software testing methodology in which individual tests (unit tests) are developed for each small part of a program. [Wiktionary. Unit Testing.]

Integration testing: is the phase in software testing in which individual software modules are combined and tested as a group. [Wikipedia. Integration testing.]

Functional testing: •Testing the features and operational behaviour of a product to ensure they correspond to its specifications. [Software testing-glossary]

System testing: Conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements; a kind of black-box testing. [Wikipedia. System testing.]

Black-box testing: Testing that does not focus on the internal details of the program but uses external requirements. [Engineering Dictionary]

White-box testing: uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. [Wikipedia. White-box testing.]

Validation: Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled [ISO 9000].

Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled. [SFS-EN 61508-4 2010]

Verification: Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled [ISO 9000].

Confirmation by examination and provision of objective evidence that the requirements have been fulfilled. [SFS-EN 61508-4 2010]

Voter: A voter is a unit that detects and masks errors by accepting a number of independently computed input messages, and delivers an output message that is based on the analysis of the inputs. [Kopetz 1997]

Watchdog timer: An independent, external timer that ensures the computer cannot enter an infinite loop. Watchdog timers are normally reset by the computer program. Expiration of the timer results in generation of an interrupt, program restart, or other function that terminates current program execution. [NASA 2004]

References

Avizienis, A., Laprie, J-C., Randell, B. & Landwehr, C. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. Institute for Systems Research TR 2004-47. 36 p.

de Lemos, R. Guerra, P.A. de C. & Rubira, C.M.F. 2006. A Fault-Tolerant Architectural Approach for Dependable Systems. IEEE Softw., March, Vol. 23, No. 2, pp. 80–87. ISSN 0740-7459. doi: 10.1109/MS.2006.35. http://dl.acm.org/citation.cfm?id=1128592.1128713.

Engineering Dictionary. Available at: www.tumcivil.com/dic/index.php.

Haikala, I. & Järvinen, H. Käyttöjärjestelmät. 2nd ed. Talentum, 2004, 246 p.

IEC 60050-191 ed1.0. 1990. International Electrotechnical Vocabulary. Chapter 191: Dependability and quality of service. International Electrotechnical Commission IEC. 149 p.

IEC 61508-3 ed2.0. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3: Software requirements. International Electrotechnical Commission. 234 s.

ISO 9000:2005. Quality Management Systems – Fundamentals and Vocabulary.

Kopetz, H. 1997. Design Principles for Distributed Embedded Applications. Boston. Kluwer Academic Publishers.

Koskimies, K. & Mikkonen, T. 2005. Ohjelmistoarkkitehtuurit. Talentum. ISBN 952-14-0862-6. 250 p.

Malm, T., Hérard, J. Bøegh, J., Kivipuro, M. 2007. Validation of safety-related wireless machine control systems. NT TECHNICAL REPORT 605. 92 p.

NASA Software Safety Guidebook. (2004). NASA TEchnical Standard NASA –GB-8719.13. 388 p

SFS-EN 61508-4. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems. Part 4: Definitions and abbreviations. Finnish Standards Association SFS. 68 p.

SFS-EN ISO 12100:en. 2010 Safety of machinery. General principles for design. Risk assessment and risk reduction [ISO 12100:2010]. Finnish Standards Association SFS. 82 p.

SFS-EN ISO 13849-1. 2008. Safety of machinery — Safety-related parts of control systems – Part 1: General principles for design. Finnish Standards Association SFS. 180 p.

Appendix A: Terminology

Software Testing Glossary. http://www.aptest.com/glossary.html.

Stålhane, T., Herard, J., Söderberg, A., Malm, T., Kylmälä, K. & Pöyhönen, I. 2000. Safety assessment of systems containing COTS software. NT TECHNICAL REPORT 460. 47 p.

WikiAnswers. 2010. What is the Difference between fault and failure. [Ref. 9.2.2010] http://wiki.answers.com/Q/What_is_the_Difference_between_fault_and_failure

Wikipedia. Agile software development. Available at: http://en.wikipedia.org/wiki/Agile_software_development.

Wikipedia. Defect. Available at: http://en.wikipedia.org/wiki/Defect

Wikipedia. Integration testing. Available at: http://en.wikipedia.org/wiki/Integration_testing

Wikipedia. Lean software development. Available at: http://en.wikipedia.org/wiki/Lean_software_development.

Wikipedia. Software bug. Available at: http://en.wikipedia.org/wiki/Software_bug.

Wikipedia. System testing. Available at: http://en.wikipedia.org/wiki/System_testing

Wikipedia. White-box testing. Available at: http://en.wikipedia.org/wiki/White-box_testing

Wiktionary. Unit testing. Available at: http://en.wiktionary.org/wiki/unit_testing

WordNet Search - 3.1. 2011. Available at: http://wordnetweb.princeton.edu/perl/webwn

Appendix B: Aspects of the design process

The following list shows the tasks that the designer needs to consider during the design process. The first tasks are related to phases before the requirements specification and define the process more specifically. At the end, there is safety checking, which aims to detect failures related to requirements, validation and assembly. This is also related to safe testing of the complete machine.

- The SW safety process is part of the overall system development process.
- Companies may have their own policies, e.g. for maintenance.

1. Identify safety-critical SW.
2. Choose the basic requirements, which must be followed.
3. Define a design process model: V-model, other.
4. Sources of specific requirements: risk assessment, experience, supplier, subcontractor, customer, standards.
5. Safety requirement specification: safety function and integrity description.
6. Plan V&V methods, operation, installation, commissioning and maintenance.
7. Consider connection between SW and HW with respect to architecture.
8. Choose methods to describe architecture (views, tools, methods).
9. Define the architectural style, e.g.: layered, data flow, distributed, event-driven, duplicated independent architecture, designated (ISO 13849-1).
10. Define architectural techniques that support safety such as: stateless design, defensive programming, fault containment regions, diverse redundancy, recovery techniques and error detecting codes.
11. Consider how faults can be avoided, detected, removed and tolerated.
12. Define required modules and subsystems and how they are connected.
13. Define coding rules (standards).
14. Coding
15. Unit testing
16. Integration
17. Module testing
18. Walkthrough, inspection
19. System testing.
20. Validation, commissioning
21. Safety checking of the system before first use

The following figure shows aspects that need to be considered when selecting a standard to apply to the design process. The standards ISO 13849-1 and IEC 62061 are harmonized; it is therefore often good to consider these standards first. When these standards are followed, the design fulfils the requirements of the Machinery Directive. In some cases, the standards do not give adequate advice for the design of the system and the IEC 61508 standard family then needs to be applied.

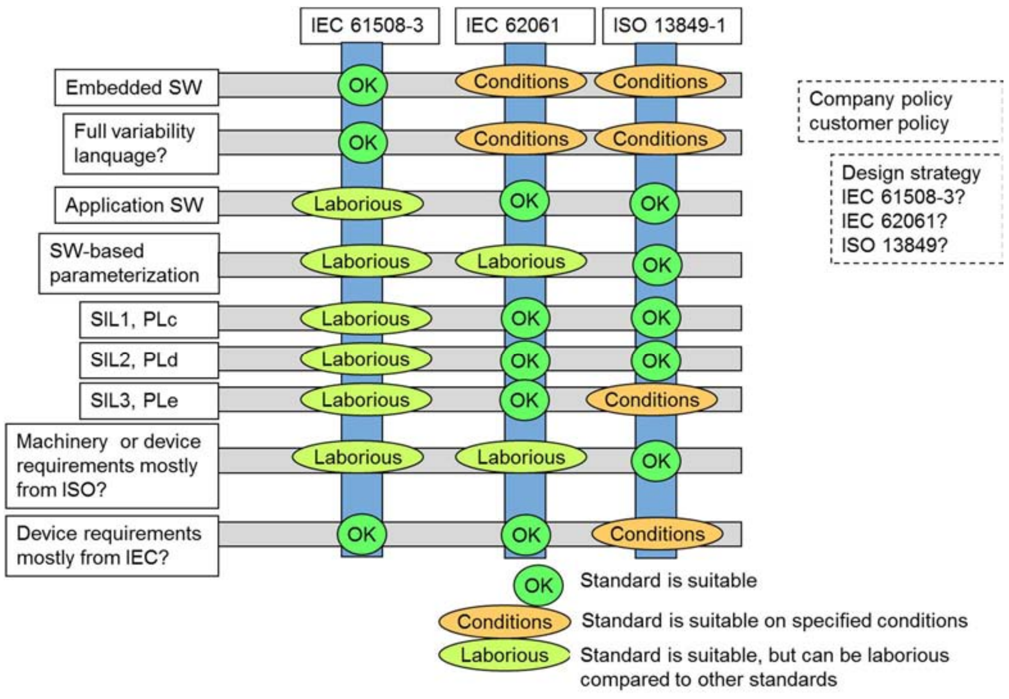

IEC 61508-3
IEC 62061
ISO 13849-1
Embedded SW
Full variability lanquage?
Application SW
SW-based parameterization
SIL1, PLc
SIL2, PLd
SIL3, PLe
Machinery or device requirements mostly from ISO?
Device requirements mostly from IEC?
OK
Conditions
Laborious
Company policy customer policy
Design strategy IEC 61508-3? IEC 62061? ISO 13849?
Standard is suitable
Standard is suitable on specified conditions
Standard is suitable, but can be laborious compared to other standards

Figure B1. Criteria to select basic functional requirements.

Series title, number and report code of publication

VTT Research Notes 2601
VTT-TIED-2601

Author(s)
Timo Malm, Matti Vuori, Jari Rauhamäki, Timo Vepsäläinen, Johannes Koskinen, Jari Seppälä, Heikki Virtanen, Marita Hietikko & Mika Katara

Title
Safety-critical software in machinery applications

Abstract
This report presents some important factors related to safety-critical software in machinery. The following subjects are considered in the text, bearing in mind the subject: the role of safety-critical software in machinery, statistics of software faults, requirements, safety and security principles, risk and hazard modelling, agile development, safety process patterns, safety-related architectures, verification and validation, phases of development and formal methods.

The general observation is that there are many methods for software design and it is difficult to choose the most relevant ones. The report shows some criteria for selecting methods and some aspects related to current topics. There are so many different safety-critical software applications in machinery that the research found the most interesting topics and then focused on them.

The statistics show that most defects arise during the requirements specification and architectural design phases of the lifecycle. This is before any coding. The statistics also show that the defect density is higher in large programs, i.e. the number of defects increases exponentially as the program size grows. It may therefore be better to separate safety-critical and standard code in order to keep the first one small. The separation of modules and keeping the connections between modules under control and narrow is recommended in order to have advantages in testing, understanding of the program, limited error spreading, program development etc. There are many kinds of self-diagnostic and monitoring functions that may be complex and increase the number of defects, but they increase safety and are needed in safety-critical code.

The standard IEC 61508-3, published in 2010, lays down many functional requirements for safety-critical programmable systems. However, there are also other standards related to functional safety and the safety of control systems.

This paper also considers some aspects related to the safety of agile methods. The standards show the requirements related to the phases of the V-model, but agile methods are not considered.

The functional safety of software is achieved through systematic (not intuitional) use of adequate methods in all phases of the programmable system lifecycle. Programmable systems contain hardware and software, both of which need to be considered in the validation process.

ISBN 978-951-38-7790-3 (URL: http://www.vtt.fi/publications/index.jsp)		
Series title and ISSN VTT Tiedotteita – Research Notes 1235-0605 (soft back ed.) 1455-0865 (URL: http://www.vtt.fi/publications/index.jsp)		Project number
Date November 2011	Language English	Pages 111 p. + app. 10 p.
Name of project		Commissioned by
Keywords Software, safety, safety-related, machinery, control system		Publisher VTT Technical Research Centre of Finland P.O. Box 1000, FI-02044 VTT, Finland Phone internat. +358 20 722 4520 Fax +358 20 722 4374

ISBN 978-951-38-7790-3 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0865 (URL: http://www.vtt.fi/publications/index.jsp)