Distributed Real-time Architecture for Mixed Criticality Systems

State of the Art of Piecewise Certification of Mixed Criticality Systems

D 5.5.1

Project Acronym	DREAMS	**Grant Agreement Number**		FP7-ICT-2013.3.4-610640	
Document Version	R 1.0	**Date**	2014-11-28	**Deliverable No.**	D 5.5.1
Contact Person	Øystein Haugen	**Organisation**		SINTEF ICT	
Phone	(+47) 913 90 914	**E-Mail**		oystein.haugen@sintef.no	

Contributors

Name	Partner
Øystein Haugen	SINTEF
Franck Chauvel	SINTEF
Asier Larrucea	IKERLAN
Jon Perez	IKERLAN
Salvador Trujillo	IKERLAN
Vicent Brocal	FENTISS

Table of Contents

1 Introduction

Individual products or systems are not often designed and developed independently. They belong to a product family so called *product line*. Products from the same product line are similar in many respects, but remain different enough to require specific development activities. This deliverable aims to describe the state of the art of certification of these systems, where it is attempted to achieve optimal certification of products that suffer some update or change, and it also pretends to expose how certification of one product can affect to the certification of other product of same product-line.

This deliverable is part of FP7 DREAMS project which aims to develop an cross-domain architecture and design tools for mixed criticality networked complex systems. It is based on results from DREAMS work packages WP1 "Architectural Style", WP2 "Languages" and WP4 "Tool Support". This review of the state of the art of modular certification for mixed criticality systems lays the foundations for novel product-line certification methods, which are one of the final innovations targeted by the DREAMS project.

1.1 What is a Mixed Criticality System?

The architecture of embedded systems in multiple domains follows a federated architecture paradigm, where the system is composed of interconnected subsystems that provide a well-defined functionality. The ever increasing demand for additional functionalities leads to a considerable complexity growth that, in some cases, limits the scalability of this approach. For example, a modern off-shore wind turbine control system manages up to three thousand inputs / outputs, several hundreds of functions are distributed over several hundred nodes grouped into eight subsystems interconnected with a fieldbus and the distributed software contains several hundred thousand lines of code. The integration of additional functionalities also leads to an increase in the number of subsystems, connectors and wires increasing the overall cost-size-weight and reducing the overall reliability of the system. For example, in the automotive domain, field data has shown that between 30-60 % of electrical failures are attributed to connector problems [1, 2].

The integration of applications of different criticality (safety, security, real-time and non-real time) in a single embedded system is referred as mixed criticality system [1]. This integration is further complicated by the recent advent of multi-core systems providing more computational power but requiring specific operating systems and software [3]. Yet, this integrated approach can improve scalability, increase reliability reducing the amount of systems-wires-connectors and reduce the overall cost-size-weight factor. However, safety certification according to industrial standards becomes a challenge because sufficient evidence must be provided to demonstrate that the resulting system is safe for its purpose [1, 2].

1.2 What is Certification?

Certification is a third party assurance of a product, system, subsystem or element establishing that the system is compliant with defined requirements. Certification is a process based on evidences for compliance with requirements, where evidences shall be established by documentation reviews, audits or testing activities (see Section 2.1.1). The certification of a product usually has a limited period of validity and has to be renewed after a certain period of time (see Section 2.1.2).

IEC 61508 [4-6] is a generic international safety standard from which different domain specific standards have been derived. Safety Integrity Level (SIL) is a discrete level corresponding to a range of safety integrity values where 4 is the highest level and 1 is the lowest. As a rule of thumb, the higher the SIL, the higher the certification cost [1, 2].

IEC 61508 [4-6] safety standard does not directly support nor restrict the certification of mixed criticality systems. Whenever a system integrates safety functions of different criticality, sufficient

independence of implementation must be shown among these functions. In case there is not sufficient evidence, all integrated functions will need to meet the highest integrity level. Sufficient independence of implementation is established showing that the probability of a dependent failure between the higher and lower integrity parts is sufficiently low in comparison to the highest safety integrity level [1, 2].

1.3 Similar Systems and Product Lines

System design is often guided by two very general, yet opposite heuristics: namely the *top-down* and the *bottom-up* approaches.

- The *top-down* approach encourages designs driven by the primary function of the system, and recursively divided into sub functions until they become concrete enough to be implemented. This leads to highly specialized functions/components, with low reusability as their usefulness is limited to the inherent system's decomposition.
- By contrast, the *bottom-up* approach advocates assembling general purpose functions into a whole that fits the purpose of the system. The bottom-up approach often leads to challenging integrations between components which have not been initially designed to interoperate.

Product lines engineering (PLE) acknowledges the limitations of both heuristics and seeks to make the most of both ideas. PLE recognizes the importance of a set of core functions, which together define a framework where reusability is worth considering. PLE thus captures the set of commonalities (and discrepancies, respectively) between products of a same "family" or "line" and permits to capitalize on well-accepted practices and standards in the application domain of interest. By fostering reuse within specific contexts, PLE has shown significant effort reduction in design, development, time to market and maintenance [7].

An intuitive example of product lines is the catalogue of wind turbines developed by Alstom, one of the DREAMS project case-studies. In order to accommodate different environmental conditions (temperatures, humidity, wind speed, etc.), wind turbines vary in many respects such as height, rotor diameter, power, etc. Yet, all wind turbines installed by Alstom share a common structure. The early identification of commonalities among turbines enables the reuse of large segments of the production chains as well as the development of large catalogues of closely related products.

Software Product Lines (SPL) are product lines where products are software systems. Among others, example of large scale SPLs includes the Linux Kernel, which is made of a set of carefully selected modules, or the Eclipse IDE that can be tailored to specific tasks (e.g., C/C++ or Java development, testing, modelling) by adding (resp. removing) some of the underlying plugins.

In the DREAMS context, we intend to model the variability inherent to mixed criticality platforms (e.g., hardware, hypervisor, scheduler configurations) and applications (e.g., fault-tolerance mechanisms). The objective is to investigate how SPL reduces the development cost of such platforms and to foster their certification by mixing modular certification and SPL testing technologies.

2 Certification

2.1 Certification of a Single System

2.1.1 Certification is a Cooperative Process

The certification of a system is a process that contains different steps that must be followed to obtain the final certificate (certification). As shown in Figure 1, a basic IEC-61508 compliant certification process of a system consists of four steps. As shown in Figure 2, a generic safety product development and certification starts from an idea. Then the company develops the safety embedded product and associated documentation using a safety life-cycle (FSM, Functional Safety Management) with the assessment of an external certification authority that provides the certificate.

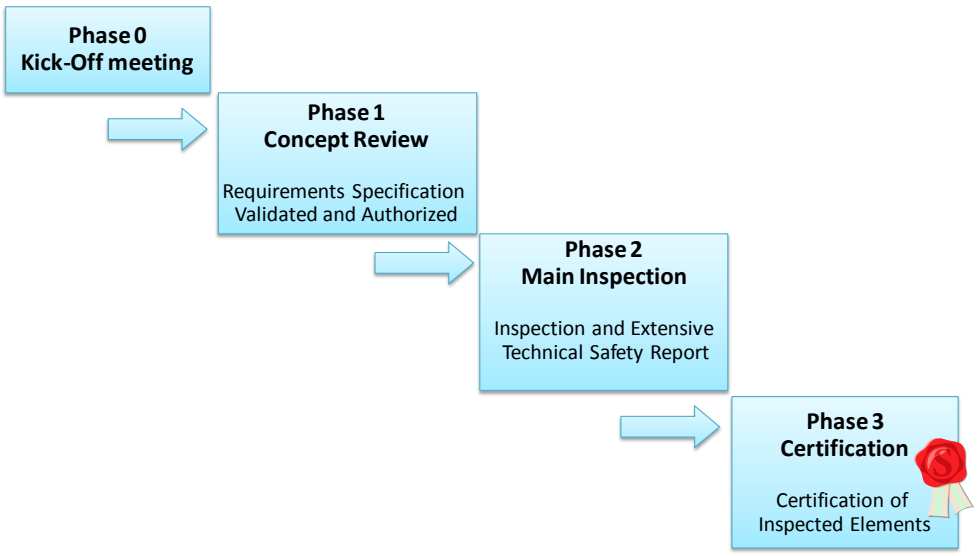

Figure 1: Example basic IEC-61508 certification process [8].

Figure 2: Example IEC-61508 certification process [8].

2.1.2 The Need for Re-certification

The need of re-certification of a system may result from improvements of system requirements and design. These updates can be given also by changes on system's requirements or standards updates. Whenever a change or update is required, impact analysis must be carried out and may possibly require a complete or partial re-certification of the system.

This re-certification process can be highly expensive if *modularity* was not taken in consideration in the system safety concept and detailed design. As explained in Section 2.3, modularity becomes a key element for certification and re-certification of systems, because it potentially reduces the impact of changes and reduces the impact in system recertification.

2.2 Certification Standards

A large number of standards have already been established by different organizations like ISO or IEC. These standards are being constantly updated to accommodate for daily needs. As shown in Figure 3, ISO/IEC Guide 51 classifies standards by type, allowing them to be applied to most actual systems.

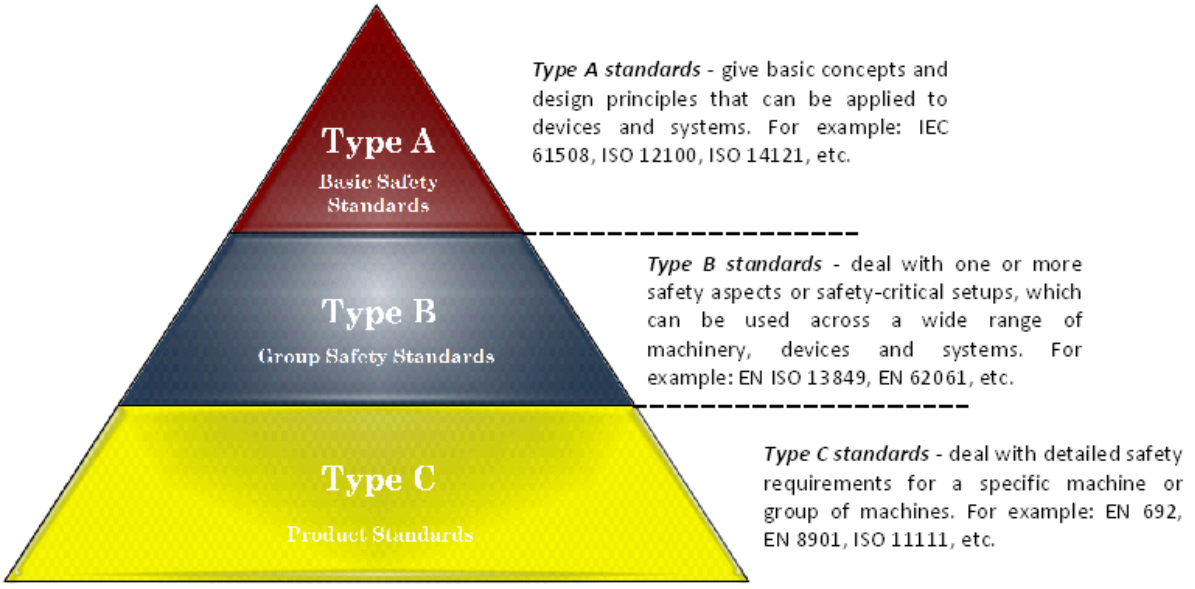

Figure 3: Hierarchical structure of Safety Standards.

IEC 61508 is an international basic standard that deals with functional safety-related systems, where *reliability* is defined by the Safety Integrity Level (SIL). The IEC 61508 is concerned with electrical, electronic and programmable safety-related systems where a failure will affect people or environment. The term safety-related describes every programmed system, whose failure may lead to damage or death of humans or catastrophic environmental destruction or degradation.

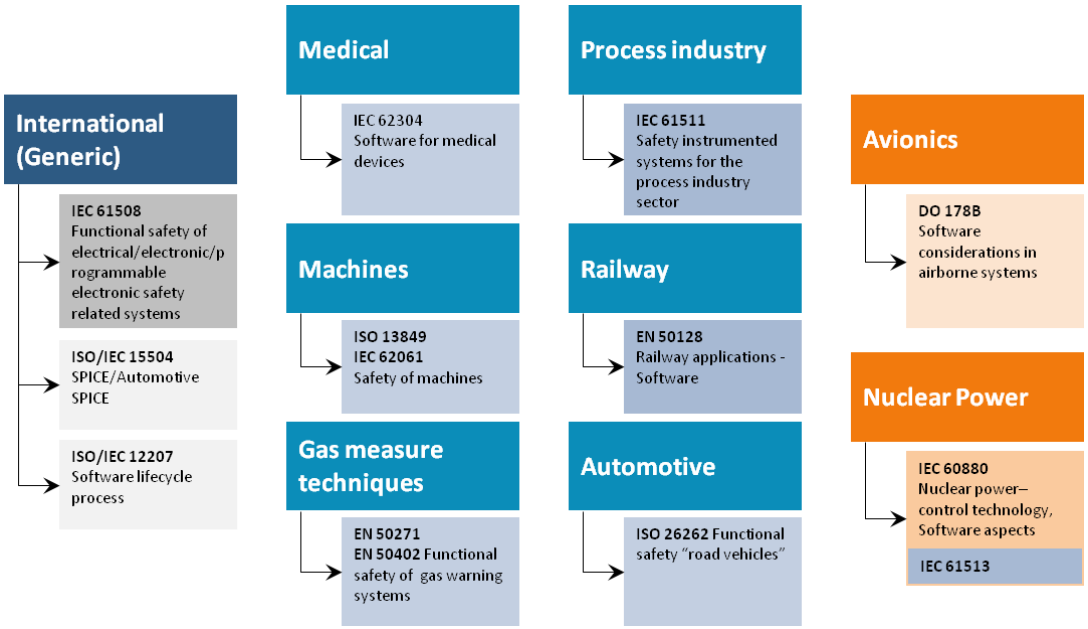

Figure 4 Overview of functional safety relevant standards in different areas (examples from VARIES [9])

As shown in Figure 4, all areas in blue are directly referred to Functional Safety and are all derived from the basic safety standard IEC 61508 (e.g., ISO 26262, EN 50139, DO 178B). Areas in orange are

standards, where IEC 61508 is not explicitly taken as basis, but they use similar approaches to address the safety of Electric/Electronic/Programmable Electronic (E/E/PE) systems.

2.2.1 IEC 61508 on Safety Certification

A safety life cycle is a series of phases covering the initiation and specification of safety requirements, the development of safety features for safety-critical systems, and ending with the decommissioning of that system. The IEC 61508 standard covers safety-related systems where a system incorporates electrical/electronic/programmable electronic devices. The standard covers possible hazards caused by failures of the safety functions of E/E/PE safety related systems. The detection of a potentially dangerous condition that results in the action of a protective or corrective mechanism to prevent hazardous events is defined as *functional safety*. IEC 61508 is concerned with the E/E/PE safety-related systems whose failure could have an impact on the safety of persons and/or environment.

The standard has two fundamental points: the safety life cycle and the safety integrity levels. The safety life cycle is defined as a process that includes all necessary steps to achieve the required functional safety.

Fehler! Verweisquelle konnte nicht gefunden werden. shows the safety life cycle defined by IEC 61508 [10], also called Functional Safety Management (FSM).

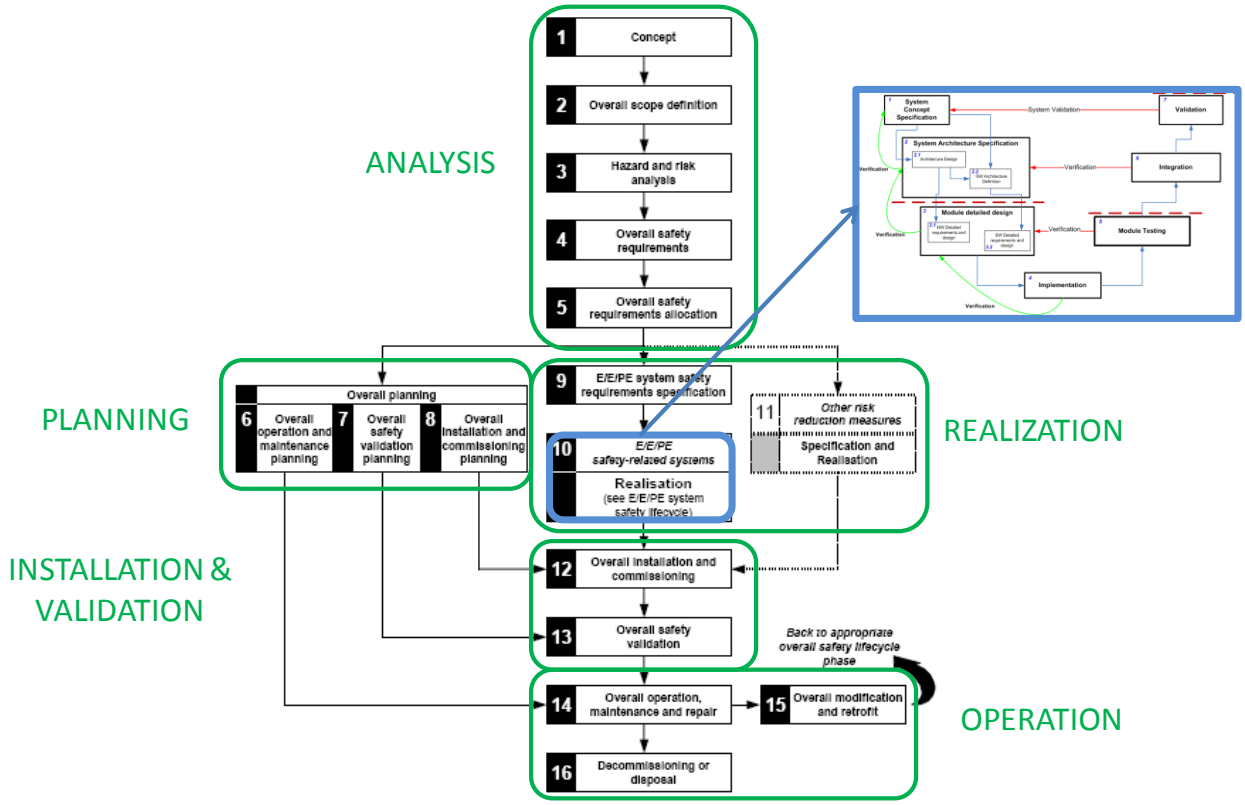

Figure 5 Safety lifecycle from IEC 61508 [8]

The FSM is divided in 16 phases grouped as follows:

- Analysis: Phases 1 and 2 entail the considerations of the safety implications of the equipment under control (EUC) and the control systems, at the system level. In Phase 3, first two phases' risk identification and analysis, assessed against tolerable criteria, are done. In phase 4, the risk-reduction measures of safety requirements are specified, and in phase 5 these are translated into the design of safety functions.
- Planning: Overall system planning for operation, validation and installation are provided in phases 6, 7 and 8 respectively.

- Realization: The system safety requirements specification is performed in phase 5 and it is realized according a V model in phase 10.
- Installation & validation: In Phase 12, the system must be installed and commissioned and in Phase 13, the system is checked to verify that all the safety related requirements have been identified and handled during building and installation.

Operation: Then the system may be put into operation, where there are safety and maintenance activities. It is also foreseen that the system can be modified during operations and therefore, incorporation of some modifications will be needed. The final phase, phase 16, is related with the disposal of the system (e.g., separations of the battery or the toxic elements to dispose them separately).According to the IEC 61508, safety refers to a system that has to be safe enough, regarding to protection of health and environment. IEC 61508 is used for the development of E/E/PE systems that carry out safety functions.

Furthermore, IEC 61508 determines that safety integrity level (SIL) of a system is determined by software failures (systematic failures) and hardware failures (systematic and random failures). SIL is the parameter that is used by IEC 61508 to categorize the required integrity of safety. It is defined as the probability that a dangerous failure of the safety related system may occur per unit of time. Table 1 shows the target failure measure for a safety function operating in one of three defined demand modes by IEC 61508 (see subsection 7.6.2.9 of IEC 61508-1).

Safety Integrity Level (SIL)	Average frequency of dangerous failure of the safety function [h^{-1}] (PFH)
4	$\geq 10^{-9}$ to $< 10^{-8}$
3	$\geq 10^{-8}$ to $< 10^{-7}$
2	$\geq 10^{-7}$ to $< 10^{-6}$
1	$\geq 10^{-6}$ to $< 10^{-5}$

Table 1: Safety integrity levels - target failure measures for a safety function operating in high demand mode of operation or continuous mode of operation (Table 3 of IEC 61508-1)

Systematic and random failures are handled by techniques/measures defined by IEC 61508 for the diagnosis, control and avoidance of failures, and they are used for defining the diagnostic coverage of the system. See Annexes A, B, C, and E of IEC 61508-2 for hardware failures and Annexes A, B and C of IEC 61508-3 for software failures.

2.3 Modular Certification

Different terms are used by different safety standards to refer to the concept of modular certification used in academia. Some basic definitions to be considered are:

- A **safety case** "*represents an argument supporting the claim that the system is safe for a given application in a given environment*" [11]
- **Modularity** is an approach that subdivides a system into smaller parts (modules) which are independently generated and used by different systems to drive functionality. The names of decomposed structure of system can vary from one developer to another, but the meaning is the same.
- **Modular Safety Cases** allow assurance of the safety of a system that has been composed from modules.

Systems and software have been designed according to modularity since many years ago. The safety cases are important to reduce safety and commercial risks. The motivation for a safety case is to: (i) provide arguments to demonstrate that safety properties are satisfied and risk has been mitigated (ii) provide a mechanism for review and (iii) provide interworking of different standards. Some progress towards the road to certification has been achieved in the MultiPARTES project. A safety concept for

mixed criticality systems based on multicore hypervisor has been defined. Most important, this safety concept has been assessed positively by TÜV Rheinland providing a compliance statement according to IEC 61508. Modelling tools for the consistency checking of mixed criticality designs have been developed as well to enable design and early validation of safety properties. In this context, a consistently designed and partitioned safety case limits the impact of changes to a reduced area of the safety case.

The modular approach is used in automotive or avionic domains for instance, where the diverse number of options and variants makes the design, and in turn the certification, extremely complex. It reduces cost of the re-certification of changed systems, because it provides a system composed from design modules, which ideally can be replaceable, such as jigsaw pieces (see Figure 6), without affecting the safety-related properties of the system.

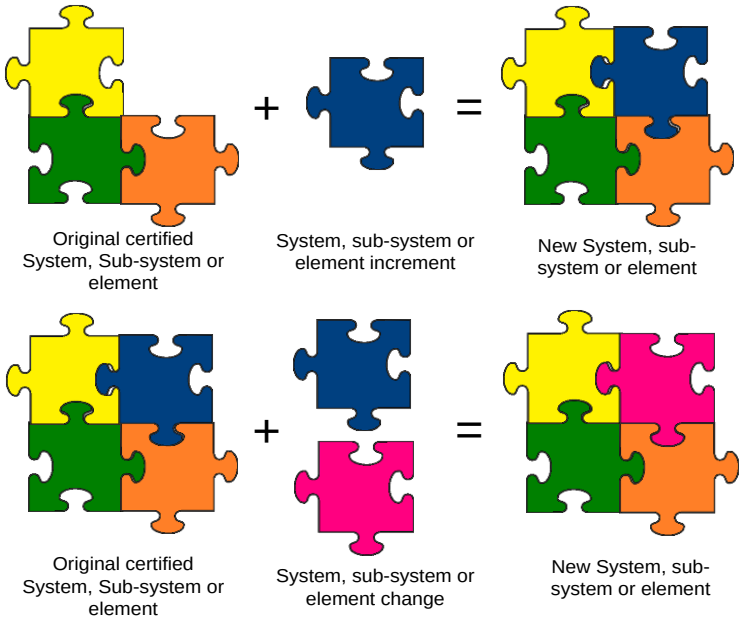

Figure 6: Modularity approach.

This section aims to expose the modularity requirements varieties among different domain standards (e.g., IEC 61508, ISO 26262, EN 50129 and DO-178 + IMA).

2.3.1 IEC 61508

Modular certification provides arguments by which already certified or qualified components or parts of the system that have been designed with safety life-cycles (even if they are not certified) can be applied in activities of the mixed-critical design.

The product families are a different use case of modular certification. If a product is built with modular safety cases and then the product suffers a modification, the reproducibility of the certification according to IEC 61508 can be less costly, since modular safety cases can be re-used and the problem solving pattern has been already provided. Modular certification implies that the used modules are pre-certified according to IEC 61508 to achieve a certain SIL Claim Level (CL). Therefore, all hardware and software components shall be developed and certified according to IEC 61508 to fulfil with the required Safety Integrity Level (SIL).

Modular Safety Cases shall cover the following aspects:

- Analysis of the system regarding safety needs;
- Strategies adopted to achieve the desirable SIL (Safety Integrity Level);
- Techniques and measures to control random faults;
- Demonstration that selected techniques are sufficient to fulfil safety needs.

The architecture of the system must be well defined and preferably divided into subsystems, where some of those subsystems can form Modular Safety Cases (e.g., Input/output Modules, Memory Units, On/Off chip communications, Safety/Non-Safety interactions, etc.) as shown in Figure 7.

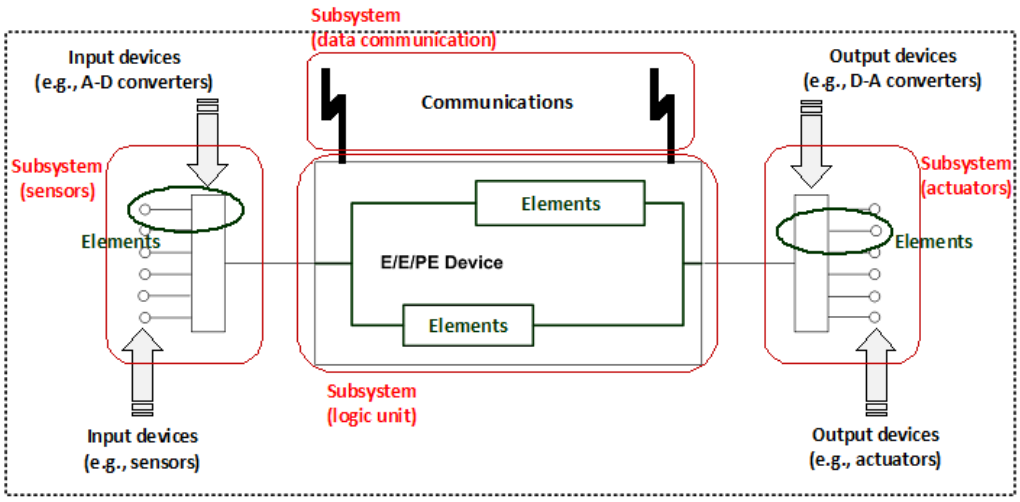

Figure 7: System structure based on sub-systems and elements. (IEC 61508-4, Figure 3) [12]

IEC 61508 is designed to be flexible enough to accommodate emerging technologies without breaking the fundamental concepts (IEC 61508-1). Fundamental concepts of IEC 61508 are applicable to COTS-based systems. IEC 61508-2 defines that whenever an existing verified subsystem shall be implemented, the total mapping of the subsystem shall be carried out for selection of required specific implementation functions or performances.

IEC 61508-3 defines that when a software design incorporates pre-existing reusable software, which may have been developed without taking into account the systems requirements, the software safety requirements specifications must be satisfied. These safety requirements specify that a pre-existing reusable software element shall meet the following requirements for systematic safety integrity:

a) meet the requirements of one of the compliance routes

- *Route 1S: Compliant Development*
  Compliance with the requirements of IEC 61508 for the avoidance and control of systematic faults in software
- *Route 2S: Proven in use*
  Provide evidence that the element is proven in use, taking into account that the relevant iterations of hardware and software shall be identified, evaluated and documented.
- *Route 3S: Assessment of non-compliant development* (not applicable).

b) Provide a safety manual for compliant items (see Annex D of IEC 61508-2 and Annex D of IEC 61508-3)

According to IEC 61508, "*a compliant item is any item (e.g., an element) on which a claim is being made respect the clauses of IEC 61508*" [13]. This standard defines two kinds of compliant items: hardware (microcontrollers, ASICs, FPGAs, etc.) and software (RTOS, communication protocol stack, etc.). These two types of compliant items shall provide a safety manual, where shall be specified their functions, that are required to ensure that the system meets with the IEC 61508 requirements. The safety manual enables the integration of the compliant item into a safety related system, sub-system or element.

In case of third party components, all relevant claims of compliance items made by the supplier and other parties shall be included in the functional safety assessment. In case that the compliant item was assessed as a part of a larger system, precise identification of system should be documented.

- **Hardware Compliant Items**

  In case of hardware compliant items, it will be necessary that the supplier of a compliant item makes available the information of the safety related system (see IEC 61508-2, Section 7.4.9.3) to the designer; providing it in the safety manual. The requirements of this safety manual are provided by Annex D of IEC 61508-2 (Compliant item in general).

  One example of a hardware compliant item is a mixed criticality network, which involves the transfer of the information between different locations. According to IEC 61508-2, there exist two transmission systems or channels (white and black channel). In this case, modularity approach makes possible that a modular safety case of a mixed criticality network contains a safety communication layer, which can be re-used as a compliant item for different developments of products, systems or sub-systems.

- **Software Compliant Items**

  As in case of hardware compliant items, it will also be necessary to make a safety manual of software compliant items. In the safety manual will be documented the information related with the compliant items. This information is required to enable the integration of the compliant item(s) into the safety related system, subsystem or element. For this reason Annex D of IEC 61508-3 shall be followed. This annex specifies the contents that shall be contained by the safety manual. Additionally, in case of software compliant items, it will be necessary to dispose of the hardware system information required by IEC 61508-2.

  For example, in case of a hypervisor, this is considered as a compliant item. In the same way, the safety partitions generated by the hypervisor are also considered as compliant items, so, they must follow IEC 61508-3 Annex D. The partitions, which are not related to the safety functions, are not considered as a compliant item. In case of hypervisor, one shall also provide the non-interference (spatial and temporal) between partitions following the techniques that are defined in IEC 61508 Annex F.

2.3.2 ISO 26262 (Road Vehicle)

ISO 26262 provides its own component model, where an item can be seen as the system or the systems under consideration. As shown in Figure 8, a system can be hierarchically structured and consists of a set of functions.

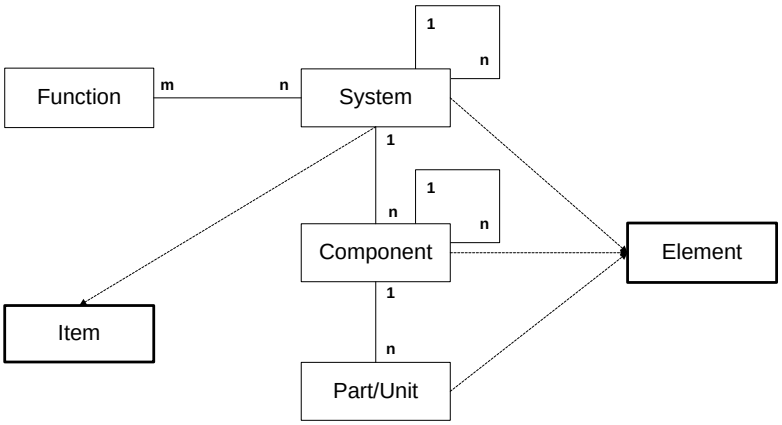

Figure 8: Relationship of system, item, element, and hardware part and software unit. [14]

Each system is composed by a set of components that can be hierarchically structured as well. Systems are elements or are composed by elements [15]. A component is a non-system level element that is

logically and technologically separable and is comprised of more than one *hardware part*[1] or of more *software units*[2].

ISO 26262 furthermore supports modular construction of items, including levels of abstraction for the elements, as shown in **Fehler! Verweisquelle konnte nicht gefunden werden.**

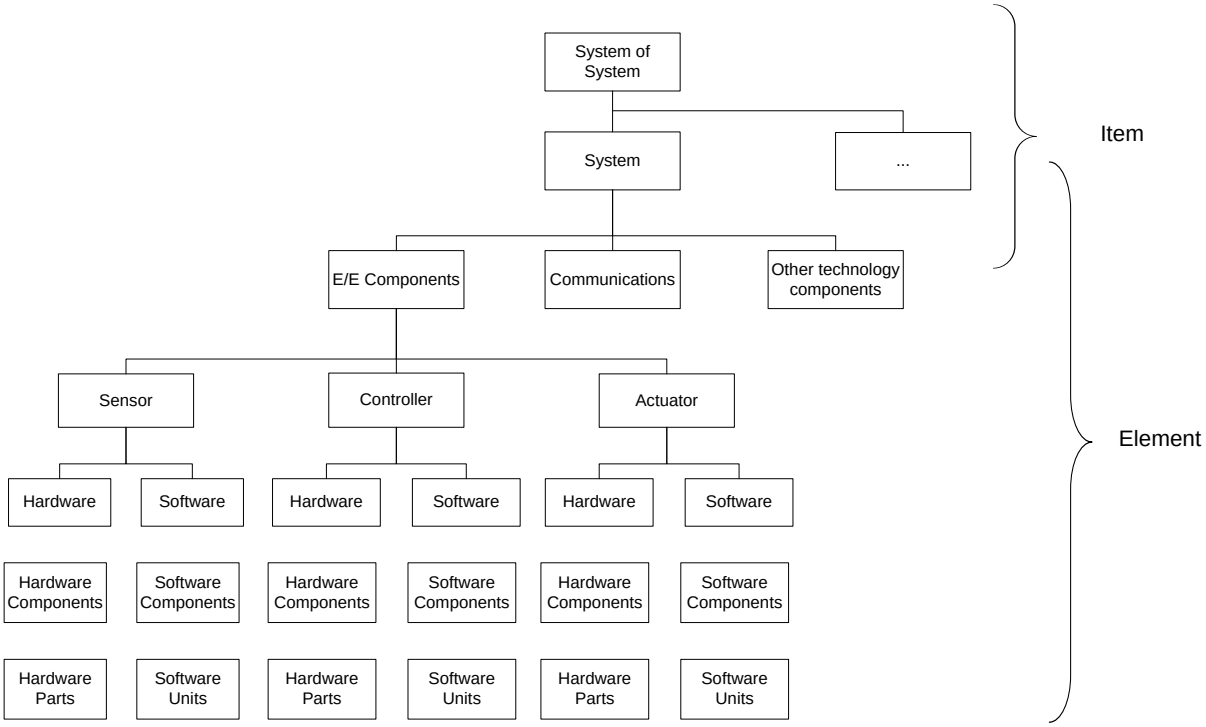

Figure 9: Items, Elements in ISO 26262 [14].

ISO 26262 defines term of Safety Element out of Context (SEooC). SEooC is a safety element for which an item does not exist at the time of the development. A SEooC can be a subsystem, a software component, or a hardware component.

The development of sub-system or hardware component out of context implies that the prerequisite work products are replaced by assumptions on ASIL capabilities. In this case, the system design specification (ISO 26262-4, Clause 7) and the technical safety concept (ISO 26262-4, Clause 6) are replaced by assumptions.

The development of software component out of context (see Figure 10), implies that the prerequisite work products are replaced by assumptions on ASIL capabilities. In this case, the system design specification (ISO 26262-4, Clause 7) and the technical safety concept (ISO 26262-4, Clause 6) are replaced by assumptions. However, the software development out of context can also start with either the software architectural design (ISO 26262-6, Clause 6) or with the software unit design and implementation (ISO 26262-6, Clause 8). So, the software safety requirements (ISO 26262-6, Clause 6) and the architectural design specification (ISO 26262-6, Clause 7), can be replaced by assumptions.

[1] Hardware which cannot be subdivided

[2] Atomic level of software component of the software architecture that can be subjected to standalone testing

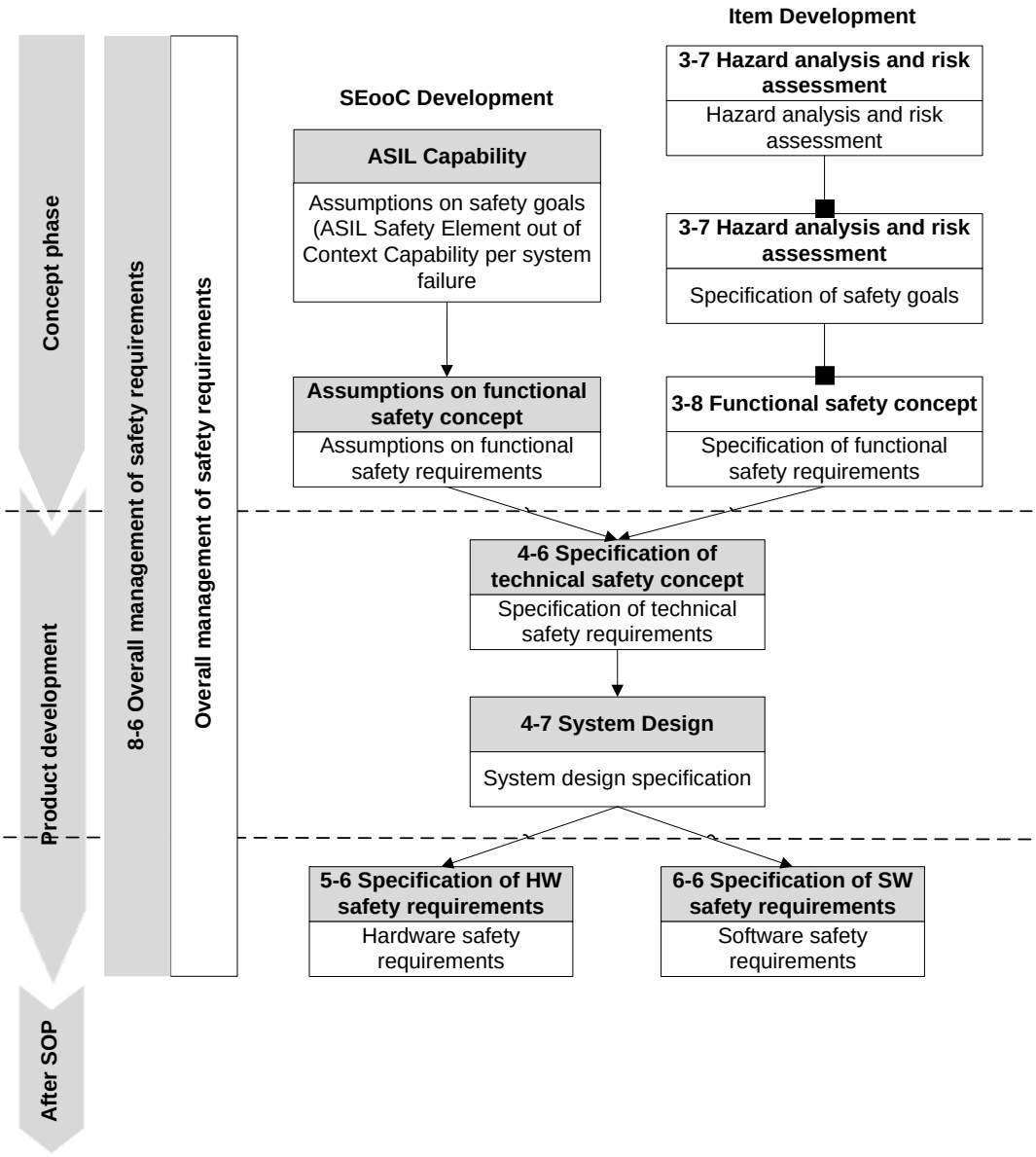

Figure 10: ISO 26262 Safety Element out of Context (SEooC). [14]

2.3.3 EN 50129 (Railway)

This standard is applicable to safety-related electronic systems (including sub-systems and equipment) for railway signalling applications [16]. This standard defines the conditions that shall be satisfied in order that a safety-related electronic railway system/subsystem/equipment can be accepted as adequately safe for its intended application.

- Evidence of quality management.
- Evidence of safety management.
- Evidence of functional and technical safety.

The documentary evidence that these conditions have been satisfied shall be included in a structured safety justification document, known as safety case. The safety case shall contain the documented safety evidence for system/sub-system/equipment, before the safety related system can be accepted as adequately safe. The safety case document structure shall be the following:

I) Definition of system/sub-system/equipment
II) Quality Management Report
III) Safety Management Report

IV) Technical Safety Report
V) Related Safety Cases
VI) Conclusions

2.3.3.1 Evidence of Quality Management

The first condition for safety acceptance implies that the quality of the system, sub-system or equipment has been, and shall be controlled by a quality management system (see EN 50126) during its life-time. Evidence to demonstrate this shall be recovered in the Quality Management Report, which forms part of the safety case document structure (II).
The purpose of the quality management system is to minimize the incident of human errors at each stage in the life-cycle, and thus to reduce the risk of systematic faults in the system, sub-system or equipment.

2.3.3.2 Evidence of Safety Management

The second condition for safety acceptance, which shall be satisfied, is that the safety of the system, subsystem or equipment has been, and shall continue to be, managed by RAMS management process described in EN 50126. The use of Safety Management process is mandatory for Safety Integrity Levels 1 to 4 inclusive. Documentary evidence to demonstrate compliance with the safety management process throughput the life cycle shall be provided in the Safety Management Report, which forms part of the Safety Case (III).

2.3.3.3 Evidence of Functional and Technical Safety

In addition to the evidence of quality and safety management, a technical evidence for the safety design shall be documented in the Technical Safety Report. This document forms the part IV of the Safety Case for the system/sub-system/equipment. This document is mandatory for safety integrity level 1 to 4. In case of safety integrity level 0, it falls outside the scope of this safety standard. Technical safety report headings are the following:

I) Introduction
II) Assurance of correct functional operation
III) Effects and Faults
IV) Operation with external influences
V) Safety-related application conditions
VI) Safety Qualification Test

2.3.3.4 Safety Assurance and Approval

This sub-clause defines the safety acceptance and approval process for safety-related electronic system/sub-system/equipment.

Three categories of Safety Case can be considered (See **Fehler! Verweisquelle konnte nicht gefunden werden.**):

- **Generic Product (GP) Safety Case:** A generic product that can be re-used for different independent applications (e.g., be an American Windows, running on a PC) (the platform).
- **Generic Application (GA) Safety Case:** A generic application can be re-used for a class/type of applications with common functions (the type).
- **Specific Application (SA) Safety Case:** A specific application is used for only one particular installation (installed product).

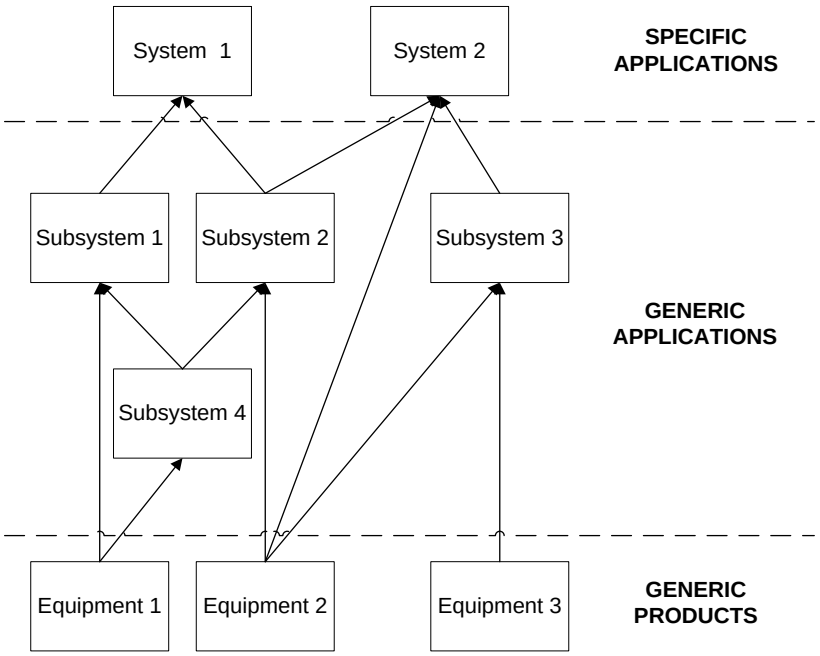

Figure 11: Dependencies between safety case and safety approval. [16]

Figure 12 shows the structure of each safety case categories and the procedure to obtaining the safety approval for each one. Although the procedures for obtaining the safety approval are basically the same, separate safety approval is needed for the application design of the system and for its physical implementation. The safety case for specific application is subdivided into two portions: I) the application design safety case that contains the safety evidence for the theoretical design of the specific application, and II) the physical implementation, which shall contain the safety evidence for the physical implementation of the specific application.

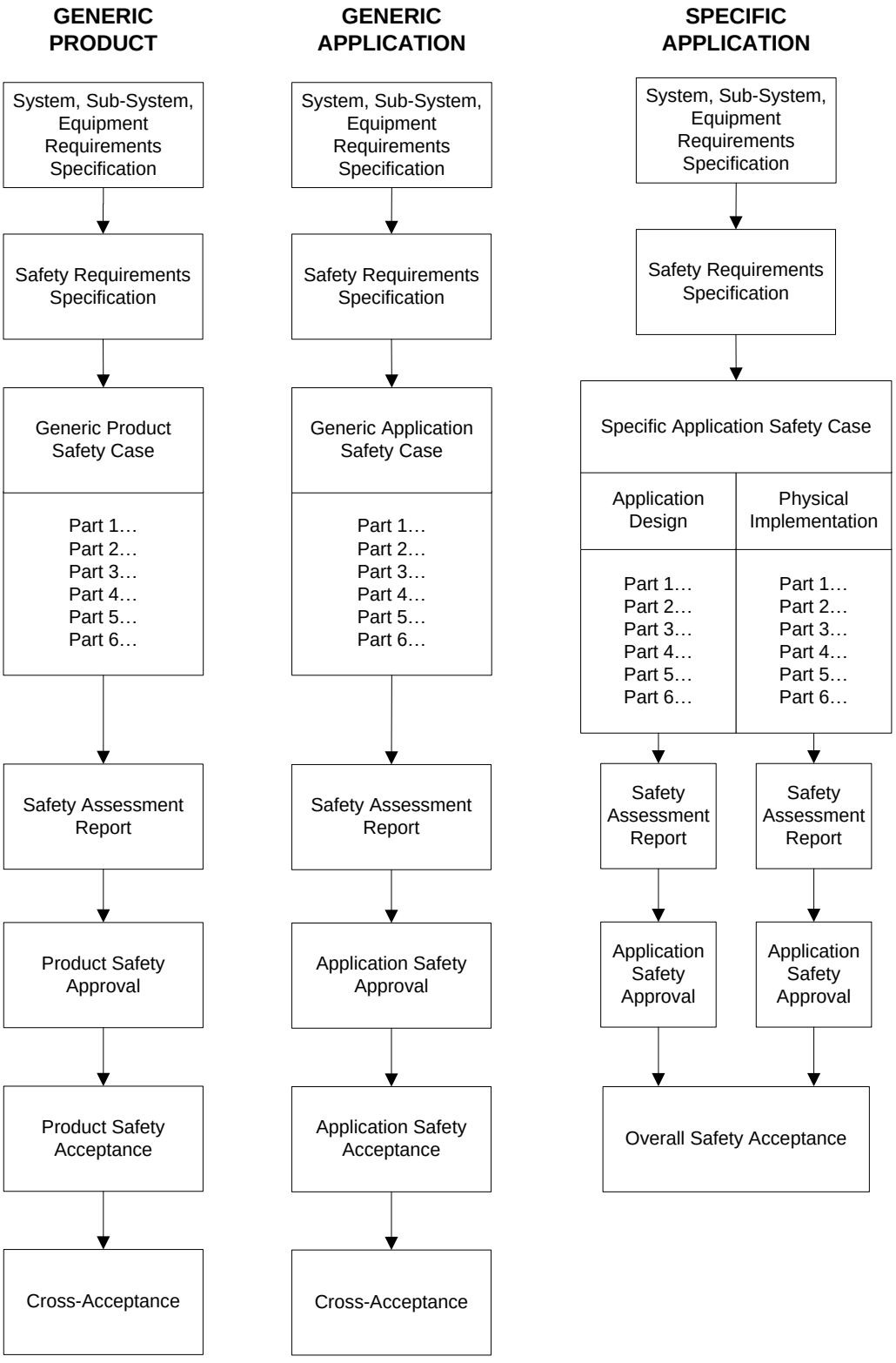

Figure 12: Safety acceptance and approval process. [16]

2.3.4 DO 178 + IMA (Avionic)

DO 178 defines a component as "*a self-contained part, combination of parts, sub-assemblies or units, which performs a distinct function of a system*" [17]. It defines also two types of components: modifiable and not modifiable components. A modifiable component is part of the software that is intended to be changed by the user, whereas a non-modifiable component is not intended to be

changed by the user. The non-modifiable component should be protected from the modifiable component to prevent interference in the safe operation of the non-modifiable component. This protection can be achieved by hardware or by software (e.g., software partitioning).

In case of some airborne systems, an equipment may include optional functions which may be selected by software options rather than be selected by hardware and which are not intended to be used in every application. In Section 5.4.3 of DO 178 are defined the considerations for deactivated code. This document also specifies that COTS software that can be included in airborne systems or equipment should satisfy the objectives of this document.

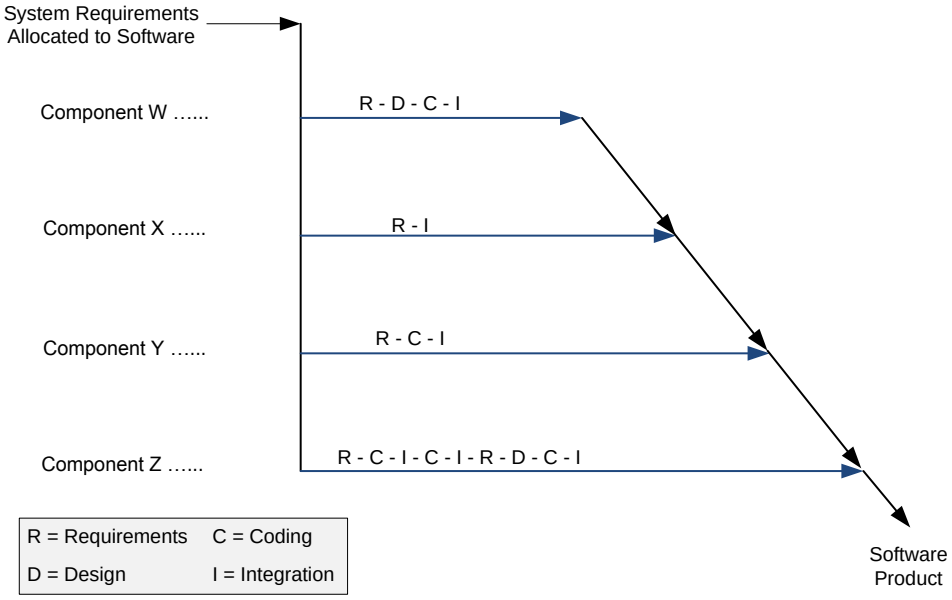

Figure 13: Component based Software Incremental Development Process. [17]

Figure 13 illustrates the software development process sequence, defined by DO-178, for components of a single software product with different software lifecycles. Component W implement and develops a set of system requirements to define a software design, which will be coded into source code, to finally integrate it into the hardware. Component X shows the use of Component W, and the integration of its requirements into the last component, this way, resulting in a new component.

Component Y illustrates the use of a simple, partitioned function that can be coded directly from the software requirements.

Component Z illustrates the use of a prototyping strategy. The goals of prototyping are to better understand the software requirements and to mitigate development and technical risks, through the continuous evaluation and continuous refinement of the software project requirements.

Some avionic companies (e.g., Boeing and Airbus) are using Integrated Modular Avionic (IMA) as their architecture for present and future product developments. IMA is a flexible distributed real-time network airborne system, which is capable of supporting numerous mixed criticality applications. The IMA concept provides an integrated modular architecture, where application software can be ported across common components (e.g., hardware, OS, middleware). It also provides, many potential benefits, including reduction of the number of platforms, which decreases the cost, performance gain due to latency reductions, simplified software updates (changes), allowing integration of new applications without hardware changes. In the other hand, although IMA provides significant benefits to the aircraft, it can present certification challenges.

2.4 IMA, Certification and Hypervisors

One of the key ideas of the Integrated Modular Avionics (IMA) is the integration in the same module of several software functions that share the same hardware resources. For this integration to be reliable and safe, these software functions are isolated into partitions in terms of space and time: a partition can only access its own memory address space and is allowed to execute only during pre-allocated time windows. This isolation principle, widely known as Time and Spatial Partitioning (TSP), is directly applicable when integrating functions of different levels of criticality.

The specific frame for the certification of IMA-based systems in the commercial aircraft industry, accepted by both the Federal Aviation Administration (FAA) and European Aviation Safety Agency (EASA) certification bodies, is the DO-297/ED-124 standard, which *"contains guidance for Integrated Modular Avionics (IMA) developers, application developers, integrators, certification applicants, and those involved in the approval and continued airworthiness of IMA systems in civil certification projects. It is focused on IMA-specific aspects of design assurance." [18]*.

A key component of the IMA architecture is the partitioning kernel, which is the software component providing the isolation capabilities among all software components operating on the same computer. The ARINC 653 [19, 20] standard defines the behaviour, properties and interface provided by IMA partitioning kernels. Certification of the software components of an airborne system is regulated by DO 178C/ED-12 for the civil aviation industry, thus including the partitioning kernel and the partitions (isolated software components). The previous section 2.3.4 provides further details on DO 178C.

While there are many technologies suitable to implement partitioning kernels in the IMA style, there has been a renewed interest in the hypervisor technology for such concern.

2.4.1 Hypervisor Technology

Hypervisor (also known as virtual machine monitor VMM) is a layer of software (or a combination of software/hardware) that allows running several independent execution environments in a single computer. Although the basic idea of virtualizing is widely understood: any way to recreate an execution environment, which is not the original (native) one; there are substantial differences between the alternative technological approaches used to achieve this goal.

The key differences between hypervisor technology and other kind of virtualization (such as Java virtual machine or software emulation) are performance and complexity. When targeted to embedded applications, bare-metal hypervisor are designed to virtualize only the critical hardware devices necessary to create the isolated partitions. This allows to limit the complexity of the software and to guarantee real-time performances.

Following the lead of IMA, TSP and virtualisation, hypervisors are a promising technology [2, 21] for modular certification and the development of product families even outside the avionics domain. The isolation of the software components allows increasing the confidence in that safety and reliable functions can be independently developed and assessed, and later on integrated with other functions without adverse effects.

XtratuM [21] is a bare-metal hypervisor designed to provide TSP for safety critical applications. Its main features are:

- Memory management: XtratuM uses the hardware mechanisms to guarantee the isolation of the memory spaces of the partitions.
- Scheduling: Partitions are scheduled according to a cyclic scheduling policy, enforcing their temporal isolation.
- Health monitor (HM): It is the part of XtratuM that detects and reacts to anomalous events or states. The purpose of the HM is to discover errors at an early stage and try to recover or confine the faulting subsystem in order to avoid or reduce the possible consequences.

2.4.2 IMA in Space Applications

In the recent year, the interest for the Integrated Modular Architecture (IMA) has reached the space domain. The European Space Agency (ESA) is currently undertaking activities, by means of the SAVOIR-IMA workgroup, for the definition of a reference avionics architecture based on the IMA concept [22]: *"The group analyses the impact of the concept of IMA on the overall current reference architecture (hardware, software and communication). Modifications to the hardware architecture are identified to improve compatibility with time and space partitioning, primarily taking into account the computer architecture [...]"*. The topic also has recently received attention from the NASA [23].

2.5 Certification and Testing

Certification and testing are two different activities, which although related, shall not be confused. As explained above, *certification* denotes the activities carried out to confirm that a product exhibits certain characteristics. The resulting certificate confirms that enough evidence supporting the claim of interest was available. Product certification mainly targets safety and quality characteristics. By contrast, testing is the activity of assessing (or measuring) specific capabilities of a given product. Software testing for instance may cover correctness, performance, security or usability to name a few. While certification is about making a decision based on collected evidences, testing is a means of obtaining such evidences.

Both certification and testing are subject to obsolescence when the product of interest changes. For software, *automation* has been the key to better cope with the impact that changes have on testing. Automated testing, where tests are automatically run as soon as the product is changed, turned testing into a key activity supporting the overall development process: detecting defects, regression, and deviation from the requirements. By contrast, certification is less prone to automation due to its human-based nature. Yet, as certification consumes the evidence produced by testing activities, automated testing may certainly contribute to faster re-certification.

Building software product lines challenges both testing and certification activities. Testing a software product line as a whole requires assessing the abilities (e.g., correctness) of every single product which can be derived from it. Similarly, certifying a product line requires collecting and scrutinizing evidences given for every single product. In the following sections, we review techniques from the product line engineering field, which could help move towards certification of product lines, especially through testing.

3 Families of Systems – Product Lines

3.1 A Brief Introduction to Variability Modelling

Modelling is a core practice in science and engineering. In short, modelling aims at easing the resolution of a particular problem while preserving our ability to gain hindsight about it. Regardless of the approach, any modelling activity is always strongly coupled to the problem it aims at. Modelling is inherently difficult because it requires striking the right balance between discarding enough of the reality to tame its complexity and still retaining enough to maintain significance. This paradox — known in engineering as the *principle of incompatibility* [24] — directly impedes our ability to reuse models.

Modelling experts tackle complexity with to two main leverages: *abstraction* and *separation of concerns*. Abstraction discards any aspect of the reality that is irrelevant to the problem of interest whereas separation of concerns divides the problem into separate sub problems of smaller complexity. When exploiting the resulting models, hindsight often comes from our ability to distinguish between what varies and what remains. Mathematics and especially geometry, topology and algebra produced many models where the hindsight comes from the duality between what

changes and what remains. Differential equations, to name only one, capture invariant relationships, which permit to understand how complex dynamics varies.

From the modelling standpoint, PLE explicitly captures variability in order to maximize reuse throughout the development and maintenance process, in order to improve productivity while reducing risk and cost. PLE therefore distinguishes between what remains and what varies in a system. The "things that remain" often reflect hard constraints in the application domain of interest whereas parts opened to variations reflect potential areas for business and innovation. The main objective is therefore to automate the derivation of a new product from the prescription of its features.

3.1.1 Capturing Similarity and Variability

Modelling the variability among a family of products relies on the notion of *feature*. A feature stands for "*a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option*" [25]. Products are thus characterized by the features they provide, and in turn, product lines are characterized by the union of all features provided by their products.

The simplest way to capture the commonalties (resp. the variability) among a set of products is using a feature table. The table relates the set of products with the set of possible features: specifying for each product the features it provides. This approach is commonly used to provide customers with a comparison of a range of products such as mobile phones, TVs, etc.

A better way to capture variability within a set of products is through a *feature model* [26, 27]. A feature model gathers the commonalities within a product line into a tree of features, where some features are mandatory (the commonalities) whereas other may be optional or exclusive (the variability). Figure 14 illustrates the graphical notation associated with such feature models on a subset of what could be the DREAMS platform. This simplified DREAMS platform encompasses hardware components, hypervisors and operating systems (OS). The platform may or may not include hypervisors, which are either based on OS virtualization or on hardware virtualization, or both. By contrast the platform will always include an OS, which will be one of the three possible alternatives listed in Figure 14.

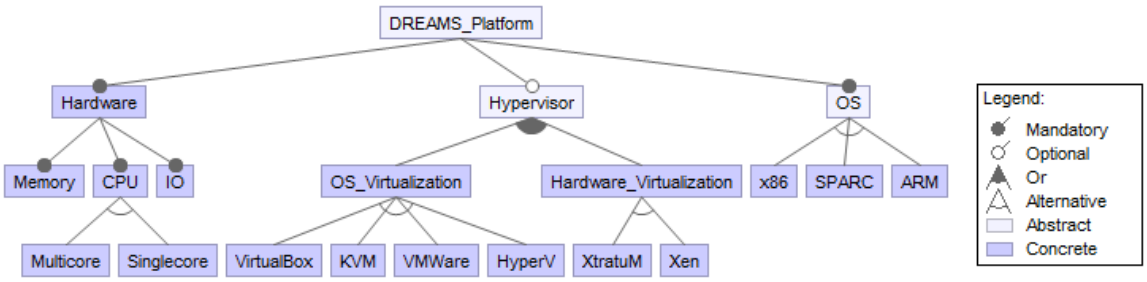

Figure 14 A simplified feature model capturing the variability of hardware platforms

The proposed notation is not expressive enough to capture all the additional constraints that may exist and restrict the set of products which can be derived from a given product lines. In our example, it may be that some OS virtualization technologies are not available for all operating system, and the choice of one (say VMWare) restricts the possible OS.

Orthogonal Variability models (OVM) [28] and its successor the common variability language (CVL) [29, 30] attempted to overcome the inherent tight coupling between the product derivation procedure and the underlying technologies. They resolve variation points and variants, not anymore in the technological space, but on a domain-specific model. The resulting model of the product can thus be finalized using the associated domain-specific tooling.

3.1.2 The BVR Model

BVR (Base Variability Resolution models) is a language built on CVL technology, but enhanced due to needs of the industrial partners of the VARIES project (http://www.varies.eu), in particular Autronica.

BVR is built on CVL, but CVL is not a subset of BVR. In BVR we have removed some of the mechanisms of CVL that we are not using in our industrial demo cases that apply BVR. We have also made improvements to what CVL had originally. For the purpose of DREAMS we may just say that BVR is a continuation of the CVL language with associated tooling. Figure 15 below illustrates the variability model of the simplified DREAMS platform in the BVR tool.

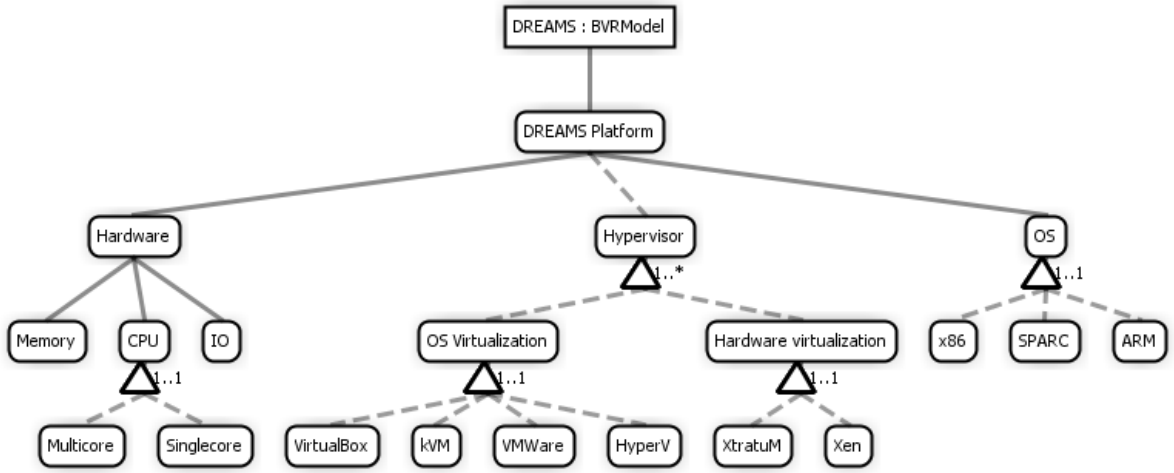

Figure 15 The variability model of the DREAMS platform modelled using the BVR tool

3.1.3 Exploiting Variability

As mentioned above, the major challenge of product lines engineering is to automate the construction of new products from the sole prescription of their features. We review below the main approaches proposed in the SPL literature, further detailed in [31].

The simplest solution is the use of pre-processor directives to disable irrelevant features in the code base, before it is compiled and linked. This approach implies a direct mapping from features to some syntactic programming constructs such as routines, modules, or classes. It also remains tightly coupled to the underlying technology (i.e., the programming language) as the feature selection mechanism is hardcoded in the source code.

A better solution is to rely on recent advances in middleware technologies such as components- or service-based platforms. In such execution environments, one can configure (and reconfigure) a running system, by deploying and "wiring" the needed components. Provided that features are implemented by single components, the feature decomposition then lends itself naturally to the underlying decomposition into components.

Features orthogonal to the breakdown into components or services (e.g., security, logging) can be managed by tools such as BVR, which resolve variability in the domain model, rather than in the technological space.

3.2 The Benefit of Product Lines

3.2.1 The Organizational Aspect of Product Lines

As mentioned above, the key benefits brought by PLE go far beyond the mere technicalities of products' derivations: PLE implies a global shift from a *technical* to a *strategic* understanding of reuse. By focusing on a family of products, PLE forces managers, analysts, designers, and other stakeholders to consider their activity in the light of the complete market niche that is targeted and how the products fit in. PLE forces to anticipate the boundaries of the application domain where reuse is worth considering. As shown by Figure 16 below, it ideally permits to amortize and capitalize on any core asset produced during the development process including for instance requirements, documentations, test plans/cases and user support.

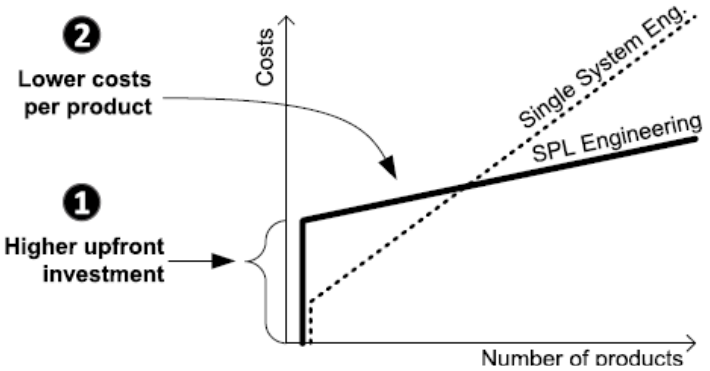

Figure 16 The cost of Software Product line [32]

This *systematic reuse* significantly improves the overall development process. The reuse of artifacts reduces the development effort as it avoids duplicating development effort. It also increases artefacts' internal quality as the probability to find and correct defects increases with the reuse rate. Reuse eventually ensures the internal spread and consolidation of the domain-specific knowledge accumulated throughout successive products developments.

Market agility also significantly gains from PLE adoption. PLE enables faster responses to new customers' needs and more generally to new market trends. Existing products can be quickly extended with other features already available in other products and new products can be built from new and yet unforeseen combinations of features requested by customers. PLE adoption may thus bring a competitive edge as it empowers the user with the ability to build the product that fits her very needs.

3.2.2 Configuring Products

Automated product derivation is the most emphasized feature of software product lines. By giving the user the possibility to select the features that she needs, it becomes possible to check the consistency of the whole product line, check the consistency of a given feature prescription, and to assemble the prescribed products.

Checking the consistency of a product line as a whole consists in ensuring that there exists at least one single product that meets all the constraints embedded in the associated feature model. Interestingly feature models (see Section 4.1.1) can be reduced to propositional logic formulae [33], and their validation thus boils down to the satisfiability problem (SAT). Although SAT is well-known to be a NP-Complete problem, recent advances in SPL [34] showed that industrial size SPLs form a very specific subset of SAT instance, which existing SAT solver can address.

Checking the validity of a specific feature prescription ensures that the prescribed features meet the constraints carried by the feature model. The prescription is valid if and only if the underlying variable assignment satisfies the associated logical formulae. SPL thus permits to detect automatically invalid configurations that will not work in practice.

Finally, assuming a given feature prescription is consistent with its enclosing SPL, it is possible to automated — possibly only partially — the construction and the validation of the associated products. This construction step is tightly coupled with the reuse capabilities of the underlying execution platform.

3.2.3 Dynamic Product Lines

The extra flexibility to adjust products to customers' needs can also be leveraged at runtime. The knowledge carried by product lines can be used to dynamically respond to changes in the system's environment by activating (resp. passivating) selected features. As its environment evolves, the system thus transitions from product to product in order to meet specific objectives, such as performance or user satisfaction.

The use of software product lines at runtime — so called *dynamic software product lines* (DSPL) [35] — provides a framework to ease and constrain future maintenance and evolution. It enables the specification of a "safe" envelope within which, maintenance and evolution have been anticipated and potentially validated. The DSPL framework also permits to automate these maintenance or evolution activities, by plugging in a reasoning engine that autonomously decides which product best fits the current environment. Such automated decisions range from the mere selection of the most suited product among a set of predefined ones to the dynamic exploration of the product space by successive feature activations (resp. passivation). Our ability to verify, validate and certify DSPL decreases with the size of the product lines and the complexity of the associated decision procedure.

3.3 Product Lines and Modularity

Building highly reusable software pieces has been the major impetus for several breakthroughs in Software Engineering: routines, modules, objects, components, services, aspects, etc. While *reusability* lacks a precise and well accepted definition, it is worth to note the distinction that exists between reused and reusable software: Reused components are not necessarily reusable *per se* (they may be the only available alternative) and reusable components are not necessarily reused (poor visibility, wrong timeline, etc.). Yet, the technologies successively proposed to develop reusable software pieces shed some lights on their key characteristics: effectiveness, generality, cohesion, substitutability and visibility, to name only a few.

- Effectiveness is the *sine qua none* condition for reusability: software that fails doing what they are supposed to do will certainly not be reused as is.
- Generality reflects the extent to which the problem solved by the software is common and directly impacts its reusability. The use of software pieces that only solve linear differential equations is for instance restricted to linear mathematical models.
- Cohesion characterizes pieces of software that have a single and well-defined responsibility. Cohesion requires some level modularity, in order for different concerns to be isolated into different units (i.e., modules). The resulting highly cohesive units are more easily understood, and in turn, more easily reused. Object-oriented technologies significantly contribute to the definition of general and cohesive abstractions.
- Substitutability calls for the presence of explicit and well defined interfaces, which enable the (dynamic) replacement of individual software pieces. Substitutability was a limitation of classical object-oriented technologies and motivates the development of components-based technologies.
- Visibility reflects the ability for a software piece to be identified and reused on the spot. Service-oriented architecture and the underlying publish-discover-invoke scheme directly promote higher visibility.

By contrast with the technologies mentioned above, a software product line is a framework which helps eventually deliver products made out of a set of reusable software pieces. To be effective, a software product line has to build upon an associated reusable technology: The more reusable are the available components, the more effective the product line will be. SPL contribute to realize the potential existing within an existing set of application specific and reusable software pieces.

In addition, a software product line defines a bounded context, within which reuse if worth considering. In practice, this context becomes delineated by the associated software architecture, which reduces the need for reusability. Software components only need to meet the reusability requirements within this architecture, and need not be generally reusable. Assumptions regard for instance communication protocols, middleware technologies, data encoding, etc. Building a software product line is thus tightly coupled to key architectural choices and eases the identification and development of the missing reusable software components.

Finally, as mentioned previously, reuse as understood in SPL goes beyond the sole underlying technology. SPL aims at reusing as much as possible any assets produced during the development cycle: requirements, analysis, tests, and possibly certification evidences.

4 Piecewise Certification

As product line technologies build upon the reusability of the underlying software technologies (e.g., objects, components, services), they form a promising approach to reduce the cost of certification and recertification, by maximizing reuse of certified units, here-after called "pieces". Yet, reuse is always challenging as it is difficult to ensure that third-party components will perform correctly in an environment for which they were not explicitly designed. Among others, the $ 500 million crash of Ariane 5 in 1996 [36], remains a strong evidence of the opposition between reuse and verification and in turn, certification. We review below the key techniques available to verify reusable piece of software and we discuss how they could be extended to certification.

4.1 What is a "Piece"?

4.1.1 Pieces Defined by Input and Output Parameters

As mentioned before, various software units of reuse have been proposed in the literature ranging from simple routines to high level services. From the standpoint of certification and recertification, a key aspect is the substitutability of a part: the ability to replace a unit by another one, while still guaranteeing that the whole is operational (e.g., safe, correct). The need for substitution of individual parts drove the development of component-based system, where *"a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition"* [37]. This general definition encompasses any piece whose dependencies with its environment are well specified so as to enable its substitution by an equivalent.

4.1.2 The Assume/Guarantee Paradigm

By analogy with the way relationships between people are specified by contracts constraining the rights and duties of each party, relationships between software pieces follow *contracts* specifying the assumption a piece makes and the guarantees it provides. Ultimately, this so called assume/guarantee paradigm aims at building the specification of a complete system in a bottom-up manner: by assembling the individual specifications of its parts.

A textbook example of contract characterizes the reuse of a square root function, called "sqrt". As described below, this contract expresses under which conditions one can calculate the square root of a real number.

```
sqrt(in x: Real, out r: Real)
        assume: x ≥ 0
        guarantee: x = r²
```

It is worth to note that the contract is completely disconnected from the algorithm actually used to compute the square root. The assume/guarantee paradigm is indeed strongly linked with information hiding as it requires pieces to expose the necessary and sufficient information to enable reuse. Ideally, this contract could also include extra-functional concerns, such as memory consumption, energy consumption or execution time, although such extensions are generally difficult to leverage for formal verification.

Design by contract [38] is a direct application of the assume/guarantee ideas in Software Engineering practices. Following ideas of the Floyd-Hoare logic [39], each routine is equipped with pre and post conditions capturing its semantics. Pre-conditions explicit the assumptions made by the routine to perform correctly, whereas the post-conditions specify the properties guaranteed after completion.

Such contracts are fragments of specification embedded at runtime into assertions in order to detect discrepancies between the specification and the actual behaviour of the routines. Coupled with testing techniques, contracts have shown to be an effective verification and diagnostic tool. As we shall see in the next section, contracts can also be used to verify various types of properties on software assemblies. Contracts are actively researched as a means to capture various software interactions, especially synchronization and quality of service [40].

4.2 Piecewise Verification and Piecewise Testing

Verification in Software Engineering is driven by two main approaches: *automated testing and formal verification*. Testing takes products at the end of development iterations and checks their adherence to specification (correctness, performance, usability, etc.) By contrast, formal verification builds mathematical models of systems and proves their correctness, performance, usability, etc. We review below the main characteristics of the two approaches.

4.2.1 Formal Contract-based Verification

A variety of assume/guarantee specifications and other contract-based specification have been developed as a means to compose (resp. decompose) formal system specifications. We shall restrict ourselves to an overview of the main ideas, but interested readers may found a comprehensive review of contract-based models in [41]. As explained assume/guarantee specification are couples (A, G) where A models the assumption a component has on its own environment, and G models the guaranties it offers to its environment. A and G are generally formal processes, whose composition may be subject to safety and liveness issues. The composition of such processes is yet not as straightforward as it may seems: mutually dependent components lead to circular reasoning which must handle carefully. To this end, various composition operators, often denoted by ||, have been developed for specific models. Communicating sequential processes (CSP) [42], the temporal logic of actions (TLA) [43, 44], Focus [45] or BIP [46] to name a few are examples of formalisms using such composition rule.

More recently, there has been an attempt to combine such formal verification techniques with architectural description languages (ADL). Wright [47] for instance pioneered the formalization of components and their assemblies into well-defined software architectures. Various ADLs have been then proposed such as ACME, ArchJava, UML 2.x [48, 49], SysML [50] or Modelica [51].

4.2.2 Testing of Product Lines

The idea of *automated testing* is to invoke a piece of software with specific inputs for which the expected outputs are known *a priori* in order to detect discrepancies with actual outputs. We review below the main techniques developed to test product lines technologies. Interested readers may find a more comprehensive treatment in [52, 53]. We highlight two main techniques, namely the 150 % model and the coverage array techniques, jointly developed by SINTEF and University of Oslo.

Testing mainly varies depending on the scope of the system under test (SUT). At the finer scale, single routines are tested independently by so called unit-tests. At a medium scale, the interaction between two or more components is performed by integration tests. Finally the complete systems or product have to be tested by the end-to-end tests, checking specific usage scenarios. Execution time, complexity and cost of maintenance all grow as scale does. Unit, integration and system tests represent different activities in the development process: end-to-end tests relate to general requirement, integration tests to general design and unit tests to detailed design. Testing is a resource intensive activity so a "brute force" to SPL testing requiring to explicitly and separately test all possible products in not feasible. Testing SPL requires each activity be reconsidered under variability: The "W" development model, presented below, is an attempt to adapt the well accepted V-model to variability.

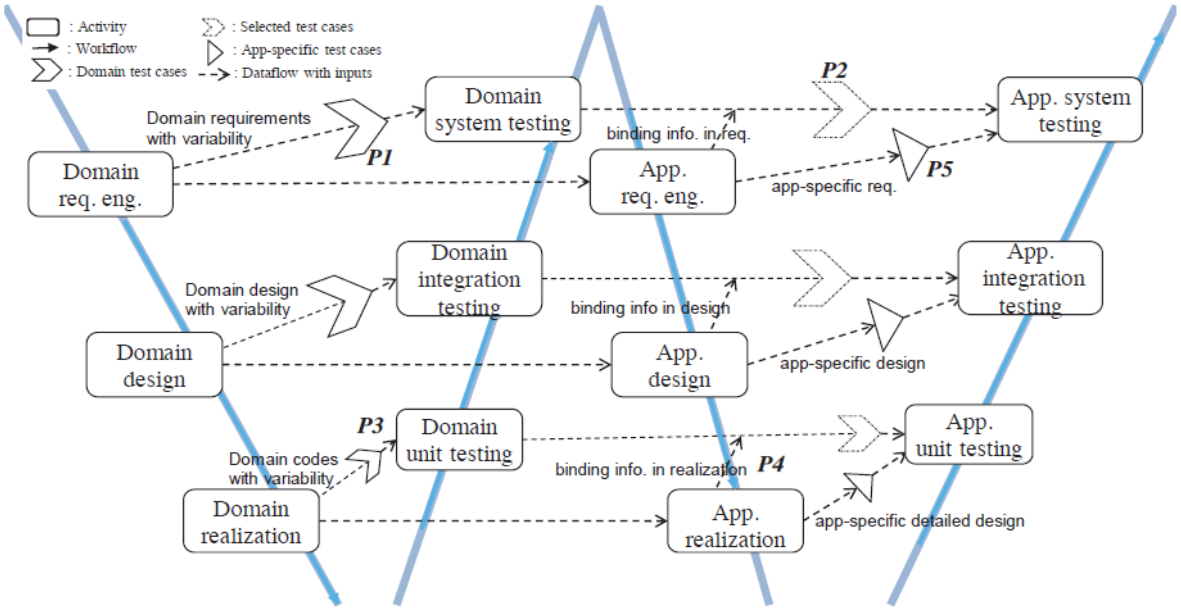

Figure 17 The "W model" [52]: an attempt to formalize the underlying testing in the context of SPL

The W-model distinguishes between domain-testing (i.e., product-line) and application (i.e., product) testing. Domain-centric activities result in test artefacts (test-cases, test plan, etc.) which shall be reused across product from the family, whereas application-centric activities result in artefacts tailored for a single specific product. It is worth to note the importance of having system tests for each possible product. Having a set of well tested components, each passing a separate large suite of tests, cannot detect issues occurring when two or more components interact. This problem, known as the *feature interaction problem* is one of the major challenges in SPL testing. We discuss below three main techniques addressing SPL testing: model-based testing, incremental testing and combinatorial testing. Interested reader may refer to [31] for additional details.

In model-based testing (MBT), models are used to capture the desired behaviour of the SUT, the testing strategies of interest, or both. For instance, one can describe the system as a finite state machine, capturing the set legal inputs and the set of associated outputs. It is thus possible to generate a test suite (a set of test cases) to reach a specific coverage criterion. In the context of SPL, Cichos et al. [54] proposed to build for instance a 150 % model as a single state machine aggregating the behaviour of all possible products. This 150 % model can thus be scoped down to generate tests covering any product resulting from the product line.

An alternative to minimize the cost of testing SPL is the use of incremental testing strategies. New test cases are generated based on the difference between the SUT and other products that have already been tested. Incremental testing exploits the relationship that binds SPL testing to regression testing: as regression testing aims at retesting a software piece that has changed, it can be used to test a new product, which conceived as an extension of an existing one.

Another promising approach to SPL testing is the use of combinatorial interaction testing. The idea is to select a small subset of products, whose executions are likely to trigger feature interaction problems. As shown by Kuhn et al [55], SPL follows some sort of 80/20 rule: most bugs are related to a few parameters configurations. SINTEF developed a technique to automatically select a minimal subset of products that maximizes the number of interaction exercised during testing [34], and in turn, the likelihood of detecting a feature interaction issue.

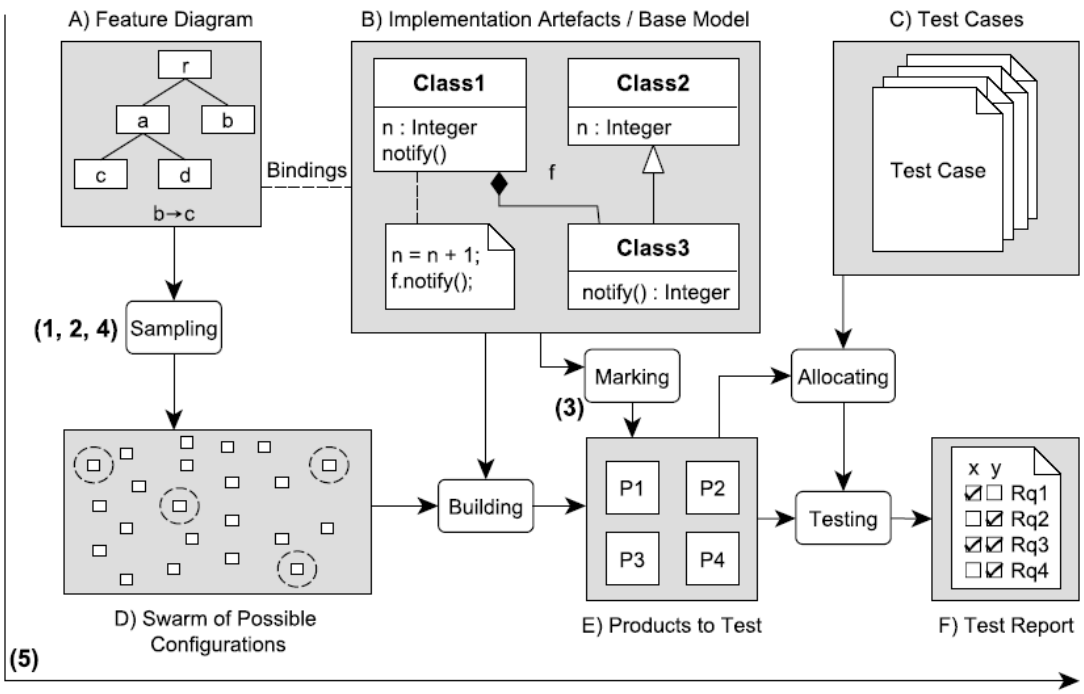

Figure 18 The ICPL tool to efficiently test product lines [31]

The associated too, called ICPL foster testing software product lines. The main contributions of this tool are summarized in Figure 18, testing SPL requires three main inputs:

- A software system and its implementation artefacts
- The feature model capture the inherent variability, and in turn the set of possible variants, which can be derived from the given software artefacts.
- A set of test cases used to validate products derived from the systems.

The first step consists in *sampling* the space of possible products, in order to cover the possible *t*-wise interactions between features (i.e., 1-wise ensures that all features are selected at least once, whereas 2-wise coverage ensures that each couple of feature is selected at least once). The resulting products can thus be automatically built by assembling existing software artefacts, and tested using the provided test cases.

4.3 What do Standards Say about Piecewise Certification?

Ideally, any changes in the product or in the related development process shall trigger the re-certification of the newly derived products. As shown below by Figure 19, re-certification triggers include changes in the product, changes in the process, but also changes in the associated standard or in the legislations that apply to the product.

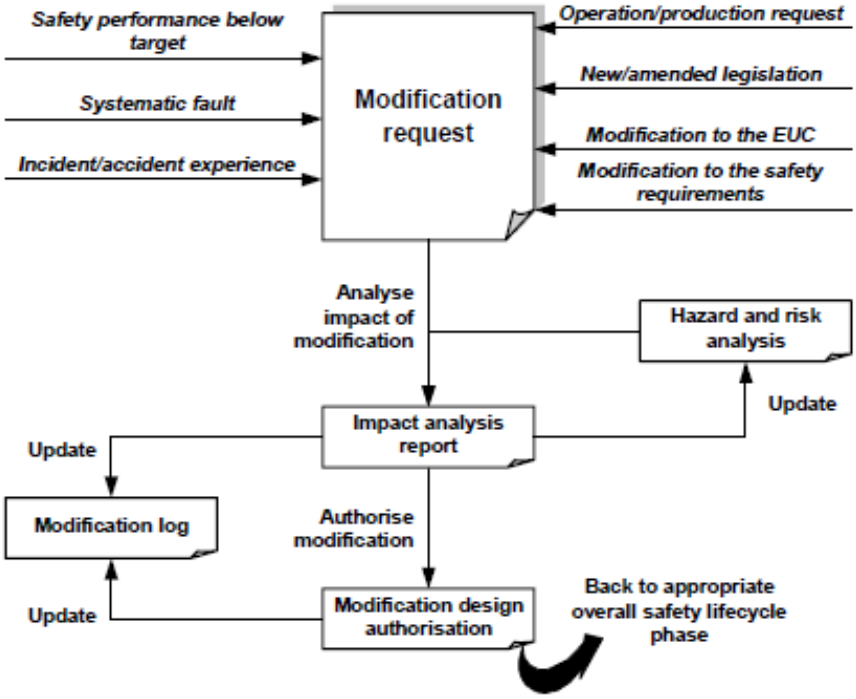

Figure 19 Modification handling process according to IEC 61508

Certification and re-certification thus induce a significant increase of the development process costs. Collected data [56] showed for instance that certification expenses with respect DO-178B increase by a factor of 3 to 5, depending on the associated criticality level. Standards and especially safety standards such as ISO 61508 or ISO 26262 recognize indeed the need for reuse in both system development and certification (see Section 2.3).

By analogy with advances in software architecture which enhanced reusability, modular certification aims at taming the high cost of certification by offering reusable certification pieces. If a system is made out of independently certified, replacing a component should only trigger the re-certification of that very component together with the overall assembly architecture. Recalling the techniques surveyed in Section 4.2, modular certification should be able to leverage existing piecewise verification techniques.

4.4 Tool Certification and its Relationship to Piecewise Certification

Certification is inherently about collecting evidence that a given product (in the case of safety) adhere to some given requirements. Eventually, there is never any absolute guarantee that the requirement hold and certification is only about consolidating the confidence one may have. Evidence can be collected from the product (functional safety), but also from the development process that was used to deliver the product. Knowledge about the process indirectly supports the product-based evidence, as a sound and generally well-established development process is more likely to yield a well understood product.

Although various software development processes are possible, a widely accepted one is the V-model (see Section 4.2.2), which requires that every design step (i.e., requirement analysis, system design and implementation) be secured by appropriate verification and validation (V&V) procedures. Traceability is needed for certification purposes and all V&V activities shall be properly documented.

Process certification also covers the tools that are used throughout the development process such as CASE tool, compiler, code generators, etc. For instance, the IEC 61508 distinguishes between three categories of tools depending on their impact on the final product. Category T1 includes all tools that do not directly impact the safety of the final product, such as text editors for instance. Category T2

includes tools contribute to improve safety, but which do not contribute to the running code such as static code analyzers or model-checkers. Finally, Category T3 includes tools that directly contribute to the running code of the product such code generators or compilers. While tools in the first category need not fulfil any specific requirements, the risk induced by using tools from Category T2 and T3 shall be assessed separately. In addition, tools from category T3 shall provide evidence that they meet their specifications.

4.5 Piecewise Certification and Product Lines

Various techniques and challenges related to the certification of safety-critical software systems are detailed in [57]. Product lines appeared as a promising approach for industries that produce families of such systems. The literature also reports about several attempts to apply product line ideas to safety critical systems in automotive or avionic, especially. Dordowsky *et al.* [58] described for instance how they built the NH90 product line of medium weight multi-role military helicopters, which resulted in 23 variants. While this work focused on using product lines to assemble software components, certification of modular components (w.r.t. DO-178B & AC20-148) is not addressed but is clearly identified as "a major cost block". Alternatively, Hutchesson and McDermid [59] showed how to leverage recent advances in model-driven engineering and component-based systems to foster to verification of high-integrity product lines. This approach contributes to provide the evidence needed to achieve modular certification.

Habli [60] made a first attempt to apply product line techniques to safety analysis. He proposed a specific meta-model to capture safety concerns in a product-line so that product-line safety and development artefacts can be jointly reused in a traceable and justifiable manner. Braga *et al.* [61] described a product line of unmanned aerial vehicles (UAV) named Tiriba. This work is a first attempt to leverage feature models to foster certification, as they capture the impact that each feature has on certification and thus can contribute to the impact analysis required when deriving new products (see Figure 19). Conmy and Bate [62] discussed the challenges of using product line and component-based design to ease the assembly of safety arguments.

Although product line engineering is perceived as a promising technique to do modular certification, research is still in its infancy. As identified above, product lines are a potential enabler for impact analysis, when products have to be re-certified. In practice re-certification will address the modification itself as well as the supporting evidence. All not affected parts of the safety related system as well as all affected evidence should be reused and need not to be re-certified. A second potential is the ability to easily identified compliant component or solutions that could meet the requirements of systems compliant with existing standards. Building blocks can be certified independently and yet be available as components off the shelf (COTS), ready for assembling new safety-critical systems. Only the resulting assemblies would therefore require certification.

5 Conclusion

Mixed criticality systems are typically multi-core embedded systems that integrate applications or software components of various criticality levels. Traditionally, certification is a key concern of safety-critical systems, and it ensures that enough evidence is provided to support the required safety integrity levels. When applied to mixed criticality systems, certification further complicated as it requires evidence that non-critical functions do not interfere with the safety-critical ones. In addition, as acknowledged by existing safety standards (e.g., ISO 61805, ISO 26262) one of the major costs in certification is the need for re-certification. Any change in the product, the requirements, or the standard triggers a new round of certification.

Product line engineering has been proposed as a way to maximize reuse across an organization product's platform. By making the commonalities between products explicit, a product line permits to maximize reuse of components, and in turn, to reduce cost and development efforts. Although in practice, product line engineering primarily focus on the derivation of new products, any related development activity such as documentation or certification can – in principle – benefit from the PLE approach.

From a process-oriented perspective, product line engineering is a promising approach to effectively reduce the cost of certification/re-certification. The existing attempts to apply product line ideas to mixed criticality systems primarily focus on reducing the development cost by maximizing reuse of software components. The issue of certification in product-lines, especially in safety-critical systems, is still in its infancy and has been mainly concerned with the construction of safety cases and with specific types of safety analysis. In the DREAMS project we intend to move beyond the state of art by combining recent advances in testing software product lines as a whole to foster the certification of safety-critical product lines.

Glossary

ADL	Architectural Description Language
ASIC	Application Specific Integrated Circuit
BIP	Behaviour, Interaction, Priority
COTS	Commercial-Off-The-Shelf
CSP	Communicating Sequential Processes
CVL	Common Variability Language
DREAMS	Distributed REal-Time Architecture for Mixed Criticality Systems
DSPL	Dynamic Software Product Lines
E/E/PE	Electrical/Electronic/Programmable Electronic
EASA	European Aviation Safety Agency
EN	European Standard
EUC	Equipment Under Control
FAA	Federal Avionic Administration
FPGA	Field Programmable Gate Arrays
FSM/QM	Functional Safety Management / Quality Management
GA	Generic Application
GP	Generic Product
ICPL	Isotope Coded Protein Labeling
IEC	International Electrotechnical Commission
IMA	Integrated Modular Avionic
ISO	International Organization for Standardization
MBT	Model Based Testing
MCS	Mixed Criticality System
OS	Operating System
OVM	Orthogonal Variability Models
PL	Product Line
PLE	Product-Line Engineering
RAM	Random Access Memory
RTOS	Real Time Operating System
SA	Specific Application
SAT	Satisfability Problem
SEooC	Safety Element out of Context
SIL	Safety Integrity Level

SIL CL	SIL Claim Level
SPL	Software Product Line
SUT	System Under Test
TLA	Temporal Logic of Actions
UAV	Unmanned Aerial Vehicles
V&V	Verification and Validation
WP	Work Package

List of Figures

Bibliography

[1] J. Perez, D. Gonzalez, S. Trujillo, A. Trapman, and J. M. Garate, "A safety concept for a wind power mixed-criticality embedded system based on multicore partitioning," in *Functional Safety in Industry Application, 11th International TÜV Rheinland Symposium*, Cologne, Germany, 2014, p. 36.

[2] J. Perez, D. Gonzalez, C. F. Nicolas, T. Trapman, and J. M. Garate, "A safety certification strategy for IEC-61508 compliant industrial mixed-criticality systems based on multicore partitioning," *Euromicro DSD/SEAA,* vol. Verona, Italy, August 2014.

[3] European Commission, "Mixed Criticality Systems - Report from the Workshop on Mixed Criticality Systems," February. 2012

[4] *IEC 61508-3: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements* IEC 61508, 2010.

[5] *IEC 61508-2: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 2: Requirements for electrical / electronic / programmable electronic safety-related systems,* IEC 61508, 2010.

[6] *IEC 61508-1: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 1: General requirements,* IEC 61508, 2010.

[7] P. Clements and L. Northrop, *Software product lines: practices and patterns*, 3rd ed.: Addison-Wesley Professional, 2001.

[8] J. Perez, "Development and certification of dependable embedded systems based on IEC-61508 - Introduction: Why, what and how - UPV/EHU seminar," IkerlanMay 2014.

[9] R. Von Hahn, C. Dirmeier, T. Jedelhauser, and M. Wagner, "D4.8 Challenges and Potentials of Certifying Product Lines," VARIES Consortium D4.8, September 2014.

[10] IEC, "IEC 61508-1: General Requirements," in *Requirements for electrical/ electronical / programmable electronic safety-related systems*, ed, 2010.

[11] U. M. o. Defence, "Safety Management Requirements for Defence Systems," in *Defence Standard 00-56 (Issue 4)*, ed, 2007.

[12] IEC, "IEC-61508-4: Definitions and Abbreviations," ed, 2010.

[13] IEC, "IEC 61508-3: Software requirements," 2010.

[14] ISO, "ISO 26262 -10," in *Road Vehicles - Functional Safety - Part 10: Guideline*, ed, 2009, p. 31.

[15] ISO, "ISO 26262-1," in *Road Vehicle - Functional Safety - Part 1: Vocabulary*, ed, 2011, p. 30.

[16] CENELEC, "EN 50129," in *Railway applications - Communications, signalling and processing systems - Safety related electronic systems for signalling*, ed, 2003, p. 94.

[17] R. Inc., "DO-178B: Software Considerations in Airborne Systems and Equipment Certification," ed, 2011.

[18] RTCA, "DO-297 Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations," ed, 2005.

[19] Aeronautical Radio Inc., "ARINC 653-1 Avionics Application Software Standard Interface. Part 1, Required Services," ed: ARINC, 2006.

[20] Aeronautical Radio Inc., "ARINC specification 653-2 — Avionics application software standard interface. Part 2 - Extended services," March 2006.

[21] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach," in *Dependable Computing Conference (EDCC), 2010 European*, 2010, pp. 67-72.

[22] SAVOIR/IMA Workgroup. (2012, Nov. 06). *Space Avionics Open Interface Architecture (SAVOIR)*. Available: http://savoir.estec.esa.int/SAVOIRIMA.htm

[23] R. L. Alena, J. P. Ossenfort, K. I. Laws, A. Goforth, and F. Figueroa, "Communications for Integrated Modular Avionics," in *Aerospace Conference, 2007 IEEE*, 2007, pp. 1-18.

[24] L. A. Zadeh, "Outline of a New Approach to the Analysis of Complex Systems and Decision Processes," *Systems, Man and Cybernetics, IEEE Transactions on,* vol. SMC-3, pp. 28-44, 1973.

[25] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," *Journal of Object Technology,* vol. 8, pp. 49-84, 2009.

[26] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA Tech. Report CMU/SEI-90-TR-21, Nov., 1990.

[27] P. Schobbens, P. Heymans, and J. C. Trigaux, "Feature Diagrams: A Survey and a Formal Semantics," in *Requirements Engineering, 14th IEEE International Conference*, 2006, pp. 139-148.

[28] K. Pohl, G\, \#252, n. B\, \#246, ckle*, et al.*, *Software Product Line Engineering: Foundations, Principles and Techniques*: Springer-Verlag New York, Inc., 2005.

[29] O. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen, "Adding Standardized Variability to Domain Specific Languages," presented at the SPLC 2008, Limerick, Ireland, 2008.

[30] O. Haugen, A. Wasowski, and K. Czarnecki, "CVL: common variability language," presented at the Proceedings of the 17th International Software Product Line Conference, Tokyo, Japan, 2013.

[31] M. F. Johansen, "Testing Product Lines of Industrial Size: Advancements in Combinatorial Interaction Testing," PhD, Department of Informatics, University of Oslo, Oslo, 2013.

[32] M. E. Janota, J. Kiniry, and G. Botterweck, "Formal methods in software product lines: concepts, survey, and guidelines," Lero, University of Limerick TR-SPL-2008-02, 2008.

[33] D. Batory, "Feature models, grammars, and propositional formulas," presented at the Proceedings of the 9th international conference on Software Product Lines, Rennes, France, 2005.

[34] M. F. Johansen, C. y. Haugen, and F. Fleurey, "Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible," in *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, 2011, pp. 638-652.

[35] M. Hinchey, P. Sooyong, and K. Schmid, "Building Dynamic Software Product Lines," *Computer,* vol. 45, pp. 22-26, 2012.

[36] J. M. Jezequel and B. Meyer, "Design by contract: the lessons of Ariane," *Computer,* vol. 30, pp. 129-130, 1997.

[37] C. A. Szyperski, *Component software - beyond object-oriented programming*: Addison-Wesley-Longman, 1998.

[38] B. Meyer, "Applying "Design by Contract"," *Computer,* vol. 25, pp. 40-51, 1992.

[39] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM,* vol. 12, pp. 576-580, 1969.

[40] A. Beugnard, J.-M. Jézéquel, N. l. Plouzeau, and D. Watkins, "Making Components Contract Aware," *Computer,* vol. 32, pp. 38-45, 1999.

[41] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier*, et al.*, "Contracts for System Design," INRIA, Research Report HAL-00757488, 2012.

[42] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM,* vol. 21, pp. 666-677, 1978.

[43] M. Abadi and L. Lamport, "Composing Specifications," *ACM Trans. Program. Lang. Syst.,* vol. 15, pp. 73-132, 1993.

[44] M. Abadi and L. Lamport, "Conjoining Specifications," *ACM Trans. Program. Lang. Syst.,* vol. 17, pp. 507-535, 1995.

[45] M. Broy, K. St\, \#248, and len, *Specification and development of interactive systems: focus on streams, interfaces, and refinement*: Springer-Verlag New York, Inc., 2001.

[46] A. Basu, M. Bozga, and J. Sifakis, "Modeling Heterogeneous Real-time Components in BIP," presented at the Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods, 2006.

[47] R. J. Allen, "A Formal Approach To Software Architecture," PhD, Carnegie Mellon University, 1997.

[48] Object Management Group, "Unified Modeling Language (UML) v2.4.1 - Infrastructure Specification," ed: Object Management Group, 2011.

[49] Object Management Group, "Unified Modeling Language (UML) v2.4.1 - Superstructure Specification," ed: Object Management Group, 2011.

[50] Object Management Group, "OMG Systems Modeling Language (OMG SysML)," ed: Object Management Group, 2012.

[51] Modelica Association, "Modelica - A Unified Object-Oriented Language for Systems Modeling (v3.3)," ed: Modelica Association, 2012.

[52] J. Lee, S. Kang, and D. Lee, "A survey on software product line testing," in *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*, 2012, pp. 31-40.

[53] M. F. Johansen, O. Haugen, and F. Fleurey, "A Survey of Empirics of Strategies for Software Product Line Testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, 2011, pp. 266-269.

[54] H. Cichos, S. Oster, M. Lochau, and SchC, "Model-Based Coverage-Driven Test Suite Generation for Software Product Lines," in *Model Driven Engineering Languages and Systems*. vol. 6981, J. Whittle, T. Clark, and Kc, Eds., ed: Springer Berlin Heidelberg, 2011, pp. 425-439.

[55] D. R. Kuhn, "Software Fault Interactions and Implications for Software Testing," *IEEE Transactions on Software Engineering,* vol. 30, pp. 418-421, 2004.

[56] E. Althammer, E. Schoitsch, G. Sonneck, H. Eriksson, and J. Vinter, "Modular certification support -- the DECOS concept of generic safety cases," in *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, 2008, pp. 258-263.

[57] A. Kornecki and J. Zalewski, "Certification of software for real-time safety-critical systems: state of the art," *Innovations in Systems and Software Engineering,* vol. 5, pp. 149-161, 2009.

[58] F. Dordowsky, R. Bridges, and H. TschC6pe, "Implementing a Software Product Line for a Complex Avionics System," in *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, 2011, pp. 241-250.

[59] S. Hutchesson and J. McDermid, "Development of High-Integrity Software Product Lines Using Model Transformation," in *Computer Safety, Reliability, and Security*. vol. 6351, E. Schoitsch, Ed., ed: Springer Berlin Heidelberg, 2010, pp. 389-401.

[60] I. M. Habli, "Model-based assurance of Safety-Critical Product lines," University of York, Department of Computer Science, 2009.

[61] R. T. V. Braga, O. T. Jc:nior, K. R. L. J. C. Branco, and J. Lee, "Incorporating certification in feature modelling of an unmanned aerial vehicle product line," in *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*, 2012, pp. 249-258.

[62] P. Conmy and I. Bate, "Assuring Safety for Component Based Software Engineering," in *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, 2014, pp. 121-128.