ECSEL Research and Innovation actions (RIA)

AMASS

**Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems**

# Baseline and requirements for architecture-driven assurance D3.1

| | |
|---|---|
| **Work Package:** | WP3: Architecture-Driven Assurance |
| **Dissemination level:** | PU = Public |
| **Status:** | Final |
| **Date:** | 9th March 2018 |
| **Responsible partner:** | Stefano Tonetta and Ramiro Demasi (FBK) |
| **Contact information:** | {tonettas, demasi} AT fbk.eu |
| **Document reference:** | AMASS_D3.1_WP3_FBK_V1. 1 |

*This deliverable is part of a project that has received funding from the ECSEL JU under grant agreement No 692474. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and from Spain, Czech Republic, Germany, Sweden, Italy, United Kingdom and France*

## Contributors

| Names | Organisation |
|---|---|
| Stefano Tonetta and Ramiro Demasi | FBK |
| Tomáš Kratochvíla, Petr Bauch | HON |
| Andreas Kager | AVL |
| Bernhard Winkler | ViF |
| Benito Caracuel | TLV |
| Stefano Puri, Silvia Mazzini | INT |
| Jose Luis de la Vara, Jose Maria Alvarez, Gonzalo Génova, Elena Gallego, Eugenio Parra, Juan Llorens | UC3 |
| Jose Fuentes, Luis Alonso, Borja López | TRC |
| Xabier Larrucea, Alejandra Ruiz, Huáscar Espinoza, Garazi Juez | TEC |
| Jiri Barnat, Ivana Cerna, Jaroslav Bendik | UOM |
| Juan Castillo | TAS |
| Andrea Critelli, Luca Macchi | RIN |
| Monajemi Behrang, Kaiser Bernhard, Suryo Buono, Yu BAI | B&M |
| Irfan Sljivo | MDH |
| Michael Soden | KMT |

## Reviewers

| Names | Organisation |
|---|---|
| [Peer Reviewer]  Morayo Adedjouma | CEA |
| [Peer Reviewer]  Elena Alaña Salazar | GMV |
| [Peer Reviewer] Javier Herrero | GMV |
| Jose Luis de la Vara | UC3 |
| Huáscar Espinoza | TEC |

## Document History

| Version | Date | Change | Author (Partner) |
|---|---|---|---|
| V1.0 | 2016-09-29 | Approved by TC | Stefano Tonetta and Ramiro Demasi (FBK) |
| V1.1 | 2018-03-09 | Revised to consider comments from EC reviewers (June 2017) | Stefano Tonetta (FBK) |

# TABLE OF CONTENTS

# List of Figures

# List of Tables

# Executive Summary

This deliverable D3.1 (Baseline and requirements for architecture-driver assurance) sets the stages of WP3. The stage is set by first of all recalling the AMASS context, motivation, objectives. Then, the problem in its multifaceted nature is stated. We analyse the state of art and the state of the practice concerning architecture-driven assurance. More specifically, the analysis aims at allowing AMASS to adopt the best features from existing approaches and to guarantee compatibility. We also analyse other ongoing and past projects, as well as available technology in the market. Moreover, when relevant, the state of the art and the state of the practice are compared in order to identify possible gaps. This comparative work ensures the identification of concrete needs, calling for new solutions, and ensuring the innovation of the project and future feasibility of exploitation of results.

Finally, a way forward is proposed. The proposal ways consist of a consolidation of the existing results achieved within OPENCOSS, SafeCer and other ongoing and past projects, and of the available technology on the market and state of practice. Most specifically, regarding system architecture modelling for assurance, it appears that there is currently a trend towards extending modelling languages (e.g. SysML) to better and explicitly support the concepts and needs from assurance standards. This is also in line with the stated need in OPENCOSS CCL for better relating it with component and system models for safety-critical systems, such as those from SafeCer and CHESS. Based on prior work, a generic UML profile-based approach could be suitable. It will also be necessary to select the system modelling languages to extend and link with assurance models. Standard languages, and especially languages used in the case studies, are the main candidates. Concerning assurance patterns library management, we have observed that further investigation needs to be carried out to develop a more enhanced argumentation library which covers not only safety argumentation patterns but also some other aspects such as security. Concerning assurance activities concerning novel technologies, several standard requirements might need to be adapted or modified to include the special requirements that novel technologies demand. This includes not only new specific requirements but also novel V&V techniques. At the same time, argumentation patterns of several concerns will be further investigated and developed in AMASS to facilitate the reuse of specific technologies. Finally, concerning contract-based assurance composition approaches, standard architectures (such as AUTOSAR in the automotive industry, IMA in avionics, ETCS in railway) require some safety/security *architectural patterns* definition and application (3-level-monitoring, E2E protection, and partitioning, among others), and auto-generation of platform models and configurations based on these patterns (e.g. for AUTOSAR and IMA). The use of patterns speeds architecture specification and facilitates the (re)use of components targeted at being used in such patterns. We also observed that the architecture can be enriched with contracts that formalize the functional requirements to ensure that the system responds correctly to some safety requirements.

To sum up, this deliverable will set the baseline for the development of the AMASS system architecture-driven assurance and will specify the requirements that it has to meet.

D3.1 relates to the AMASS deliverables D3.2 and D3.3 which are the outputs for task 3.2 (Conceptual Approach for Architecture-driven Assurance).

# 1. Introduction

This introductory chapter is aimed at recalling the context of the AMASS project as well as the objectives and expected results that pertain to this document.

Embedded systems have significantly increased in number, technical complexity, and sophistication, moving towards open, interconnected, networked systems (such as "the connected car" and the cloud), integrating the physical and digital world, thus justifying the term “cyber-physical systems” (CPS). This “cyber-physical” dimension is exacerbating the problem of ensuring safety, security, availability, robustness and reliability in the presence of human, environmental and technological risks. Furthermore, the products into which these Cyber-Physical Systems (CPS) are integrated (e.g. aircrafts) need to respect applicable standards for assurance and in some areas they even need certification. The dimension of the certification issue becomes clear if we look at the passenger plane B 787 as a recent example – it is reported in [181] that the certification process lasted 8 years and has consumed 200.000 staff hours at the FAA, just for technical work. The staff hours of the manufacturer even exceeded this figure as more than 1500 regulations had to be fulfilled, with evidence reflected onto 4000+ documents. Although aircrafts are an extremely safety-critical product with many of such regulations, the situation in other areas (railway, automotive, medical devices etc.) is similar.

Unlike practices in electrical and mechanical equipment engineering, CPS do not have a set of standardized and harmonized practices for assurance and certification that ensure safe, secure and reliable operation with typical software and hardware architectures. As a result, the CPS community often finds it difficult to apply existing certification guidance. Ultimately, the pace of assurance and certification will be determined by the ability of both industry and certification/assessment authorities to overcome technical, regulatory, and operational challenges. A key regulatory-related challenge has to be faced when trying to reuse CPS products from one application domain in another because they are constrained by different standards and the full assurance and certification process must be applied as if it were a totally new product, thus reducing the return on investment of such reuse decisions. Similarly, reuse is hindered often even within the same domain, when trying to reuse CPS products from one project to another, where assumptions change together with the criticality level.

To face all these challenges, the AMASS approach focuses on the development and consolidation of an open and holistic assurance and certification framework for CPS, which constitutes the evolution of the OPENCOSS and SafeCer approaches towards an architecture-driven, multi-concern assurance, and seamlessly interoperable tool platform.

The AMASS tangible expected results are:

a) The AMASS Reference Tool Architecture, which will extend the OPENCOSS and SafeCer conceptual modelling and methodological frameworks for architecture-driven and multi-concern assurance, as well as for further cross-domain and intra-domain reuse capabilities and seamless interoperability mechanisms (based on OSLC specifications).
b) The AMASS Open Tool Platform, which will correspond to a collaborative tool environment supporting CPS assurance and certification. This platform represents a concrete implementation of the AMASS Reference Tool Architecture, with a capability for evolution and adaptation, which will be released as an open technological solution by the AMASS project. AMASS openness is based on both standard OSLC APIs with external tools (e.g. engineering tools including V&V tools) and on open-source release of the AMASS building blocks.
c) The Open AMASS Community, which will manage the project outcomes, for maintenance, evolution and industrialization. The Open Community will be supported by a governance board, and by rules,

policies, and quality models. This includes support for AMASS base tools (tool infrastructure for database and access management, among others) and extension tools (enriching AMASS functionality). As Eclipse Foundation is part of the AMASS consortium, the Polarsys/Eclipse community (www.polarsys.org) is a strong candidate to host AMASS.

To achieve the AMASS results, as depicted in Figure 1, the multiple challenges and corresponding project scientific and technical objectives are addressed by different work-packages.

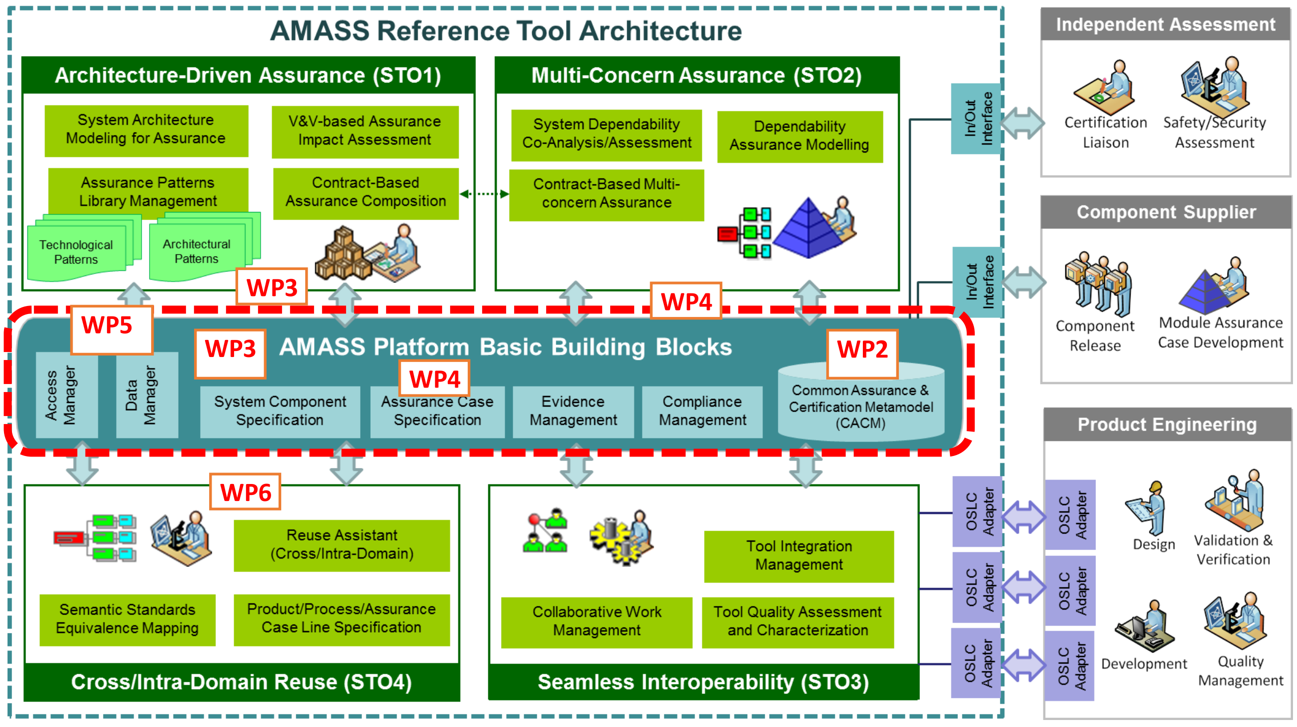

AMASS Reference Tool Architecture
Architecture-Driven Assurance (STO1)
System Architecture Modeling for Assurance
V&V-based Assurance Impact Assessment
Assurance Patterns Library Management
Contract-Based Assurance Composition
Technological Patterns
Architectural Patterns
Multi-Concern Assurance (STO2)
System Dependability Co-Analysis/Assessment
Dependability Assurance Modelling
Contract-Based Multi-concern Assurance
WP3
WP4
WP5
WP3
WP4
WP2
AMASS Platform Basic Building Blocks
Access Manager
Data Manager
System Component Specification
Assurance Case Specification
Evidence Management
Compliance Management
Common Assurance & Certification Metamodel (CACM)
WP6
Reuse Assistant (Cross/Intra-Domain)
Semantic Standards Equivalence Mapping
Product/Process/Assurance Case Line Specification
Cross/Intra-Domain Reuse (STO4)
Tool Integration Management
Collaborative Work Management
Tool Quality Assessment and Characterization
Seamless Interoperability (STO3)
In/Out Interface
In/Out Interface
OSLC Adapter
Independent Assessment
Certification Liaison
Safety/Security Assessment
Component Supplier
Component Release
Module Assurance Case Development
Product Engineering
Design
Validation & Verification
Development
Quality Management

**Figure 1: AMASS Building blocks**

WP3 aims at addressing Architecture-Driven Assurance. More specifically, with respect to the AMASS goals, this deliverable presents the background in terms of problem and solution space related to: Goal 1 (G1) and Goal 3 (G3), the corresponding project objective O1, and to the project scientific and technical objective (STO) 1. G1, G3, O1 and STO1 are recalled here to make the deliverable self-contained.

**G1:** to demonstrate a potential gain for design efficiency of complex CPS by reducing their assurance and certification/qualification effort by 50%.

**G3:** to demonstrate a potential raise of technology innovation led by 35% reduction of assurance and certification/qualification risks of new safety/security-critical products.

**O1:** define a holistic approach for *architecture-driven assurance* to leverage the reuse opportunities in assurance and certification by directly and explicitly addressing current technologies and HW/SW architectures needs

**STO1** focuses on Architecture-Driven Assurance, including: a) System Architecture Modelling for Assurance, b) Architectural Patterns for Assurance, c) Assurance of specific technologies, d) Contract-Based Assurance, e) V&V-based Assurance.

This document is deliverable D3.1 (Baseline and requirements for architecture-driven assurance), which is the output of task 3.1 (Consolidation of Current Approaches for Architecture-driven Assurance), released by the AMASS WP3 (Architecture-Driven Assurance). WP3 shall develop the means necessary for providing the system architecture-driven assurance approach of AMASS. Such an approach will be based on the extension of OPENCOSS and SafeCer conceptual results and platform in order to:

1. deal with architectural assurance patterns and with the assurance and certification needs of specific technologies (e.g., multicore), and
2. link assurance and certification models with system models (e.g. the latter represented with SysML) and standard software architectures (e.g. AUTOSAR and IMA). Moreover, WP3 will integrate OPENCOSS and SafeCer approaches and will extend them in order to consider standard software architectures.

Moreover, to achieve STO1, WP3 is structured into three tasks. The purpose of this deliverable is to document the work conducted during Task 3.1. More specifically, the goal of the deliverable is multi-fold:

1) to analyse the problem related to architecture-driven assurance
2) to present a corresponding state of the art
3) to present the current state of the practice; and finally, based on these findings
4) to present a consolidation of existing results and profit from ongoing and past projects.

The rest of the deliverable is organised as follows. Section 2 states the main concepts and objectives on Architecture-Driven Assurance (ADA) in AMASS. In section 3 is described the state of the art on ADA. Subsequently, section 4 describes the state of the practice on ADA. Finally, section 5 presents a summary of the main points from previous sections, detecting the gaps between state of the art and state of the practice, and the way forward in AMASS.

# 2. Problem Statement and Concepts

One of the main contributions of the AMASS project is to provide a modelling language (metamodel), tools, and techniques to support an architecture-driven assurance, i.e., an assurance that exploits and is linked to the system architecture in order to provide more structured evidences and arguments to show that the system is free of vulnerabilities. In particular, the system architecture is used for model-driven engineering, contract-based and pattern-based design and argumentation. In this respect, there are a number of challenges that must be addressed by the project, as discussed in the following sections.

## 2.1 System Architecture Modelling for Assurance

### 2.1.1 Exploiting the System Architecture in the Assurance Case

The system architecture is one of the first artefacts produced by the development process and includes many design choices that should be reflected in the assurance case. Therefore, we have to understand which elements of the system architecture are important for the assurance case. The existing OPENCOSS CCL (Common Certification Language) metamodel corresponds mainly to an assurance metamodel, and should be extended with (or linked to) other modelling formalisms to enable a more detailed definition of system and analysis of system’s dependability. In general, CCL needs to be extended for dealing with the linkage between its assurance framework and system architecture models. This will facilitate a finer-grained management of artefacts, such as those involved in the management of a hazard log in the railway domain: a hazard in a fault tree analysis, a safety requirement in a requirements specification, a block in an architecture specification, an interface in a design specification, a step in a verification report, a test case in a validation report, a section of a safety case, and so on. For example, the CCL should be extended with concepts such as component decomposition and contract refinement, as developed in SafeCer, to enable an architecture-driven reuse of models and assurance artefacts.

### 2.1.2 System Architecture Languages

There are many languages suitable to describe the system architecture, but most of them share the main concepts that are relevant for the system architecture are in common to these languages. Therefore, we have to face the problem of defining a meta-model for a generic system component specification with such architectural concepts. The SafeCer project created a first generic model, implemented in CHESS, but the link with other architecture description languages remained at a conceptual level. The current OPENCOSS models allow the treatment of artefacts only at a coarse "black box" level. These models will be extended in AMASS so that they are linked to modelling formalisms for safety information (information necessary to realize, analyse and verify systems’ safety) and to system modelling. The plan is to study the relation of the OPENCOSS and SafeCer assurance models with different system modelling languages (UML, SysML, AADL, EAST-ADL, etc.), safety modelling profiles, and specific platform models and architectures like AUTOSAR for automotive and IMA for avionics.

### 2.1.3 Architectures Trade-Off and Comparison

During the system development process, it is often the case that different system architectures are compared or one architecture is replaced by another one to trade-off different aspects. For example, a single-point-of-failure component is replaced by some redundant components, components may be removed or replaced for reducing the cost because of the project budget, the deployment and physical partitioning may have a completely different topology with respect to the logical decomposition of the system. Providing support to compare different system architectures will allow industry to make more informed decisions regarding what can be reused between systems (including difference versions of systems) and reuse consequences.

## 2.2 Architectural Patterns for Assurance

### 2.2.1 Architectural Patterns

An architectural pattern is a partial specification of a part of the system architecture that can be instantiated/used in a project-specific design. There are many opportunities to define architectural patterns, as the result of standard of practice applied in a specific domain or coming from the standards. Many patterns can be defined for fault tolerant mechanisms, including redundancy schemas and components for fault detection, isolation, and recovery. Other patterns can be created for specific domains (although they can be probably reused in other domains). For example, in the space domain, when considering the design of a satellite, the top-level architecture listing the sub-systems (AOCS, thermal, power, …) is almost the same for all projects; in the automotive domain, the AUTOSAR specifies how the communication should be protected from failures. Architectural patterns can be used to reduce the cost of design, increase the quality of the developed system, but also for auto-generation of platform models and configurations based on these patterns.

### 2.2.2 Interaction between Assurance and Architectural Patterns

OPENCOSS and SafeCer have straightforward mechanisms to specify assurance patterns for argumentation and for compliance with standards. However, further research and case studies are necessary to integrate cohesively these patterns with the architectural patterns and to integrate them into specific assurance and certification activities. The use of patterns speeds architecture specification and facilitates the (re)use of components targeted at being used in such patterns. Moreover, it enables the reuse of analysis results associated with the patterns. Therefore, we want to address the problem of defining the assurance patterns that can be associated to specific architecture patterns or design mechanisms. For example, a specific assurance pattern can be associated to the tolerance on failure communication associated with E2E protection of AUTOSAR or to the security-related non-interference associated with partitioning in MILS systems.

### 2.2.3 Architectural Patterns from Standards

There are many standards, in many domains, that specify parts of the system architecture such as sub-systems decomposition, component interfaces, communication packets. For example, the ETCS standard specifies how the train on-board system should interface with the track-side system, the AUTOSAR provides standardized interfaces for components at different layers of the design; in the space domain, the ECSS Packet Utilization Standard (PUS) specifies telecommands and telemetry packets for asynchronous communication to and from satellites. We will use architectural patterns to formalize the architectural elements specified in standards.

## 2.3 Assurance of Specific Technologies

Some specific technologies offer many benefits in terms of performance, reconfiguration or adaptability. However, their use in safety critical domains still lacks from maturity due to certification issues. Devices such as FPGA need to be safely deployed so that they can be certified.

Taking into account that OPENCOSS and SafeCer results are technology-agnostic, the assurance and certification of many characteristics of the new technologies of CPS are not supported. AMASS will tackle those issues addressing different technology patterns. More specifically, certification issues regarding MultiProcessor System-on-a-Chip (FPGA or Microcontroller based), Programmable Logic Devices, Commercial Off-The-Shelf, IMA, AUTOSAR or adaptive systems will be addressed. The detailed characteristics of the most recent and future technologies for CPSs will set under what circumstances reuse, assure, and certification of CPSs is possible.

## 2.4 Contract-Based Assurance

### 2.4.1 Assurance Patterns for Contract-Based Design

An important issue in AMASS is the integration and consolidation of the concept of contracts from the existing results of the OPENCOSS and SafeCer projects. In particular, the AMASS assurance for the argumentation that a system architecture is compliant with the system properties will follow the contract refinement defined in the system model.

### 2.4.2 Enriching the Evidence Produced by Contract-Based Design

A general challenge for those tools (e.g., OCRA) that are used for analysing and verifying contract specifications using formal methods is to provide useful evidences in order to enrich the contracts refinements argument. Similarly, in the context of safety analysis based on the contract specification, the goal is to enrich the assurance case with fault trees showing the dependency of system failures on the component failures.

### 2.4.3 Automation in Contract-Based Design

Another challenge is to increase the automation capabilities that are provided by tools supporting contracts. Such automations will include pre-defined properties and contracts derived from the standards as done for the Catalogue of System and Software Properties (CSSP) defined in the CATSY project [184] or associated to architectural patterns derived from the standards. Also, user guidance during the design of safety critical systems (like the selection, based on contracts, of appropriate components under consideration of their safety properties) is a subject for a higher automation degree that would also increase the re-usability of critical system components even cross-domain.

## 2.5 V&V-based Assurance

Another challenge of the AMASS project is to enrich the OPENCOSS and SafeCer assurance approaches with V&V techniques. For example, formal techniques can be used to validate that a requirements specification is complete, correct, and unambiguous and to verify that the deployed system satisfies those requirements. In fact, many safety issues in the deployed system are due to errors in the requirements specification, which are typically discovered very late in the development process. Therefore, the AMASS assurance approach will make sure that evidence for arguing that the requirements specification is valid are provided as part of the assurance case. When the requirements specification is formalised and validated, it is then employed in the development of system designs and of the final system. All intermediate stages will also be checked for compliance with requirements using automated V&V tools. The situation and needs described for requirements specifications could also be applied to other artefact types, e.g. V&V of design models.

# 3. State of the Art on Architecture-Driven Assurance

This chapter provides an overview concerning the state of the art on Architecture-Driven Assurance. The overview is structured by topics, where many results came from previous related projects to AMASS such as SafeCer and OPENCOSS.

## 3.1 Contract-based approaches

In this section we review diverse contract-based approaches related to subsection 2.2 "Architectural Patterns for Assurance" and 2.4 "Contract-based Assurance". We study the concepts of contracts defined in OPENCOSS and SafeCer, which will be integrated in AMASS. In addition, we present some formalisms to specify and analyse contracts and the related tool supports implemented in different projects. We also analyse the extension of some of the approaches to cover safety analysis.

### 3.1.1 Contract-based approaches based on Temporal logic

Contract-based design, first conceived for software specification [10] and now applied also to embedded systems [1], [2], [3], [4], [5], [6], [7], [8], [9], [12], [13], is a very promising paradigm, amenable to stepwise refinement, compositional reasoning, and reuse of components. The idea is to annotate the architectural decomposition with contracts that specify the relevant behavioural aspects of each component interface. More specifically, a contract is composed by an assumption and a guarantee. The former specifies the expected behaviour of the component environment, and the latter specifies how the component must behave in response. The system resulting from the composition of implementations satisfying the contracts according to the annotated architecture is guaranteed to satisfy the overall system contracts.

In many formal modelling approaches, in the underlying model of communication, the components receiving an input are blocking, in the sense that if they cannot receive the input, they block the component generating such data or event. As in some architecture languages, input/output are just a syntactic way to represent shared labels. For example, in the framework described in [7], contracts are specified with a temporal logic defined over a set of variables, their product is given by language intersection, and the contract satisfaction and refinement is defined in terms of language inclusion. This framework has been implemented in a tool, called OCRA [20]. In more details, the approach is very efficient, because the overall correctness proof is decomposed into proofs local to each component. However, part of the complexity is delegated to the designer, who has the burden of specifying the contracts. Typical problems include understanding which contracts are necessary, and how they can be simplified without breaking the correctness of the refinement. In [86], the authors tackle these problems by proposing a new technique to understand and simplify a contract refinement. The technique, called *tightening*, is based on parameter synthesis. The idea is to generate a set of parametric proof obligations, where each parameter evaluation corresponds to a variant of the original contract refinement, and to search for tighter variants of the contracts that still ensure the correctness of the refinement.

One of the main concerns in model-based system engineering is to design the architecture of systems so that the components are properly integrated in order to satisfy the system properties. Architecture description languages specify the syntactic interfaces of components in terms of data and event ports, their connections and decomposition. Contract-based design provides a formal framework to specify the semantic interface of components detailing the assumptions on the input received from the environment and the guarantee on the input/output relationship.

In SafeCer, Linear-time Temporal Logic (LTL) [11] has been used to express the assertions in the contracts over data and event ports. In this context, the behavioural model of a component is verified to satisfy the contract associated to that component. In order to be **reused**, the behavioural model must also be compatible with the environment of the component provided by the system design. It exploits the contract specification to ensure that the component implementation is compatible with any environment satisfying the assumption of the contract. In addition, the framework for contract-based design has been extended to take explicitly into account the problem of the component to be compatible with any environment that satisfies the assumptions, in the sense that the component must accept all the inputs that are produced by such an environment. Moreover, a compositional method is provided to solve this problem exploiting the refinement of contracts.

As we said, Linear-time Temporal Logic (LTL) has been used to express the assertions in the contracts over data and event port, where they reduce the problem to LTL model checking. The main novelty came from the fact that the compatibility is local and is composed exploiting trace-based inclusion checks based on the contracts. This is important to remark due to it is a great advantage, given that the receptiveness check turns out to be very expensive, and tackling it on the final system implementation is impractical.

One of the main contributions in this area is the extension of the framework in [7] to input/output components, and taking into account standard problems of interface theory such as the compatibility of the implementations. A key finding is that the same notion of contract refinement based on trace inclusion can be used as compositional rule for checking the receptiveness of component implementations. This enables us to fully exploit mature technology of temporal satisfiability and symbolic model checking.

### 3.1.2 Contract-based approaches based on agreements among components

OPENCOSS project defined compositional assurance approaches where safety case contracts are playing a key role [158].

Contract-based approaches are hard to apply from a safety perspective. In fact, several companies and standardisation bodies are reluctant to consider this kind of contracts during certification processes. However, we can consider that a contract can be designed in a way that it considers agreements among components. The main interfaces are defined in order to facilitate interoperability and integration among these components. A component has a correct functionality when the interaction with its interfaces is well defined, and its preconditions and postconditions are satisfied. From a safety perspective, system properties should be verified as a whole. A system composed by safe components may not be safe. Therefore, quality assurance activities, including safety assessment, are set in order to verify and validate system properties. Several aspects should be considered. For example, failures modes are also analysed, and all assumptions and contexts are considered. Standards address this problem in different ways. ISO 26262 defines a development Interface Agreement (DIA) document, which is an agreement between OEM and suppliers, by defining procedures and responsibilities. In addition, ISO26262 also refers to Safety Element Out Of Context (SEooC) which defines and interprets concepts, procedures and functionalities (also non-functionalities) by manufacturers, suppliers and developers. In the avionics domain we can find similar requirements while talking about modules and application reuse on an IMA (Integrated Modular Avionics) platform. DO-297 requires the definition of component limitations and assumptions, among others, for component acceptance. In this context we need to analyse its usage domain to ensure that it is being reused in the same way as it was originally intended.

The process of composition varies depending on the focus:

- **Compositional Argumentation**. A contract module contains the relationship between two modules, and how a claim in one module supports the argument in another. Arguments are encapsulated in a module, or in a set of modules. The argumentation editor of OPENCOSS is able to define argumentations from a compositional point of view, so-called "modular argumentation".

- **Project Reuse**. A safety assurance project can reuse parts of another project. Traditionally this is carried out in industrial settings. Artefacts used as evidences to support assurance on a given project, want to be used as evidences in another project. Contracts can play a keystone for the reuse of components.
- **Component composition**. This is the traditional approach component based engineering approaches.

Guidelines and standards prescribe the information needed to manage at the assurance project level. When analysing guidelines and standards, we noticed that the data required for assurance are classified in three main categories (Figure 2):

- *Artefacts*: referring to the data required by an entity when doing the safety assessment
- *Properties*: these are characteristics that must be present after the integration in order to confirm that there are no concerns or an emerging unknown behaviour. The properties need to be verified, and the verification needs to be included as part of the evidence.
- *Processes*: refers to the activities that shall be performed in order to prepare the reuse and after the reuse itself in order to comply with the standards requirements.

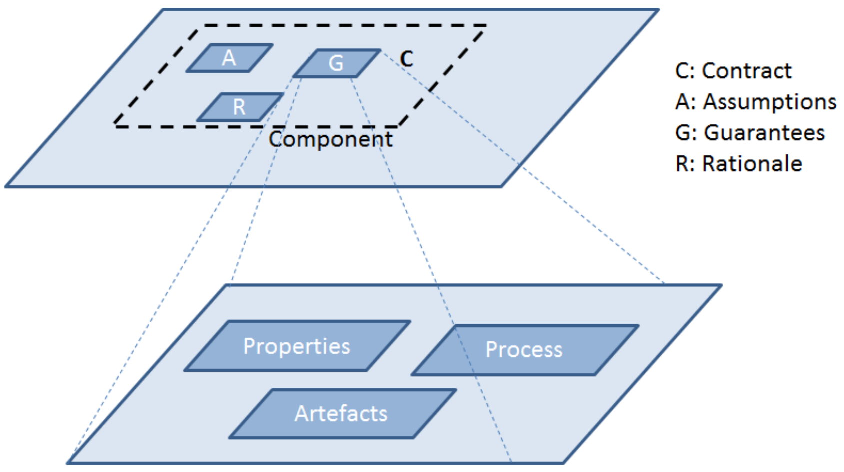


**Figure 2: Contracts View for Component**

A contract is characterised by the following sections (Figure 3):

Definition

What is assumed
- Activities/processes that shall be done by the integrator of the component
- Properties of the component that shall be checked after the integration
- Artefacts that shall be completed or done after the integration of the component

What is guaranteed
- Activities/processes that shall be done by the developer of the component
- Properties of the component that shall be checked after the integration
- Artefacts that should be completed or done after the integration of the component

Strategy
- Impacts on the guarantees if any of the assumptions is not valid
- Trace between risk mitigation needs and protection mechanism
- Rationale about the limits, conditions, and use of the component

**Figure 3: Contracts definition template**

OPENCOSS defined a Common Certification Language (CCL) ([160]) relying on three aspects: Compliance management, safety argumentation and evidences management. Contracts are based on this CCL. Contract data related to a component need to include information about assured properties and behaviours of that component, the artefacts that should be accessible to the authorities and the evidence of the process and activities executed to fulfil the component's assurance requirements. All this information is related to an evidence meta-model Contract references to the artefacts and their properties, and the rest of the information from evidence meta-model is considered as a black box. Activity on the contract defines a unit of behaviour for the component lifecycle that must be executed to demonstrate compliance. Activity is the modelling entity, which relates the contract with the process meta-model.

All these aspects are linked intertwined. Contract validation is necessary to take into account not only component properties, but also how properties have been assured. An argumentation meta-model captures these entities. Claims reference properties (which define the behaviour) and standards' objectives, and Information Elements reference the evidences used to support those claims.

### 3.1.3 Semi-formal notation of contracts

Contract-based design has shown many evidences of its applicability in industry. They have also been proposed for the functional safety domain (e.g., [38], [39], [40]). Although such contract-based developments have gained popularity as an approach for supporting distributed development by explicitly annotating assumptions and guarantees to components, an integrated process covering specification of the nominal behavior and safety was missing. In [42] such an integrated development approach is presented. The approach encompasses the systematic breakdown of the nominal system behavior using contracts, the consistent derivation of the safety analysis by interpreting several types of contract violations as a specification for failure modes, and the subsequent integration of safety mechanisms that cover these failure modes through safety contracts. There have also been proposals for the usage of contracts to specify real-time properties of continuous-valued controller structures and the control error of technical systems (e.g., [29]). Furthermore, contracts have also been applied to UML/SysML models as well as Simulink models (e.g., [30]).

Contracts can be specified in a natural language, in a set of semi-formal languages (such as template-languages), or in formal languages.

Formal languages (as temporal logics [34] or IO-Automata [35]) allow automatic verification of refinement and implementation of contracts, but they are often hard to understand for practitioners from the different involved disciplines and therefore difficult to promote in industry. A proposal that bridges this gap are the pattern-based Requirements Specification Language RSL [36] or the Contract Specification Language CSL from the SPEEDS project [37]. Text patterns, consisting of static text elements and attributes, provide both a well-defined syntax and semantics. To cope with the needs of the different aspects of a design, various sets of patterns have been defined [31], they build upon parametrized requirements patterns that have been known for a long time (e.g. [32], [33]).

Natural language contracts are often accompanied by ambiguity, incompleteness or inconsistency. Some proposals have been made with semi-formal languages (the syntax is defined and restricted, but verification has to be performed by human experts) to avoid these drawbacks of natural language while providing an understandable language for experts from different domains (e.g., [41]). In [41] a semi-formal approach is proposed, which allows specifying assumptions and guarantees at component interfaces in a language with well-defined syntax, but leaving the verification of fulfillment of the contract by a component to expert decision. However, the approach allows formalizing and automatically checking some relevant refinement relations. The paper presents a prototypical Eclipse tool (SAVONA), which allows the annotation of components with assumptions and guarantees, and the partial checking of the decomposition. It also shows its applicability based on an automotive electric drive system case study.

The OPENCOSS Common Certification Language (CCL) provides a Thesaurus-type vocabulary [161], which defines and records key concepts relevant to safety assurance within the target domains and the relationships between them. A further use case for the vocabulary aspects of OPENCOSS is to provide a means for regularizing the structure of expressions used in claims in assurance argumentation. This work has been written up in detail in [161].

Having a common syntactic structure for argument claims makes it easier for a reader to parse claims, and avoids issues such as confusion over the scope of a given term in a sentence. It is quite common in "engineering language" for there to be uncertainties over the interpretation of the scope of qualifiers, as in the phrase "failure modes and effect analysis", where "analysis" serves to qualify both "failure mode" and "effect" and where "failure" qualifies both "mode" and "effect". A non-specialist reader (or a machine) would be likely to look only for the most limited range, and associate "analysis" only with "effect" and "failure" only with "mode".

Structured expressions can be used to characterise the types of concepts that are discussed at a particular point in an argument, and the relevant features which can be asserted about them. Typically, a structured expression comprises a fixed verb phrase, which carries the main sense of the claim, while noun phrases, providing the subject and object over which the verb phrase ranges, are parameterisable. For example, a very simple structured expression in an assurance case claim might take the form "{fault of type systematic fault} is adequately mitigated by {fault mitigation technique}", where both "fault of type systematic fault" and "fault mitigation technique" are broad parameters. Simple expression structures can be combined to form larger syntactic units, e.g., form "{fault of type systematic fault} is adequately mitigated by {fault mitigation technique} which addresses {hazard}".

In the OPENCOSS approach defines a series of generic structures. They are used to refine the logical structures summarised in argument fragment templates captured in patterns such as those defined in [165] by specifying the types of concept which are in focus at particular points in the argument.

### 3.1.4 Safety Analysis with contracts

Fault-tree analysis is a safety technique applied to a system description in order to check the dependency of system properties on the occurrences of faults. In particular, given a system implementation, a set of faults events, and a top-level failure condition, the fault-tree analysis produces a tree structure that shows how the top-level condition depends on different sets of occurrences of fault events.

Contract-based design and fault-tree analysis can be integrated in order to improve both techniques. In particular, SafeCer applies safety analysis to the contract-based specification by extending the specification with failure conditions and analyzing when the contract refinement is preserved in case of failures. On the other side, the contract failures are exploited as intermediate events in order to enrich the fault-tree analysis. The integration is described in [14].

In a nominal architectural design, the one that is later implemented, the failures of components are not modelled explicitly. Failures may be artificially introduced in the model for the analysis of the safety mechanisms. However, the design that is later implemented in real software and hardware components contains only the nominal interfaces and behaviours. It may contain redundancy mechanism or failure monitoring, but not the failures themselves. Unfortunately, there is often a gap between the design of the nominal architecture and the safety analysis, which are carried out by different teams, possibly on out-of-sync components. This requires substantial effort, and it is often based on unclear semantics.
FBK proposed a new formal methodology to support a tight integration between the architectural design and the analysis of failures. The approach builds on two main ingredients: Contract-Based Design (CBD) and symbolic fault injection.

An important aspect of CBD is the ability to provide feedback in the early stages of the process, by specifying properties of blocks in abstract terms (e.g. in terms of temporal logic), without the need of a behavioural model (e.g. in terms of finite state machines).

The idea of (symbolic) fault injection, also referred to as model extension, is to transform a nominal model into one that encompasses the faults. This is done by introducing additional variables, describing whether a fault has occurred or not and controlling whether the system is behaving according to the original (nominal) or extended (faulty) model. Within this setting, it is possible to generate automatically fault trees by means of model checking techniques. This approach focuses on behavioural models, and is flat, thus unable to exploit the hierarchical decomposition of the system.

The novel contribution of the proposed approach is the extension of CBD for safety analysis: given a contract-based decomposition of the system under nominal conditions, we obtain automatically a contract-based decomposition of the model with fault injections. The insight is that the failure mode variables are directly extracted from the structure of the nominal description; basically, a failure variable models the failure of the component to satisfy its contract. The approach is proved to preserve the correctness of refinement: the extension of a correct refinement of nominal contracts yields an extended model where the refinements are still correct. Once the contracts are extended, it is possible to construct automatically hierarchical fault trees that mimic the structure of the decomposition, and formally characterize how lower level or environmental failures (to satisfy the respective contracts) may cause failures at higher levels.

The approach has several important features. First, it is fully automated since the models required to support the analysis of safety mechanisms are directly obtained from the models used in design, without the need of further human intervention. Second, the approach can be applied early in the development process and stepwise along the refinement of the design providing a tight connection between the design and the safety analysis. Third, it produces artefacts that are fundamental in safety assessment, in particular fault trees, which follow the hierarchical decomposition of the system architecture.

The framework has been implemented extending the OCRA tool, where the input is an OCRA description of the architecture with the specification of component contracts and the output of the new functionality is a hierarchical fault tree. An application of this framework has been used to formalize and analyse the AIR6110 Wheel Brake System.

Both the integration of nominal system development with failure analysis and the augmentation of failure analysis with fault injection, as elaborated in the SafeCer approach, can profit from the formalization of failure modes through the contracts: As the set of contracts specify the correct behaviour of a component in all relevant operation situations, it is obvious that a contract violation of a component (i.e. one of its guarantees is violated, although all of its assumptions towards the environment hold) constitutes a failure. Additional classification schemes for failure modes help automating the model-based failure analysis to a high degree (e.g. For ports delivering a continuous signal, the failure modes “too low” and “too high” are standard ones. If the guarantee says that a value shall be between 1 and 99 in a certain situation, then clearly 0 or everything below is a “too low” failure, while 100 and everything above is a “too high” failure). Details about the integration of contract-based development with safety analysis can be found in Section 4 of [42].

### 3.1.5 Patterns of contracts

The increasing complexity through highly integrated and advanced driver assistance systems challenges the safety and security demands on the different domains at industry. Patterns for safety and security contracts enable robust and fast design for such complex systems, while at the same time enabling the reuse of design/components across systems and domains. Nowadays we see a number of best practices in the industry for safety/security design patterns ranging from high-level system architectures down to idioms for the design of components and basic parts. Examples include redundancy (dual channels, 1oo2, 2oo3, lockstep, etc.), online failure monitoring (3-level-monitoring, watchdogs, EDCs, intrusion detection, etc.), or communication and data protection (E2E, checksums, error bits, partitioning), and so on.

At the core, safety (and security) contracts define how fault avoidance, fault detection and recovery and/or mitigation is handled by a single element or between a set of elements. For this purpose, the contract must include a statement about potential failures of the element(s) and mechanisms how they are covered. Therefore, safety contracts are often regarded as an extension to an element's interface with a set of specific requires/guarantees properties, and there exists a number of proposals on how they can be expressed and formalized (cp. Section 3.1.3). However, as state of the art there is almost no formalization and explicit modelling of these patterns/contracts applied during the development process on the different domains at industry.

The key points to enable safety and security contracts are the following:

1. A formalization of the design is required, where the nominal specification of a hierarchically decomposed system can be augmented with relevant information for safety/security contracts. This information includes the formalization of the contracts by means of interfaces as well as the addition of failures, mechanisms, constraints, context/environment dependencies, etc. to the design.
2. A formal definition of failures, failure dependencies, and safety mechanisms as well as their detailed semantics in relation to the model. This implies the element behaviour in case of fault occurrence and the characteristics of the countermeasures, e.g. detection capabilities with a certain diagnostic coverage, timing aspects, avoidance/protection against the fault, and so on.
3. A method to define and exchange patterns/contracts and to build libraries, so that actual reuse can be accomplished across systems. Modelling wise there is the need to have a technology that allows the definition of contracts and exchange of components that encompass the safety/security contracts (e.g. data or exchange format).

Model-driven design and development defines the state of the art addressing these points and providing a solid and sound foundation for the required extensions. Section 3.5 gives an overview on typically used languages such as UML and SysML. Thereby, the type and interface concepts are key to a contractual design, since ports and interfaces make up the interactions/connections of a structural element.

The usage of *metamodels* for the modelling languages constitute a precise and concise technique to define the information from a structural point of view. The definition of the safety contract related information – such as failures, dependencies, or safety mechanisms – can be achieved by extending the language metamodels with the safety/security related concepts. An example of such a metamodel is the safety core part of medini$^{TM}$ analyze as shown in the figure below.

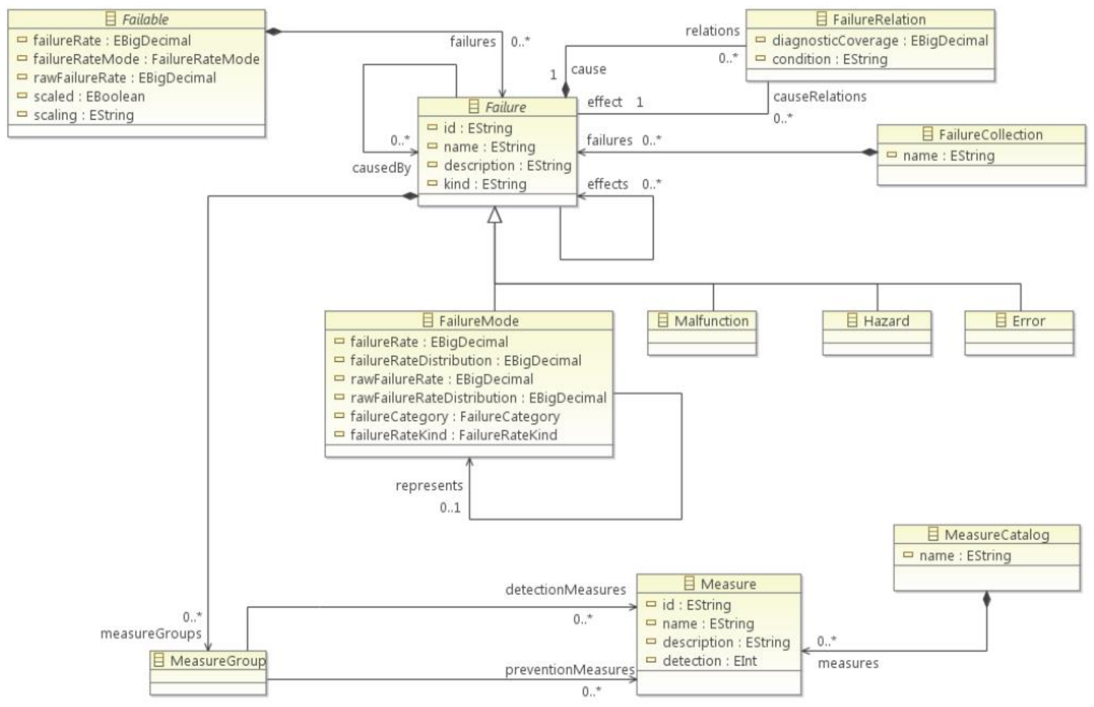


**Figure 4: Arguments over the use of enabling monitor**

The safety core model defines the modelling concepts of *failures* and *measures* as a central metamodel that is shared across all language models such as SysML, UML, AUTOSAR, Simulink, etc. We use the term "failure" as a general classification across domains of all abnormal conditions that are in the focus of an analysis, i.e. Hazard Analysis and Risk Assessment, FMEA, FMEDA, or FTA. Failures can be connected via a cause-effect relationship to model *failure nets* (usually hierarchies). In structural/physical models, they're quantified by means of a failure rate (e.g. from computations along the SN29500, IEC62380 or other sources). *Measures* are the generic concept to express all means to prevent, detect, control, mitigate or correct failures of a system. In addition, *Safety Mechanisms* are specific measures that are implemented into systems and that provide a *diagnostic coverage* (DC) of failures, i.e. they cover a proportion of the failure rate of a failure mode (not shown in the excerpt). These definitions are intended to be as generic as possible to fit to multiple domains (e.g. ISO 26262, IEC 61508, DO 178, etc.) and imply only a few limitations in their usage regarding the semantics of the underlying model. For example, malfunctions are contained only in behavioural elements (activities/actions/operations) and functions, failure modes are contained in all components or parts, errors are dedicated to software/logical blocks, and so on.

Based on these state of practice metamodels, the definition of a safety contract is modelled by an extended type definition where the type itself and its ports receive annotated failure modes/malfunctions, failure cause/effect relations, and safety mechanisms that address the failures. Thereby, the ports of a type as well as the type itself maintain a list of potential failures of its instances. Consequently, the failure (modes) of all instances of a type/ports are synchronized with the type definition, but the specific cause-effect relations are specific to the instance, since they often depend on location and connections of a concrete part. Similarly, the failure rates and distributions are modelled at the type and synchronized with all instances, which define context specific stress parameters based on a given mission profile (e.g. temperature, voltage/current, mechanical stress, and so on).

Given a set of parts, failures/failure modes, and failure rates, the safety mechanisms model the fault detection and control at the instance level and are linked to the failures of the instances. The safety mechanisms of a system usually come with a maximum Detection Coverage (DC), which might not always be achieved for all instances, so some flexibility is required to adapt the DC per instance. Hence, it is important to note that many safety properties – failure rates and diagnostic coverage values – vary depending on the application context (system) and mission profiles (environment). Therefore, the designer of a safety contract (usually the safety engineer) must review whether all assumed conditions are met and adapt the design or analysis based on specific product constraints.

In order to enable reuse of contracts and patterns, all the safety design information must be stored in some sort of database of library. (Element) Libraries are used to share definitions of re-usable elements of system models across project boundaries and provide the basis to "instantiate" patterns for a concrete system design. The usage of libraries is only partly established and this topic is subject of further research, especially in the context of WP6.

### 3.1.6 Generation of Safety Case Argument-Fragments from Contracts

In the context of SYNOPSIS project [162], component contracts were used to (semi)-automatically generate safety-case argument fragments [142]. SYNOPSIS assumes that the component contracts and the safety requirements allocated to the same component are closely related, but not the same. On the one hand, safety requirements describe behaviours that a certain context/system requires from a component. On the other hand, the contracts represent the actual behaviours of the component, which can be used to check whether the component satisfies the allocated requirements by checking the component guarantees with the corresponding safety requirements.

The safety case is represented with an argument that connects the requirements with the supporting evidence. Such argument should demonstrate how the specified safety requirements have been satisfied in a particular context. Since both the contracts and the argument serve the same purpose to show that the requirements are satisfied, the existing contracts could be used to speed up the creation of the corresponding arguments. A SEooCMM metamodel [142], see Figure 5, is defined as an extension of the SafeCer generic component metamodel, to capture the needed information around the notion of contracts and enable the generation of argument-fragments directly from such contracts. Since SEooCMM addresses the out-of-context setting it deals with strong and weak contracts. Regardless of that, the requirements and evidence in SEooCMM are related to the abstract safety contract class to allow for extensions of the metamodel to include different types of contracts.

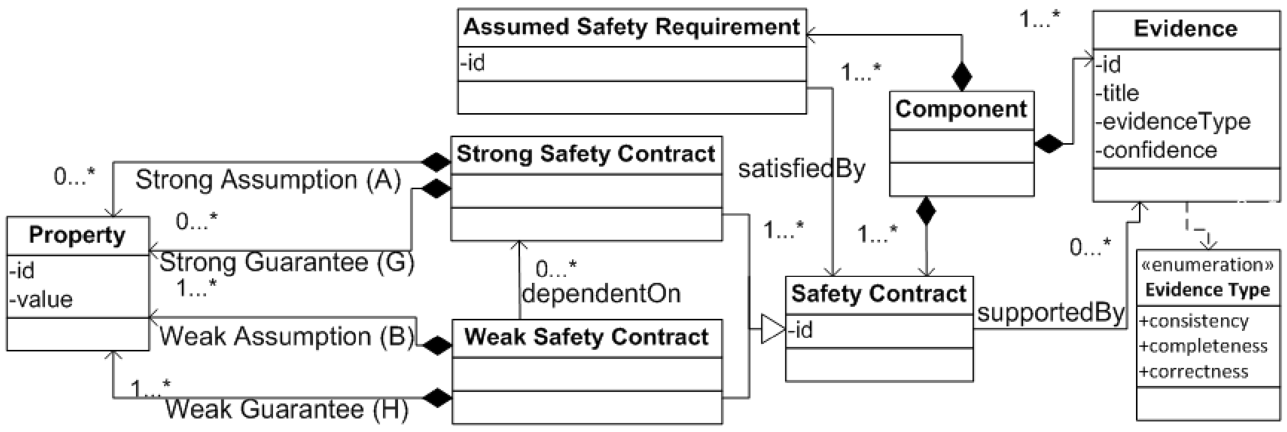


**Figure 5: Safety Element out-of-context Metamodel (SEooCMM)**

To generate the argument-fragment from SEooCMM, we map the contract guarantees with the argument claims, the supporting evidence with solutions, and the contract assumptions with claims and contexts. Based on this mapping, the contract satisfaction argumentation pattern (Figure 6) is used to generate an argument-fragment that a particular component contract is satisfied with sufficient confidence. A set of such argument-fragments is used to support the satisfaction of a safety requirement allocated to the component.

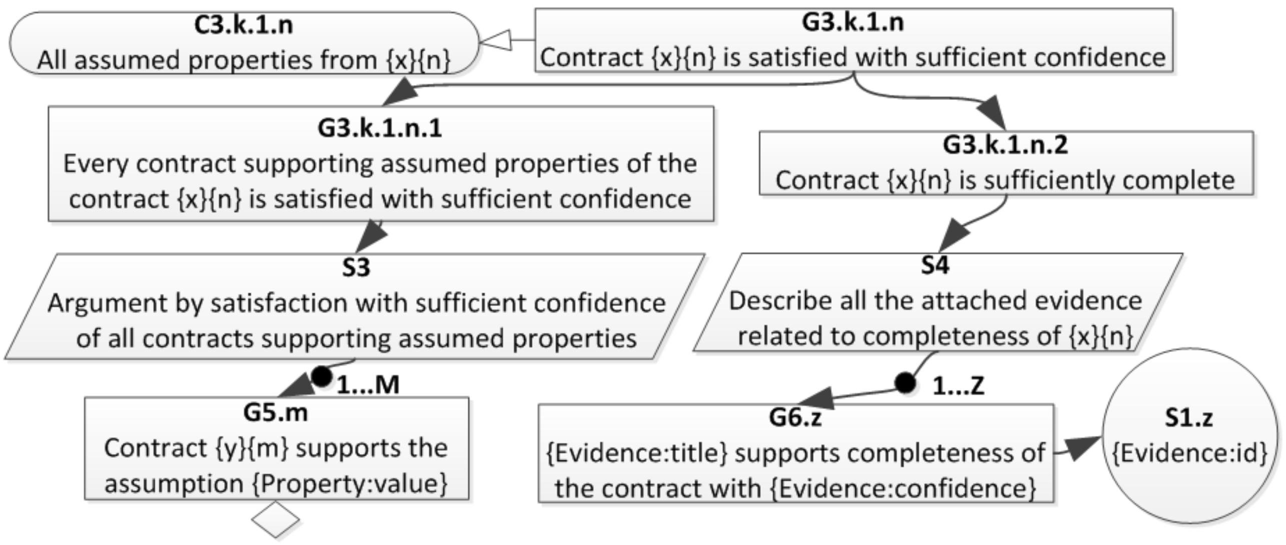

C3.k.1.n
All assumed properties from {x}{n}
G3.k.1.n
Contract {x}{n} is satisfied with sufficient confidence
G3.k.1.n.1
Every contract supporting assumed properties of the contract {x}{n} is satisfied with sufficient confidence
G3.k.1.n.2
Contract {x}{n} is sufficiently complete
S3
Argument by satisfaction with sufficient confidence of all contracts supporting assumed properties
S4
Describe all the attached evidence related to completeness of {x}{n}
1...M
1...Z
G5.m
Contract {y}{m} supports the assumption {Property:value}
G6.z
{Evidence:title} supports completeness of the contract with {Evidence:confidence}
S1.z
{Evidence:id}

**Figure 6: Contract satisfaction argument pattern [142]**

The contract satisfaction argument-pattern starts with a claim that a contract is satisfied with sufficient confidence, which means that the contract guarantees are offered. To support such claim, the pattern argues over the satisfaction of the contract assumptions, and the confidence in sufficient completeness of the contract. While the argument over each assumption points to the contracts in the environment that satisfy that assumption, the contract completeness sub-argument should establish sufficient confidence that the specified contract assumptions are sufficient to claim that the guarantee is offered.

Based on the potential for generation of contract-based argument-fragments from SEooCMM, the metamodel and the contract-based argument-fragments can be used to instantiate different pre-established argumentation patterns such as Handling of Software Failure Modes argument pattern [143]. Furthermore, to facilitate better evidence management in SEooCMM, an extension is proposed in [144] where SEooCMM is aligned with the standardised SACM evidence metamodel.

## 3.2 Requirement Specification

In this section we review diverse requirement specification approaches related to subsection 2.5 “V&V-based Assurance”.

### 3.2.1 Domain Ontology Authoring

The system engineers need to have considerable knowledge and experience in the domain in order to define the system requirements and design the system architecture. They also need to have clear vision about the ultimate result of the development effort that will raise from the implementation of their system architecture and requirements. This section provides yet another example, how the system architecture and requirement authoring activities can be based on an ontology and what benefits that brings (e.g. the ontology can be used as a unified consistent language).

**Ontology as a language**

The ontology is perceived as a kind of specification language, which offers the user:

1. A list of textual expressions (names of types/sorts/classes of things, names of individual things/values/parameters/constants, names of processes, names of relations, possibly

supplemented with corresponding definitions of these concepts and bound to examples of contexts in which they occur) tightly related to (stemming from) the application domain. Such list of symbols is sometimes called the *signature*. The signature helps to suppress the ambiguity of the text, e.g. by limiting the usage of several synonyms for the same thing, which is okay when stylistic issues are important, but which is undesirable from an engineering point of view. The symbols of the signature together with the symbols of the logical operations like conjunction, negation, etc. of the chosen logic represent the constituent blocks of formal system specifications.

2. The possible compositions of the lower-level expressions into higher-level expressions and even into the whole sentences. These potential compositions are inscribed in the diagrammatic structure of the underlying ontology captured in the UML. When the sentences are created, the user traverses appropriate continuous paths in the UML diagrams from one concept (class) to another concept via the existing connections (associations, generalizations, aggregations) and composes the names of classes and relations encountered along the way into a sentence.

**Ontology employment**

The basic structure of the process to apply ontology in the system architecture creation and requirement authoring and formalization can be summarized in the following steps:

1. Create the (UML) ontology.
2. Write a (tentative, sketchy) informal system architecture and requirement.
3. For the most important notions of the informal system architecture and requirement find the corresponding terms in the ontology.
4. Select the most appropriate paths in the ontology graph, which connect/include the important notions found in the previous step.
5. Compose meaningful sentences by concatenating the names of the elements (classes, objects, relations) traversed along the selected paths.
6. Repeat the steps 2 – 5 for all informal artefacts.

This approach is also applicable when the goal is to improve the quality of the current system architecture or requirements and rewrite them in a more clear a consistent form. In that case, the step 2. does not involve writing of new parts of system architecture or requirements, but taking the existing artefacts from their last iteration.

If we consider an extreme but desirable case that a new system's architecture is just a composition of reused components that have been developed separately around different ontologies, it is obvious that the ontology of the new and more complex system should be some composition of the simpler ontologies of its subsystems. Therefore, for the development of Cyber-Physical Systems it is highly desirable to have appropriate means of gluing ontologies together to obtain ontologies that are more complex.

### 3.2.2 Requirement Grammars Authoring

Both requirement authoring supported with domain ontology and requirements formalization increases the quality of requirements and improves the capacity later to verify compliance to these requirements. In the case of requirements formalization, the benefits that this process brings are automatic formal verification, guaranteed verifiability, and the removal of ambiguity among requirements.

In some aerospace domains, e.g. Flight Controls, Flight Management Systems, Display and Graphics, the Honeywell requirements are written in a structured and restricted way to improve their quality. Yet these restrictions are not sufficient to guarantee machine readability and the subsequent automatic verification. The requirements language needs to be further restricted to be unambiguous and to have clear semantics, before a machine could read such. Honeywell internal tool ForReq [57] allows requirement authoring based on a grammar for structured English requirements that serves two separate purposes. For the requirements already written that conform to this grammar, ForReq allows automatic translation into Linear Temporal Logic (LTL) and thus automatic verification. Yet, more importantly, the machine readability can be enforced for new requirements by the use of *auto-completion*. This new functionality suggests the requirements engineer the set of possible words to continue their requirements within the boundaries of the restricted requirements language. Thus, the requirements engineers save effort that would be needed for writing twice each of the requirements, i.e. the human readable version for stakeholders and the machine-readable version for verification.

In the case requirements do not use exact artefacts (variables or states) from the system (for example some system requirement are prohibited to contain such link), the ForReq tool now guides the user to create mapping from artefacts in requirements to the corresponding artefacts in the system. Moreover, requirements defining mapping between variable names and its textual descriptions used in requirements are supported to automate fully the process. These requirements are also verified and any inconsistency is reported to be fixed by the user.

However, the user has to specify the exact timing of each requirement, i.e. whether the effects shall happen immediately or in the next time step or after specified number of time steps or seconds. Honeywell ForReq tool supports this requirement formalization process as depicted in Figure 7 in order to enable automatic semantic requirement analysis as described in Section 3.7.1 and automated formal verification against system design as described in Section 3.7.3.

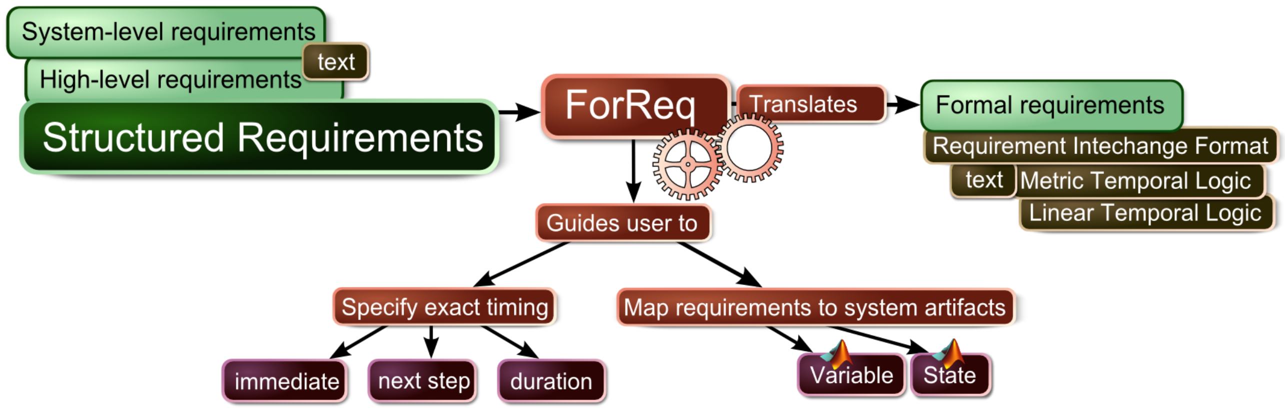


**Figure 7: Process of formalization of structured requirements using ForReq tool**

The requirements formalization is not a straightforward process and a considerable number of steps is required for incorporating formalization into real-world development of embedded systems. The goal of the ForReq development is to guarantee that the authored requirements are unambiguous, automatically verifiable (machine-readable) and conforming to the requirements reference (template, pattern, boilerplate, standard). Auto-completion, requirement standard grammar and requirement guidelines were implemented in ForReq to cover this need and to proceed further towards fully incorporating requirements formalization into the development process.

### 3.2.3 Requirements-Based Engineering Approach

The Requirements-Based Engineering (RBE) approach was developed by The REUSE Company, Carlos III University of Madrid and OFFIS in the CRYSTAL project. It provides an answer to the use cases initially drawn by SAGEM and CASSIDIAN as well as other use cases that stated their interest on this RBE approach during the project. It relates to the objective "V&V-based Assurance", as it provides automatic V&V techniques, e.g., validate that a requirements specification is complete, correct, and unambiguous.

The main goals of this RBE approach can be summarized as follows:

- Improve the quality of requirements specifications, both at individual level for each individual requirement, but also at a global level for requirements documents or even a complete specification
- Leverage the role of Requirements Author as a key element for requirements quality improvement, and not only the role of quality control
- Application of knowledge management techniques as the enabler of the aforementioned goals

The RBE approach envisaged on CRYSTAL was, in turn, based on the results of the previous ARTEMIS CESAR Project that can be described as:

- CCC (Correctness, Consistency and Completeness) approach for requirements quality checking: a requirements analysis focus on three main dimensions, Correctness, Consistency and Completeness
- Boilerplate approach: to represent the structure (grammar) of different types of requirements that, once agreed in the organization, can be used by requirements authors and checked by the quality teams

Using this background from CESAR, the RBE approach described in CRYSTAL was based on the following principles (internally called Knowledge-Centric Approach):

- A Knowledge Repository known as System Knowledge Repository (SKR)
- A Knowledge Model also known as System Knowledge Base
- A set of Boilerplates and Patterns
- A Semantic Indexing process
- The CCC approach for requirements quality checking

Aside of these principles, the final concern was the proper definition of the interoperability structure needed to operate all the elements of the proposer architecture. This interoperability layer was based on OSLC (Open Services for Lifecycle Collaboration). This interoperability layer was based on OSLC. The OSLC initiative [170] is a joint effort between academia and industry to boost data sharing and interoperability among applications applying the Linked Data principles [171]. OSLC is based on a set of specifications that take advantage of web-based standards such as the Resource Description Framework (RDF) [172]] and the Hypertext Transfer Protocol (HTTP) to share data under a common a data model (RDF) and protocol (HTTP). Every OSLC specification defines a shape for a particular type of resource. For instance, requirements, changes, test cases, models (the OSLC-MBSE[1] specification Model-Based Systems Engineering by the Object Management Group) or estimation and measurement metrics to name a few have already a defined shape (also called OSLC Resource Shape). Currently, OSLC[2] is part of the OASIS Open Standards Network. More specifically, OSLC Lifecycle Integration Core (OSLC Core), OSLC Lifecycle Integration for Automation (OSLC Automation), OSLC Lifecycle Integration for Change and Configuration Management (OSLC CCM) and OSLC Lifecycle Integration for Project Management of Contracted Delivery (OSLC PROMCODE) are the technical committees working on OSLC specifications at OASIS. Finally, OSLC Core version 3.0 is under public review[3] (28th September, 2016).

The following subsections describe, one by one, all the aforementioned principles, as well as the OSLC domains developed to cope with the initial goals.

[1] http://www.omgwiki.org/OMGSysML/doku.php?id=sysml-oslc:oslc4mbse_working_group

[2] http://www.oasis-oslc.org/

[3] https://www.oasis-open.org/news/announcements/30-day-public-review-for-oslc-core-version-3-0-ends-sept-28th

#### 3.2.3.1 System Knowledge Repository

A System Knowledge Repository (SKR) is where all the information related to improving the quality of requirements specifications is to be stored. Technically speaking, the SKR depicted in CRYSTAL is not including the storage for the requirements themselves, since different approaches for that exist. However, it could be extended with additional room for requirements to provide a more self-contained representation.

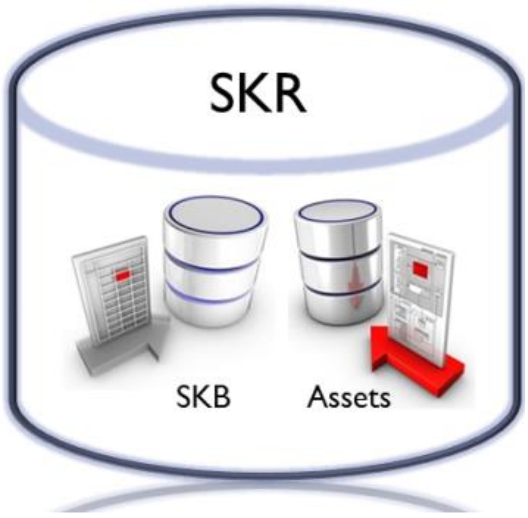


**Figure 8: System Knowledge Repository**

Globally speaking, a System Knowledge Repository is made up of the following elements:

- A System Knowledge Base (SKB): where information about a specific business domain is stored. More information about this is provided in the following section.
- A System Assets Store (SAS): where the requirements are stored. This store might include the textual definition of the requirements (as well as other attributes commonly managed in a Requirements Management Tool) but what is a key element of this SAS is the formal representation distilled from each requirement through the application of boilerplates and patterns.

#### 3.2.3.2 System Knowledge Base

A System Knowledge Base (SKB) is the formal representation of the knowledge about a specific business domain. Based on the content of a classical ontology, but with the aim to apply these knowledge bases for the enhancement of the quality of requirements specifications, the knowledge bases here described includes the following layers, which are funded one on top of the other:

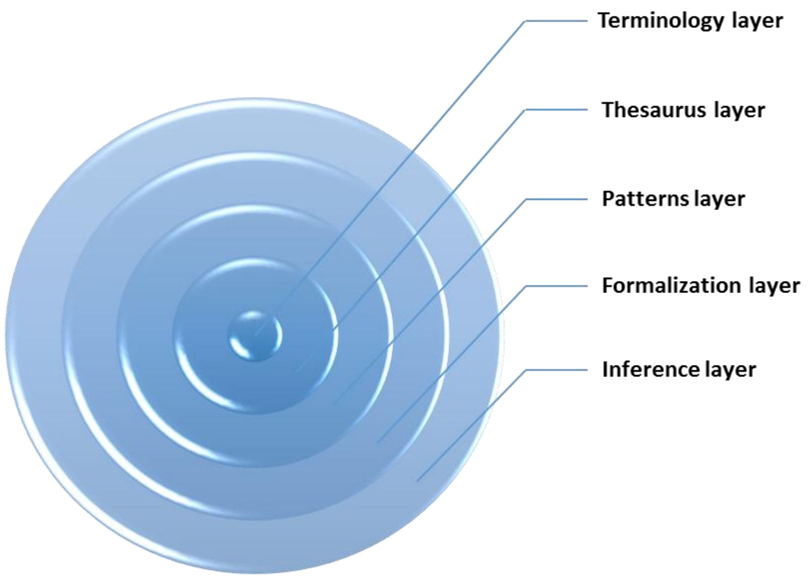


**Figure 9: System Knowledge Base**

More detailed, these layers, which are funded one on top of the other, contains the following information:

- **Terminology layer:** where the specific terms describing the domain specific concepts are stored. The dictionaries described here includes not only domain specific terms, but also some other common terms such as determiners, prepositions… Finally, it also includes some terms and expressions to be avoided (detected as previous step for avoiding) in the requirements (e.g. maximise, minimise, user-friendly…)
- **Thesaurus layer:** where the business concepts from the previous layer are better classified/clustered and linked together to provide even more semantics about the modelled domain. Examples of such semantic clusters could be: <system>, <system element>, <stakeholder>, <state>…, while examples of different types of relationships among concepts may include: synonymy, association, dependency, and aggregation.
- **Pattern layer:** described in more detail in the following section (section 3.2.3.3), they represent the set of agreed upon grammars (structures) that the requirements should conform to
- **Formalization layer:** describes the semi-formal transformation that a textual requirement follows once it matches with one of the aforementioned patterns. This transformation follows the steps described in section 3.2.3.4, known as the semantic indexing
- **Inference layer:** unlike other more classical ontologies, the ones used in this approach are not based on axioms and rules, but rather on a set of CCC rules (see section 3.2.3.5)

#### 3.2.3.3 Boilerplates and Patterns

An easy way to promote consistent requirements in large specifications is by following a set of agreed-upon grammars (boilerplates/patterns), together with the consistent vocabulary already described in section 3.2.3.2.

In order to do so, a taxonomy of different types of requirements shall be described. Later, for each different type of requirement one or more patterns is described. Those patterns are made up of a sequential list of items (slots), where each item may include:

- A specific term: e.g. *The…*
- A Part-of-speech item (or grammatical category): e.g. NOUN, VERB, …
- A semantic cluster: e.g. <System_Element>, <User>…
- A combination of the last two items

- Another pattern: a.k.a. sub-pattern, they provide the recursively structure of the language

Figure 10 depicts an example of pattern.

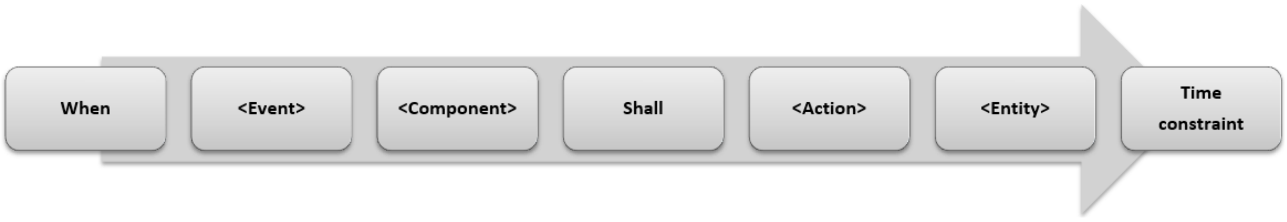


**Figure 10: Example of boilerplates/pattern**

#### 3.2.3.4 Semantic indexing and retrieval processes

The semantic indexing process is represented as a sequence of steps (following a pipeline architecture) that allows a textual requirement to be first matched with any of the agreed patterns, then formalized based on a formal representation and lastly analysed for CCC (Correctness, Consistency and Completeness).

The steps followed in this process are the following:

- **Tokenization:** where the individual components of a sentence (requirement) are divided, first based on basic approach using a set of word separators, but lately using semantic meaningful tokens
- **Normalization:** where some of the semantic tokens are reduced into a canonical form (masculine/singular for nouns, infinitive for verb forms…)
- **Disambiguation:** the most challenging step in this indexing process, where one suitable form shall be selected in those cases where more than one concept exits the previous step. One example of the need for disambiguation is the typical example "Time flies like an arrow; fruit flies like a banana", where some of the words can play different roles according to the structure and meaning of the sentence
- **Pattern matching:** where each requirement is hopefully matched with one of the agreed upon patterns in the SKB
- **Formalization:** following the pattern matching, the textual requirement is transformed into a semantic graph plus a set of metaproperties. This formal structure is the basis of some of the CCC quality rules, as well as the base for semantic retrieval/reuse of requirements

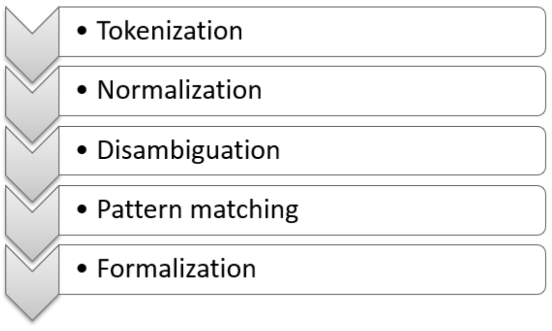


**Figure 11: Indexing process**

Figure 12 represents an example on how a textual requirement is first matched with one of the existing patterns, and then transformed into its formal representation (a semantic graph).

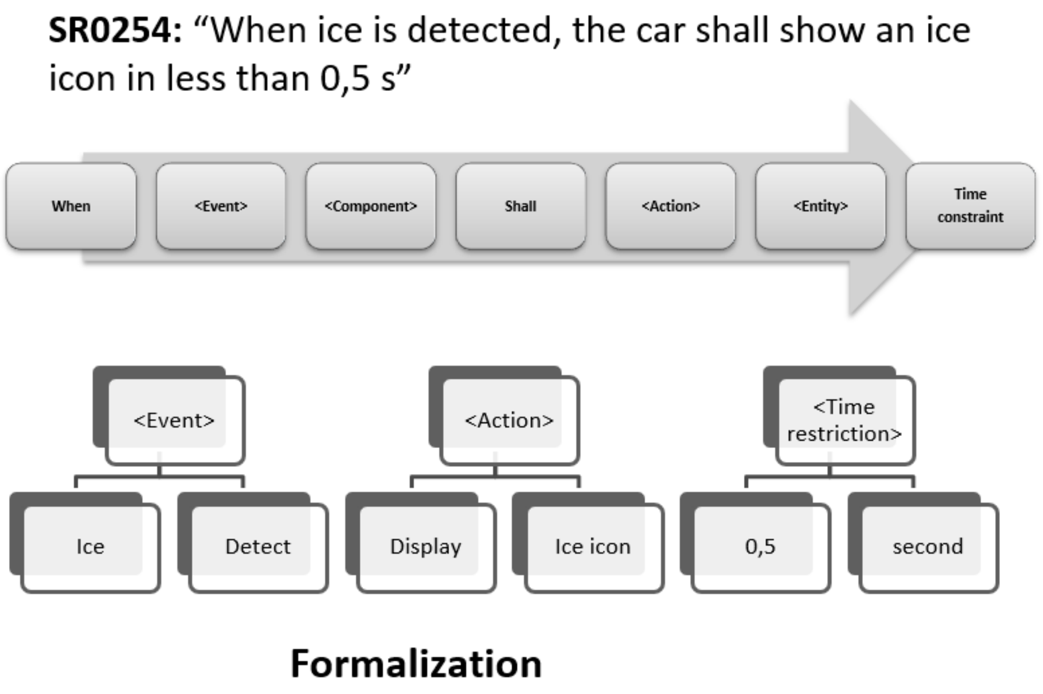


Figure 12: Formalisation requirements

The semantic transformation is not only the base of the quality assessment approach (the CCC approach), but also the basis for the semantic search approach that, based on the formal representation of a requirement, allows more retrieval accuracy as a previous step for different techniques such as:

- Identification of duplicated/overlapped requirements
- Easy identification of related requirements
- Detection of missing links
- Identification of similar in previous projects as a requirement reuse mechanism

Figure 13 shows an example where, based on the formal representation of the two requirements, and the information that may be contained in the System Knowledge Base (synonyms, linked concepts…), two requirements that could be apparently not similar are identified as fully equivalents from a semantic point of view.

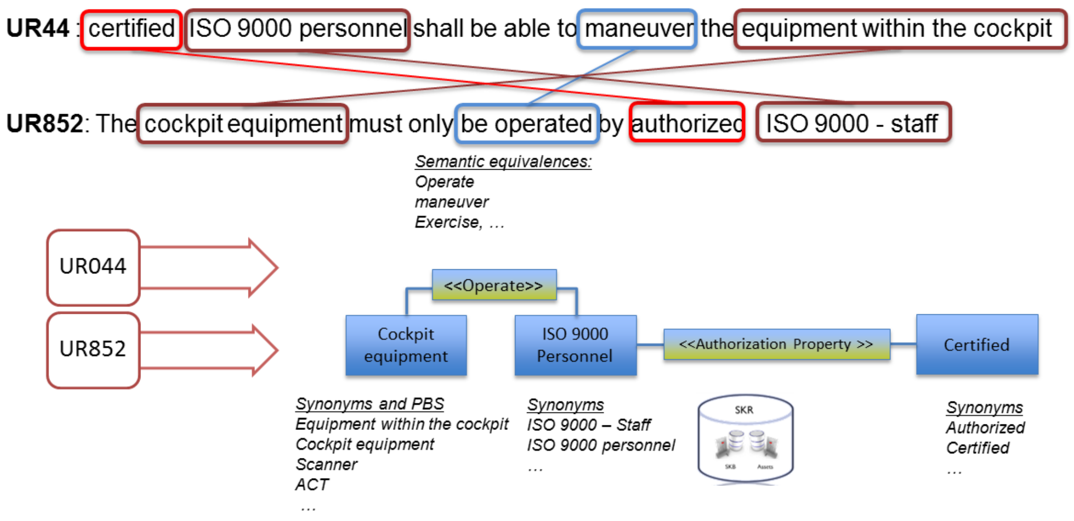


Figure 13: Semantic search engine

#### 3.2.3.5 CCC Approach

This approach tackles the need of assessing the level of quality of a requirements specification based on three main dimensions: Correctness for individual requirements, Consistency and Completeness for sets of requirements. It follows the set of quality metrics already described in CESAR, together with rules described in other standards and guides such as IEEE Std 830-1998, ISO/IEC 29148, and the INCOSE Guide for Writing Requirements.

The RBE approach in CRYSTAL concluded that any of the three aforementioned quality dimensions could be tackled by 3 different types of metrics (quality rules):

- Rules based on fixed algorithms: detection of passive voice, text length…
- Parameterisable rules: where the rule depends of a parameter provided when the quality baselines are being tailored
- Custom-code: allowing end-users of the tools designed to provide their own algorithm to cope with a new rule/metric

Thus, these three dimensions include, among many others, rules such as:

- Correctness:
  - Metrics based on information coming from the RMS:
    - Attributes, links, versions…
  - Metrics based on patterns:
    - Compliance with different types of requirements patterns
  - Detection of specific structures within the requirements
    - Metrics based on the conformance with models:
    - Concepts in your requirements coming from PBS, FBS…
  - Metrics based on linguistic algorithms:
    - Detection of passive voice, imperative tense…
    - Text length, misspelling….
  - Metrics based on lists of terms:
    - Forbidden: ambiguous…
    - Restricted: negations, pronouns…
    - Mandatory: 'shall'
- Consistency: answering the following questions:
  - Are all the expected requirements types involved in your specifications?
  - Are all the key concepts (from the ontology or from other models) properly covered?
  - Are your requirements properly linked? At the different levels?
- Completeness: answering the following questions:
  - Are your requirements consistent with each other?
  - Are your requirements consistent with the models of your projects?
  - Do you have duplicated requirements in your specifications?
  - Are you using the proper measurement units in your requirements?

#### 3.2.3.6 Requirements Interoperability

Interoperability was considered in the whole CRYSTAL Project as a key success factor. Thus, and relying on OSLC the project depicted a complete set of OSLC domains that, based on the *Client* and *Provider* paradigm, allowed the full interoperability scenario, not only for requirements, but also for knowledge items, quality rules and quality results (KPIs).

Figure 14 depicts the whole OSLC schema as implemented for requirements-based engineering. The OSLC domains involved are:

- OSLC for Requirements Management: as consumer of the requirements managed and stored in a requirements management repository

- OSLC for Knowledge Management: designed during the CRYSTAL project in order to exchange semantic information from the Knowledge Base
- OSLC for System KPI: designed during the CRYSTAL project as the way to exchange information regarding with quality metrics and quality assessment results

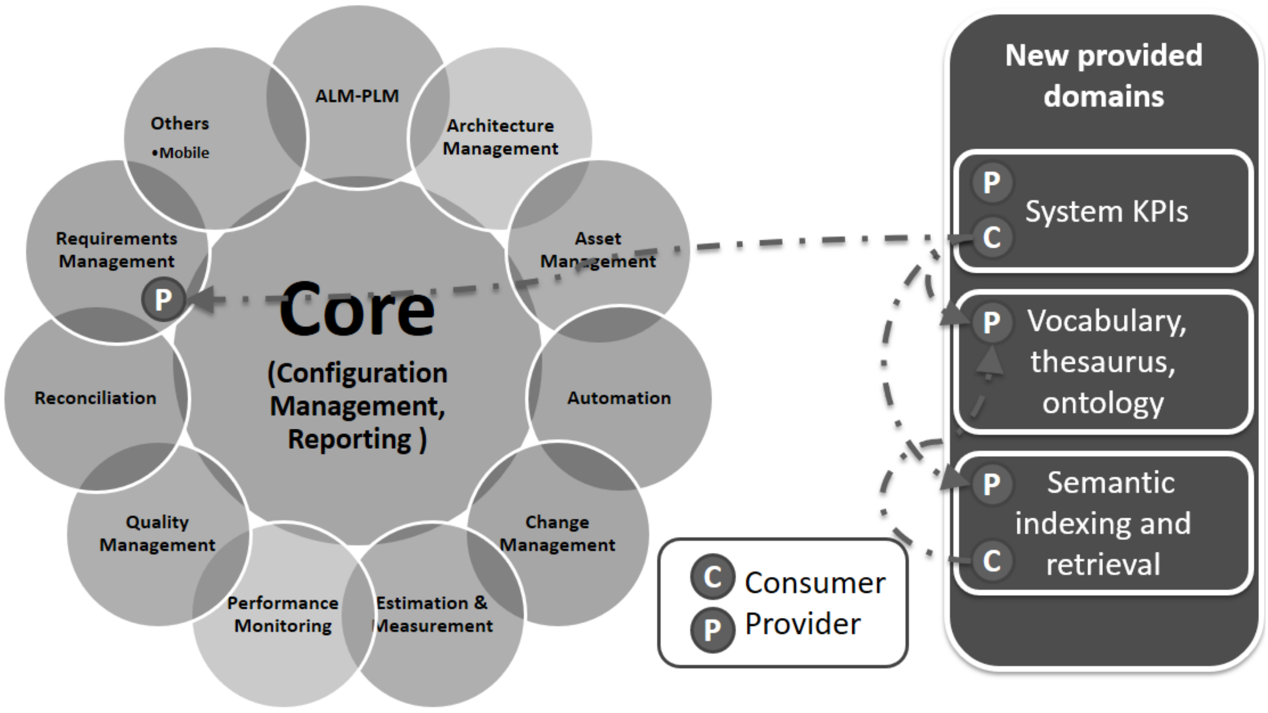


**Figure 14: OSCL domains**

### *3.2.3.6.1 Tooling*

Finally, all these approaches and techniques were implemented in a set of tools, the Requirements Quality Suite (by The REUSE Company, http://www.reusecompany.com), that includes the following tools:

- RQA – Requirements Quality Analyzer:
  - Customizes RQS with your own quality policies and checklist
  - Checks the correctness of your requirements specification
  - CCC: Correctness, Consistency and Completeness Analysis

- RAT – Requirements Authoring Tool:
  - Write your requirements easily by using an assistant: pattern based
  - Correctness analysis on the fly
  - Consistency analysis on the fly

- KM – Knowledge Manager:
  - Management of all the domain knowledge behind the quality analysis
  - Management of glossaries, taxonomies, thesauri and ontologies
  - Management of requirements Patterns to be used by RQA and RAT

## 3.3 Pattern-based approaches

In this section we review assurance patterns for argumentation and for compliance with standards related to the objective “Assurance Patterns Library Management” introduced in Section 2.

Christopher Alexander proposed the idea of design patterns for the first time in order to describe and document recurring solutions for design problems [91]. Accordingly, a design pattern is a description of a set of successful solutions of a recurring problem within a context. A general design problem that occurs repeatedly in many applications can be solved by this approach and it is usually made of three pillars: a problem, a context and a solution [92].

An important issue to address in the design of safety-critical embedded systems is the integration of implications on non-functional properties in the existing design pattern concept since non-functional requirements are an important aspect in the design of such systems. Armoush [93] proposed a pattern representation for the design of these systems by including fields for the implications and side effects of the represented design pattern on the non-functional requirements (safety, reliability, modifiability, cost and execution time) of the systems. The benefits of their applicability are quite clear: design quality, avoidance of introducing systematic faults in the design, flexibility and productivity can be fully exploited. They can be used either in hardware ([94], [95], [96], [97]) or software development to improve their corresponding development processes. Together with the aforementioned benefits, they are used to support and help designers and system architects to choose a suitable solution for a recurring design problem among available collection of successful solutions. Moreover, design patterns can simplify communication between practitioners and provide a good way for documentation of proven design techniques [98].

Concerning fault tolerance systems, design patterns can also be applied to allow modelling fault tolerance tactics with reuseable aspects by implementing them in a well-known architecture pattern [99]. Harrison and Avgeriou carry out an empirical study to investigate regarding the importance of having information about the effects on the architectural design pattern of fault tolerance tactics. In order to allow the designers to select among the best fault tolerance tactics, the impact of integrating them into architectural patterns must be available from the beginning. In [100] a model driven framework is created to be used in the development and testing of fault tolerance systems.

[101] depicts different types of architectural design patterns concerning fault tolerance and can be implemented either in hardware or software. Table 1 represents some of the most remarkable ones, which were the basis for developing the design pattern catalog implemented in a tool [102] to the patterns that are appropriate for the real-time applications based on its design problems:

**Table 1: Hardware and Software fault tolerance patterns**

| Hardware Patterns | Software Patterns |
|---|---|
| Triple Modular Redundancy Pattern | N-Version Programming Pattern |
| Homogeneous Redundancy Pattern | Recovery Block Pattern |
| Heterogeneous Redundancy Pattern | Acceptance Voting Pattern |
| M-Out-Of-N Pattern | N-Self Checking Programming Pattern |
| Monitor-Actuator Pattern | Recovery Block with Backup Voting Pattern |
| Sanity Check Pattern | |
| Watchdog Pattern | |
| Safety Executive Pattern | |

Likewise, we believe that there is a need of establishing certain design suggestions not only based on the previous concepts but also to achieve the need safety integrity level required by functional safety standards. Consequently, this source of information can be applied when trying to decide about certain architectures or safety mechanisms.

One project addressing the preceding issues is the so-called SafeAdapt [103] It investigated in the area of modelling design patterns and defined a UML package that captures the problem, the solution and the applicability in textual form (stereotyped UML comments) [98]. These design patterns help to define configurations automatically as they determine a set of replicas with an allocation constraint eventually by means of a constraint solver. In the deliverable D4.1 of SafeAdapt [98], the model of a passive fault-tolerance pattern with diversified replicas is shown in:

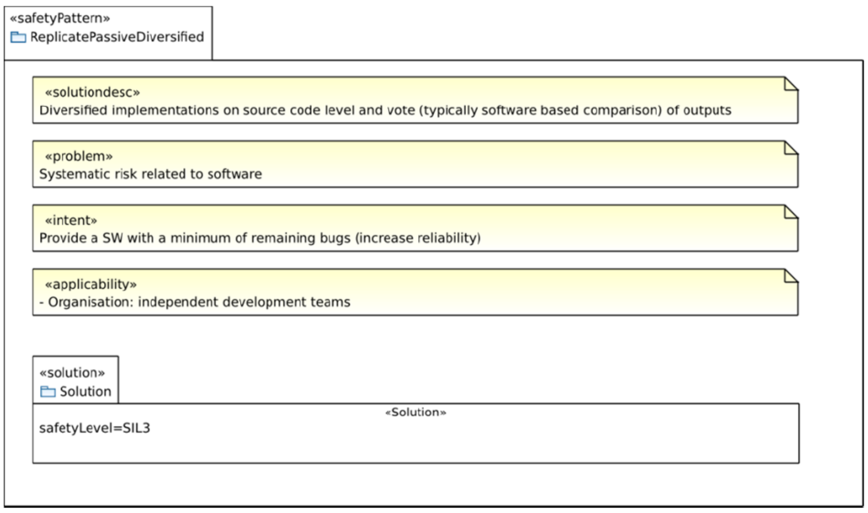

«safetyPattern»
ReplicatePassiveDiversified
«solutiondesc»
Diversified implementations on source code level and vote (typically software based comparison) of outputs
«problem»
Systematic risk related to software
«intent»
Provide a SW with a minimum of remaining bugs (increase reliability)
«applicability»
- Organisation: independent development teams
«solution»
Solution
«Solution»
safetyLevel=SIL3

**Figure 15: Model of passive fault-tolerance pattern with diversification**

Along this same line, the concept of auto-generating platform models and configurations based on those patterns for AUTOSAR is tackled.

To be more precise, the AUTOSAR gateway tool exports AUTOSAR from UML-EAST-ADL models (Papyrus add-on) [108]. Thus, it provides an enhanced transformation from EASTADL2 design architecture to AUTOSAR vehicle architecture design and initial system configuration. Some predefined strategies are applied to generate the preliminary AUTOSAR model. Based on EAST-ADL2 hardware platform specification, a hardware platform according to AUTOSAR concept is derived and runnables are grouped into software components, which are allocated to the Electronic Control Units (ECUs) based on runnables allocation.

With this aim in mind and in the context of SafeAdapt [104] proposed an approach to deal with the challenges of fail-operational behaviour of safety-critical networked embedded systems in which engineers are supported by design concepts for realizing safety, reliability and adaptability requirements through the use of architectural patterns. They highlighted how fail-operational concepts can be expressed at software architectural level and the effort of developing such systems is significantly reduced. To do so, they represent patterns as meta-models extension, which can be instantiated at the software architectural level.

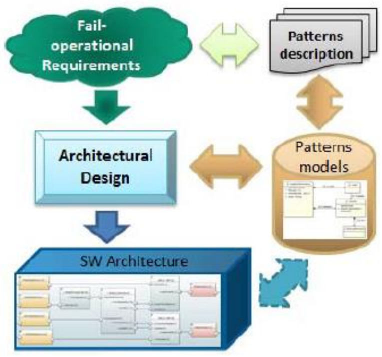


**Figure 16: Overview of architectural pattern definition**

Specifically, different meta-models are proposed such the ones for adaptive fail-operational redundancy pattern or the one for fail-operational graceful degradation pattern.

Similar concepts are addressed by [92] where an object-oriented approach to describe different safety patterns for control – monitor structure or Triple Modular Redundancy in UML are described.

Some other projects such as parMERASA [105] provided recommendations to the execution models of AUTOSAR and IMA as well.

Together with safety or availability issues, some other concerns such as security is discussed. SAFURE project [106] tackles not only safety but also the concept of security patterns, since the concept of architecture models and patterns for safety and security is addressed. To be more precise, the Hardware Security Module as defined in EVITA [107] and the separation kernel as described in the MILS recommendations are described.

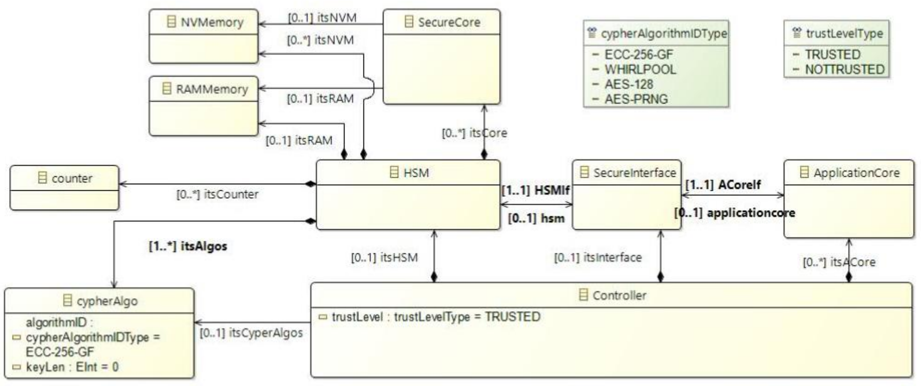


**Figure 17: Modelling elements to represent platform elements for security [106]**

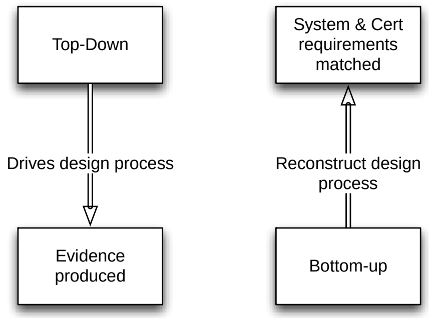

Top-Down
Drives design process
Evidence produced
System & Cert requirements matched
Reconstruct design process
Bottom-up

**Figure 18: Top down and bottom up approaches to system certification**

### 3.3.1 Argumentation patterns targeting fault tolerance

As it has recently been pointed out, some work has already been carried out w.r.t. argumentation patterns and its application on technology patterns such as multicore. Another important field gaining more interest from both academia and industry is the one trying to settle the issue of conceiving argumentation patterns for different fault tolerance design patterns. Fault tolerance in a system can be achieved through different design patterns such as redundancy, diversity, heartbeat or system monitor [81]. W. Wu and Tim Kelly [82] explained how it is possible to establish a set of argumentation patterns corresponding to different fault tolerant architectures. Figure 19, Figure 20, Figure 21, and Figure 22 illustrate arguments over the use of crosschecking (comparison and functional redundancy), voting (with functional redundancy), watchdog and enabling monitor (with analytic redundancy).

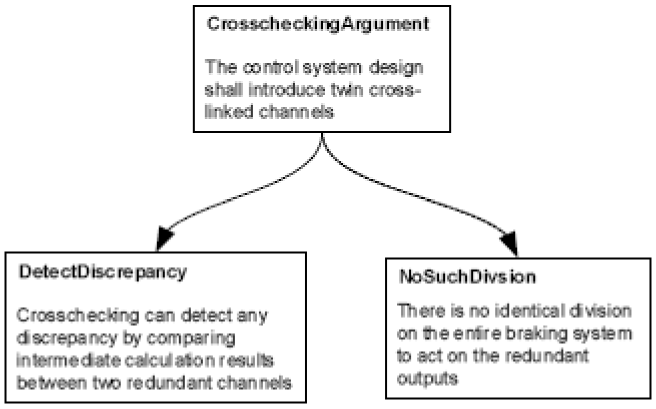

CrosscheckingArgument
The control system design shall introduce twin cross-linked channels
DetectDiscrepancy
Crosschecking can detect any discrepancy by comparing intermediate calculation results between two redundant channels
NoSuchDivsion
There is no identical division on the entire braking system to act on the redundant outputs

**Figure 19: Arguments over the use of crosschecking**

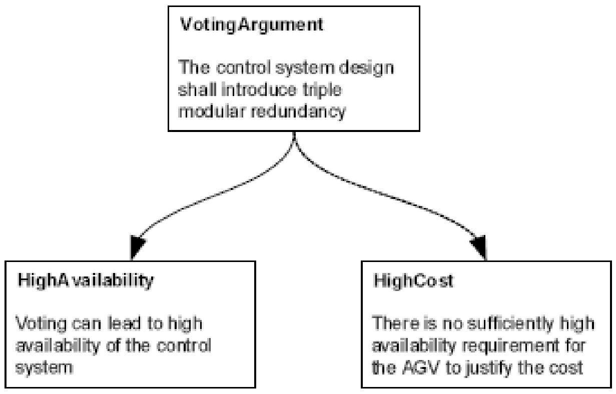

VotingArgument
The control system design shall introduce triple modular redundancy
HighAvailability
Voting can lead to high availability of the control system
HighCost
There is no sufficiently high availability requirement for the AGV to justify the cost

**Figure 20: Arguments over the use of voting**

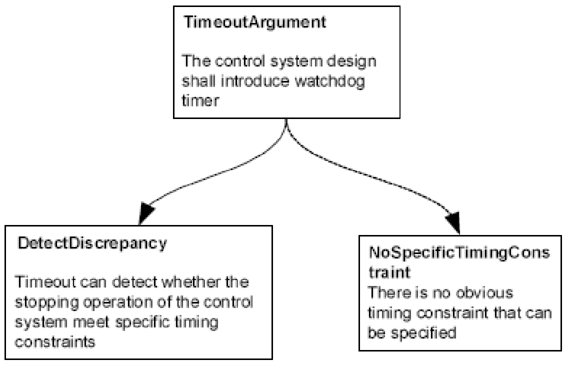

TimeoutArgument
The control system design shall introduce watchdog timer
DetectDiscrepancy
Timeout can detect whether the stopping operation of the control system meet specific timing constraints
NoSpecificTimingConstraint
There is no obvious timing constraint that can be specified

**Figure 21: Arguments over the use of timeout (watchdog)**

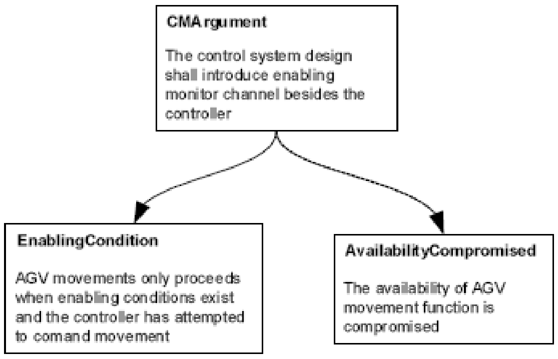

CMArgument
The control system design shall introduce enabling monitor channel besides the controller
EnablingCondition
AGV movements only proceeds when enabling conditions exist and the controller has attempted to comand movement
AvailabilityCompromised
The availability of AGV movement function is compromised

**Figure 22: Arguments over the use of enabling monitor**

### 3.3.2 Argumentation patterns for specific technologies

In the same way that fault tolerance argumentation patterns can be built up, the same approach is applicable to create specific technology ones, making the reuse process easier. [140] proposes a safety argumentation pattern for multicore to make the reuse easier. Figure 23 depicts an excerpt of safety argumentation used for safe access to shared resources related to multicore.

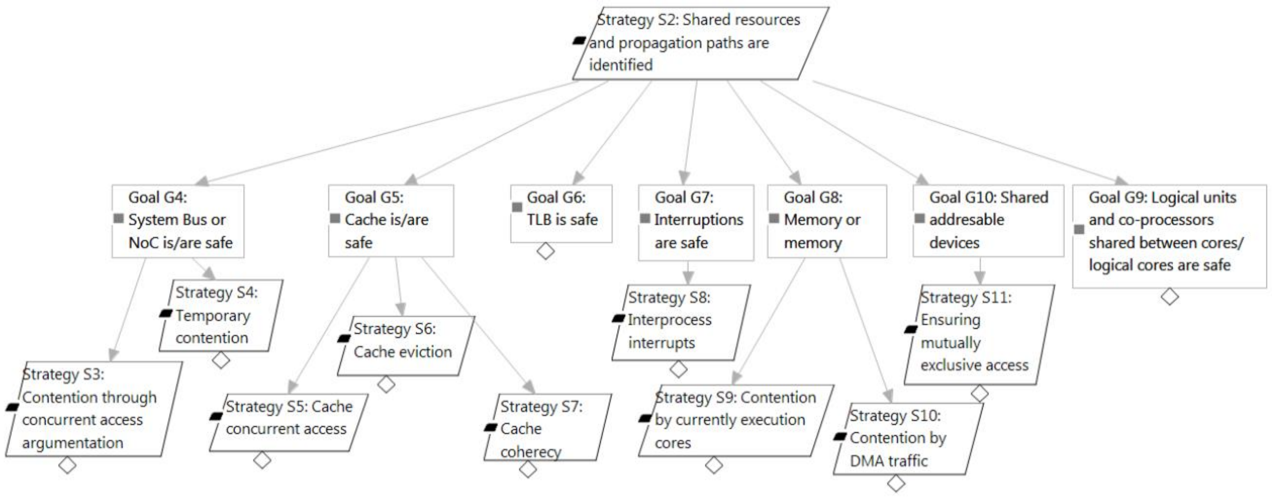

Strategy S2: Shared resources and propagation paths are identified
Goal G4: System Bus or NoC is/are safe
Goal G5: Cache is/are safe
Goal G6: TLB is safe
Goal G7: Interruptions are safe
Goal G8: Memory or memory
Goal G10: Shared addresable devices
Goal G9: Logical units and co-processors shared between cores/ logical cores are safe
Strategy S4: Temporary contention
Strategy S3: Contention through concurrent access argumentation
Strategy S6: Cache eviction
Strategy S5: Cache concurrent access
Strategy S7: Cache coherecy
Strategy S8: Interprocess interrupts
Strategy S9: Contention by currently execution cores
Strategy S10: Contention by DMA traffic
Strategy S11: Ensuring mutually exclusive access

**Figure 23: Excerpt of safety argumentation used for safe access to shared resources**

Figure 24 shows a common pattern regarding sharing resources so all instantiations of it will have a similar structure that will help the system on a future step to compare decisions from past cases and how the decisions have affected the arguments.

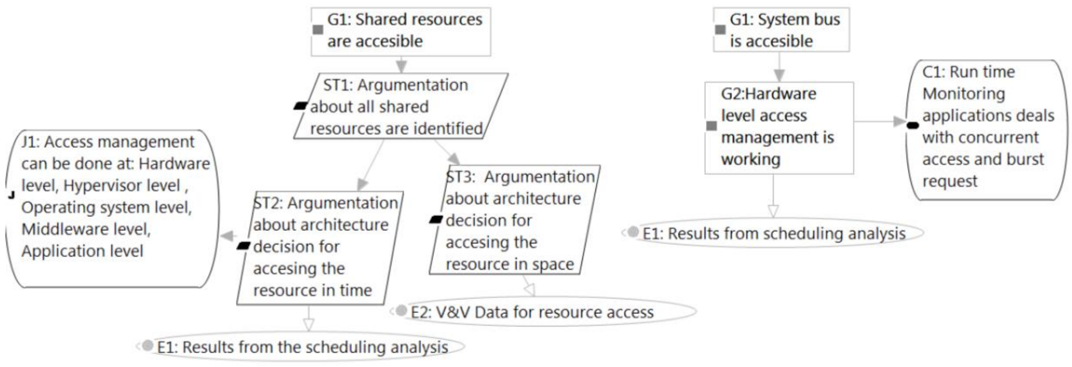

G1: Shared resources are accesible
ST1: Argumentation about all shared resources are identified
J1: Access management can be done at: Hardware level, Hypervisor level , Operating system level, Middleware level, Application level
ST2: Argumentation about architecture decision for accesing the resource in time
ST3: Argumentation about architecture decision for accesing the resource in space
E1: Results from the scheduling analysis
E2: V&V Data for resource access
G1: System bus is accesible
G2:Hardware level access management is working
C1: Run time Monitoring applications deals with concurrent access and burst request
E1: Results from scheduling analysis

**Figure 24: Preliminary pattern and its use on an example for shared resources**

However, mainly of the work has been done in relation with safety case patterns so far. Thus, assurance case patterns addressing technology specific solutions and considering some other concerns such as security will be further investigated in AMASS.

Together with assurance and certification issues related to the previous specific technologies, some others like deterministic communication technologies, remote diagnosis or software upgrading will be further investigated in AMASS in order to see under what circumstances can be reused, assured and certified.

## 3.4 Assurance of specific technologies

In this section we review different assurance and certification activities concerning novel technologies. This is related to subsection 2.3 “Assurance of Specific Technologies”.

The fact that new technologies have arisen over the past few years challenges traditional assurance and certification procedures. A challenging issue in safety-critical domains such as automotive or aerospace industry is that those novel technologies emerge before certification processes can be adapted in time. Consequently, new requirements might be included as part of the current or future standards because of the usage of a specific technology. Furthermore, specific assurance case patterns related to a specific technology can be developed. In the same way, new methods and requirements covering multi-concern assurance of the evolving new technologies/architectures might be needed to balance the trade-off between the different concerns as tackled in WP4.

Because of the aforementioned pace of technological change, some entities like NASA have published standardized design process and assurance activities for complex electronics. Figure 25 illustrates the devices that can be roughly grouped by function and complexity [136]:

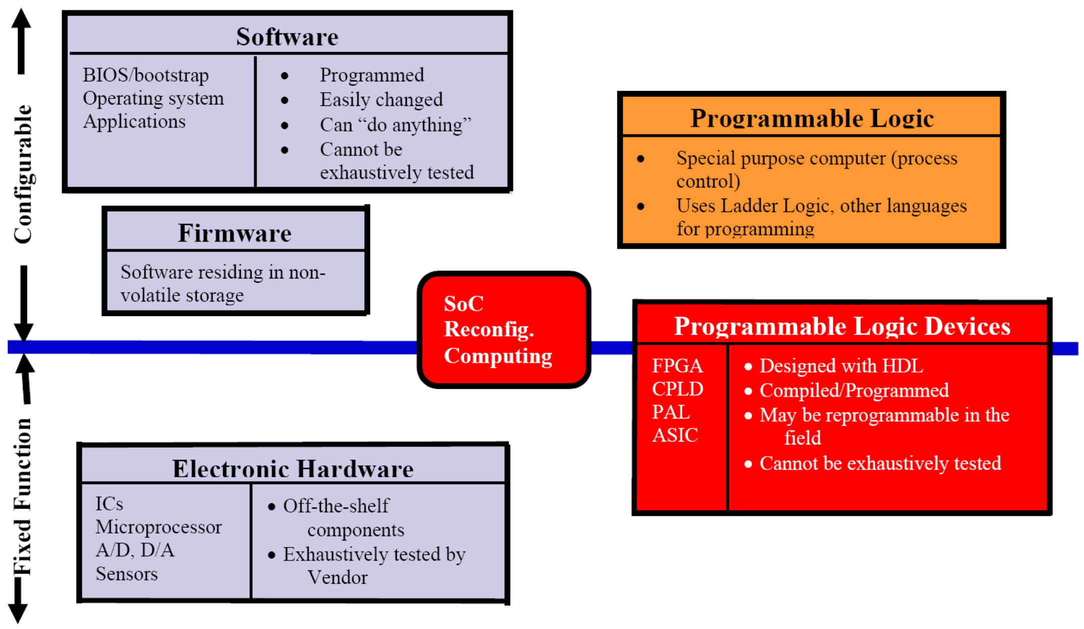

Configurable
Fixed Function
Software
BIOS/bootstrap
Operating system
Applications
• Programmed
• Easily changed
• Can "do anything"
• Cannot be exhaustively tested
Firmware
Software residing in non-volatile storage
SoC Reconfig. Computing
Programmable Logic
• Special purpose computer (process control)
• Uses Ladder Logic, other languages for programming
Programmable Logic Devices
FPGA
CPLD
PAL
ASIC
• Designed with HDL
• Compiled/Programmed
• May be reprogrammable in the field
• Cannot be exhaustively tested
Electronic Hardware
ICs
Microprocessor
A/D, D/A
Sensors
• Off-the-shelf components
• Exhaustively tested by Vendor

**Figure 25: Classification of complex electronics**

### 3.4.1 MultiProcessor System-on-a-Chip: multi-core

One of the most relevant and recent platforms that has gained increasing relevance is the so-called MPSoC (Multiprocessor System-on-a-Chip) which uses multiple processors on a single chip.

In fact, one of the very significant areas for AMASS is the rise of multi-core processors which use is rapidly increasing. They can be part of microprocessor based on SoC (standards microprocessor cores completed with various hardware functions which are not customizable and only configurable through programming registers) or part of custom SoC/ System on Programmable Chip (SoPC) (included programmable logic).
The benefits from integrating multiple functionalities in a single chip are quite clear. There are substantial savings with respect to volume/reduction of size, weight, cost and energy consumption. Parallel processing and the physical separation of mixed criticality software arise as possible implementations of these architectures. Together with the previous benefits, the total number of computational nodes and wires is considerably reduced increasing the robustness due to connector failures.

In spite of these benefits, multi-core processors present some pitfalls that need to be overcome and further investigate. These limitations include difficulties in the certification process due to the high complexity of these kinds of systems, the lacking temporal determinism and problems related to error propagation between subsystems [137].

The hard-real time constraints that many safety-critical embedded applications require stand for systems having to guarantee worst-case execution time (WCET). In fact, missing a specified deadline could lead to catastrophic failures. Thus, we should keep in mind that several problems still arise with existing multi-core architectures. Likewise, another drawback is the high complexity they present. The European Aviation Safety Agency and the Federal Aviation Administration classify them as "highly complex microcontrollers" and proposes as a compromise solution limiting its use to a single core, turning off all the other cores, even though this is not really the desirable solution. The need of guaranteeing spatial and temporal partitioning requires also from adequate partitioning mechanisms.

Even if some of the existing multi-core architectures do not contain special mechanisms to ensure freedom of interference by providing temporal and spatial isolation, in the recent years new multi-core based architectures have been developed having these requirements in mind. One remarkable example is the platform architecture developed in RECOMP [156]. It stands for Reduced Certification Costs Using Trusted Multi-core Platforms. and deals with mixed criticality issues what is a serious certification cost driver and limits update capabilities of non-critical functions, as safety requires from synchronism and determinism. In particular, this EU-funded project developed HW and SW architectures, design methods, and tools to efficiently design and (re-)certify MPSoCs for mixed critical systems.

The European ARTEMIS ACROSS project (2010-2013) [173][174] evolved a heterogeneous muti-core architecture that enables certification for the highest criticality levels. It supported temporally deterministic applications via a time-triggered Network-on-Chip and ensured the freedom of interference between applications from different criticality levels.

Another project addressing the multi core architectures challenges is CONCERTO [145]; CONCERTO, an ARTEMIS JU project recently ended, provides a cross-domain model driven methodology supporting the design, verification and implementation of critical real-time software systems that target multicore processors. In particular, CONCERTO extended the results of the CHESS project [146][26] allowing modelling and verification of dependability and timing properties for software components allocated to multicore processors. Run-time monitoring analyses are also provided to analyse traces on the actual application execution in its run-time environment.

On top of the cross domain support, CONCERTO provides some domain specific support; for instance, in order to better cover specific needs for the avionics, the modelling language and analysis tool support have been extended to cover the requirements of the IMA avionics architectures. Using the CONCERTO/CHESS extended approach, IMA and ARINC-653 functional/application partitions can be modeled, together with their allocation to cores; then their timing properties can be derived and statically analyzed.

Regarding certification of critical real-time software systems that target multicore processors, additional work should be done to better understand the potential support of the analysis methods proposed by CONCERTO.

As EASA considered inevitable the introduction of processor multi-core in Embedded Aircraft Systems, they decided to publish a study [155] for information purposes, as the outcome of MULCORS (The Use of MULticoreproCessORS in Airborne Systems) project. This study aims to provide a survey of Multi-core processors market availability, define multi-core processors assessment and selection criteria, perform investigations on a representative multi-core processor, identify mitigation means, design and usage rules and limitations, suggest recommendations for multi-core processor introduction and to suggest complementary or modification to EASA guidance.

In the same way, the EU FP7 projects CONTREX (Design of embedded mixed-criticality CONTRol systems under consideration of EXtra-functional properties) [148], DREAMS[149] and PROXIMA [150] collaborate in an European Mixed-Criticality Cluster (MCC) [147] to identify future challenges in the design and development of mixed-criticality multicore systems. They establish essential challenges to overcome in the area of timing, certification, extra-functional properties and development methods. One key factor is the need of ensuring temporal and spatial partitioning. To do so, theses architectures include the appropriate mechanisms put in place at different levels. Regarding certification CERTAINTY (Certification of Real Time Applications designed for mixed criticality), MultiPARTES, PROXIMA or DREAMS work in the context of mixed-criticality systems. CERTAINTY facilitated the path towards certifiability in the avionics domain by providing a set of hardware and software principles, methodologies and techniques mastering system predictability and determinism while ensuring isolation between tasks of different criticality during shared hardware resource accesses. Besides, it contributed to define NoC (Network on Chip) protocols to ensure system robustness with very low overhead together with new design methodologies and accompanying tools to simplify the timing verification of safety-critical applications. Another important topic w.r.t. certification is the one related to modular safety cases what was one of the research concerns of DREAMS whereas the goal of was to deliver evidence and arguments on its probabilistic approach. Such systems may be heterogeneous with needs and requirements going up against in contest in terms os safety and security purposes.

It should be mentioned that some current MPSoC architectures already include safety and security related features to ensure product and process integrity. For instance, the The UltraScale MPSoC architecture builds on Zynq-7000 include multiple military-class security protocols to prevent any conceivable unauthorized access. One of the main issues to address is how to leverage the benefits of FPGA and at the same time meet the functional safety requirements.

### 3.4.2 Programmable Logic Devices

In [153] Bate and Conmy discussed the most significant issues and the possible solutions for the certification of FPGAs. Clegg [154] explained how faults and failures can be introduced into a FPGA, what possible mitigation techniques can be used, and the need for arguments to demonstrate how a FPGA meets its safety requirements. FPGA companies such Xilinx[175] or Altera [178][179] provide a functional safety design flow for both FPGA and SoC. This includes DO-254 compliant IP cores [177], certificate and reports and FPGA design and verification tools and methodologies. Xilinx functional safety design flow solution has been certified by TÜV SUD [176], reducing the whole certification time process by several months. Furthermore, regarding fault tolerance, Xilinx Isolation Design flow and Isolation verification tools provide a certified methodology to separate areas on the FPGA. This ensures that a fault of an isolate region (fault containment region) does not propagate to a different one generating a cascading failure. Besides it provides DO-254 compliant IP modules.

### 3.4.3 COTS (Commercial Off-The-Self)

COTS technology also presents assurance and certification challenges. Two different groups can be distinguished: COTS IC (Integrated Circuit) and COTS IP (Intellectual Property)

- COTS IC: Any COTS digital or hybrid electronic device which does not execute software in a specific core e.g., bus controllers, flip-flop, multiplexers, converters, memories…
- COTS IP (Commercial Off-The-Self Intellectual Property: any commercially available electronic function designed to be reused as a portion of a device which may be classified in the following three categories Soft IP, Firm IP or Hard IP)

Even if some work has been already done, AMASS will try to contribute to this enhancement.

### 3.4.4 IMA (Integrated Modular Avionics)

The development of multi-core based high performance IMA platforms will be a necessary step to reach a larger scale integration on function level. The Federal Aviation Authority's (FAA) Certification Authorities Software Team (CAST) has issued a position paper [141] where they limit the use of one core for IMA and two cores for other systems.

Rudolf Fuchsen [138] also tackled the concept of how to address Certification for Multi-Core Based IMA Platforms on the published professional article. It emphasized that integration of software services with diverging functional, robustness, safety and security needs requires an embedded platform, which provides sufficient CPU performance, memory resources and I/O bandwidth. At the same time, it should be sufficiently reliable, have a deterministic and predictable real-time behaviour, and support a safe and secure resource and CPU time partitioning. To finish with, it should be certifiable according to the applicable safety and security standards.

### 3.4.5 AUTOSAR

Not only new complex electronics architectures pose a challenge with respect to certification, but also the open integrated architectures like AUTOSAR or the previous mentioned IMA. AUTOSAR is considered a layered software architecture with about 80 modules and several functional safety measures are considered in [183]. As for IMA, it is of vital importance to ensure the freedom of interference for software components with different ASIL ratings.

As mentioned, an important issue is to fulfil ISO 26262 in an AUTOSAR architecture. The Safe Automotive soFtware architEcture (SAFE) [182] project addressed the previous concern speeding up the efficient development of safety features in cars. The main objective was to extend the AUTOSAR architectural model, enhance methods for defining safety goals and define development processes complying with the new ISO 26262 standard for functional safety in automotive electrical and electronic systems. Three of the most relevant key results regarding AUTOSAR and ISO 26262 were the following:

- Definition of an assessment model in order to show ISO 26262 compliance in terms of AUTOSAR architecture, determining which safety-related inputs/outputs are needed on each design phase, determining which safety-related inputs/outputs are needed on each design phase;
- AUTOSAR architecture recommendation and configuration through application rules
- A proposal for extension of the AUTOSAR standard, based on resulting final results of system, hardware and software meta-model, with specialised properties for safety analysis.

### 3.4.6 Adaptive Systems

In the same way, new multi-core based platforms need to reach the same level of determinism as single core platform and this feature needs to be demonstrated, the same applies to other emerging technologies such as adaptive systems. Some properties such as time and space partitioning need to be ensured and verified.

The more systems e.g. automotive systems are evolving towards open systems, the bigger is the potential applicability of concepts targeting fully self-adaptive systems. Moreover, adaptation is believed to be the forthcoming breakthrough in computing. This implies that traditional safety approaches addressing a complete safety-assessment at development time will not be fully applicable. New safety assessment scenarios will arise where systems and components that are not known at design time need to be integrated with already existing ones. Consequently, these future challenges may require from a complete revision of several standards e.g. ISO 26262 that these systems would need to comply with. SafeAdapt [103] investigated to propose a set of example of methods and techniques that would need further analysis to comply with ISO 26262 in terms of adaptive systems by inducing a critical thinking so that a safer system could be built.

**Table 2: Summary of methods/techniques that need further analysis to be applied on adaptive systems[139]**

| Method/technique | Level of appliance | Rationale |
|---|---|---|
| Formal methods | System / hardware / software | Formal methods can be used to verify each of the possible adaptation situations the system could run into. |
| Safety analysis | System | As a possible safety analysis, the combination of state-space and non-state-space model based techniques. |
| STPA (new hazard analysis technique)[152] | System | A New Hazard Analysis Technique has been used for a preliminary HARA analysis on different domains where complex interactions occur. However, its use on adaptive systems needs further research. |

| Method/technique | Level of appliance | Rationale |
|---|---|---|
| Model-based development | System | It enables low cost, iterative investigation and early verification and validation being able to re-design before the real implementation is carried out. In the same way, the adaptation concept and its development could be not only verified but also evaluated and re-designed depending on the obtained results. |
| Simulation | System / Hardware / Software | For functional behaviour verification For fault injection for safety and dependability validation |

These kinds of systems would need from new verification and validation methods.

## 3.5 System Modelling Languages

Many system (and system-related) modelling languages that have been proposed over the last couple of decades can be used in AMASS for architecture-driven assurance purposes, and more concretely in relation to the objective “System Architecture Modelling for Assurance”. The corresponding models (1) could be analysed to determine the extent to which they comply with the applicable standards’ requirements, such as requirements regarding their completeness or consistency, or (2) could be used to support assurance analysis and to store the corresponding information, such degree of compliance, in a structure way.

The following subsections review the main system modelling languages that the AMASS consortium currently regard as the most relevant ones for architecture-driven assurance which is related to subsection 2.1 “System Architecture Modelling Assurance”.

### 3.5.1 UML

The Unified Modeling Language (UML) is a general-purpose modeling language, which is intended to provide a standard way to visualize the design of a software engineering system [109]. It was originally motivated by the desire to standardize the disparate notational systems and approaches to software design developed by Grady Booch, Ivar Jacobson and James Rumbaugh at Rational Software in 1994–95, with further development led by them through 1996. It was adopted in 1997 as a standard by the Object Management Group (OMG), and later in 2005 by the International Organization for Standardization (ISO) as an approved ISO standard.

The current version is UML 2.5, officially released in June 2015 [110]. With great efforts devoted in academia to provide UML with strong formal foundations, it has pervaded education and industry, having become a de-facto standard in the field of software engineering. Examples of tools that support UML modelling include IBM Rational (e.g. Rhapsody), Papyrus (open source), PTC Integrity Modeler, Modelio, Enterprise Architect, and MagicDraw.

**UML and the 4+1 View Model.** The 4+1 View Model (Figure 26) was designed by Philippe Kruchten for "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views" [111]. This model, or some variant of it, has been widely adopted to describe software systems. It starts from the principle that a single view is not enough to represent adequately the viewpoints of different stakeholders, such as end-users, developers and project managers. Therefore, one needs several views to describe the system from those different viewpoints. Each view uses a different language (or UML-sublanguage) to address the particular needs of each stakeholder.

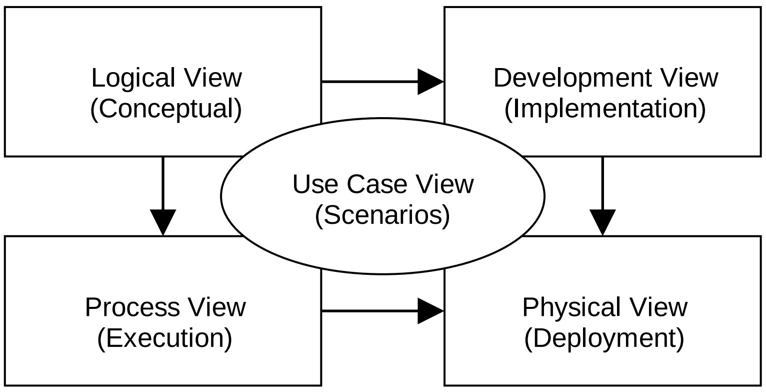


**Figure 26: The 4+1 Model or Architectural Views**

The four views of the system are logical, process, development and physical view. In addition to these basic views, use cases or scenarios are used to illustrate the architecture, i.e. the use cases view is the fifth view in the set. There are multiple versions of the 4+1 model and its relationship to UML resources. We provide here a brief summary:

- **Logical view (or Conceptual view):** The logical view is a static information model concerned with the functionality that the system provides to end-users. The main elements of UML used to convey this view are Class diagrams.
- **Process view (or Execution view):** The process view deals with the dynamic aspects of the system, i.e. it explains the system processes and how they communicate, and focuses on the runtime behavior of the system. The process view addresses mainly the needs of system integrators regarding concurrency, distribution, performance, and scalability, etc. The UML diagrams used to represent the process view include Activity diagrams, Statecharts, Communication diagrams, Sequence diagrams, Interaction diagrams, and so on (UML is particularly rich in different kinds of diagrams to represent the dynamics of systems).
- **Development view (or Implementation view):** The development view documents a system from the designer and programmer's perspectives, and it is concerned with software decomposition and management, where the expression of usage relationships, interfaces and dependencies is crucial. It uses the UML Component diagram to describe system components, often combined with the use of Package diagrams.
- **Physical view (or Deployment view):** The physical view depicts the system from a system engineer's point of view. It is concerned with the hardware-software mapping, i.e. the topology of software components on the physical layer, as well as the physical connections between these physical components. The Deployment diagram is the main UML resource used to represent the physical view, where the predefined icons are frequently tailored to meet specific needs and representational conventions.
- **Use Case view (or Scenarios view):** The description of an architecture is illustrated using a small set of use cases, or scenarios, which become a fifth view. Use cases define in an abstract way the services provided by the system as the external agents (called Actors) perceive them. Scenarios are concrete examples of interactions between Actors and System, sometimes including also interactions between system's internal objects and processes. Use cases are used to identify architectural elements and to illustrate and validate the architecture design, since the services provided by the system can (though not always) be allocated to different architectural subsystems and components. Scenarios also serve as a starting point for tests of an architecture prototype.

Table 3: Characteristics of each view in the 4+1 Model

| View | Logical (Conceptual) | Process (Execution) | Development (Implementation) | Physical (Deployment) |
|---|---|---|---|---|
| **Aspect** | Information model | Concurrency and synchronization | Software management and working environment | Software-Hardware mapping |
| **Stakeholders** | End users | System integrators | Designers and programmers | Systems engineers |
| **Requirements** | Functional | Performance<br>Availability<br>Reliability<br>Concurrency<br>Distribution<br>Security/Safety | Software management<br>Reuse<br>Portability<br>Maintainability<br>Platform/Language constraints | Performance<br>Availability<br>Reliability<br>Scalability<br>Topology<br>Communications |
| **Notation** | Classes and associations | Processes and communications | Components and usage relationships | Nodes and communication paths |

### 3.5.2 SysML

SysML [112] is a modelling language for systems engineering applications that allows the representation of various systems using a standardized set of symbols and supports the specification, analysis, design, verification and validation of a wide range of simple and complex systems and systems of systems. SysML allows a graphical definition of technical and conceptual aspects of a system, e.g. a "Plane". SysML is based on UML, and extends it with additional capabilities that allow its application in systems of general scope and not only related to software. The overall aim of SysML is to provide modelling techniques to a variety of systems including physical equipment, software, data, people, procedures, and facilities.

The benefits of using SysML lie in the potential of representation offered, which exceeds the natural barriers that exist in the various disciplines of knowledge. The language supports the creation of relatively simple representations of understanding, for systems covering disciplines such as thermodynamics, hydraulics, electronics and computer. The fact that SysML is not limited to a specific area of knowledge is an important because of the possibilities of integrating knowledge from multiple disciplines and knowledge reuse. Among the possibilities offered by the SysML, a user can model requirements and apply parametric restrictions. This allows the implementation of requirements engineering and performance analysis, which are essential when designing and assuring critical and complex systems.

The SysML diagrams are a specialization of UML diagrams. Some UML diagrams are used, other UML diagrams are adjusted so that they have greater coverage of representation and not just limited to software engineering, and other diagrams have been added only for SysML. Figure 27 shows the SysML diagrams taxonomy. **Behavior Diagrams** can represent different behaviors and states of systems, and consist in the following four diagrams: Activity Diagram, Sequence Diagram, State Machine Diagram, and Use Case Diagram. The **Requirements Diagram** is a new type of diagram that allows the application of requirements engineering in the design of systems. It also allows the definition of text-based requirements. **Structure Diagrams** allows the representation of structural and composition of the systems by means of Block Definition Diagram, Internal Block Diagram, Parametric Diagram, and Package Diagram.

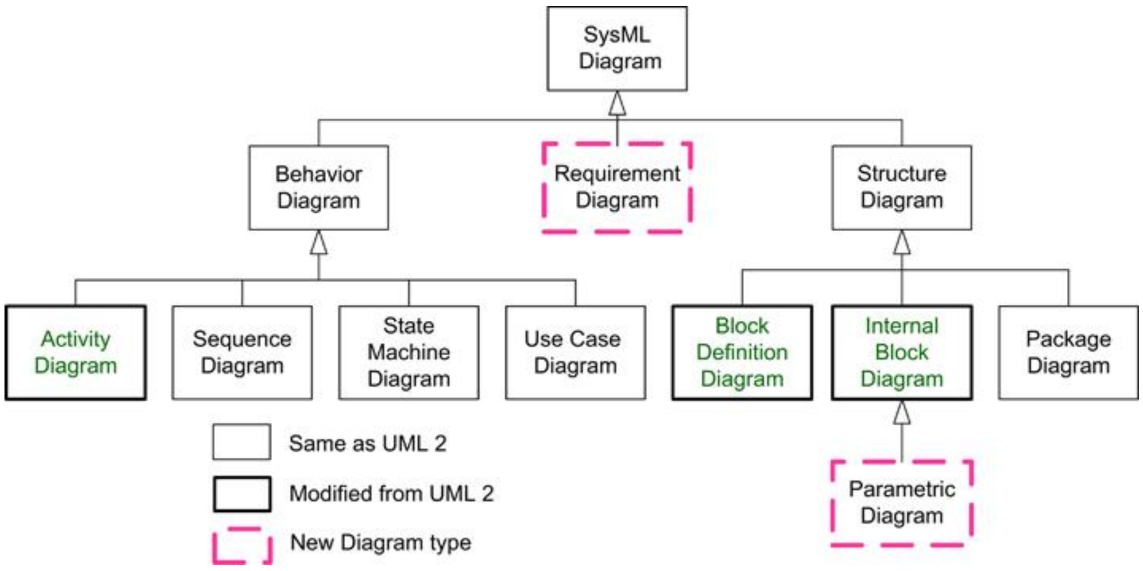


**Figure 27: SysML diagrams**

Currently there are many tools on the market that offer the possibility of representing systems through SysML, such as Rational Rhapsody (IBM), Papyrus (Polarsys), Integrity Modeler (PTC), Magic Draw (No Magic), and Enterprise Architect (Sparx Systems).

### 3.5.3 CHESS

CHESS [26] is a model driven methodology for the design, verification and implementation of critical software system. CHESS comes with its own component model and modelling language, the latter implemented as UML, MARTE (see 3.5.7) and SysML profile and called CHESSML [157]. Both the methodology and CHESSML can be put in practice by using the supporting toolset implemented on top of Eclipse Papyrus UML editor and made available as Eclipse open source Polarsys project [163].

The CHESS methodology relies on the CHESS Component Model [158], which is built around the concepts of components, containers and connectors. It supports the separation of concerns principle, strictly separating the functional aspects of a component from the non-functional ones.

According to the CHESS Component Model, a component represents a purely functional unit, whereas the non-functional aspects are in charge of the component’s infrastructure and delegated to the container, and connectors are responsible for the communication between containers.

From the interaction perspective, components are considered as black boxes that only expose their provided and required interfaces. Non-functional attributes are specified by decorating the component’s interfaces with non-functional properties related for instance to real-time concerns (e.g. a real-time activation pattern for an operation).

The declarative specification of non-functional attributes of a component, together with its communication concerns, is used in CHESS for the automated generation of the containers and connectors that embody the system’s infrastructure.

The CHESSML Dependability profile is used to enrich functional models of the system with information regarding its behaviour with respect to faults and failures, thus allowing properties like reliability, availability, and safety to be documented and analysed.

Moreover, CHESSML, by reusing and extending the MARTE UML profile, provides capabilities to specify timing related constraints on top of the component interfaces.

By using the CHESS methodology, modelling language and supporting toolset the modelled software system can be validated by applying specialised state-of-the-art schedulability analysis and dependability analysis, like state based analysis and failure propagation analysis. Such analysis results can then be used as artefacts for the qualification of the components and the overall system. Finally, code can be automatically generated for the designed and validated component architecture.

Consistent with the principles of component-based software engineering, the CHESS component model is hence characterized by the definition of “strong interfaces” that are the basis for structural composability, and precursors of contracts.

Indeed, in the direction to further support qualification and then certification of components, CHESSML has been extended to support the modelling of contracts, as addressed by the SafeCer generic component model (see 3.5.4). A dedicated profile for contract has been implemented of top of CHESSML (see appendix B). Then integration with the OCRA tool has been developed to support formal validation of contract refinement on top of the CHESS model.

CONCERTO ARTEMIS JU project [145] recently extended CHESS methodology, component model and toolset, aiming at the development of support technology for the use of model-based engineering artefacts and solutions for the end-to-end development process of critical real-time software systems that target multicore processors, as discussed in section 3.4.

### 3.5.4 SafeCer generic component model

Component Based Software Engineering (CBSE) promotes building software from existing software units, called software components, to facilitate software reuse and reduce development costs [23]. The approach emphasizes the notion of components being independent units with explicitly defined interfaces that capture all communication and all dependencies towards the rest of the system.

The systems development processes are then typically a mixed top-down and bottom-up approach, in particular by focusing on the decomposition of the system into the proper entities to develop/reuse in the top-down case and by composing the independently designed or reused entities during the second case.

When applied in the context of safety critical applications subject to certification, the traditional CBSE concepts must be extended with the information needed to support safety related concerns, in order to support the assurance case specification and so the certification process.

In fact, when a component is reused, it should be possible to reuse any safety argument and evidence, the latter collected during the component development or during a previous usage of the component itself. This means that for a given component, beside the checks concerning interfaces specification and compatibility, it should be possible to check if the associated safety related information can hold in the new environment where the component should operate.

In the context of the SafeCer project a generic component meta-model was defined [28], based upon common notions available in standard modelling languages and component models, like UML/SysML, and in particular:

- hierarchical components, at type and instance level;
- component’s required/provided data, event and service ports

The reader can refer to appendix A to find detailed information about how the aforementioned concepts are supported by the SafeCer component model.

Starting from a common notion of component model, new principles and mechanisms for safety contracts and safety argumentation formulation (in general term) have been added to the concept of component (see Figure 28).

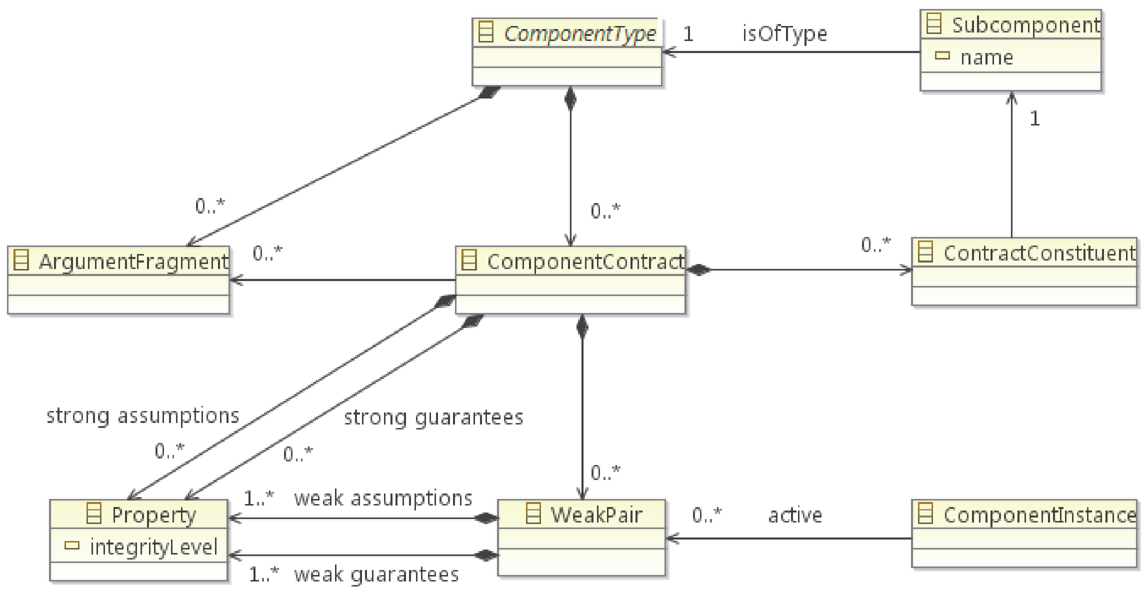


**Figure 28: Contracts, properties and argumentation.**

Component contracts and argumentation fragments represent reusable information associated with a component type, i.e., information that is valid for any instantiation of the component type in any context. Component contracts define properties together with the context assumptions under which they are guaranteed, and argument fragments represent a part of the safety argumentation related to the component type with no (or few but documented) context dependencies.

The metamodel supports the definition of strong and weak contracts. As discussed in [24] and [25], while strong assumptions define compatible environments in which the component type can be used, weak assumptions define specific contexts where additional information is available. Hence, a component type should never be used in a context where some strong assumptions are violated, but if some weak assumptions do not hold, it just means that the corresponding guarantees cannot be relied on.

Component instance inherits contracts and argument fragments from the component type, containing argumentation supporting the claims of the contracts. Moreover, for a given component instance it is possible to specify the set of weak contracts associated to the typing component which are active in a given system.

In a hierarchical system, a given contract of a decomposed component can be actually decomposed/refined by contracts of the internal parts decomposing the component itself; so for a given contract of a component it shall be possible to model the set of contracts of the child components which decompose/refine the contract of the parent component.

A component type can be associated with a number of safety argument fragments. An argument fragment provides a structured representation of the different evidence that a certain safety goal is met, and the reasoning why these evidences together form a sufficiently trustworthy case. A component contract can refer to argument fragments that describe the various evidence and the reasoning why the component can be trusted to fulfil the contract. For example, argument fragments can be represented as modules in GSN, a notation for systematic structuring of safety argumentation. The integrity level attribute available for contract's properties denotes the confidence in the contracts, i.e., the strength of the argumentation why the component can be trusted to fulfil the contract.

### 3.5.5 Modelica

Modelica [113] is an object-oriented programming language that allows physical systems modelling. Physical system models are built to represent an abstraction of the real world where the model is going to be used and its response to particular stimuli.

The Modelica language can be specified semantically by a set of rules that translate the designed model into a simpler flat Modelica structure. Models created with Modelica can be expressed by means of differential, algebraic, and discrete equations, by giving the possibility to the designer to develop models without needing any graphical representation. This flat structure allows keeping the model information, as functions, packages and partial models, allowing correctness to be verified before building the simulation model [114]. The language has raised great interest from system engineers.

The overall aim of Modelica is to ease models transformation into symbolic models in order to perform better simulations, which even in many cases could not be done. From these transformations, the opportunity to better share and reuse Modelica models by reducing the modelling effort also emerges [115].

Modelica provides a grammar that allows representing many building blocks of Modelica, such as characters and lexical units including identifiers and literals. The smallest building blocks in Modelica are single characters that belongs to a set; these elements are combined to form lexical units or tokens that are detected by the lexical analysis part. Table 4 shows some rules from the whole Modelica grammar specification. This information is the basis for representing physical laws that govern the model, without graphical representation information or comments, and for understanding the essentials as the information related to the components, and their properties as values and their relations with the rest of the components of the model.

Tools such as Dymola, JModelica, MapleSim, and OMEdit support the creation of Modelica models. Figure 29 shows an example of a Modelica model.

**Table 4: Selected Production rules of the Regular Tree Grammar of Modelica**

| | |
|---|---|
| class_definition ::= | class_prefixes class_specifier |
| class_prefixes ::= | (model) |
| class_specifier::= | long_class_specifier \|short_class_specifier |
| long_class_specifier ::= | IDENT string_comment composition end IDENT\| extends IDENT [class_modification] string_comment composition end IDENT |
| short_class_specifier ::= | IDENT "=" base_prefix name [array_subscripts] [class_modification] comment\| IDENT "=" enumeration "(" ( [enum_list] \| ":" ) ")" comment |
| component_clause ::= | type_prefix type_specifier [array_subscripts] component_list |
| type_specifier ::= | name |
| name ::= | ["."] IDENT {"." IDENT } |
| component_list ::= | component_declaration { "," component_declaration } |
| component_declaration ::= | declaration [condition_attribute] comment |
| declaration ::= | IDENT [array_subscripts] [modification] ->KE \| Term |
| connect_clause ::= | connect "(" component_reference "," component_reference ") |
| component_reference ::= | ["."] IDENT [array_subscripts] {"." IDENT array_subscripts]} |

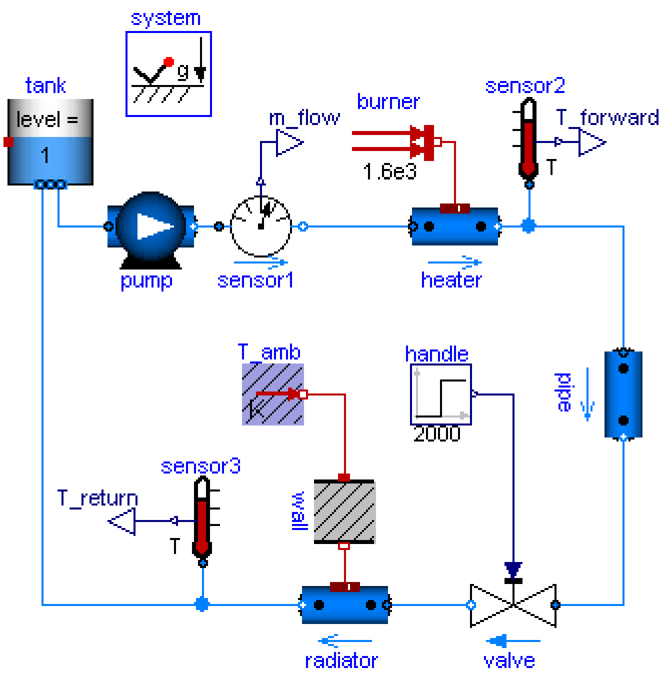


**Figure 29: Modelica model [115]**

### 3.5.6 RSHP

RSHP (arship) [117][118] is a universal knowledge representation model based on the ground idea that whatever information can be described as a group of relationships between concepts. Therefore, the leading element of an information unit is the relationship. For example, Entity/Relationship data models are certainly represented as relationships between entity types; software object models can also be represented as relationships among objects or classes; in the process modelling area, processes can be represented as causal/sequential relationships between sub-processes. Moreover, UML or SysML metamodels can also be modelled as a set of relationships between metamodel elements.

RSHP provides a repository model (Figure 30) to store information and relationships with the aim of reusing all kind of knowledge chunks. Furthermore, free text information can certainly be represented as relationships between terms by means of the same structure. Indeed, to represent human language text, a set of well-constructed sentences, including the subject+verb+predicate (SVP) should be used. The SVP structure can be then considered as a relationship typed V between the subject S and the predicated P. More specifically, the RSHP formal representation model is based on the following principles:

- The main description element is the relationship since it is the element in charge of linking knowledge elements.
- A Knowledge Element (KE) is an atomic knowledge component that appears into an artefact and that is linked by one or more relationships with other KEs to build information. It is defined by a concept, and it can also be an artefact (an information container found inside a wider artefact). A concept is represented by a normalized term (a keyword coming from a controlled vocabulary, or domain). Artefacts are knowledge containers of KEs and their relationships.

In RSHP, the simple representation model for describing the content of whatever artefact type (requirements, risks, models, tests, maps, text docs or source code) should be: α = i_α = {(RSHP_1 ), (RSHP_2 ),…, (RSHP_n )}, where every single RSHP is called RSHP-description and must be described using KE.

One important consequence of this representation model is that there is no restriction to represent a particular type of knowledge. Furthermore, RSHP has been used as the underlying information model to build general-purpose indexing and retrieval systems, domain representation models, approaches for quality assessment of requirements, and knowledge management tools.

RSHP is currently the main modelling language and thus is supported by the Requirements Quality Suite (RQS; https://www.reusecompany.com/requirements-quality-suite).

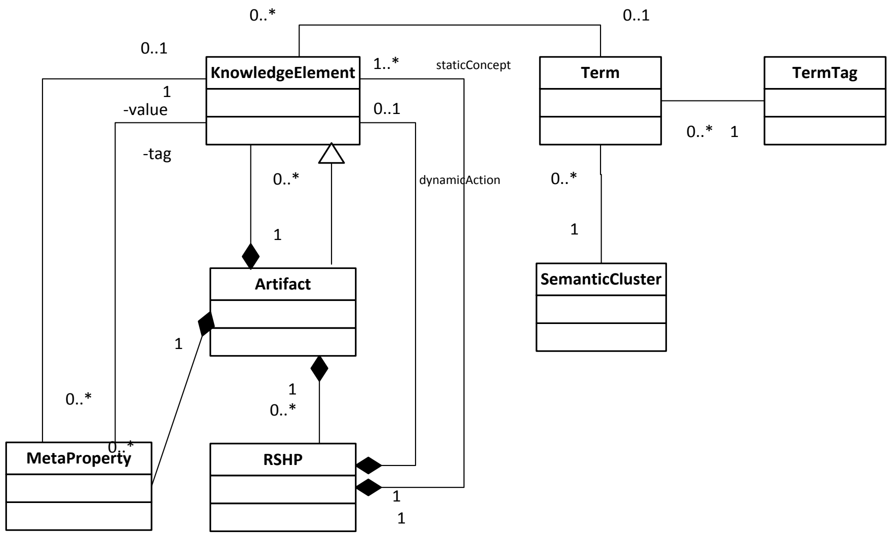


**Figure 30: The RSHP representation model using UML**

### 3.5.7 Other Languages

In addition to the languages presented in the previous sections, the following modelling languages that might be relevant for architecture-driven assurance in AMASS.

**AADL** (Architecture Analysis and Design Language; [119]) is a SAE (Society of Automotive Engineers) standard designed for the specification, analysis, automated integration and code generation of real-time performance-critical (timing, safety, schedulability, fault tolerant, security, etc.) distributed computer systems. It provides a new vehicle to allow analysis of system designs (and system of systems) prior to development and supports a model-based, model-driven development approach throughout the system life cycle.

**DAF** (Dependability Assurance Framework for Safety-Sensitive Consumer Devices; [120]) is an OMG (Object Management Group) specification that aims to provide a new system assurance methodology for the dependability argumentation for consumer devices, which is achieved by integrating conventional system assurance approaches such as risk analysis and assessments with a new way of approaching unique characteristics of consumer devices.

**EAST-ADL** [121] is a metamodel and an ontology for representing engineering information for automotive embedded systems. As such, it can be applied in different ways: As a framework to characterise engineering artefacts, as an exchange format between tools, as a UML Profile or as a native EAST-ADL modeller.

**KDM** (Knowledge Discovery Metamodel; [122]) is an OMG specification that provides a common intermediate representation for existing software systems and their operating environments

**MARTE** (Modeling and Analysis of Real Time and Embedded systems; [123]) is an OMG specification for modelling real-time and embedded applications with an UML extension in the form of a profile. MARTE consists of four main parts: (1) a core framework defining the basic concepts required to support real-time and embedded domain; (2) a first specialization (refinement) of this core package to support pure modelling of applications (e.g. hardware and software platform modelling); (3) a second specialization (refinement) of this core package to support quantitative analysis of UML models, specially schedulability and performance analysis; and, (4) a last part gathering all the MARTE annexes such as the one defining a textual language for value specification within UML models.

**ODM** (Ontology Definition Metamodel; [124]) is an OMG specification that aims to make the concepts of Model-Driven Architecture applicable to the engineering of ontologies. It links CL (Common Logic), OWL (Web Ontology Language), and RDF (Resource Description Framework).

**RAS** (Reusable Asset Specification; [125]) is an OMG specification to package digital artefacts. The specification is a set of guidelines and recommendations about the structure, content, and descriptions of reusable software assets. The main purpose of RAS is to increase the reusability of the software assets by encouraging consistent and standard packaging.

**SBVR** (Semantics of Business Vocabulary and Rules; [126]) is an OMG specification that can to be used as the the basis for formal and detailed natural language declarative description of a complex entity, such as a business. SBVR is intended to formalize complex compliance rules, such as operational rules for an enterprise, security policy, standard compliance, or regulatory compliance rules.

**SMM** (Structured Metrics Metamodel; [127]) is an OMG specification to define, represent, and exchange both measures and measurement information related to any structured information model.

### 3.5.8 Link of Assurance and System Models

Regarding the possibility to link assurance and system models, the SafeCer generic approach presented in section 3.5.4 addresses the problem by identifying possible relationships between the two models. This section reviews other approaches that have addressed the problem of how to link system models and assurance information. More specifically, the approaches have aimed to show how a system model matches the requirements or terminology of a given assurance standard. To this end, the approaches have proposed either refinement mechanisms to specify system models from assurance ones, or extended modelling languages with concepts from assurance standards.

In this category of approaches:

- Biggs et al. [128] describe a SysML profile designed for modelling the safety-related concerns of a system. The profile models common safety concepts from IEC 61508, ISO 12100, and ISOS 26262 and integrates them with system design information. The authors claim that, through increased traceability and integration, the profile allows for greater consistency between safety information and system design information and can aid in communicating that information to stakeholders.
- Gribov and Voos [129] address the problem of modelling software applications for robotics according to ISO 13482.
- Kuschnerus et al. [130] propose a UML profile that extends to support the development of safety-critical embedded software in accordance to IEC 61508. The authors aim to help software developers precisely express certification-related information using UML.
- Panesar-Walawege et al. [131] use Model-Driven Engineering to systematically guide system designers in establishing a sound relationship between a domain model of a safety–critical application and the evidence model for a certification standard. The authors use UML class diagrams to create domain models of the application and conceptual models of the safety standard (IEC 61508). They then use the extension mechanisms of UML and create a UML profile of the safety standard based on its conceptual model. The profile is then augmented with verifiable constraints, written in the Object Constraint Language, to help system suppliers in relating the concepts in the standard to the concepts in the application domain.

- Rupanov et al. [132] provide a framework for architecture model analysis and selection of safety mechanisms according to ISO 26262.
- Stallbaum and Rzepka [133] deal with DO-178-compliant model-based testing. They propose a UML profile that can be used to extend standard UML test models with information that is relevant for DO-178 certification.
- Wu et al. [134] present a modelling methodology including a UML profile to specify safety requirements for a component-based architecture model in compliance with DO-178. The methodology enforces safety requirements automatically.
- Zoughbi et al. [135] propose an approach to improve communication and collaboration among safety engineers, software engineers, and certification authorities in the context of DO-178. This is achieved by utilizing a UML profile that allows software engineers to model safety-related concepts and properties in UML. A conceptual meta-model is defined based on DO-178, and then a corresponding UML profile is designed to enable its precise modelling.

## 3.6 The MILS Architectural approach

In this section we review the MILS architecture approach and its extension, which is related to subsection 2.1 “System Architecture Modelling for Assurance” and 2.4 “Contract-Based Assurance”.

### 3.6.1 A brief overview of MILS architectural approach

The *MILS*[4] *architectural approach* [82] to the design and implementation of critical systems involves two principal phases: the development of an abstract architecture intended to achieve the stated purpose, and the implementation of that architecture on a robust technology platform. During the first phase, essential properties are identified that the system is expected to exhibit, and the contributions to the achievement of those properties by the architectural structure and by the behavioural attributes of key components are analysed and justified.

The MILS approach leverages system architecture to support vital system-level properties. The architecture reflects an intended pattern of information flow and causality referred to as the policy architecture, while key components of the architecture enforce local policies through specific behavioural properties. By reasoning compositionally over the components about the policy architecture and the local policies, many useful system-level properties may be established.

The MILS platform provides the technology for the concrete realisation of an abstract system architecture. A separation kernel [83][84], the underlying foundational component of the MILS platform, is used to establish and enforce the system architecture according to its configuration data.

The assurance of a system’s properties depends not only on the analysis of its design but on the correct implementation and deployment of that design. The configuration of the separation kernel must faithfully implement the specified architecture. This is guaranteed by the MILS platform configuration compiler that is driven by a model of the architecture and the constraints of the target platform to synthesize viable and semantically correct configuration data corresponding to the specified architecture.

[4] MILS was originally an acronym for “Multiple Independent Levels of Security”. Today, “MILS” is used as a proper name for an architectural approach and an implementation framework, promulgated by a community of interested parties, and elaborated by ongoing MILS research and development efforts.

### 3.6.2 DMILS

The distributed MILS (D-MILS) approach to high-assurance systems is based on an architecture-driven end-to-end methodology that encompasses techniques and tools for modelling the system architecture, contract-based analysis of the architecture, automatic configuration of the platform, and assurance case generation from patterns. The D-MILS concept extends the capacity of MILS to implement a single unified policy architecture to a network of separation kernels. To accomplish this, each separation kernel is combined with a new MILS foundational component, the MILS networking system (MNS), producing the effect of a distributed separation kernel. Time-Triggered Ethernet (TTE) [77] is employed in the D-MILS Project [74] as the MILS "backplane", which allows to extend the robustness and determinism benefits of a single MILS node to the network of D-MILS nodes, referred to as the distributed MILS platform4 [75][76].

Since D-MILS systems are intended for critical applications, assurance of the system's critical properties is a necessary byproduct of its development. In such applications, evidence supporting the claimed properties must often be presented for consideration by objective third-party system certifiers. To achieve assurance requires diligence at all phases of design, development, and deployment; and, at all levels of abstraction: from the abstract architecture to the details of configuration and scheduling of physical resources within each separation kernel and within the TTE interfaces and switches. Correct operation of the deployed system depends upon the correctness of the configuration details, of the component composition, of key system components, and of the D-MILS platform itself. Configuration is particularly challenging, because the scalability that D-MILS is intended to provide causes the magnitude of the configuration problem to scale as well. The concrete configuration data and scheduling details of the numerous separation kernels and of the TTE are at a very fine level of granularity, and must be complete, correct, and coherent.

The only reasonable prospect of achieving these various aspects of correctness, separately and jointly, is through pervasive and coordinated automation as embodied in the D-MILS tool chain. Inputs to the tool chain include, a declarative model of the system expressed in the MILS dialect of the Architecture Analysis and Design Language (AADL) [78], facts about the target hard-ware platform, properties of separately developed system components, designer-imposed constraints and system property specifications, and human guidance to the construction of the assurance case. Components of the tool chain perform parsing of the languages, transformations among the various internal forms, analysis and verification, configuration data synthesis and rendering, and pattern-based assurance case construction [80][81]. Outputs of the tool chain include, proofs of specified system properties, configuration data for the D-MILS platform, and an assurance case expressed in Goal Structuring Notation (GSN) [79].

The integration of the MILS architectural approach with contract-based design and analysis is described in [65]. Both MILS and contract-based approaches focus on architecture, and do so in a complementary way. MILS [66] regards information flow policy as an abstraction of architecture, and seeks to maximize the correspondence between architectural structure and the desired information flow policy of a system, which may rely on the behaviour of key components to enforce local policies that further restrict the maximal information flow permitted by the architecture. In more details, the approach relies on the OCRA tool [20] to formally prove that the global system requirements are met, provided local policies are guaranteed by application components. Safety-related and security-related requirements are both considered and analysed the decomposition also taking into account the possibility of component failures.

An architecture is only as valuable as the integrity of its components and connections. Recognizing the importance of integrity, MILS provides an implementation platform that can be configured to the ``shape'' of the architecture by initializing it with specific configuration data compiled to embody the global information flow policy.

The two methods are complementary and their combination yields a strong result. The contract-based method proves that the composition of components that satisfy their contracts will meet the system requirements, provided that their integrity is protected. The MILS platform guarantees the integrity of components and their configured connections, preventing interference that could cause a verified component to fail to satisfy its contract.

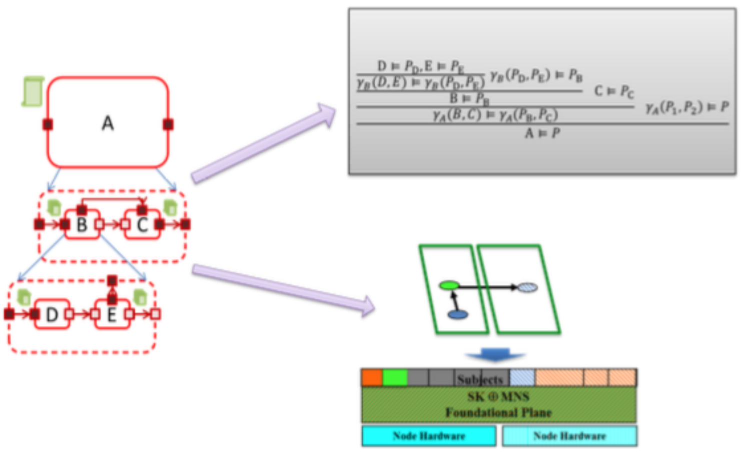


**Figure 31: The architecture is used for 1) formal reasoning to prove that the system requirements are assured by the local policies, 2) configuration of the platform to ensure the global information flow policy and the integrity of the architecture.**

In Figure 31, it is shown the approach applied to an abstract example taken from [65]. The system *A* is decomposed into subsystems *B* and *C*, and *B* in turn is decomposed into *D* and *E*. Each component is enriched with a contract (represented here by green scrolls). If the contract refinement is correct, we have associated with the architecture a formal proof that the system is correct provided that the leaf components (*D*, *E*, and *C*) satisfy their contracts. Namely, if *D* and *E* satisfy their contracts ($D \vDash P_D$, $E \vDash P_E$) and the contract refinement of *B* is correct ($\gamma_B (P_D, P_E) \preccurlyeq P_B$), then the composition of *D* and *E* satisfies the contract of *B* ($\gamma_B (D, E) \vDash P_B$). Moreover, if *C* satisfies its contract ($C \vDash P_C$) and the contract refinement of *A* is correct ($\gamma_A (P_B, P_C) \preccurlyeq P_A$), then the composition of *B* and *C* satisfies the contract of *A* ($\gamma_A (B, C) \vDash P_A$).

In MILS terms, the architecture defines three subjects (*D*, *E* and *C*) and prescribes that the only allowed communications must be the ones between *D* and *E* and between *E* and *C*. This is translated into a configuration for the D-MILS platform (taking into account other deployment constraints in terms of available resources), which in this example encompasses two MILS nodes.

In D-MILS, the architecture is specified in a variant of AADL, called MILS-AADL [78], similar to the SLIM language developed in the COMPASS project [68]. The COMPASS tool set has been extended in order to support the new language and to enrich the components with annotations that specify different verification properties such as contracts. The language used to specify the component contracts is the one provided by the OCRA tool [20]. It consists of a textual human-readable version of a First-Order Linear-time Temporal Logic. The logic has been extended in D-MILS to support uninterpreted functions, i.e. functional symbols that do not have a specific interpretation but are used to abstract procedures and the related results (such as CRC checksum or encryption), or to label data with user-defined tags (such as ``is_high'' or ``low-level'', etc.).

Regarding tool support, the refinement is first translated by OCRA into a set of entailment problems in temporal logic. nuXmv [67] translates this into a liveness model-checking problem with a classic automata-theoretic approach [69]. The resulting problem requires proving that a certain liveness condition can be visited only finitely many times along an (infinite) execution. This problem is in turn reduced to proving an invariant on the reachable states with the K-liveness techniques described in [70]. This has been extended to infinite-state systems and to take into account real-time aspects in [71]. Finally, the invariant is proved with an efficient combination of induction-based reasoning, explicit-state search, and predicate abstraction, extending the IC3 algorithm [73] to the infinite-state case, as described in [72].

Finally, apart of this architecture approach, we would like to remark that in the CATSY (**Cat**alogue of **Sy**stem and Software Properties) project [184], an extension of the SLIM was defined to specify properties and contracts in an AADL-compliant way, using an AADL property set. The CSSP (**C**a**t**alogue of **S**ystem and **S**oftware **P**roperties) property set was defined, which is a catalogue of AADL properties with associated a predefined formalisation in temporal logic, to ease the formalisation of design properties that are typically found during the development lifecycle. Finally, a COMPASS extension was designed to support the compilation of these property sets and to interact with the OCRA tool to perform their validation and contract-based refinement.

## 3.7 Formal techniques for V&V-based Assurance

In this subsection, we describe different formal techniques to validate that a requirements specification is complete, correct, and unambiguous and that the deployed system satisfies those requirements. These techniques will enrich the OPENCOSS and SafeCer assurance approaches, making sure that evidence for arguing that the requirements specification is valid are provided as part of the assurance case. The analysis presented in this section is related to subsection 2.5 “V&V-based Assurance”.

### 3.7.1 Semantic Requirement Analysis

There are certain properties of software requirements that greatly decrease the overall cost of the development. To list a few the requirements should be accurate, atomic, attainable, cohesive, complete, consistent, current, independent, modifiable, non-redundant, traceable, unambiguous, and verifiable. The more of the above properties the requirements have the more easily the requirements are used later in the development. Spending more time on requirements engineering and analysis proves greatly efficient in terms of the effort needed for validation and verification. Many of these properties are a matter of conscientious and rigorous engineering but some properties are difficult to maintain and to demonstrate. Formalization of requirements automatically provides accurate, modifiable, traceable, unambiguous, and verifiable requirements from its definition. For other properties, such as consistency and non-redundancy, formal requirements provide a direct route to test automatically these properties. ForReq allows the engineers to test sanity of requirements (see Section 3.7.2), by facilitating the functionality of the DiVinE sanity checker [49] and the nuXmv tools [67].

Another tool that support the formal analysis of requirements is RAT [87], developed in the Prosyd Project [88], where a designer can explore the requirements’ semantics and automatically performs different checks to assess whether she/he has written the right set of properties, where the authors call this set of checks as *Property Assurance*. First, it is possible to check that the requirements are consistent and do not contain a contradiction. Second, the designer can provide two inputs: a set of properties $\Phi_A$ that must be guaranteed, and $\Phi_P$ a set of possibilities that describes corner cases that must be allowed by the requirements. On the one hand, using assertions, a designer can check whether the set of formal properties $\Phi_A$ entails another formal property $\Phi_P$ representing the negation of the undesired behaviour. On the other hand, with possibilities, the designer can check that either the set of formal properties $\Phi A$ is consistent with an additional formal property $\Phi P$ representing the desired behaviour.

An interesting formal approach is presented in [16] for the validation of functional requirements of hybrid systems, where discrete components and continuous components are tightly intertwined. The validation problem in this approach is similar to the previous one, which is associated with different set of checks (performed automatically by the verification engine), each consisting of the satisfiability problem for a conjunction of formulas. The approach was the basis of an industrial project aiming at the validation of the European Train Control System (ETCS) requirements specification.

Finally, the OCRA tool also supports the validation of requirement specification, which was inspired from RAT. In more details, three kinds of validation problem can be specified. *Consistency problem*: either the set of formal properties is consistent and a trace is shown as witness or it is inconsistent and an unsat core (subset still inconsistent) is given to the user. *Possibility problem*: either the set of formal properties is consistent with an additional formal property and a trace is shown as witness or it is inconsistent and an unsat core is presented to the user. *Assertion problem:* either the set of formal properties entails another formal property representing the negation of the undesired behaviour and an unsat core is presented to the user or it violates the additional formal property and a trace is shown as counterexample.

### 3.7.2 Requirements Sanity Checking

Recently, there have been tendencies to use the mathematical language of temporal logics, e.g. the Linear Temporal Logic (LTL) [11], to specify functional system requirements. Restating requirements in a rigorous, formal way enables the requirement engineers to scrutinise their insight into the problem and allows for a considerably more thorough analysis of the final requirement documents [43]. When the requirements are given and a model is designed, the formal verification tools can provide a proof of correctness of the system being developed with respect to formally written requirements. The model of the system or even the system itself can be checked using model checking [44][45] or theorem proving [46] tools. If there are some requirements the system does not meet, the cause has to be found and the development reverted. The longer it takes to discover an error in the development, the more expensive the error is to mend. Consequently, errors made during requirements specification are among the most expensive ones in the whole development.

Model checking is particularly efficient in finding bugs in the design, however, it exhibits some shortcomings when it is applied to requirement analysis. In particular, if the system satisfies the formula then the model checking procedure only provides an affirmative answer and does not elaborate for the reason of the satisfaction. It could be the case that, e.g., the specified formula is a tautology, hence, it is satisfied for any system. To mitigate the situation a subsidiary approach, the sanity checking, was proposed to check vacuity and coverage of requirements [47]. Yet the existence of a model is still a prerequisite, which postpones the verification until later phases in the development cycle.

In CRYSTAL, the idea of liberating sanity checking from the necessity of having a model of the system under development has been studied [48][49][50]. The notion of model-free sanity checking of requirements written as LTL formulae has been defined, implemented and evaluated. Sanity checking commonly consists of three parts: consistency checking, redundancy checking, and checking completeness of requirements. A consistent set of requirements is one that can be implemented in a single system. A vacuous requirement is satisfied trivially by a given system and redundancy is the equivalent of vacuity for the model-free case. Finally, a complete set of requirements extends to all sensible behaviours of a system.

The presented approach to consistency and vacuity checking is novel in identifying all inconsistent (or vacuous) subsets of the input set of requirements. This considerably simplifies the work of requirements engineers because it pinpoints all sources of inconsistencies. For completeness checking, a new behaviour-based coverage metric has been proposed. Assuming that the user specifies what behaviour of the system is sensible, the proposed coverage metric calculates what portion of this behaviour is described by the requirements specifying the system itself. The method further suggests new requirements to the user that would improve the coverage and thus ensure more accurate description of users' expectations. The efficiency and usability of this approach to sanity checking has been verified in an experimental evaluation and a case study.

Since checking sanity is computationally demanding, and the demands grow with the number of related requirements, ForReq further optimizes the verification method beyond the capacity offered by either of the tools. Within ForReq, the requirements are first analysed to be divided into categories where each category refers only to a subset of input/output variables. Since no two requirements from distinct categories refer to the same variable, these categories can be processed independently, which provides the sanity checking with considerably better scalability.

### 3.7.3 Automated Formal Verification

Once the requirements are formalized and their sanity is checked they can be automatically verified against corresponding system design. Such chain of activities leading to automated verification is described in Section 3.7.4, where explicit model checking is used as the verification method. Honeywell ForReq tool serves as a broker of this toolchain and allows for each task in the chain to be executed using multiple tools in parallel. Similarly, as for sanity checking, where ForReq employs DiVinE sanity checker and nuXmv, for model checking it also provides alternatives. DIVINE model checker [85] facilitates the explicit model checking and either NuSMV or nuXmv facilitate the symbolic model checking. Hence the verification expert can choose which tool is better applicable for the verification task at hand without repeating any of the task within the chain that were already performed. As a final alternative, there is also a test case generation tool.

If any of the model checkers succeed in their verification task the assurance tool chain returns a proof that given requirements are satisfied by the system design. The supported design languages are Simulink, C, and C++; and the proof has the form of the enumeration of all possible behaviours of the system, each satisfying the requirements. When all the model checkers fail, the toolchain returns at least the requirement-based test with coverage according to the aerospace RTCA DO-178B/C standard. A similar system that uses evidence integration and design assurance frameworks is an Evidential Tooling Bus from SRI [166].

### 3.7.4 Toolchain for Automated Formal Verification

Verification of developed embedded systems is one of the major consumers in the overall development process in terms of time and cost. The safety-critical nature of embedded systems, e.g., in avionics industry makes the situation even worse, as it calls for thorough process of validation, verification, and certification. Automated formal verification methods, such as model checking [51], are used to significantly alleviate the burden of the cost of verification and validation process in this case. An important lesson learned from the effort spent on verification of avionics systems [52] shows that the major bottleneck preventing successful application of model checking to verification of avionics systems is the scalability of verification tools. While symbolic model checking tools have been successfully applied in model-based development based on Mathworks Simulink [53], their scalability is unfortunately quite limited. One of the factors limiting scalability is the absence of distributed tools: none of the symbolic model checkers can distribute the work among multiple computation nodes [54]. On the other hand, explicit-state model checking has been repeatedly reported to scale well in a distributed-memory environment [55][56].

In Crystal, an integrated toolchain [57] composing tools used for the development of embedded systems in avionics has been created. In particular, Simulink and the explicit-state model checker DIVINE [58] have been interconnected. This connection allows a design engineer to verify systems under development against properties specified in Linear Temporal Logic (LTL). LTL is the standard formalism for expressing important behavioural properties; see e.g. Process Specification Language [59]. The choice of DIVINE model checker stemmed from the simple fact that to our best knowledge, DIVINE has been the only LTL model checker that could fight the scalability limits by means of parallel and distributed-memory processing.

The primary goal of the toolchain as described in [57] was to enable the verification of Simulink designs with an explicit-state model checker. While this was achieved in principle, minor attention was paid to help the design engineer to understand outputs of the verification process. The model checking verification consumes a model of the system under verification and a specification the model should meet. For those two inputs it performs an algorithmic decision about the validity of the system with respect to the specification. If the system meets the specification, the model checking procedure simply returns a message informing the user about the fact. However, should any behaviour of the system under verification violate the specification, the negative answer to the verification is supported with the so called counterexample, i.e. behaviour of the system witnessing the invalidity of the specification.

The process of interpretation of the counterexample as returned by the model checker back to the Simulink environment has been focused on in [60]. For the purpose of counterexample interpretation, a Simulink template model has been introduced, which can be used to simulate any Simulink system according to a counterexample.

Further research in the area of checking Simulink diagrams has been made in [61][62][63]. The process of model checking of Simulink diagrams has been formalized, building on the concept of set-based reduction of the underlying state space as proposed in [64]. This reduction can significantly decrease the number of states, but does not lead to any reduction in the worst case. In addition, the representation used for the sets of variable evaluations determines the resulting complexity, which can differ exponentially for particular representations. The set-based reduction using explicit sets as one representation and using bit-vector formulas as another representation has been implemented and both variants have been experimentally evaluated in [63].

## 3.8 Model-Based Safety Analysis

Safety-critical systems are traditionally subject to safety assessment techniques (e.g., Fault Tree Analysis (FTA) [186][187] or Failure Modes and Effects Analysis (FMEA)), to analyze the behavior of the system in presence of faults. The dramatic increase in complexity of computer-based systems has motivated, in recent years, a growing industrial interest in model-based safety analysis (MBSA) methods. In contrast to traditional safety assessment, MBSA methods are based on a single “safety model” of a system, typically written in a suitable modelling language. Analyses with respect to particular safety-significant events are conducted on the basis of such “safety models” with a high degree of automation [188]. Many of the MBSA techniques allow the verification of the functional correctness of the system (e.g., with respect to functional safety requirements) as well as to assess system behavior in the presence of faults [189][190][191]. In particular, formal verification tools based on model checking have been extended to automate the generation of artifacts such as Fault Trees (FTs) and FMEA tables (e.g., in FSAP [192][193], COMPASS [68], and xSAP [196]).

Since early 1990s a large number of MBSA languages and notations have become available. The most prominent languages include Failure Propagation and Transformation Notation and Calculus (FPTN and FPTC) [197][198], Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) [199], Error Model Annex of Architecture Analysis and Design Language (AADL), and AltaRica [194][195]. In particular, AltaRica is at the core of several industrial Integrated Modelling and Simulation Environments: Cecilia OCAS (Dassault Aviation), Simfia (EADS Apsys) and Safety Designer (Dassault Systemes). AltaRica can be also translated into SMV models, the input language of NuSMV and nuXmv, and analyzed symbolic model checking techniques [201].

xSAP [196] is an extension of FSAP [193] that integrates nuXmv and allows the safety analysis directly of SMV models, including different features such as a library-based specification of faults, fault effects, and fault dynamics, an automatic model-extension with fault specifications, FTA based on the generation of Minimal Cut Sets (MCS) for dynamic systems, for both the monotonic and non-monotonic case, FMEA, Fault propagation analysis based on Timed Failure Propagation Graphs (TFPG) and Common Cause Analysis (CCA).

# 4. State of the Practice on Architecture-Driven Assurance

In this chapter, an overview of the state of practice on Architecture-Driven Assurance is given. The overview is given per domain (Automotive, Railway, Avionics, Space, and Automation). For sake of clarity it should be noted that architecture-driven assurance is not yet spread in industry.

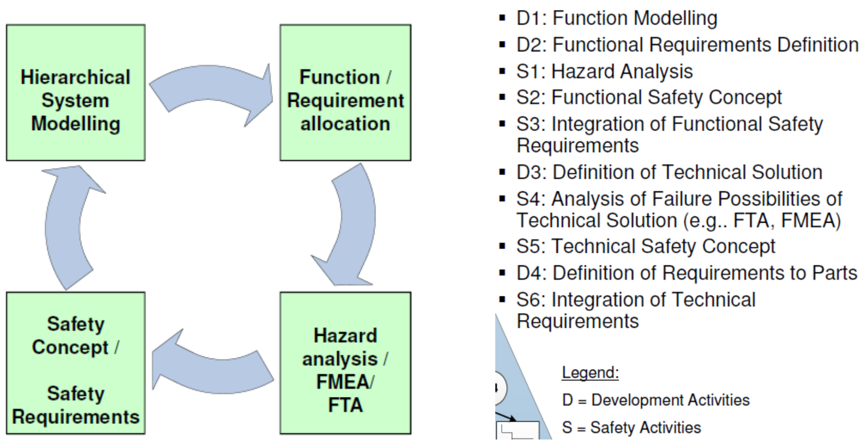
## 4.1 Automotive domain

With the trend of increasing technological complexity, software content and mechatronic implementation, there are increasing risks from systematic failures and random hardware failures. ISO 26262 ("Road vehicles – Functional safety") is the most important standard concerning functional safety in the automotive industry. The standard was published at the end of 2011. This means, that it is applied to new projects since 2011. ISO 26262 includes guidance to avoid these risks by providing appropriate requirements and processes. System safety is achieved through a number of safety measures, which are implemented in a variety of technologies (e.g. mechanical, hydraulic, pneumatic, electrical, electronic, programmable electronic) and applied at the various levels of the development process. ISO 26262 is intended to be applied to safety-related systems that include one or more electrical and/or electronic (E/E) systems and that are installed in series production passenger cars with a maximum gross vehicle mass up to 3500 kg.

Safety concepts are a required work product in an ISO 26262 safety case. Safety concepts define all measures against safety-critical failures and the allocation of safety requirements to the architecture elements. Main aspects of safety concepts (Acc. to Berner&Mattner [180]):

- Integration Systems Engineering / Functional Safety
- Clear Separation Functional / Technical Safety Concept
- Safety Functions as Entry Points for Technical Safety Concept
- Stepwise Refinement with Traceability
- Structuring the TSC according Technical Architecture
- Relation Matrix Failure Modes vs. Safety Mechanisms
- Establishing Safety Patterns
- Separation between different functions on same ECU

The Functional Safety Concept (FSC) is realized by allocating functional safety requirements to the system architecture. The Technical Safety Concept (TSC) is realized by deriving technical safety requirements from the functional safety requirements and by allocating them on the technical architecture (ECU/HW/SW).


Hierarchical System Modelling
Function / Requirement allocation
Hazard analysis / FMEA/ FTA
Safety Concept / Safety Requirements
D1: Function Modelling
D2: Functional Requirements Definition
S1: Hazard Analysis
S2: Functional Safety Concept
S3: Integration of Functional Safety Requirements
D3: Definition of Technical Solution
S4: Analysis of Failure Possibilities of Technical Solution (e.g.. FTA, FMEA)
S5: Technical Safety Concept
D4: Definition of Requirements to Parts
S6: Integration of Technical Requirements
Legend:
D = Development Activities
S = Safety Activities

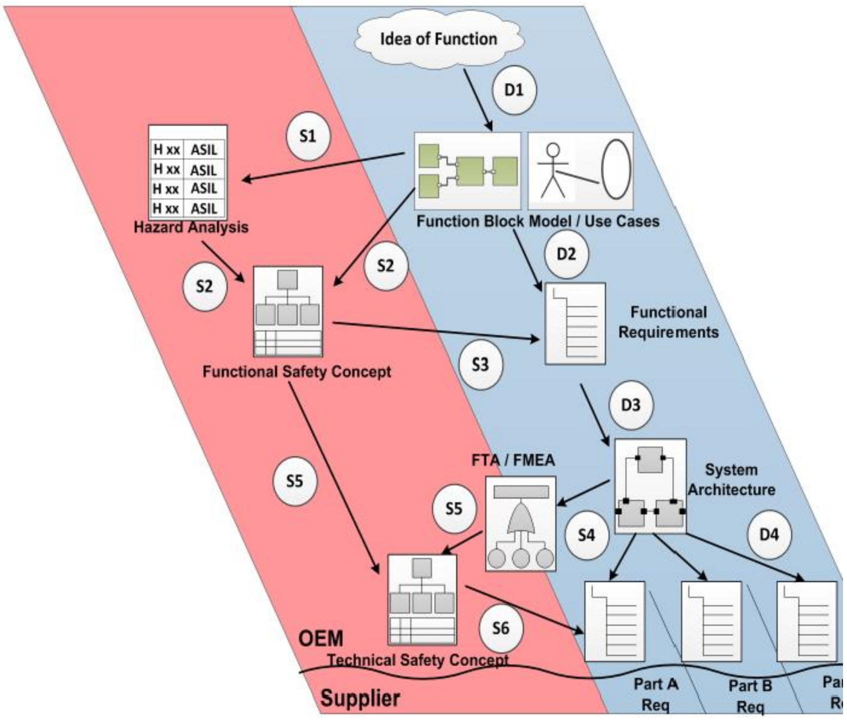


Figure 32: Approaches Towards Reusable Safety Concepts, 2012 Berner&Mattner, © Bernhard Kaiser

## 4.2 Railway domain

The main reference standard framework in the railway domain concerning an ERTMS/ETCS subsystem is defined by the European Directive 2008/57/EC on the interoperability of the rail system within the Community and the related Technical Specification for Interoperability (TSI) for Control-Command and Signalling sector (European commission decision 2012/88/EU of 25 January 2012 amended by the Decision 2015/14/EU of 05 January 2015). The TSI defines the requirements for interoperability and imposes among the others the safety assessment of the developed ERTMS systems with regards to the mandatory safety standards CENELEC EN 50126, EN 50128, EN 50129 and EN 50159-1 and EN 50159-2. These standards constitute the reference for all the stakeholders which act in the railway domain, like manufacturers, assessors, National Safety Authorities (NSAs), railway infrastructure managers, railway undertakings and so on.

Each of these standards presents in many parts a structure which, at least in some cases, allows the development of an architecture driven approach by the various stakeholders during the designing, manufacturing and assessment processes. With the scope to underline the state of practice on architecture driven assurance in the Railway domain, in the following paragraph some of these structures are highlighted in the following paragraphs.

The EN 50126 standard covers the specification and demonstration of safety for all railway applications and at all levels of such an application, as appropriate, from complete railway routes to major systems within a railway route, and to individual and combined sub-systems and components within these major systems, including those containing software and hardware. The standard also addresses Reliability, Availability, and Maintainability (RAM) as essential aspects of a railway system that contribute to safety.

In order to define a management process which will enable the control of RAMS factors specific to railway application, a system lifecycle is defined in this standard. The system lifecycle, shown in Figure 33, is a sequence of phases, each containing tasks, covering the total life of a system from initial concept through to decommissioning and disposal.

This lifecycle concept, which is fundamental to the successful implementation of this standard, can be identified as an example of system architecture approach. The lifecycle provides a structure for planning, managing, controlling and monitoring all aspects of a system, including RAMS, as the system progresses through the phases, in order to deliver the right product within the agreed time scales. The processes can be also tailored, provided that the modifications do not have any consequence on standard safety lifecycle and are well motivated. The lifecycle defined in EN50126 starting from the requirements, passing through the definition of the functions covering the requirements and the apportionment of them to the sub-functions and to the specific equipment is a typical example of a structured architecture driven approach. Following the lifecycle, the design of a system/sub-system/generic product starts from the highest level and it is then developed until the lowest level by respecting the architecture flow in a typical top-down approach.

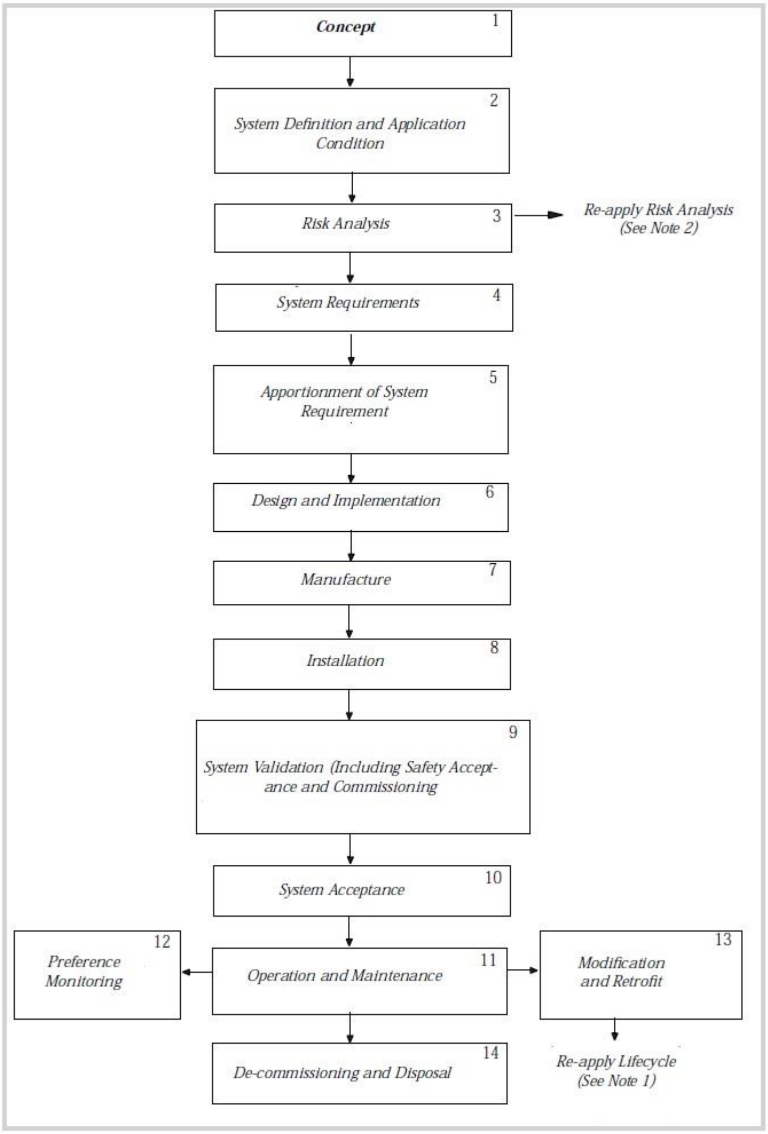

Concept 1
System Definition and Application Condition 2
Risk Analysis 3
Re-apply Risk Analysis (See Note 2)
System Requirements 4
Apportionment of System Requirement 5
Design and Implementation 6
Manufacture 7
Installation 8
System Validation (Including Safety Accept-ance and Commissioning 9
System Acceptance 10
Operation and Maintenance 11
Preference Monitoring 12
Modification and Retrofit 13
Re-apply Lifecycle (See Note 1)
De-commissioning and Disposal 14

**Figure 33: Lifecycle defined in EN 50126 standard**

The EN 50128 standard defines the requirements for software design and creation intended to be used in safety critical application belonging to railway domain. Among the requirements defined in the standard an

architectural approach can be identified in the definition of the reference software architecture: for example, the standard prescribes a creation of modular software where all the functional blocks shall be created, verified and validated.
This model defines the approach for the manufacturer for the creation of the software and provides a valid method of evaluation that is able to guide the assessor in the assessment of the product both at general and specific application level.

The EN 50129 is the first European Standard defining requirements for the acceptance and approval of safety-related electronic systems in the railway signalling field. It defines how the conditions for safety acceptance and approval shall be presented. The conditions shall cover three major themes:

- Quality Management;
- Safety Management;
- Functional and Technical Safety.

The documentary evidence that these conditions have been satisfied shall be included in a structured safety justification document defined as the Safety Case.

Acceptance by qualified organisations and national regulatory bodies of the Safety Case, through the activities of approval, assessment, and cross acceptance, is the ultimate step allowing the railway system to enter passenger service.

Also in this case the approach required by the standard is modular: there is an emphasis on the model of the system as broken down into sub-systems and constituent components (generic products), associated for the purpose of safety assurance to a tree-like structure of safety cases embedded within each other. In the Figure 34, an example of a possible hierarchy of Safety Cases is represented. Each Safety Case is associated to one of the following categories:

- Generic Product Safety Case: addresses a core product used in Generic applications;
- Generic Application Safety Case: addresses standard developments common to several projects;
- Specific Application Safety Case: addresses specific data configuration and specific installation for a dedicated project

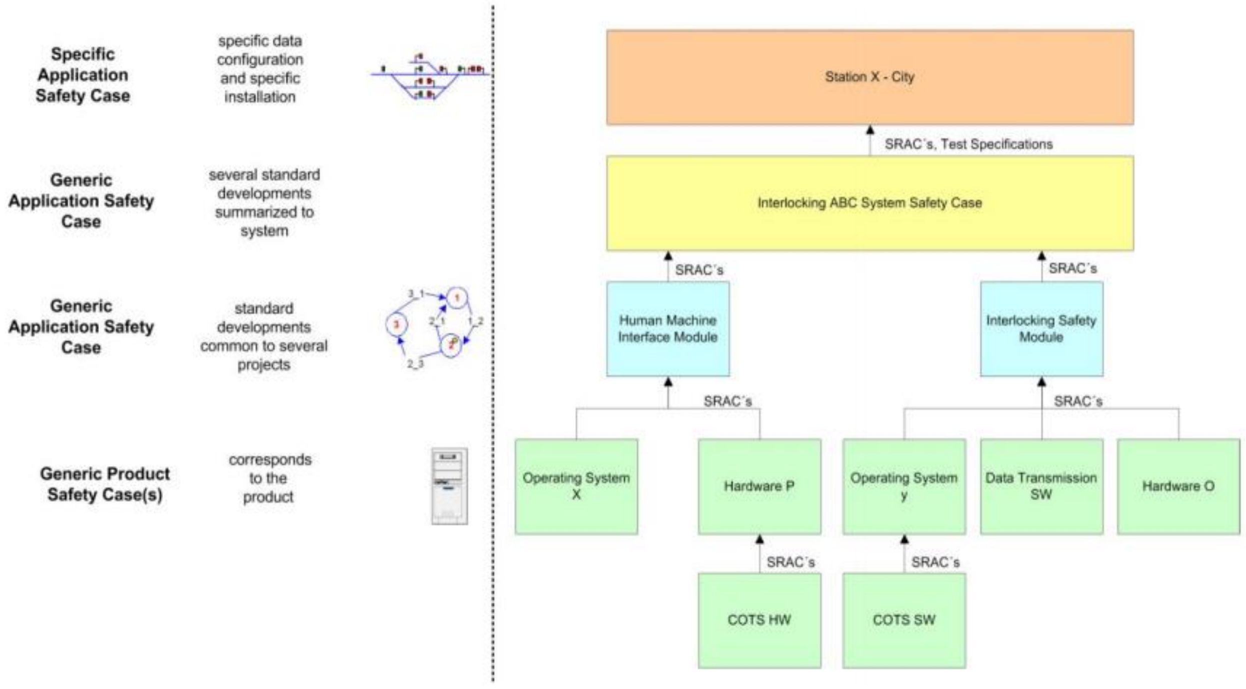

Specific Application Safety Case
specific data configuration and specific installation
Generic Application Safety Case
several standard developments summarized to system
Generic Application Safety Case
standard developments common to several projects
Generic Product Safety Case(s)
corresponds to the product
Station X - City
SRAC´s, Test Specifications
Interlocking ABC System Safety Case
SRAC´s
SRAC´s
Human Machine Interface Module
Interlocking Safety Module
SRAC´s
SRAC´s
Operating System X
Hardware P
Operating System y
Data Transmission SW
Hardware O
SRAC´s
SRAC´s
COTS HW
COTS SW

**Figure 34: Example of a possible hierarchy of Safety Case**

Another example of system architecture approach can be found in the technique of Fault Tree Analysis (FTA) prescribed by the EN50129 standard.

A safety-related system/sub-system/equipment requires an independent assessment by an approved Independent Safety Assessor (ISA). The objective of assessment is to arrive at a judgement that the product or the system (subsystem, equipment) is of the defined safety integrity level (SIL) based on credible evidence and is fit for its intended purpose.

A basic parameter to evaluate the SIL is the Tolerable Hazard Rate (THR). The standard indicates as possible technique to determine the THR value the Fault Tree Analysis (FTA) which is based on a typical top-down architecture starting from the Top Hazard descending down to the component level. The Figure 35 represents an example of FTA scheme with the Top Hazard identified by an undue door opening.

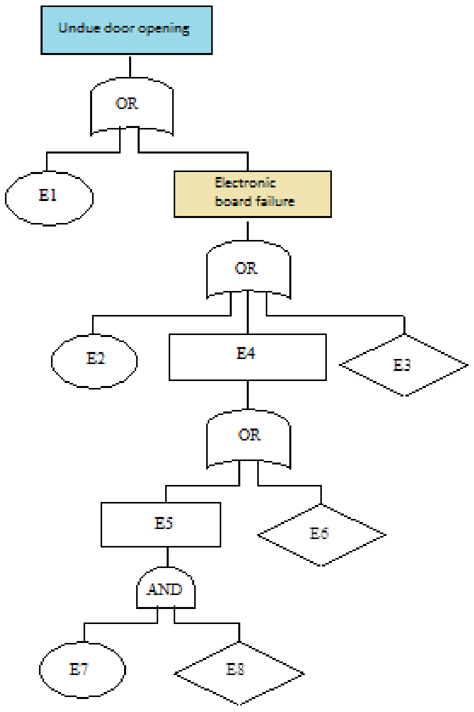


**Figure 35: Example of Fault Tree Analysis (FTA)**

At the time being, the modelling efforts concerning the railway domain reference framework are oriented in two directions:

- Modelling of the high level reference standards constituted by the CENELEC norms EN50126, EN50128 and EN50129;
- Modelling of specific standards for railway subsystems or applications, in which, due to their relevance and complexity, the need of modelisation is present.

Concerning the first point, a unique and common model of the CENELEC standards is not well diffused among the railway domain stakeholders. In this direction the European project OPENCOSS project studied a modelling of the EN 50129 standard with the scope to define a model of the safety case based on CCL (Common Certification Language). This would help manufacturers and assessors in their activities with a final reduction of time and costs. Concerning the second point, an example could be found in the UNISIG SUBSET 026 “ERTMS/ETCS System Requirements Specification”. The present modelling activity is based on formal methods and UML languages: in particular, it could be of interest the development of a language able to model specific on-board functions as defined in the subset. The final purpose would consist in making testing and certification process easier and faster, reducing the possible different interpretations of the standards.

## 4.3 Avionics domain

ARP4761 (Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment) is and Aerospace Recommended Practice. Jointly with ARP4754A (Guidelines for Development of Civil Aircraft and Systems), it constitutes recommended practices for development and safety assessment processes for the avionics domain.
Relation of these ARPs to software and hardware development guidelines are depicted in the following Figure 36:

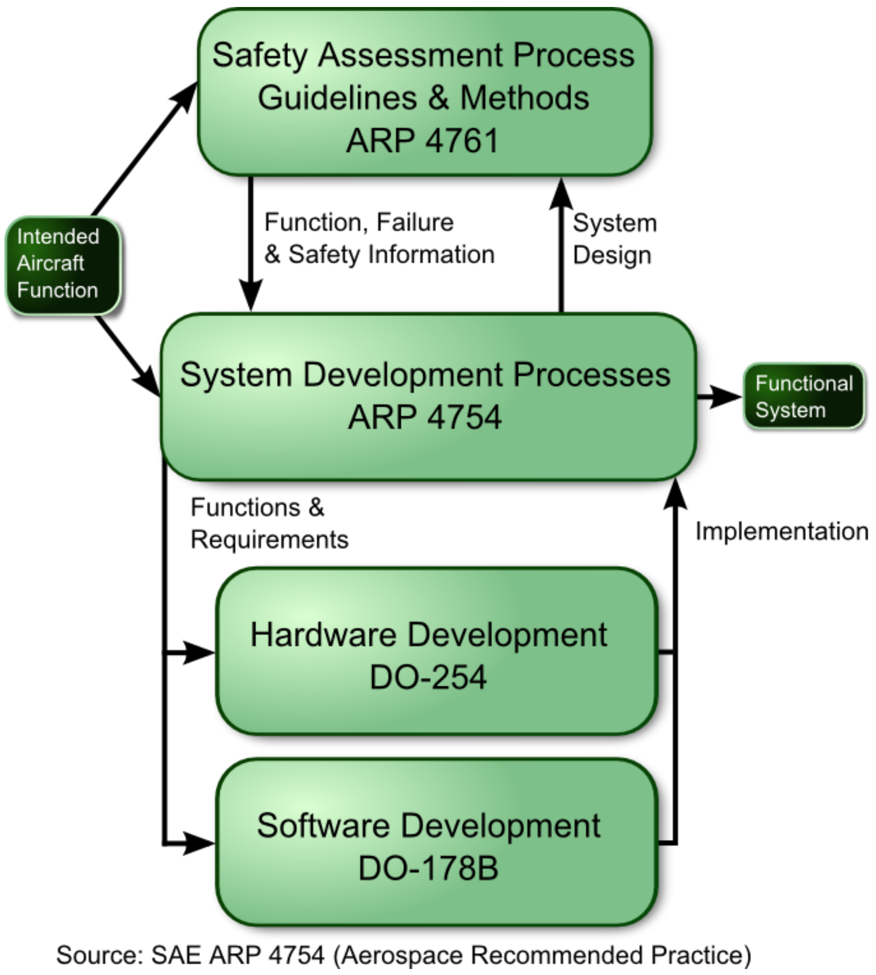

Safety Assessment Process Guidelines & Methods ARP 4761
Intended Aircraft Function
Function, Failure & Safety Information
System Design
System Development Processes ARP 4754
Functional System
Functions & Requirements
Implementation
Hardware Development DO-254
Software Development DO-178B
Source: SAE ARP 4754 (Aerospace Recommended Practice)

**Figure 36: Relation of ARP4761 and ARP4754A software and hardware development guidelines**

The main methods covered by the aerospace recommended practices:

- Functional Hazard Assessment (FHA)
- Preliminary System Safety Assessment (PSSA)
- System Safety Assessment (SSA)
- Fault Tree Analysis (FTA)
- Failure Mode and Effects Analysis (FMEA)
- Failure Modes and Effects Summary (FMES)
- Common Cause Analysis (CCA)
    - Zonal Safety Analysis (ZSA)

- Particular Risks Analysis (PRA)
- Common Mode Analysis (CMA)

The standard flow of the safety process based on the ARP4761 is as follows:

1. Perform the aircraft level FHA in parallel with development of aircraft level requirements.
2. Perform the system level FHA in parallel with allocation of aircraft functions to system functions, and initiate the CCA.
3. Perform the PSSA in parallel with system architecture development, and update the CCA.
4. Iterate the CCA and PSSA as the system is allocated into hardware and software components.
5. Perform the SSA in parallel with system implementation, and complete the CCA.
6. Feed the results into the certification process.

System development and safety assessment and analysis in practice are performed mostly manually. When Model-Based System Engineering is performed usually, some of the following modelling environments are used:

1. Mathworks Simulink or SCADE – for behavioral system design
2. Enterprise Architect or IBM Ratinonal (SysML/UML) – for system, requirements and architecture
3. AADL/SysML – for platform configuration/analysis

## 4.4 Space domain

### 4.4.1 Avionics Architecture Modelling Language (AAML)

The ESA AAML (Avionics Architecture Modelling Language) study aims at advancing the avionics engineering practices towards a model based approach. The AAML consortium is led by GMV in collaboration with ESA and Thales Alenia Space.

In the space domain, the definition of the system architecture is usually based on:

- The architect’s expertise and background
- The avionics specific analyses

Traditionally, each type of analysis is based on a dedicated model (sometimes ad-hoc to solve a particular problem). The AAML study [164] proposes the usage of a single architecture that can be used to perform different analysis in avionics, covering most of the phases of the development life-cycle.

Examples of analyses of interest for the avionics design in the different phases of development are:

- Satellite mode definition, RAMS, FDIR and autonomy concept
- Design consistency and correctness checks
- Commandability and observability
- Bus load and data latency
- Space/ground communication
- Avionics resources
- On-board functions and performance
- Power and mass

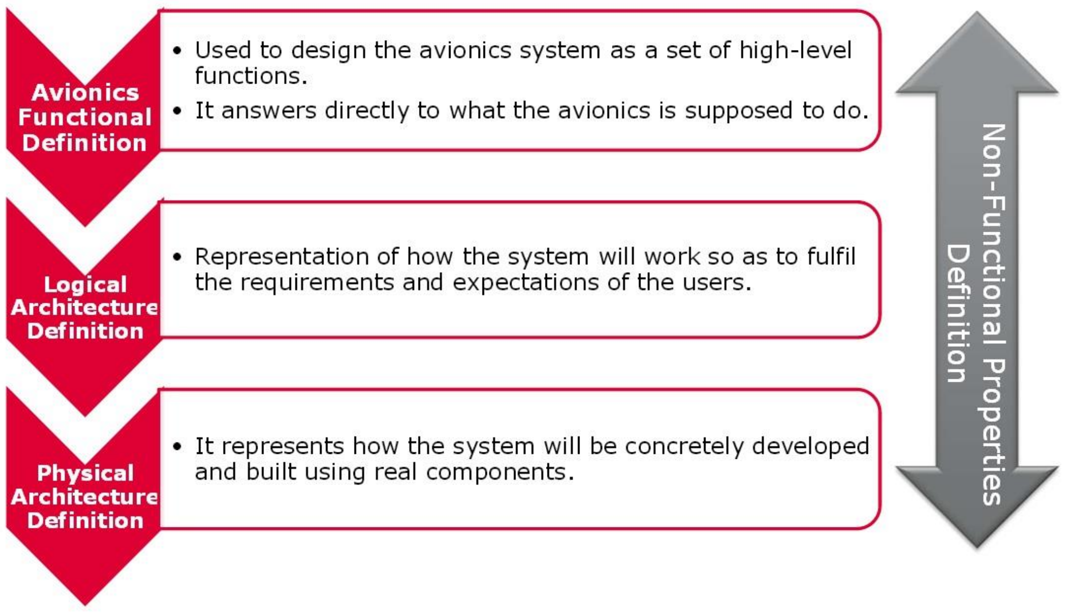

Avionics Functional Definition
• Used to design the avionics system as a set of high-level functions.
• It answers directly to what the avionics is supposed to do.
Logical Architecture Definition
• Representation of how the system will work so as to fulfil the requirements and expectations of the users.
Physical Architecture Definition
• It represents how the system will be concretely developed and built using real components.
Non-Functional Properties Definition

**Figure 37: AAML Modelling Process**

To model the avionics system, the AAML language defines:

- Entities (general entities, data model, avionic entities (e.g., avionic function, logical component, physical components), communication entities (e.g., operation), interaction entities (e.g., interface port), state modelling.
- Non-functional properties to describe the entities in the various non-functional dimensions of interest.

## 4.4.2 System architecture in the development process

The system architecture is discussed during the System Architecture Review. The main objectives of the SAR are the followings:

- Confirm the proper translation and allocation of the subsystem requirements and all lower level component to a consistent architecture.
- Verify the completeness, adequacy, and consistency of the design and development plan and associated Assembly, Integration and Verification (AIV) plan with the development schedule, considering the long lead item procurement.
- Verify the compliance of the design with the applicable requirements.
- Verify the adequacy and completeness of the verification and testing (qualification and acceptance) approach, addressing verification method, verification level, sequencing (to ensure coverage at minimum effort and risk) and planning.
- Verify that the proposed preliminary design is technically feasible and the predicted performances meet the requirements. Confirm the adequacy of the allocation of the performances, budgets and margins to the various components and its compliance with the requirements.
- Review the adequacy of all Product Assurance (PA) programs including quality assurance, safety, reliability, EEE parts, materials and process and cleanliness contamination control.
- Verify completeness, adequacy and consistency of the internal and external interfaces. Verify the validity of the operational concepts at a level of detail commensurate with the Preliminary Design Review (PDR).
- Confirm the completeness, in terms of technical and programmatic aspects, of the risk assessment and confirm adequacy of proposed mitigation actions.

- Confirm the design will be producible and maintainable within achievable and committed cost and schedule objectives

### 4.4.3 Applicable standards

Applicable to software products, the European Space Agency (ESA) has developed standards (ECSS series) defining processes for software engineering (ECSS-E-ST-40C) and software quality assurance (ECSS-Q-ST-80C). These processes are based on a series of customer-client meetings where the documentation related to design, analysis and test of the SW product is reviewed in-depth. The standard specifies the transformation of software requirements into a software architecture that:

- describes its top–level structure
- identifies the software components, ensuring that all the requirements for the software item are allocated to its software components and later refined to facilitate detailed design
- covers as a minimum hierarchy, dependency, interfaces and operational usage for the software components
- documents the process, data and control aspects of the product
- describes the architecture static decomposition into software elements such as packages, classes or units
- describes the dynamic architecture, which involves the identification of active objects such as threads, tasks and processes
- describes the software behaviour

As part of the software quality assurance, ECSS defines the requirements for the dependability (ECSS-Q-ST-30C) and safety (ECSS-Q-ST-40C) assurance programmes in space projects. Dependability and safety analysis techniques are tailored according to the space mission characteristics. The traditional techniques used in the space domain are:

- Functional Hazard Assessment (FHA)
- Fault Tree Analysis (FTA)
- Failure Mode and Effects Analysis (FMEA)

## 4.5 Industrial automation domain

Over the last years, there have been a number of initiatives worldwide to develop guidelines and standards to enable the safe exploitation of programmable electronic systems used for safety applications. In the context of industrial applications, a major initiative has been focused on IEC 61508 and this standard is emerging as a key international standard in the industrial automation domain.

IEC 61508 is the standard governing the functional safety of programmable electronic systems. This standard is well established in the industrial process control and automation industry and it is finding a foothold in other fields where safety and reliability are important. IEC 61508 covers functional safety of safety-related systems that use electrical and/or electronic and/or programmable electronic (E/E/PE) technologies. The standard applies to these systems irrespective of their application.

IEC 61508 specifies everything that should be analysed to become able to develop the Functional Safety Management system. It also provides the sequence in which these phases should be executed. Figure 38 shows all parts in which IEC 61508 is laid out as well as the relationship among these parts. The technical requirements are clearly distinguished from the non-technical ones:

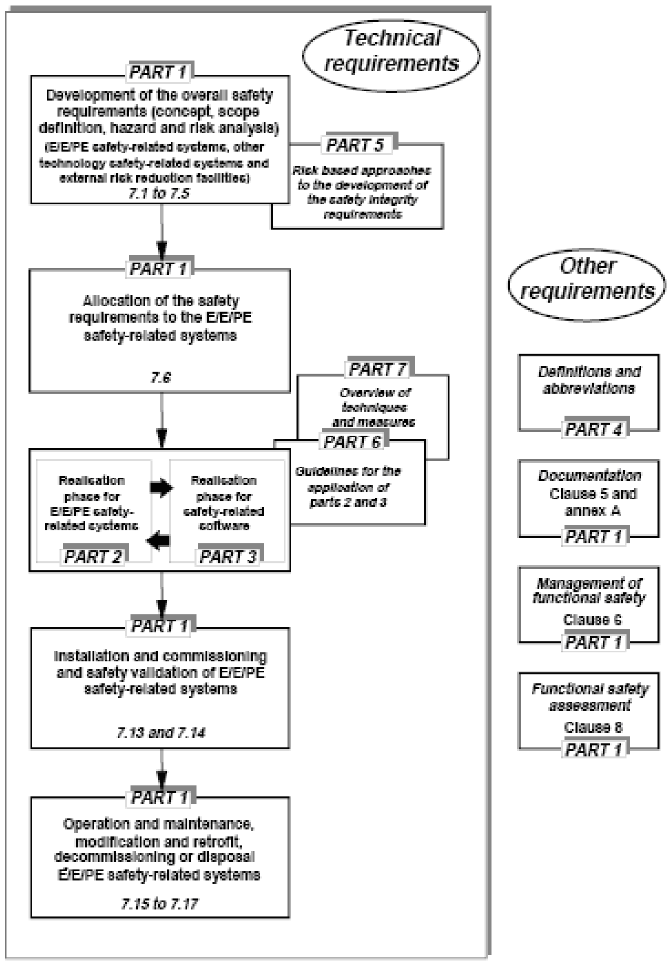

Technical requirements
PART 1
Development of the overall safety requirements (concept, scope definition, hazard and risk analysis) (E/E/PE safety-related systems, other technology safety-related systems and external risk reduction facilities) 7.1 to 7.5
PART 5
Risk based approaches to the development of the safety integrity requirements
PART 1
Allocation of the safety requirements to the E/E/PE safety-related systems 7.6
PART 7
Overview of techniques and measures
PART 6
Guidelines for the application of parts 2 and 3
Realisation phase for E/E/PE safety-related systems
PART 2
Realisation phase for safety-related software
PART 3
PART 1
Installation and commissioning and safety validation of E/E/PE safety-related systems 7.13 and 7.14
PART 1
Operation and maintenance, modification and retrofit, decommissioning or disposal E/E/PE safety-related systems 7.15 to 7.17
Other requirements
Definitions and abbreviations
PART 4
Documentation Clause 5 and annex A
PART 1
Management of functional safety Clause 6
PART 1
Functional safety assessment Clause 8
PART 1

**Figure 38: Overall framework of IEC 61508**

IEC 61508 schedule would be as follows:

- IEC 61508-1: “General requirements”
- IEC 61508-2: “Requirements for electrical/electronic/programmable electronic safety-related systems”
- IEC 61508-3: “Software requirements”
- IEC 61508-4: “Definitions and abbreviations”
- IEC 61508-5: “Examples of methods for the determination of safety integrity levels”
- IEC 61508-6: “Guidelines on the application of IEC 61508-2 and IEC 61508-3”
- IEC 61508-7: “Overview of measures and techniques”

The IEC 61508 standard specifies the risk assessment and the measures to be taken in the design of safety functions for the avoidance and control of faults. In fact, IEC 61508 provides a complete safety life cycle that accounts for possible risk of physical injury and damage to the environment. Acceptable levels of risk are determined and procedures for residual risk management over time are established.

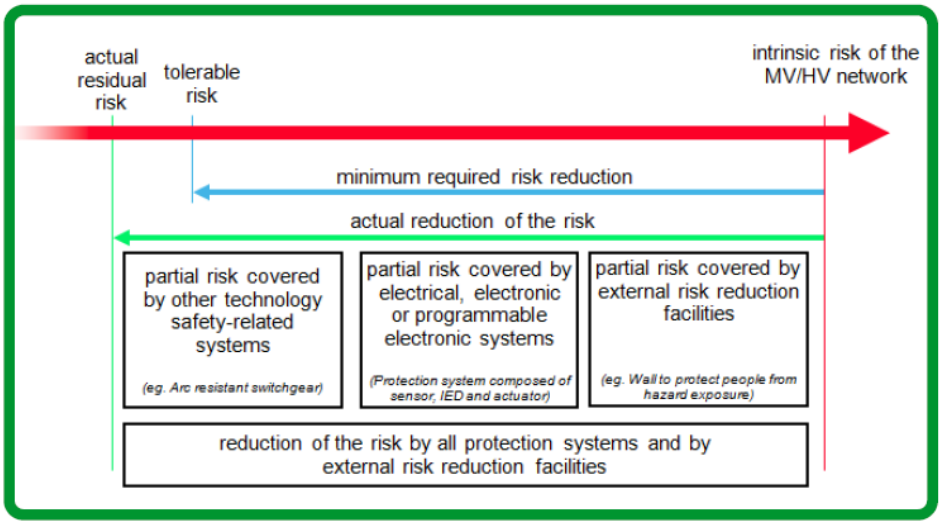
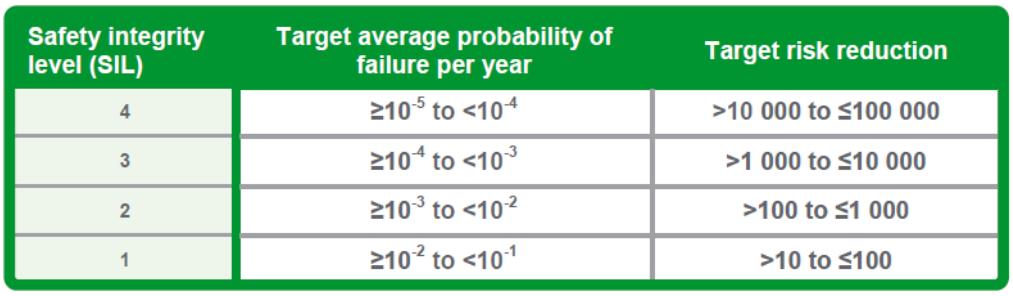

actual residual risk
tolerable risk
intrinsic risk of the MV/HV network
minimum required risk reduction
actual reduction of the risk
partial risk covered by other technology safety-related systems
(eg. Arc resistant switchgear)
partial risk covered by electrical, electronic or programmable electronic systems
(Protection system composed of sensor, IED and actuator)
partial risk covered by external risk reduction facilities
(eg. Wall to protect people from hazard exposure)
reduction of the risk by all protection systems and by external risk reduction facilities

**Figure 39: Functional safety and risk reduction [185]**

The standard also requires that hardware be designed to tolerate a certain level of random hardware faults, and to demonstrate safe operation in harsh environments. It also calculates the probability of failure of each safety function.

In order to achieve the necessary Safety Integrity Level (SIL), the standard requires a proof of residual risk, which is based on the probability of dangerous failure. The calculation is based on the equipment components that influence the entire safety loop. The failure probabilities of each component are considered together so that the safety level of the holistic architecture can be determined.



| Safety integrity level (SIL) | Target average probability of failure per year | Target risk reduction |
|---|---|---|
| 4 | ≥$10^{-5}$ to <$10^{-4}$ | >10 000 to ≤100 000 |
| 3 | ≥$10^{-4}$ to <$10^{-3}$ | >1 000 to ≤10 000 |
| 2 | ≥$10^{-3}$ to <$10^{-2}$ | >100 to ≤1 000 |
| 1 | ≥$10^{-2}$ to <$10^{-1}$ | >10 to ≤100 |

**Figure 40: Safety integrity level (SIL) estimates the probability of failure [185]**

The standard is quite comprehensive and addresses hardware failures, software failures, systematic failures, and environmental and operational failures. The standard recommends a set of techniques and measures for controlling these failures. The standard also provides requirements regarding development methods, competence of the project team, project management, change management, tracking of requirements, and documentation.

# 5. Consolidation and Way Forward

This chapter is aimed at indicating a way forward that enables a co-evolution of the state of the art (SOTA) together with the state of practice (SOPT) on Architecture-Driven Assurance. We also look into the possible gaps between SOTA and SOTP. The way forward is based on the findings obtained via the investigation of the current status in academy and in industry. This chapter is structured according to the objectives defined in Section 2.

***System Architecture Modelling for Assurance***

Information related to system architecture modelling, and more generally system modelling, has been presented when reviewing the state of the art in Sections 3.2, 3.5, and 3.7. The languages analysed deal mainly with the specification of the requirements, architecture and its components, and design of a system. When having a look at the state of the practice in Section 4, it appears that further artefact types will need to be considered when dealing with system architecture modelling for assurance. First, other artefacts, e.g. safety analysis results, either constrain architecture specification or are influenced by it. Second, confirmation measures about system and architecture properties, e.g. in the form of verification results, must be provided for compliance with the applicable standards. Therefore, system architecture modelling for assurance will not only have to deal with architecture models, and the scope of the work might need to be extended, when compared to the scope initially envisioned. This aspect will have to be carefully analysed as part of the work on case study, requirements, and design specification in AMASS. A clear understanding of the needs from the case studies is necessary, as well as of their constraints. Requirements from standards on system modelling must also be thoroughly studied, focusing on only a given phase (e.g. architectural design) but taking into account several ones (e.g. also verification).

It appears that there is currently a trend towards extending modelling languages (e.g. SysML) to better and explicitly support the concepts and needs from assurance standards. This is also in line with the stated need in OPENCOSS CCL for better relating it with component and system models for safety-critical systems, such as those from SafeCer and CHESS. How to link assurance models and system has received some attention in the literature, but not much in our opinion. The current solutions are also ad-hoc, tailored to specific standards. More general solutions seem necessary. Based on prior work, a generic UML profile-based approach could be suitable. It will also be necessary to select the system modelling languages to extend and link with assurance models. Standard languages, and especially languages used in the case studies, are the main candidates.

Ontologies are already successfully applied in practice for requirements quality assurance, as a concrete example of V&V-based assurance in compliance with the applicable standards. We conjecture that a similar approach could be adopted in system architecture modelling for assurance. Ontologies could be used as basis and support for certain analysis, e.g. completeness, correctness, and consistency of system models. It also seems interesting to study how to combine ontologies and formal methods for system model analysis. The approaches have been shown to be powerful and address complementary assurance needs.

Finally, the effective link between (1) system architecture models and other related artefacts, and (2) assurance (or safety) cases is another area to address in system architecture modelling for assurance. The provision of an assurance case is a requirement in many standards, and it must be ensured that the work on system architecture modelling allows a system supplier to assure system dependability (safety, security, etc.), comply with the applicable standards, and demonstrate both properties.

***Assurance Patterns Library Management***

In 3.3.1 we have highlighted the importance of pattern-based design and, specifically, the one related to fault tolerance. Most of the work done so far refers to the development of argumentation patterns for safety architectures. However, further investigation needs to be carried out to develop a more enhanced argumentation library which covers not only safety argumentation patterns but also some other aspects such as security. An example might be an argumentation pattern regarding EVITA hardware security module for the automotive domain. As pointed out, AMASS will also investigate on argumentation patterns such as security controls or novel architectures, which take in account both, safety and security concepts.

***Assurance of Specific Technologies***

The need of revising assurance and certification activities concerning novel technologies has been reviewed in subsection 3.4. Several complex electronics devices such as MPSoC have been analysed and the main concerns w.r.t. multicore certification issues mentioned. As pointed out, MPSoCs have a tremendous potential in the domain of embedded systems due to its energy efficiency and computational capacity, however, their use in safety-critical applications still needs further research. The difficulties in the certification process due to the high complexity of these kinds of systems, the lack of temporal determinism, and problems related to error propagation between subsystems should be investigated.

Besides, we have identified some of the main challenges concerning the assurance of adaptive systems. This poses a challenge regarding determinism and safety, as in principle it is not known what function will run and where. Although some ISO 26262 requirements that might need modifications have been presented, more work should be done in this area and in other safety-critical domains.

To sum up, several standard requirements might need to be adapted or modified to include the special requirements that novel technologies demand. This includes not only new specific requirements but also novel V&V techniques. At the same time, argumentation patterns of several concerns will be further investigated and developed in AMASS to facilitate the reuse of specific technologies.

***Contract-Based Assurance Composition***

In subsection 3.1, we have presented different contract-based approaches based on temporal logics and agreement among components. We have also reviewed some formalism to specify and analyse contracts and the related tool supports implemented in different projects like SafeCer and OPENCOSS. We also studied the extension of some of the approaches to cover safety analysis (e.g., FTA).

We have observed some connections regarding the needs summarized in section 4 and some of these techniques. For example, different standards like EN50129, ARP4761, and ARP4754A recommends practices for supporting Fault Tree Analysis (FTA). An interesting case study is reported in [19], which shows how formal methods can be applied to model and analyse the AIR6110 Wheel Brake System. The formal modelling and analysis are based on integration of different techniques like the contract-based approach supported by OCRA (which allows to reuse both models and contracts), architectural decomposition, model-based safety analysis (XSAP [89]), and contract-based safety analysis. We can observe that a link to the standard and the related documentation is missing.

Similarly, standard architectures (such as AUTOSAR in the automotive industry, IMA in avionics, ETCS in railway) require some safety/security *architectural patterns* definition and application (3-level-monitoring, E2E protection, and partitioning, among others), and auto-generation of platform models and configurations based on these patterns (e.g. for AUTOSAR and IMA). The use of patterns speeds architecture specification and facilitates the (re)use of components targeted at being used in such patterns.

The architecture can be enriched with contracts that formalize the functional requirements to ensure that the system responds correctly to some safety requirements. For instance, an AUTOSAR case study is presented in [90], where a contract-based approach has been used to formalize the safety requirements to detect communication failures. A way forward related to this is that we can build on this case study to create a pattern of this E2E protection with contracts, and then it can be used inside another system.

***V&V-based Assurance***

The state of the art on the development of embedded systems relies heavily on the use of requirements. As Section 3.2 describes, requirements can systematically built and authorized using ontologies, have to be checked for sanity, can be used to generate contracts and automated test cases, and must be later verified against the system design or executable code. The requirements are tracked during the whole development lifecycle to increase the assurance that the needs of customers will be satisfied. Automating many of these tasks is the primary goal of V&V-based assurance, yet it needs that the requirements are formalized first. In addition, as Section 4.3 shows, current practices allow automatic formalization of a considerable portion of manually written requirements.

Once the requirements are formalized, a number of verification methods are applicable to detect problems or demonstrate correctness, thus improving the assurance (see Section 3.7). Since the scale of the system under verification always limits formal verification (number of requirements checked for sanity, size of the design, lines of code, etc.), each improvement in requirements formalization will need adequate improvement in respective verification methods.

On the other hand, even though high automation in V&V is currently available, many of the current V&V processes are performed manually in practice. This suggests that the imminent goal should be to automate as many of the V&V process as possible in order to reduce the development time and cost. The prerequisite to enable V&V automation are formalized specification, system architecture, and system design. Once formalized, these artefacts have clear, unambiguous semantics, and can thus be read and processed by a machine.

In the area of requirements engineering, the formal requirements are not yet standardized and therefore most of the requirement authoring tools offer customization so that every company can use their own requirement standard. The automatic generation of certifiable requirement-based tests is being developed and formal methods increase the coverage of those tests. Formal techniques are especially useful in requirements engineering since many V&V objectives are impossible to be achieved by testing – for example: completeness, correctness, and unambiguity. Therefore, the AMASS assurance approach will make sure that evidence is provided to argue that a requirements specification is valid, as part of the assurance case.

In the area of system architecture, contract-based assurance as described in the previous sections is one of the ways forward and formalized requirements will greatly simplify the production and tracking of those contracts.

For system design, the formal methods are especially useful in at least three areas. First, the V&V of critical parts of the system – for example control subsystem or Stateflow charts – for which exhaustive verification in the form of model checking can be employed. Again, the scalability in the size of those systems is one of the goals for AMASS. Second, the verification of safety requirements, where the scalability issue is also the aspect to be improved. Third, the verification of untestable requirements, e.g. those that would require infinitely long tests, which are currently verified by manual analysis only.

# Abbreviations and Definitions

| | |
|---|---|
| AADL | Architecture Analysis and Design Language |
| ADA | Architecture-Driven Assurance |
| ARTEMIS | ARTEMIS Industry Association is the association for actors in Embedded Intelligent Systems within Europe |
| AUTOSAR | AUTomotive Open System Architecture |
| CBD | Contract-Based Design |
| CCA | Common Cause Analysis |
| CCL | Common Certification Language |
| CHESSML | CHESS Modelling Language |
| CL | Common Logic |
| CPS | Cyber-Physical Systems |
| DAF | Dependability Assurance Framework for Safety-Sensitive Consumer Devices |
| EAB | External Advisory Board |
| ECSEL | Electronic Components and Systems for European Leadership |
| FMEA | Failure Modes and Effects Analysis |
| FMEDA | Failure Modes, Effects and Diagnostic Analysis |
| FPTC | Failure Propagation and Transformation Calculus |
| FPTN | Failure Propagation and Transformation Notation |
| FTA | Fault Tree Analysis |
| IMA | Integrated Modular Avionics |
| KDM | Knowledge Discovery Metamodel |
| KE | Knowledge Element |
| MARTE | Modelling and Analysis of Real Time and Embedded systems |
| MBSA | Model-Based Safety Analysis |
| MCS | Minimal Cut Sets |
| ODM | Ontology Definition Metamodel |
| OMG | Object Management Group |
| OWL | Web Ontology Language |
| RAS | Reusable Asset Specification |
| RDF | Resource Description Framework |
| SAE | Society of Automotive Engineers |
| SBVR | Semantics of Business Vocabulary and Rules |
| SMM | Structured Metrics Metamodel |
| SMV | Symbolic Model Verifier |
| SVP | Subject + Verb + Predicate |
| SysML | System Modelling Language |
| TFPG | Timed Failure Propagation Graphs |
| UML | Unified Modelling Language |
| V&V | Verification and Validation |
| WP | Work Package |

# References

[1] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple Viewpoint Contract-Based Specification and Design. In FMCO, pages 200–225, 2007.
[2] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K.G. Larsen. Contracts for System Design. Technical Report RR-8147, INRIA, November 2012.
[3] A. Benveniste, B. Caillaud, and R. Passerone. A Generic Model of Contracts for Embedded Systems. Technical report, INRIA, 2007.
[4] S.S. Bauer, A. David, R. Hennicker, K.G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Moving from Specifications to Contracts in Component-Based Design. In FASE, pages 43–58, 2012.
[5] M. Broy. Towards a Theory of Architectural Contracts: Schemes and Patterns of Assumption/Promise Based System Specification. In Software and Systems Safety Specification and Verification, pages 33–87. 2011.
[6] D.D. Cofer, A. Gacek, S.P. Miller, M.W. Whalen, B. LaValley, and L. Sha. Compositional Verification of Architectural Models. In NFM, pages 126–140, 2012.
[7] A. Cimatti and S. Tonetta. A Property-Based Proof System for Contract-Based Design. In SEAA, 2012.
[8] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand. Using contract-based component specifications for virtual integration testing and architecture design. In DATE, pages 1023–1028, 2011.
[9] S. Graf, R. Passerone, and S. Quinton. Contract-Based Reasoning for Component Systems with Complex Interactions. In TIMOBD’11, 2011.
[10] B. Meyer. Applying ”Design by Contract”. Computer, 25(10):40–51, 1992.
[11] A. Pnueli. The temporal logic of programs. In Proceedings of 18th IEEE Symp. on Foundation of Computer Science, pages 46–57, 1977.
[12] S. Quinton and S. Graf. Contract-Based Verification of Hierarchical Systems of Components. In SEFM, pages 377–381, 2008.
[13] A.L. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. Eur. J. Control, 18(3):217–238, 2012.
[14] Marco Bozzano, Alessandro Cimatti, Cristian Mattarei, Stefano Tonetta. Formal Safety Assessment via Contract-Based Design. In ATVA, pages 81-97, 2014.
[15] Jiri Barnat, Petr Bauch, Lubos Brim: Checking Sanity of Software Requirements. SEFM 2012: 48-62
[16] Alessandro Cimatti, Marco Roveri, Angelo Susi, Stefano Tonetta: Validation of requirements for hybrid systems: A formal approach. ACM Trans. Softw. Eng. Methodol. 21(4): 22 (2012)
[17] Angelo Chiappini, Alessandro Cimatti, Luca Macchi, Oscar Rebollo, Marco Roveri, Angelo Susi, Stefano Tonetta, Berardino Vittorini: Formalization and validation of a subset of the European Train Control System. ICSE (2) 2010: 109-118
[18] OASIS committee, "OASIS Service Oriented Architecture Reference Model TC", https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm , Dec 2012.
[19] M. Bozzano, A. Cimatti, A. F. Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta, “Formal design and safety analysis of AIR6110 wheel brake system,” in CAV, 2015: 518-535.
[20] Alessandro Cimatti, Michele Dorigatti, Stefano Tonetta: OCRA: A tool for checking the refinement of temporal contracts. ASE 2013: 702-705
[21] Ewen Denney and Ganesh Pai, "A Formal Basis for Safety Case Patterns", In Proceedings of the 32nd International Conference on Computer Safety, Reliability and Security (SafeComp '13), Toulouse, France, September 2013.
[22] J. Rushby: Noninterference, Transitivity, and Channel-Control Security Policies. Tech. Report SRI-CSL-92-02, December 1992.
[23] I. Crnkovic and M. Larsson, Building reliable component-based software systems, Norwood, MA, USA: Artech house, 2002.
[24] A. Benveniste, W. Damm, A. Sangiovanni-Vincentelli, D. Nickovic, R. Passerone and P. Reinkemeier,

"Contracts for the Design of Embedded Systems Part I: Methodology and Use Cases," Submitted for publication, 2011.
[25] A. Benveniste, J.-B. Raclet, B. Caillaud, D. Nickovic, R. Passerone, A. Sangiovanni-Vincentelli, T. Henzinger and K. G. Larsen, "Contracts for the Design of Embedded Systems - Part II: Theory," Submitted for publication, 2011.
[26] Mazzini S., Puri S., Veran G., Vardanega T., Panunzio M., Santamaria C., Zovi A.: Model-Driven and Component-Based Engineering with the CHESS Methodology. Proc. of DASIA Conference, Malta, May 2011.
[27] CHESS Polarsys Eclipse project: https://www.polarsys.org/chess/
[28] SafeCer Deliverable D132.2, Generic component meta-model v1.0, 2014-12-19
[29] S. Aboubekr, G. Delaval, R. Pissard-Gibollet, É. Rutten, and D. Simon, "Automatic Generation of Discrete Handlers of Real-Time Continuous Control Tasks," in Proceedings of the 18th IFAC World Congress, 2011, vol. 18.
[30] P. Boström, "Contract-Based Verification of Simulink Models," in ICFEM, 2011, vol. 6991, pp. 291–306.
[31] A. Baumgart, E. Böde, M. Büker, W. Damm, G. Ehmen, T. Gezgin, S. Henkler, H. Hungar, B. Josko, M. Oertel, and others, "Architecture modelling," OFFIS eV, Oldenburg, Technical Report, 2011.
[32] E. Hull, K. Jackson, and J. Dick, Requirements Engineering. Springer, 2004.
[33] C. Kühl, "Formalisierung von Requirements durch Nutzung von Templates," Master Thesis Freie Universität Berlin, 2010.
[34] A. Pnueli, "The temporal logic of programs," in Foundations of Computer Science, 1977., 18th Annual Symposium on, 1977, pp. 46–57.
[35] N. A. Lynch and M. R. Tuttle, "An Introduction to I/O Automata," CWI Quarterly, vol. 2, no. 3, pp. 219–246, 1989.
[36] P. Reinkemeier, I. Stierand, P. Rehkop, and S. Henkler, "A pattern-based requirement specification language: Mapping automotive specific timing requirements," in Software Engineering (Workshops), 2011, vol. 184, pp. 99–108.
[37] SPEEDS, "D.2.5.4 Contract Specification Language (CSL), available at: http://speeds.eu.com/downloads/D_2_5_4_RE_Contract_Specification_Language.pdf (retrieved on 03-Aug-2015)," 2008.
[38] I. Bate, R. Hawkins, and J. McDermid, "A Contract-based Approach to Designing Safe Systems," in Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software - Volume 33, 2003, pp. 25–36.
[39] J. Fenn, R. Hawkins, P. Williams, and T. Kelly, "Safety case composition using contracts-refinements based on feedback from an industrial case study," in The Safety of Systems, Springer, 2007, pp. 133–146.
[40] A. Baumgart, P. Reinkemeier, A. Rettberg, I. Stierand, E. Thaden, and R. Weber, "A Model-Based Design Methodology with Contracts to Enhance the Development Process of Safety-Critical Systems.," in SEUS, 2010, vol. 6399, pp. 59–70.
[41] B. Kaiser, S. Sonski, S. Buono, H. Petersen, and J. Zander, "Lightweight Contracts for Safety-Critical Automotive Systems," in Tagungsband 13. Workshop Automotive Software Engineering. INFORMATIK 2015 - 29.9.2015, Cottbus, 2015.
[42] B. Kaiser, R. Weber, M. Oertel, E. Böde, B. Monajemi Nejad, J. Zander, „Contract-Based Design of Embedded Systems Integrating Nominal Behavior and Safety", in Complex Systems Informatics and Modeling Quarterly CSIMQ, 2015.
[43] M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T. Margaria. Software Engineering and Formal Methods. Commun. ACM, 51:54–59, 2008.
[44] J. Barnat, L. Brim, M. Ceˇ ska, and P. Ro ˇ ckai. DiVinE: Parallel Distributed Model Checker. ˇ In Proc. of HiBi/PDMC, pages 4–7, 2010.
[45] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In CAV, volume 2404 of LNCS, pages 241–268. Springer, 2002.

[46] S. Blom, W. Fokkink, J. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. µCRL: A Toolset for Analysing Algebraic Specifications. In CAV, volume 2102 of LNCS, pages 250–254. Springer, 2001.

[47] O. Kupferman. Sanity Checks in Formal Verification. In CONCUR, volume 4137 of LNCS, pages 37–51. Springer, 2006.

[48] J. Barnat, P. Bauch, and L. Brim. Checking Sanity of Software Requirements. In Proc. of SEFM, pages 48–52, 2012.

[49] J. Barnat, P. Bauch, N. Benes, L. Brim, J. Beran, andT. Kratochvila. Analysing Sanity of Requirements for Avionics Systems. In *Formal Aspects of Computing (FAoC'16)*, volume 1, pages 1--19, 2016.

[50] J. Bendik, N. Benes, J. Barnat, and I. Cerna. Finding Boundary Elements in Ordered Sets with Application to Safety and Requirements Analysis. In Proc. of SEFM 2016

[51] E. Clarke, O. Grumberg, and D. Peled, Model Checking. MIT press, 1999.

[52] D. Bhatt and K. Schloegel, "Effective Verification of Flight Critical Software Systems: Issues and Approaches," Presented at NSF/Microsoft Research Workshop on Usable Verification, November 2010.

[53] Mathworks. Simulink. [Online]. Available: http://www. mathworks.com/products/simulink/

[54] G. Ciardo, Y. Zhao, and X. Jin, "Parallel symbolic state-space exploration is difficult, but what is the alternative?" in Paralll and Distributed Methods in Verification (PDMC), ser. EPTCS, vol. 14, 2009, pp. 1–17.

[55] K. Verstoep, H. Bal, J. Barnat, and L. Brim, "Efficient Large-Scale Model Checking," in 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2009). IEEE, 2009.

[56] B. Bingham, J. Bingham, F. de Paula, J. Erickson, and M. Singh, G.and Reitblatt, "Industrial Strength Distributed Explicit State Model Checking," in Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC). IEEE, 2010, pp. 28–36.

[57] J. Barnat, L. Brim, J. Beran, and T. Kratochvila, "Partial Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs," in Submitted to FMICS., 2012.

[58] J. Barnat, L. Brim, I. Cˇ erna´, P. Moravec, P. Rocˇkai, and P. ˇSimeˇcek, "DiVinE – A Tool for Distributed Verification (Tool Paper)," in Computer Aided Verification, ser. LNCS, vol. 4144. Springer-Verlag, 2006, pp. 278–281.

[59] C. Schlenoff, M. Gruninger, F. Tissot, J. Valois, T. C. Road, S. Inc, J. Lubell, and J. Lee, "The Process Specification Language (PSL) Overview and Version 1.0 Specification," 1999.

[60] J. Barnat, L. Brim, J. Beran, T. Kratochvila, and I. R. Oliveira. Executing Model Checking Counterexamples in Simulink. Sixth International Symposium on Theoretical Aspects of Software Engineering (TASE 2012), IEEE Computer Society, 2012, 245-248.

[61] J. Barnat, P. Bauch, V. Havel. Temporal Verification of Simulink Diagrams. In Proceedings of the 15th IEEE International Symposium on High Assurance Systems Engineering (HASE'14), pages 81--88, 2014.

[62] J. Barnat, P. Bauch, V. Havel. Model Checking Parallel Programs with Inputs. In Proceedings of the 22nd Euromicro International Conference on Parallel, distributed and network-based Processing (PDP'14), pages 756--759, 2014.

[63] P. Bauch, V. Havel, J. Barnat. Accelerating Temporal Verification of Simulink Diagrams Using Satisfiability Modulo Theories. In Software Quality Journal (SQJ'14), volume 22, pages 1--27, 2014.

[64] P. Bauch, V. Havel, J. Barnat. Control Explicit---Data Symbolic Model Checking. To appear in ACM Transactions on Software Engineering and Methodology (TOSEM'16), 2016.

[65] A. Cimatti, R. DeLong, D. Marcantonio, S. Tonetta, "Combining MILS with Contract-Based Design for Safety and Security", Requirements.SAFECOMP Workshops, 2015.

[66] C. Boettcher, R. DeLong, J. Rushby, and W. Sifre, The MILS Component Integration Approach to Secure Information Sharing, in 27thAIAA/IEEE Digital Avionics Systems Conference, St. Paul, MN, Oct. 2008.

[67] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta. The nuXmv Symbolic Model Checker. CAV 2014: 334-342.

[68] M. Bozzano, A. Cimatti, J-P. Katoen, V. Nguyen, T. Noll, M. Roveri. Safety, Dependability and Performance Analysis of Extended AADL Models. Comput. J. 54(5): 754-775, 2011.

[69] M. Y. Vardi, An Automata-Theoretic Approach to Linear Temporal Logic, in Banff Higher Order Workshop, pp. 238–266, 1995.
[70] K. Claessen and N. Srensson, A liveness checking algorithm that counts, in FMCAD, pp. 52–59, 2012.
[71] A. Cimatti, A. Griggio, S. Mover, S. Tonetta. Verifying LTL Properties of Hybrid Systems with K-Liveness. CAV, 424-440, 2014.
[72] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, IC3 Modulo Theories via Implicit Predicate Abstraction, in TACAS, 2014, pp. 46–61.
[73] A. R. Bradley, SAT-Based Model Checking without Unrolling, in VMCAI, 2011, pp. 70–87.
[74] D-MILS Project. http://http://www.d-mils.org/.
[75] Extended separation kernel capable of global exported resource addressing, Tech. Rep. D6.1, Version 2.0, D-MILS Project, Mar. 2014. http://www.d-mils.org/
[76] Mils network system supporting TTEthernet, Tech. Rep. D6.3, Version 1.0, D-MILS Project, Mar. 2014. http://www.d-mils.org/page/results.
[77] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, The Time- Triggered Ethernet (TTE) Design, 8th IEEE International Symposium on Object- oriented Real-time distributed Computing (ISORC), Seattle, Washington, 2005.
[78] Specification of MILS-AADL, Tech. Rep. D2.1, Version 2.0, D-MILS Project, July 2014. http://www.d-mils.org/page/results.
[79] GSN community standard, tech. rep., Origin Consulting (York) Limited, 2011.
[80] Compositional assurance cases and arguments for distributed mils, Tech. Rep. D4.2, Version 1.0, D-MILS Project, Apr. 2014. http://www.d-mils.org/page/results.
[81] Integration of formal evidence and expression in mils assurance case, Tech. Rep. D4.3, Version 0.7, D-MILS Project, Mar. 2015. http://www.d-mils.org/page/results.
[82] C. Boettcher, R. DeLong, J. Rushby, and W. Sifre, The MILS Component Integration Approach to Secure Information Sharing, in 27thAIAA/IEEE Digital Avionics Systems Conference, St. Paul, MN, Oct. 2008.
[83] J. Rushby, The Design and Verification of Secure Systems, in EighthACM Sympo- sium on Operating System Principles, Asilomar, CA, Dec. 1981, pp. 12–21. (ACM Operating Systems Review, Vol. 15, No. 5)
[84] Information Assurance Directorate, National Security Agency, U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, Fort George G. Meade, MD 20755-6000, June 2007. Version 1.03.
[85] J. Barnat, L. Brim, V. Havel, J. Havliček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. DiVinE 3.0 --- An Explicit-State Model Checker for Multithreaded C&C++ Programs. CAV, pp 863—868, LNCS, vol. 8044, 2013
[86] A. Cimatti, R. Demasi, S. Tonetta. Tightening a Contract Refinement. SEFM 2016: 386-402.
[87] R. Bloem, R. Cavada, I. Pill, M. Roveri, A. Tchaltsev. RAT: A Tool for the Formal Analysis of Requirements. CAV 2007: 263-267.
[88] Prosyd Project. http://www.prosyd.org/
[89] xSAP, a tool for safety assessment of synchronous finite-state and infinite-state systems (https://xsap.fbk.eu/)
[90] T. Arts, M. Dorigatti, S. Tonetta: Making Implicit Safety Requirements Explicit - An AUTOSAR Safety Case. SAFECOMP 2014: 81-92
[91] C. Alexander. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, New York, 1977.
[92] Hemangi Gawand, R.S, Mundada, P. Swaminathn, "Design patterns to implement safety and Fault Tolerance, International Journal of Computer Applications(0975 –8887), Volume 18 -No. 2, March 2011
[93] Armoush, A., "Design Patterns for Safety-Critical Embedded Systems," PhD Thesis, 2010

[94] R. Damaševicius, G. Majauskas, and V. Štuikys. Application of design patterns for hardware design. In DAC '03: Proceedings of the 40th annual Design Automation Conference, pages 48–53, New York, NY, USA, 2003. ACM.
[95] R. Damaševicius and V. Štuikys. Application of UML for hardware design based on design process model. In ASP-DAC '04: Proceedings of the 2004 Asia and South Pacific Design Automation Conference, pages 244–249, Piscataway, NJ, USA, 2004. IEEE Press.
[96] F. Rincon, F. Moya, J. Barba, and J. C. Lopez. Model reuse through hardware design patterns. In DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, pages 324–329, Washington, DC, USA, 2005. IEEE Computer Society.
[97] N. Yoshida. Design patterns applied to object-oriented SOC design. In 10th Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI 2001), Nara, Japan, Oct. 2001.
[98] SafeAdapt, "D4.1 Concept for modelling safe adaptive system behaviour", Project Deliverable, 2015
[99] N. B. Harrison and P. Avgeriou, "Incorporating fault tolerance tactics in software architecture patterns," presented at the SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, New York, New York, USA, 2008, p. 9.
[100] C. A. Lewis, R. W. Smith, and A. Beaulieu, "A model driven framework for N-version programming," presented at the Systems Conference (SysCon), 2011 IEEE International, 2011, pp. 59–65.
[101] Naif Abdullah M. AlzahranAutomatic Derivation of Dependability and Fault Tolerance Analysis Models from Software Architecture, PhD, 2015
[102] U.V.R. Sarma, S. Ramp, A Catalog of Architectural Design Patterns for Safety-Critical Real-Time Systems, Vol.3, Issue 1, January-February 2013, pp.125-131
[103] Safe Adaptive Software for Fully Electric Vehicles, SafeAdapt, http://www.safeadapt.eu/
[104] Dulcineia Penha; Gereon Weiss; Alexander Stante, "Patterns-Based Approach for Designing Fail-Operational Safety-Critical Embedded Systems", Embedded and Ubiquitous Computing (EUC), 21-23 Oct. 2015
[105] T. Ungerer, C. Bradatsch ; M. Gerdes ; F. Kluge ; R. Jahr ; J. Mische, …, A. Pyka "parMERASA – Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability", In: 16th Euromicro Conference on Digital System Design (DSD 2013), 4 September 2013 - 6 September 2013 (Santander, Spain).
[106] SAFURE, https://safure.eu/
[107] EVITA, http://www.evita-project.org/
[108] SafeAdapt, "D4.2 Specification of the design process for safe adaptive embedded systems and tool support for V&V adaptive system behaviour", Project Deliverable, 2015
[109] Grady Booch, James Rumbaugh, Ivar Jacobson. The Unified Modelling Language User Guide (2nd ed.). Addison-Wesley, 2005.
[110] UML specification: http://www.omg.org/spec/UML/
[111] Kruchten, Philippe (1995). Architectural Blueprints — The "4+1" View Model of Software Architecture. IEEE Software 12 (6), pp. 42-50.
[112] SysML specification: http://sysml.org/
[113] Modelica language: https://www.modelica.org/
[114] C. Martin-Villalba, A. Urquia, and S. Dormido, "An approach to virtual-lab implementation using Modelica," Math. Comput. Model. Dyn. Syst., vol. 14, no. 4, pp. 341–360, 2008.
[115] M. Dempsey, "Dymola for multi-engineering modelling and simulation," 2006 IEEE Veh. Power Propuls. Conf. VPPC 2006, 2006.
[116] Modelica_Fluid 1.0 Released: https://www.modelica.org/news_items/release_of_modelica_fluid_1_0
[117] J. Llorens, J. Morato, and G. Genova, "RSHP: an information representation model based on relationships," in Soft Computing in Software Engineering, vol. 159, E. Damiani, M. Madravio, and L. Jain, Eds. Springer Berlin Heidelberg, 2004, pp. 221–253.
[118] I. Diaz, J. Llorens, G. Genova, and J. M. Fuentes, "Generating domain representations using a relationship model," Inf. Syst., vol. 30, no. 1, pp. 1–19, Mar. 2005
[119] AADL language: http://www.aadl.info

[120] DAF specification: http://www.omg.org/spec/DAF/
[121] EAST-ADL language: http://www.east-adl.info/
[122] KDM specification: http://www.omg.org/spec/KDM/
[123] MARTE specification: http://www.omg.org/spec/MARTE/
[124] ODM specification: http://www.omg.org/spec/ODM/
[125] RAS specification: http://www.omg.org/spec/RAS/
[126] SBVR specification: http://www.omg.org/spec/SBVR/
[127] SMM specification: http://www.omg.org/spec/SMM/
[128] Geoffrey Biggs, Takeshi Sakamoto, Tetsuo Kotoku: A profile and tool for modelling safety information with design information in SysML. Software and System Modelling 15(1): 147-178 (2016)
[129] Vladislav Gribov, Holger Voos: Safety oriented software engineering process for autonomous robots. ETFA 2013: 1-8
[130] D. Kuschnerus, F. Bruns, A. Bilgic, T. Musch, A UML profile for the development of IEC 61508 compliant embedded software, in: Embedded Real-Time Software and Systems (ERTS 2012), 2012.
[131] Rajwinder Kaur Panesar-Walawege, Mehrdad Sabetzadeh, Lionel C. Briand: Supporting the verification of compliance to safety standards via model-driven engineering: Approach, tool-support and empirical validation. Information & Software Technology 55(5): 836-864 (2013)
[132] Vladimir Rupanov, Christian Buckl, Ludger Fiege, Michael Armbruster, Alois Knoll, Gernot Spiegelberg: Employing early model-based safety evaluation to iteratively derive E/E architecture design. Sci. Comput. Program. 90: 161-179 (2014)
[133] H. Stallbaum, M. Rzepka, Toward DO-178B compliant test models, in: Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa 2010), 2010, pp. 25–30.
[134] Ji Wu, Tao Yue, Shaukat Ali, Huihui Zhang: Ensuring Safety of Avionics Software at the Architecture Design Level: An Industrial Case Study. QSIC 2013: 55-64
[135] Gregory Zoughbi, Lionel C. Briand, Yvan Labiche: Modelling safety and airworthiness (RTCA DO-178B) information: conceptual model and UML profile. Software and System Modelling 10(3): 337-367 (2011)
[136] National Aeronautics and Space Administration, "Nasa Complex Electronics handbook for assurance professionals", 2016
[137] H. Isakovic, "Use Case - MPSoC, SoA, Dynamicity - Engaging Challenges in Cyber-Physical Systems Using Emerging Technologies", workshop on EMC$^2$: Mixed Criticality Applications and Implementation Approaches, HIPEAC, 2015
[138] R. Fuchsen, "How to address Certification for Multi-Core Based IMA Platforms", professional article available on www.sysgo.com
[139] SafeAdapt, "D3.3 Specification of ISO 26262 Safety Goals for Self-Adaptation Scenarios", Project Deliverable, 2015
[140] A Ruiz, I Habli, H Espinoza, "Towards a case-based reasoning approach for safety assurance reuse, International Conference on Computer Safety, Reliability, and Security, 22-35, 2012
[141] FAA/CAST "CAST 32 - Position Paper, Multi-core processors", May 2014, http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf
[142] I. Šljivo, B. Gallina, J. Carlson, H. Hansson. "Generation of safety case argument-fragments from safety contracts." International Conference on Computer Safety, Reliability, and Security. Springer International Publishing, 2014.
[143] I. Šljivo, B. Gallina, J. Carlson, H. Hansson. "A method to generate reusable safety case fragments from compositional safety analysis." International Conference on Software Reuse. Springer International Publishing, 2015.
[144] I. Šljivo, B. Gallina, J. Carlson, H. Hansson, S. Puri, A Method to Generate Reusable Safety Case Argument-Fragments from Compositional Safety Analysis, Journal of Systems and Software, Available online 25 July 2016, ISSN 0164-1212, http://dx.doi.org/10.1016/j.jss.2016.07.034
[145] CONCERTO ARTEMIS JU project: http://www.concerto-project.org/
[146] CHESS ARTEMIS JU project: http://www.chess-project.org/

[147] Mixed Criticality Cluster http://www.mixedcriticalityforum.org/home/
[148] CONTREX https://contrex.offis.de/home/
[149] DREAMS http://www.dreams-project.eu/
[150] PROXIMA http://www.proxima-project.eu/
[151] CERTAINTY http://www.certainty-project.eu/
[152] N. G. Leveson, «Engineering a Safer World: Systems Thinking Applied to Safety», Cambridge, Mass.: The MIT Press, 2012.
[153] Bate, Iain; Conmy, Philippa / Certification of FPGAs - Current Issues and Possible Solutions. SAFETY-CRITICAL SYSTEMS: PROBLEMS, PROCESS AND PRACTICE. ed. / C Dale; T Anderson. NEW YORK: SPRINGER, 2009. p. 149-165.
[154] J. Clegg, Arguing the safety of FPGAs within safety critical systems, in: 4th IET International Conference on Systems Safety, 2009, 52–52.
[155] European Aviation Safety Agency, MULCORS-Use of MULticore proCessORs in airbone Systems, https://www.easa.europa.eu/system/files/dfu/CCC_12_006898-REV07%20-%20MULCORS%20Final%20Report.pdf , 2011
[156] RECOMP http://atcproyectos.ugr.es/recomp/
[157] CHESSML profile https://www.polarsys.org/chess/publis/CHESSMLprofile.pdf
[158] D2.2 The CONCERTO Component Model http://www.concerto-project.org/results
[159] D5.2 Compositional Certification Framework: Methodological Guide (OPENCOSS) http://www.opencoss-project.eu/sites/default/files/D5.6_Compositional_Certification_Framework_Methodological_Guide_v3_Mar2015.pdf
[160] D4.7 Methodological Guide Common Certification Language (OPENCOSS) http://www.opencoss-project.eu/sites/default/files/D4.7_Methodological_Guide_Common_Certification_Language_V1.0.pdf
[161] D5.3 Compositional Certification Conceptual (OPENCOSS) http://www.opencoss-project.eu/sites/default/files/D5.3_Compositional_Certification_Conceptual_Framework_final.pdf
[162] SYNOPSIS project: http://www.es.mdh.se/SYNOPSIS/
[163] Polarsys project: https://www.polarsys.org/
[164] E. Alaña, H. Naranjo, R. Valencia, A. Medina, C. Honvault, A. Rugina, M. Panunzio, B. Dellandrea, Gerald García, Avionics Architecture Modelling Language, DASIA 2014.
[165] D5.3 version 1, Compositional Certification Conceptual Framework, OPENCOSS Project, November 2013
[166] John M. Rushby. An evidential tool bus. In Kung-Kiu Lau and Richard Banach, editors, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, volume 3785 of Lecture Notes in Computer Science, pages 36–36. Springer, 2005.
[167] Society of Automotive Engineers (SAE). ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.
[168] Society of Automotive Engineers (SAE). ARP 4754A, Guidelines for Development of Civil Aircraft and Systems, December 2010.
[169] Society of Automotive Engineers (SAE). AIR 6110, Contiguous Aircraft/ System Development Process Example, December 2011.
[170] A. G. Ryman, A. L. Hors, and S. Speicher, "OSLC Resource Shape: A language for defining constraints on Linked Data," in LDOW, 2013.
[171] C. Bizer, T. Heath, and T. Berners-Lee, "Linked Data - The Story So Far" Int. J. Semantic Web Inf. Syst., vol. 5, no. 3, pp. 1–22, 33 2009.
[172] P. Hayes, "RDF Semantics," World Wide Web Consortium, Feb. 2004.
[173] C. El Salloum, "A cross-domain approach for mixed-criticality integration based on heterogeneous MPSoCs", HiPEAC Conference, http://cordis.europa.eu/fp7/ict/embedded-systems-engineering/presentations/ell-salloum.pdf, January 2012
[174] ACROSS ("A Cross Domain Approach For Mixed Criticality Integration based on heterogeneous MPSoCs) Project, www.across-project.eu

[175] Xilinx, www.xilinx.com
[176] Xilinx, “Xilinx Reduces Risk and Increases Efficiency for IEC61508 and ISO26262 Certified Safety Applications,», http://www.xilinx.com/support/documentation/white_papers/wp461-functional-safety.pdf, 2015.
[177] Xilinx, “Practical Use of FPGAs and IP in DO-254 Compliant Systems (Whitepaper)”, 2011.
[178] Altera, https://www.altera.com/
[179] Altera, “FPGA-based Safety Separation Design Flow for Rapid Functional Safety Certification”, https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/an/an704.pdf, 2015
[180] Fabian Feyerherd, Methodische Erstellung eines Sicherheitskonzeptes für einen elektrischen Antrieb, Master Thesis, Dec. 2014
[181] “Certifying boeing’s airplanes,” http://787updates.newairplane.com/Certification-Process, accessed: 2016-07-27.
[182] SAFE project (Safe Automotive soFtware architecture), http://www.safe-project.eu/
[183] “Overview of Functional Safety Measures in AUTOSAR”, https://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/auxiliary/AUTOSAR_EXP_FunctionalSafetyMeasures.pdf
[184] CATSY project: https://es.fbk.eu/projects/catsy
[185] M. Bonnet, M. Laforge, J-B Samuel, Impact of IEC 61508 Standards on Intelligent Electrical Networks and Safety Improvement, Schneider Electric. White paper, 21-Feb-2014.
[186] W. Vesely, F. Goldberg, N. Roberts, D. Haasl, Fault tree handbook, Tech. Rep. NUREG-0492, Systems and Reliability Research Office of Nuclear Regulatory Research U.S. Nuclear Regulatory Commission (1981).
[187] W. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, J. Minarick III, J. Railsback, Fault Tree Handbook with Aerospace Applications, Tech. rep., NASA (2002).
[188] O. Lisagor, T. Kelly, R. Niu, Model-based safety assessment: Review of the discipline and its challenges, in: ICRMS, 2011, pp. 625–632.
[189] M. Bozzano, et al., ESACS: An Integrated Methodology for Design and Safety Analysis of Complex Systems, in: Proc. ESREL, Balkema Publisher, 2003, pp. 237–245.
[190] O. Akerlund, P. Bieber, E. Boede, M. Bozzano, M. Bretschneider, C. Castel, A. Cavallo, M. Cifaldi, J. Gauthier, A. Griffault, et al., ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects, Proc. ERTS 2006 (2006) 1–11.
[191] M. Bozzano, A. Villafiorita, Design and Safety Assessment of Critical Systems, CRC Press (Taylor and Francis), an Auerbach Book, 2010.
[192] The FSAP/NuSMV-SA platform. http://es.fbk.eu/tools/FSAP
[193] M. Bozzano, A. Villafiorita, The FSAP/NuSMV-SA Safety Analysis Platform, Software Tools for Technology Transfer 9 (1) (2007) 5–24.
[194] The AltaRica language. http://altarica.labri.fr/forge
[195] A. Arnold, A. Griffault, G. Point, A. Rauzy, The AltaRica formalism for describing concurrent systems, Fundamenta Informaticae 40 (2000) 109–124.
[196] Benjamin Bittner, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Marco Gario, Alberto Griggio, Cristian Mattarei, Andrea Micheli, Gianni Zampedri: The xSAP Safety Analysis Platform. TACAS 2016: 533-539
[197] P. Fenelon, J. A. McDermid, An integrated tool set for software safety analysis, Journal of Systems and Software 21 (3) (1993) 279–290.
[198] M. Wallace, Modular architectural representation and analysis of fault propagation and transformation, Electronic Notes in Theoretical Computer Science 141 (3) (2005) 53–71.
[199] Y. Papadopoulos, J. McDermid, R. Sasse, G. Heiner, Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure, Reliability Engineering & System Safety 71 (3) (2001) 229–247.
[200] P. H. Feiler, B. A. Lewis, S. Vestal, The sae architecture analysis & design language (aadl) a standard for engineering performance critical systems, in: Computer Aided Control System Design, 2006 IEEE

International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE, IEEE, 2006, pp. 1206–1211.

[201] Marco Bozzano, Alessandro Cimatti, Oleg Lisagor, Cristian Mattarei, Sergio Mover, Marco Roveri, Stefano Tonetta: Safety assessment of AltaRica models via symbolic model checking. Sci. Comput. Program. 98: 464-483 (2015)

# Appendix A. SafeCer generic component model

In order to facilitate the specification of reusable information, one key feature of the SafeCer component meta-model is the separation of component types and component instances. The former represents those aspects that are common to all occurrences of the component in various systems, including the interface, defining the means by which the component interacts with its surroundings, but also component contracts and argument fragments (discussed in the following section).

The component type comes with provided or required operation ports and input/output data and event ports. Component types can be decomposed in sub components, the latter representing instance of components in the context of the parent component type, so not in the context of a given system.

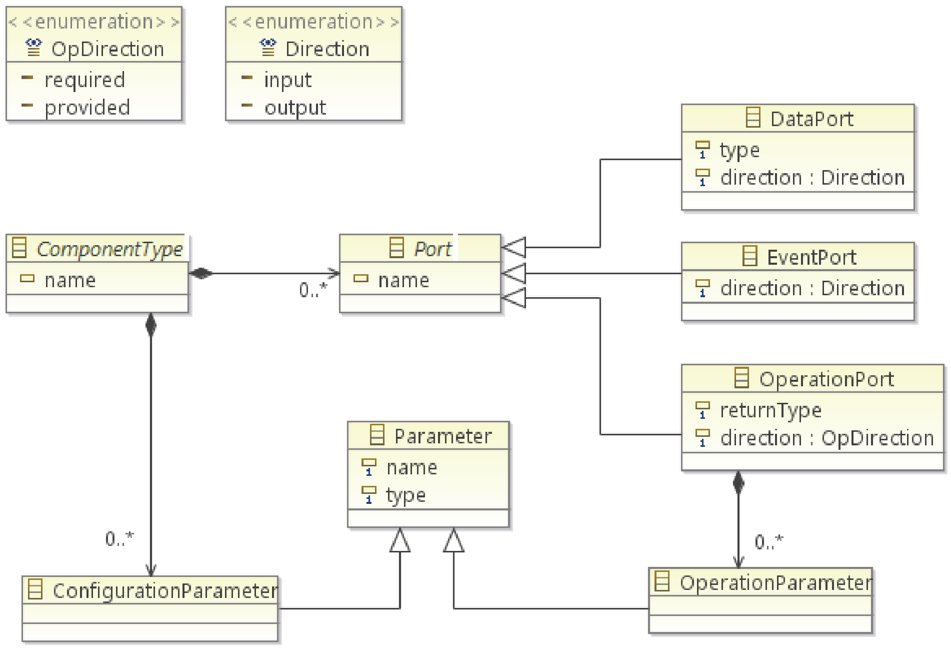


**Figure 41: Component type, ports and parameters**

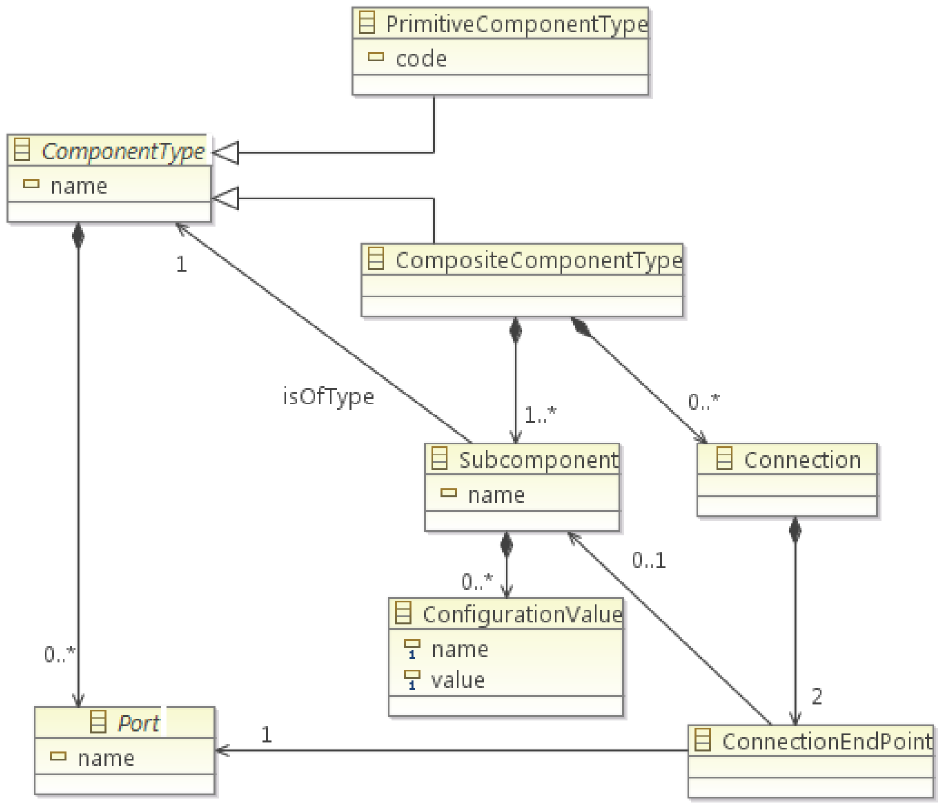


**Figure 42: Primitive and composite component types**

Component instances represent instantiation of a given component type in a particular system to be develop or running; mapping (i.e. deployment) of component instances to specific hardware node available in the system can be represented. The component instance inherits the specification associated to the component type. The component type can be designed to offer some variability in the implementation, represented with configuration parameters in the component model, to be fixed when the component type is instantiated. If a decomposed component type is instantiated, then the instances of the internal parts are instantiated as well, obtain a hierarchy of component instances.

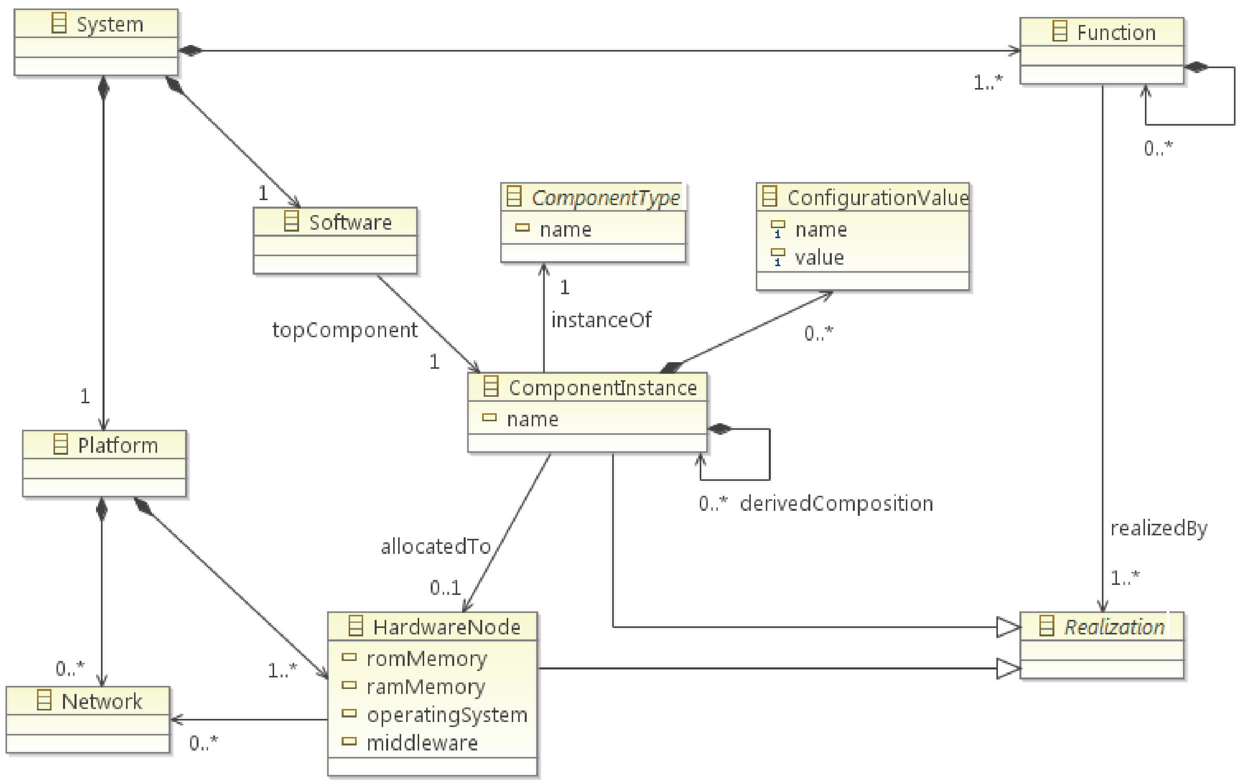


**Figure 43: Component instances and system model**

For what regards the SafeCer component model support for contract modelling the reader can refer to section 3.1.1.

# Appendix B. CHESS profile for contracts

The CHESS profile for contracts introduces the <<Contract>> stereotype (extending the SysML ConstraintBlock entity). <<Contract>> aggregates two special kind of UML Constraint, the latter representing the assumption and guarantee of the contract.
The profile does not impose any particular language to be used for the specification of the assume and guarantee contract's properties (the CHESS tool support for the integration with the OCRA tools requires that the contract properties are expressed using the language supported by OCRA).

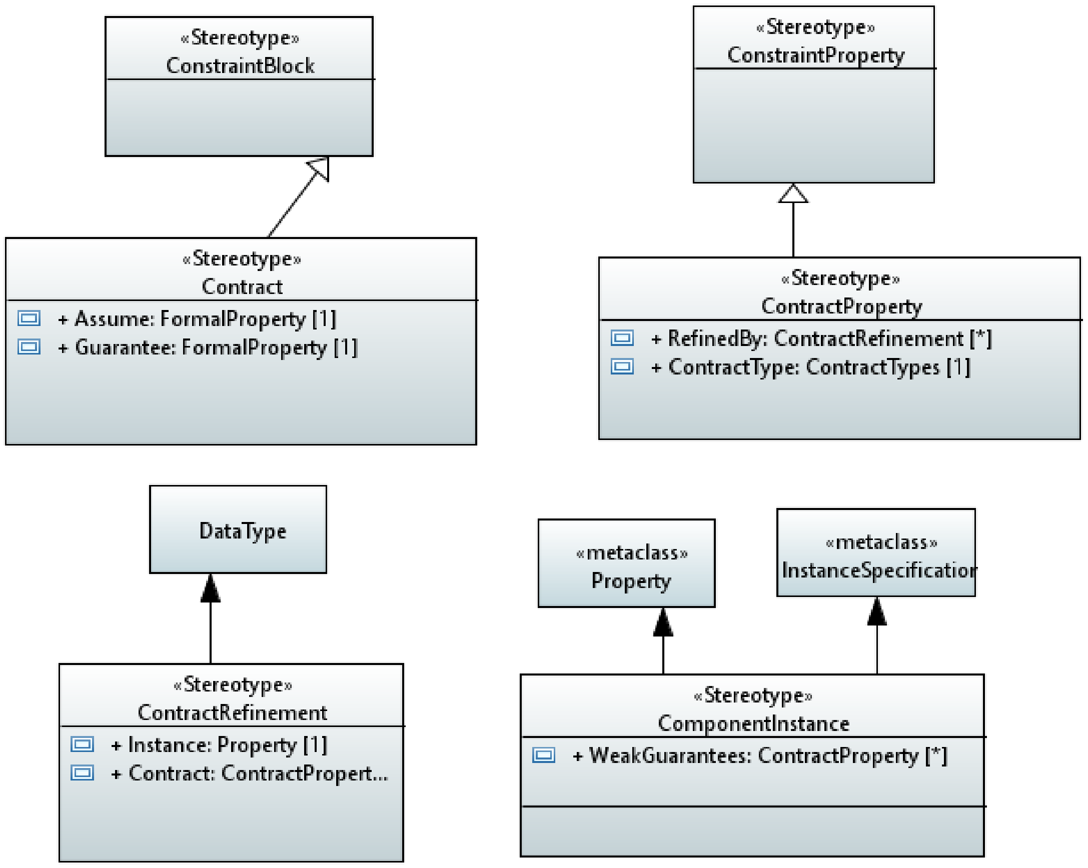

«Stereotype» ConstraintBlock
«Stereotype» ConstraintProperty
«Stereotype» Contract
+ Assume: FormalProperty [1]
+ Guarantee: FormalProperty [1]
«Stereotype» ContractProperty
+ RefinedBy: ContractRefinement [*]
+ ContractType: ContractTypes [1]
DataType
«metaclass» Property
«metaclass» InstanceSpecification
«Stereotype» ContractRefinement
+ Instance: Property [1]
+ Contract: ContractPropert...
«Stereotype» ComponentInstance
+ WeakGuarantees: ContractProperty [*]

**Figure 44: CHESS profile for Contract**

In the CHESS profile for contract the type/instance dichotomy, available at component level, is applied to the concept of contract too. So a contract can be modelled in isolation, for instance as formalization of a given requirement, and then it can be instantiated (i.e. assigned) later for a given component; in this way the same contract can be reused and associated to different components.

The association between a <<Contract>> and a component (the latter defined in UML) is obtained by adding a <<ContractProperty>> attribute typed with <<Contract>> to the component.

The information about weak or strong of a given contract is provided when the contract is associated/instantiated to a given component, so through the <<ContractProperty>> stereotype; in this way the same contract can vary its weak or strong specification depending on the component where it is instantiated.

<<ContractProperty>>, through its RefinedBy attribute, allows to model the decomposition of contracts in the context of a component decomposition, as foreseen by the SafeCer component model; i.e. it can be used to model how a contract of a given component is decomposed into a set of contract associated to the child components. Given that <<ContractProperty>> represents an instance of a Contract, the decomposition is actually modelled for contract instances; this allow to model different decompositions for different instances of the same contract.

Finally, <<ComponentInstance>> stereotype extends the concept of instance available in UML by adding the list of weak contracts (i.e. contracts instances) which actually hold for the given component instance.