

Mario Trapp

Assuring Functional Safety in Open Systems of Systems

Assuring Functional Safety in Open Systems of Systems

Vom Fachbereich Informatik der TU Kaiserslautern
im Rahmen des Habilitationsverfahrens
akzeptierte Habilitationsschrift

von

Dr. Mario Trapp

01.06.2015

Abstract

Interconnected, autonomously driving cars shall realize the vision of a zero-accident, low energy mobility in spite of a fast increasing traffic volume. Tightly interconnected medical devices and health care systems shall ensure the health of an aging society. And interconnected virtual power plants based on renewable energy sources shall ensure a clean energy supply in a society that consumes more energy than ever before. Such open systems of systems will play an essential role for economy and society.

Open systems of systems dynamically connect to each other in order to collectively provide a superordinate functionality, which could not be provided by a single system alone. The structure as well as the behavior of an open system of system dynamically emerge at runtime leading to very flexible solutions working under various different environmental conditions. This flexibility and adaptivity of systems of systems are a key for realizing the above mentioned scenarios.

On the other hand, however, this leads to uncertainties since the emerging structure and behavior of a system of system can hardly be anticipated at design time. This impedes the indispensable safety assessment of such systems in safety-critical application domains. Existing safety assurance approaches presume that a system is completely specified and configured prior to a safety assessment. Therefore, they cannot be applied to open systems of systems. In consequence, safety assurance of open systems of systems could easily become a bottleneck impeding or even preventing the success of this promising new generation of embedded systems.

For this reason, this thesis introduces an approach for the safety assurance of open systems of systems. To this end, we shift parts of the safety assurance lifecycle into runtime in order to dynamically assess the safety of the emerging system of system. We use so-called safety models at runtime for enabling systems to assess the safety of an emerging system of system themselves. This leads to a very flexible runtime safety assurance framework.

To this end, this thesis describes the fundamental knowledge on safety assurance and model-driven development, which are the indispensable prerequisites for defining safety models at runtime. Based on these fundamentals, we illustrate how we modularized and formalized conventional safety assurance techniques using model-based representations and analyses. Finally, we explain how we advanced these design time safety models to safety models that can be used by the systems themselves at runtime and how we use these safety models at runtime to create an efficient and flexible runtime safety assurance framework for open systems of systems.

Table of Contents

1.	Introduction	6
1.1.	Motivation	6
1.2.	Safety Challenges in Open Systems of Systems	7
1.2.1.	Open Adaptive Systems	7
1.2.2.	Safety Challenges	8
1.2.3.	Solution Overview	10
1.2.4.	Document Structure	15
2.	Fundamentals of Safety Assurance	17
2.1.	Terms and Definitions	17
2.2.	Safety Engineering in a Nutshell	24
2.3.	Hazard Analysis and Risk Assessment	28
2.3.1.	Hazard Identification	28
2.3.2.	Risk Assessment	29
2.4.	Safety Analysis Techniques	33
2.4.1.	Failure Mode and Effects Analysis (FMEA)	33
2.4.2.	Hazard and Operability Analysis (HAZOP)	35
2.4.3.	Event Tree Analysis (ETA)	38
2.4.4.	Fault Tree Analysis (FTA)	39
2.4.5.	Reliability Block Diagrams (RBD)	44
2.5.	Safety Concepts and Safety Cases	45
2.5.1.	Goal Structuring Notation	46
2.6.	Software Fault Tolerance Mechanisms	51
2.6.1.	Overview	52
2.6.2.	Error Detection	54
2.6.3.	Programming Techniques	59
2.6.4.	Design-Diversity	59
2.6.5.	Data-Diversity	61
3.	Fundamentals of Model-Driven Engineering	63
3.1.	The Model-Driven Engineering Paradigm	63
3.2.	Meta-Modeling	67
3.3.	The Model Driven Architecture – MDA®	70

3.4.	Model-Driven Engineering of Embedded Systems	71
3.5.	Model Driven Optimization.....	74
4.	Model-Driven Safety Engineering.....	77
4.1.	Motivation	78
4.1.1.	Problem Analysis.....	78
4.1.2.	Advantages of Model-Driven Safety Engineering	80
4.2.	Safety Modelling	86
4.2.1.	Modular Safety Assurance	86
4.2.2.	Meta-modeling	89
4.3.	Model Driven Safety Analyses.....	92
4.3.1.	Failure Logic Modeling.....	92
4.3.2.	Safety Analysis Meta-Models	94
4.4.	Model-Driven Safety Concepts and Safety Cases.....	106
4.4.1.	Modular Certification Approaches	107
4.4.2.	Safety Concept Meta-Models	109
5.	Runtime Safety Assurance.....	113
5.1.	Overview on safety models for runtime evaluation.....	114
5.2.	SafetyCertificates@Runtime	115
5.3.	SafetyCases@Runtime	118
5.4.	V&V-Models@Runtime.....	120
5.5.	SafetyAnalysis@Runtime	121
5.6.	Hazard Analysis and Risk Assessment@Runtime (HRA@Runtime).....	122
5.7.	Evaluation of the different approaches	123
5.8.	Conceptual safety assurance framework for Open Adaptive Systems.....	125
5.9.	Conditional Safety Certificates – ConSerts	128
5.9.1.	General Assumptions on Modeling Open Systems of Systems	129
5.9.2.	Safety Guarantees and Demands	131
5.9.3.	Runtime Evidences	133
5.9.4.	Mapping Functions and Runtime Evaluation	134
5.9.5.	Specification Technique	137
6.	Summary and Future Work	141
7.	References	144

1. Introduction

1.1. Motivation

We live in a digital society. Using PCs, smart phones or any other kind of mobile device we are used to be globally connected – everywhere, every time. Digital systems have been the main driver for recent innovations that shaped the character of our society and economy. Besides the obvious world of smartphones and the internet, digital systems have been the key enabler for most product innovations of the last decade. Hidden from the user, embedded systems have enabled new product features like driver assistance systems in cars, new medical devices for cancer therapy, or precision farming in the agricultural domain.

Up to now, traditional IT systems and embedded systems have been rather isolated from each other. In general, embedded systems used to be isolated systems with a limited connectivity to other systems. In the near future, however, embedded systems will be open in the sense that they dynamically interconnect to other systems or the ‘cloud’. In order to manage such a permanently changing runtime context, they will dynamically adapt their structure and their behavior to optimize their functionality to the current runtime context. In recent years, the term ‘Cyber Physical System’ has emerged as a buzzword describing this new generation of embedded systems. In research such systems are often called ‘open adaptive systems’ reflecting their most important characteristics.

Regarding industry roadmaps across various application domains, such open systems of systems seem to be the technological answer to a series of societal challenges: Interconnected, autonomously driving cars shall realize the vision of a zero-accident, low energy mobility in spite of a fast increasing traffic volume. Tightly interconnected medical devices and health care systems shall ensure the health of an aging society. And interconnected virtual power plants based on renewable energy sources shall ensure a clean energy supply in a society that consumes more energy than ever before.

As a consequence, society and economy will depend on embedded systems more than ever before. This leads to an increasing importance of the systems’ safety and reliability. This is particularly true since most application scenarios can be found in safety-critical application domains. This means that a failure in such complex systems could easily lead to catastrophic consequences.

Assuring safety in open systems of systems is however a non-trivial task. Openness and adaptivity are key properties facilitating the economic potential of cyber physical systems. However, this flexibility in the context of cyber physical systems leads to unpredictability, which is a major challenge for safety assurance. Available safety assurance approaches on the one hand usually assume that all safety-relevant uncertainties have been resolved before the system is certified. Current safety research is still rather focused on development time assurance not considering runtime adaptivity. Engineering approaches supporting open connectivity and adaptivity on the other hand

have considered safety only as one out of many non-functional properties without regarding the specific characteristics and challenges of safety assurance. Safety can therefore easily become a bottleneck impeding or even preventing the promising vision of cyber physical systems. Thus, new safety assurance approaches are indispensable to open the full potential of this new generation of embedded systems. Enabling the required flexibility without endangering the systems' safety is therefore one of the most important research challenges today.

Considering that there is for example still no commonly accepted approach available that supports modular safety assurance at development time, it is obviously quite a long journey from traditional safety engineering to the safety assurance of open systems of systems. This thesis therefore introduces a framework for assuring safety in open systems of systems. To this end, it points out the major challenges and explains the major steps we have taken to advance traditional safety engineering approaches for supporting the safety assurance of open systems of systems.

This thesis summarizes the results of several years of research, which has led to more than fifty publications and six dissertations. Therefore, it provides an introduction to the required basic fundamentals and gives a detailed overview on the resulting overall safety assurance framework incorporating the different single constituents, which have been subject to our research over the last years. To this end, it describes the most important concepts we have developed and it particularly shows the big picture of how they work together in an integrated framework. Moreover, we elaborate key findings based on our overall research, which are of general importance for the safety assurance of open systems of systems.

1.2. Safety Challenges in Open Systems of Systems

Terms like cyber physical systems, internet of things or ambient systems helped to shape a new generation of embedded systems. From a scientific point of view, however, those terms provide no precise definition of a system class. Therefore, this section refines the research problem of assuring safety in open systems of systems. To this end, Section 1.2.1 first provides a more precise definition of the system characteristics that are of relevance for safety assurance. Based on these core characteristics, Section 1.2.2 analyzes the resulting challenges for safety assurance of such systems. Section 1.2.3 provides a solution overview how we addressed these challenges in the safety assurance framework we describe in this thesis. Based on this solution overview, section 1.2.4 explains the structure of the remaining document.

1.2.1. Open Adaptive Systems

Openness and adaptivity are key properties facilitating the business goals associated with cyber physical systems [1].

Openness is required to yield a *collective* functionality of the resulting system of systems. Collective means that each system contributes its specific capabilities in order to fulfill a superordinate functionality together with other systems. For example, cars dynamically interconnect to each other in order to collectively realize a cross road assistant – a functionality that could not be realized by a single system alone. Openness in this context therefore means that the systems can dynamically interconnect to other systems. Open systems of systems are able to handle the connectivity themselves - possibly with the support of end users like operators or drivers. In order to yield collective behavior, dynamic connectivity is not limited to interfaces and interoperability but it includes the behavioral level since the systems dynamically form a distributed algorithm, e.g., for a distributed cross road management.

In consequence this means that the prospective collaboration partners of a system and their specific characteristics are not known during development time. Additionally, cyber physical systems are used very flexibly in various different usage contexts, which cannot be predicted completely. This leads to significant *uncertainties* that cannot be resolved at development time, i.e. uncertainties impede a complete anticipation of the context. For this reason, open systems of systems must be *adaptive* so that they can dynamically adapt their behavior and/or structure to the given context at runtime.

It is important to note that there is a difference between adaptability and adaptivity. *Adaptability* refers to the property of the system that it can be easily adapted by someone to a new context. This means that adaptability requires the involvement of a person to modify the system (usually a developer). *Adaptivity* on the other hand refers to the property of the system that it is able to adapt itself to a new context – any involvement of persons is not required. In order to underline this important difference, research often talks about *self-adaptive* systems. When the terms adaptive or adaptivity are used in this document, they always refer to *self-adaptive* and *self-adaptivity*, respectively.

1.2.2. Safety Challenges

From a safety assurance point of view, the main challenge of open systems is given by their inherent uncertainties. All existing safety assurance approaches assume that the system is completely specified prior to a safety assessment and that the system exposes a completely predictable and deterministic behavior. Consequently, the dynamic integration of systems to systems of systems or the dynamic adaptation of the systems' structure and behavior is not supported.

This basic principle of determinism and predictability is also reflected in safety standards. Thus, they prohibit any kind of intelligent solutions. For example, the use of artificial intelligence for fault correction is prohibited by the functional safety standard IEC 61508 [2]. Even less sophisticated approaches like the dynamic re-deployment of software in case of hardware faults is prohibited (prohibition of dynamic reconfiguration in [2]).

In general, safety standards demand a complete and deterministic system specification, following the reasonable paradigm that it is only possible to certify something that is completely known and understood. For example, standards demand that all possible hardware and software interactions are determined and evaluated in a software architecture specification [2], which is obviously not possible, if the interaction partners are not known during development time. Furthermore, safety standards demand that all modes of operation of the systems themselves and the modes of operation of all collaborating systems must be considered in all safety assurance activities. This demand presumes that all modes of operation can be determined at development time, which is not the case for open systems of systems. As a further aspect, if flexible self-adaptations at runtime ought to be supported, they must comply with the same normative demands like system modifications at development time in a traditional development process. This would mean that an impact analysis and a revalidation of modified elements or even of the overall system must be performed at runtime.

Though safety standards certainly reflect the commonly accepted state of the art, they are usually based on outdated methodologies and can obviously hardly be applied to leading edge technology like open adaptive systems. Nonetheless, the basic ideas underlying safety standards shape a corridor we cannot leave if we aim for accepted safety assurance approaches for open systems of systems. In consequence, this means that we have to show that the safety assurance of open systems leads to systems, which are at least as safe as conventional systems, today. To this end, it is first necessary to map runtime activities of open systems of systems to the corresponding development time activities. In the second step it is necessary to show that the runtime activities undergo a safety assurance that is comparable to the safety assurance as it is demanded by safety standards for the corresponding development activities:

In order to realize *openness*, systems dynamically connect with other systems to build a system of systems. This step is comparable to the integration of components at development time. Experience has shown that many problems do not arise within the single units of the system but during their integration. Therefore, the integration of safe systems does not necessarily lead to a safe system of systems. In fact, it is a challenging task to check whether or not the interaction of the components is safe. Answering this question at runtime requires sophisticated interface checks and possibly additional runtime verification mechanisms like integration tests.

In order to realize *adaptivity*, the system changes its structure and/or behavior. If adaptivity is limited to a dynamic reconfiguration, this runtime activity is comparable to a system configuration. If more flexible adaptations are required, this can be seen as a system modification at runtime. This leads to several challenges since any change of the system requires an impact analysis and a re-verification of modified parts, or – depending on the required safety integrity level– even a re-verification of the overall system. Today, both activities require a lot of human interaction making it very difficult to perform these steps automatically and autonomously at runtime.

As an additional aspect it must be considered that there is no superordinate root element during runtime integration, which has the knowledge about the overall system of systems and which could therefore coordinate the runtime integration. In fact, all systems are equal partners, which have been unknown during development time and which must build an ad-hoc collective in order to jointly coordinate the integration check. Instead of a hierarchy as it can be found in traditional systems, open systems of systems expose a heterarchical structure.

This also leads to a non-technical but nonetheless important challenge. In open systems of systems, there is no expert in the loop, who finally checks the systems' safety. In fact, the systems themselves decide on whether or not they are safe. This means a change of a basic paradigm of safety assurance. The knowledge that a human expert checked the system is an essential prerequisite for societal acceptance. Without this trust, it is unlikely that a new product would be accepted by the market. For this reason, societal acceptance is an important issue, but it is not a technical challenge and will therefore not be considered in further detail in this thesis.

Summarizing, the main challenges for safety assurance arise from the uncertainty that is immanent to open and adaptive systems. As a consequence, it is not possible to completely assure the systems' safety at development time, already. In fact, it is necessary to shift parts of the assurance process to runtime. In order to yield certifiably safe systems, it is however necessary that runtime assurance leads to a quality that is at least as good as that of development time assurance. Namely, this means that

- (1) the runtime integration of systems of systems must be as safe as systems integrated at development time.
- (2) system adaptations at runtime must be as safe as system modifications at development time.

1.2.3. Solution Overview

Regarding the challenges described in the previous section, there are various issues to be solved, which will certainly require a lot of further research. This document does not address all of those challenges. In fact, we focus on the safety engineering process since the latter builds the backbone of the development of safe systems. The general concepts of safety engineering lifecycles will be discussed in chapter 2. However, Figure 1-1 gives a coarse overview on the relation between the safety engineering lifecycle on the one hand and the development lifecycle on the other hand. A development can roughly be subdivided into the phases *requirements*, *architecture*, *design*, *implementation* and *validation & verification*. The safety engineering lifecycle is performed in parallel in close interaction with the development lifecycle. Starting with a *hazard analysis* based on the first requirements, the hazards are identified and the associated risks are assessed. As a result of this step, the safety goals are identified and added to the requirements specification as top level safety requirements. In order to identify required counter

measures, *safety analyses* such as FMEA or FTA are performed iteratively based on the system architecture and design specifications for identifying the cause-effect relationships potentially leading to a violation of the safety goals. Based on the analysis results, the safety requirements are step wisely refined and documented in a *safety concept*. The safety concept defines modifications of the requirements, architecture, or design of the system that are necessary to ensure the system's safety. For example, it could demand plausibility checks, redundancy architectures, or specific validation activities. Once the safety requirements have been implemented, validation and verification activities are required to provide evidence that the implementation is correct. These evidences are attached to the according requirements of the safety concept in order to create a *safety case*, which provides a seamless argument on the system's safety and therefore builds the basis for a safety certification.

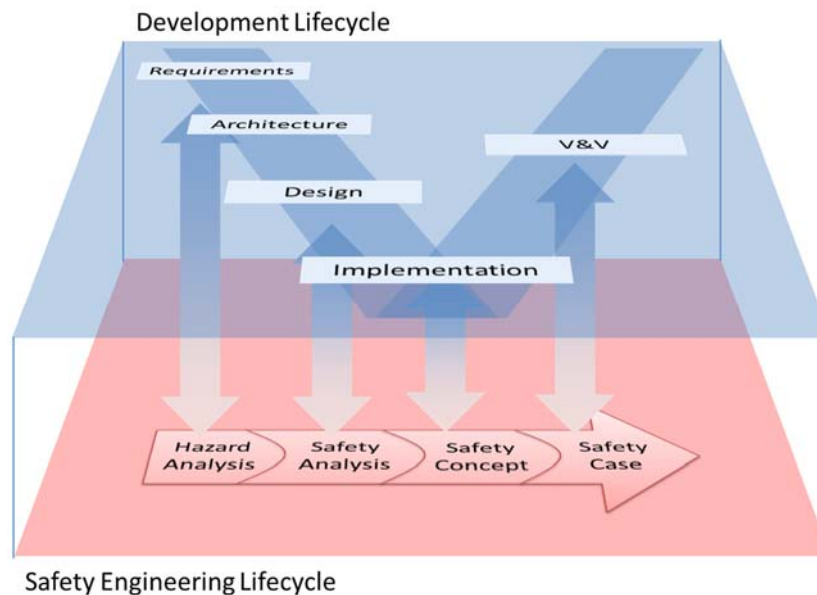


Figure 1-1 Relation between development lifecycle and safety engineering lifecycle.

Obviously, the safety engineering activities build the backbone of safety assurance by identifying hazards and assessing the associated risks, identifying the causes and providing the final argument that all causes have been addressed by appropriate counter measures in order to sufficiently reduce the residual risk of the system. Safety assurance is impossible without these steps. Therefore, we focus on safety engineering activities that are required in order to support the safety assurance of open systems of systems. Certainly, there are further modifications of development and quality assurance activities that are required for the assurance of open adaptive systems. For example, there might be a need for runtime verification activities. Nonetheless, such extensions will not be considered in this document. Instead, we will refer to related work as far as such extensions are required. In fact, however, it should always be one of the main objectives to keep as many safety assurance activities as possible at development time and to avoid complex mechanisms like runtime verification in order to increase the likelihood that a safety assurance approach is accepted by certification bodies.

A first major step to overcome the challenges identified in the previous section is to consider a system not as a monolithic whole, but as a composition of sub systems and to adopt available safety engineering techniques to support modular safety assurance. More recent standards like the ISO 26262 [4] already define concepts like the so-called ‘safety element out of context (SEooC)’, which provide a normative basis for modular safety assurance. Though they do not provide concrete suggestions on how to realize modular safety assurance, we have developed a sophisticated set of modular safety engineering techniques over the last years, which build a sound basis for modular safety assurance. However, conventional modular safety assurance approaches assume that the systems are integrated at development time and they do not support runtime modifications in the sense of self-adaptivity. In order to extend these approaches for supporting open adaptive systems, we used approaches that are available in the field of software engineering for self-adaptive systems as it will be described in more detail later on.

Summarizing this means that the combination of modular safety assurance approaches with engineering principles for self-adaptive systems is a key to advance safety engineering techniques in order to support the safety assurance of open systems of systems. Following this basic idea, our research approach was structured as shown in Figure 1-2, namely we combined two threads of research:

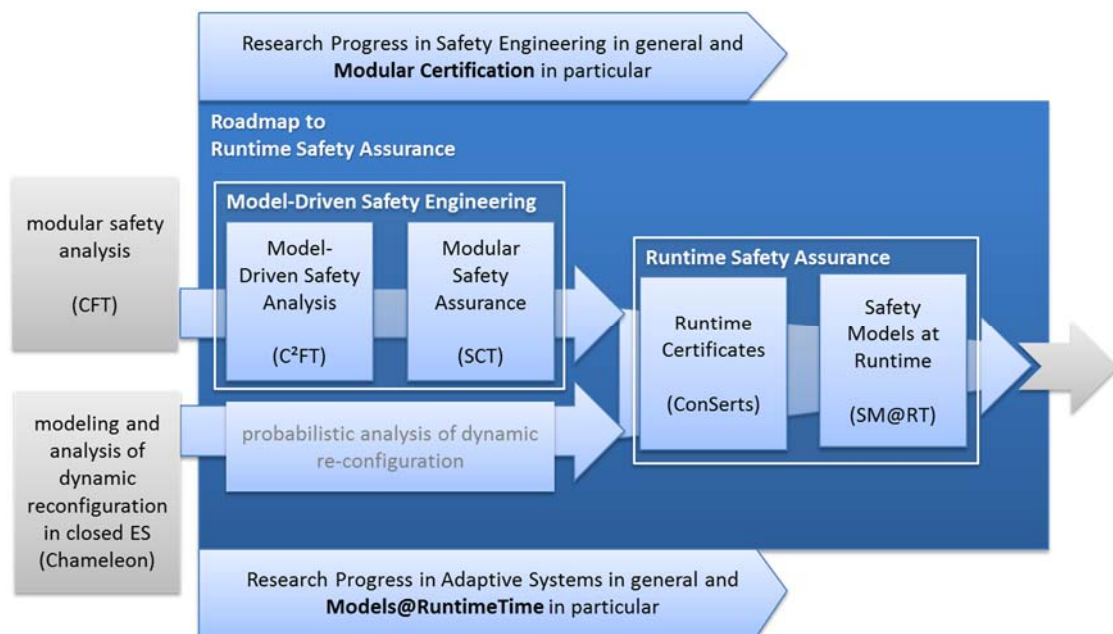


Figure 1-2 Roadmap of our research on safety assurance of open adaptive systems.

First, safety assurance builds the basis for a safety assurance framework for open systems of systems. More precisely, modular safety assurance is an indispensable prerequisite for runtime safety assurance. Traditional safety analysis approaches, for example, are not modular and can only be applied to monolithic systems as a whole. The modular, compositional character of systems of systems can therefore not be reflected in

traditional safety assurance approaches. There are a few approaches available supporting modular safety assurance, which will be described in more detail in chapter 4. In the first step, we therefore modularized safety analysis techniques. As a starting point we used component fault trees [26], a modularization of the established fault tree analysis technique, which was introduced by Kaiser et al. In general, modular certification is a major ongoing trend in the safety engineering community, so that the assurance of open systems of systems profits from continuous results in this field. In addition, we modularized the failure modes and effects analysis in order to support the second important safety analysis technique. Since safety analyses are only one safety artifact, we additionally modularized safety requirements in form of safety concept trees [65][68].

Modularization is however not sufficient. Since safety assurance of open adaptive systems partially takes place at runtime without human interaction, the safety artifacts must be formalized so that a system is enabled to perform the assurance steps and to interpret the results at runtime. We therefore applied model-driven development principles to safety assurance activities, which is an ideal basis to provide a sufficiently formal basis (cf. box ‘model-driven safety engineering’ in Figure 1-2). First, models provide an intuitive yet sufficiently formal means to specify safety aspects. Second, model-driven development of embedded systems is gaining more and more acceptance in research as well as industry. And the seamless integration of development models on the one hand and safety models on the other hand provides several advantages. First, it improves the consistency between development and safety models. Second, the developers are used to the modeling notations so that it is easier for them to define safety models using similar notations. Third, safety aspects become an integral part of the model. If, for example, a component is reused, its safety model is reused, as well. And if components are to be connected, the safety models can be included in interoperability checks etc. Fourth, the seamless integration helps to improve the semantics of safety models. If, for example, a failure mode is associated with a signal, it can be formally mapped to the respective signal in the architecture model. This link therefore provides an additional semantic information for the analysis and interpretation of safety models, which is very important for automating single steps of the safety assurance process. Based on our work, all safety assurance artifacts are available as part of an integrated model-driven development process [64]. Most important are model-driven representations of safety analyses and safety assurance models incorporating safety concepts and safety cases. The former are available as so-called component-integrated component fault trees (C²FT), which we developed as an extension of CFTs [57]. Moreover, an extension of safety concept trees incorporates a model of safety requirements and safety cases [65][68]. The concepts of model-driven safety engineering will be described in detail in Chapter 4.

Model-driven safety assurance is a very important thread of research in the safety assurance community, in general. As regards runtime safety assurance, the underlying formalisms provide an essential basis, but they are not sufficient. Therefore, we advanced these results to runtime safety assurance approaches (cf. box ‘Runtime Safety Assurance’

in Figure 1-2). In order to find an appropriate approach to transfer model-driven safety assurance approaches to runtime, we regarded a second thread of research from a completely different research community: the development of open self-adaptive systems.

In general, the modeling of the adaptation behavior of an adaptive system can be used as a basis to understand how dynamically adaptive systems work and how they can be systematically engineered. We developed sophisticated approaches for the modeling of adaptive systems [201] [202]. And we also derived approaches for performing probabilistic analyses based on models of adaptive systems [93], which can be used to support safety analyses for adaptive systems. Therefore, a broad basis for analyzing the adaptation behavior of adaptive systems is already available. However, the details of modeling adaptation will not be regarded in detail in this document. Instead, we will focus on the general underlying idea of Models@Runtime as one major paradigm for engineering open adaptive systems as it is shown in Figure 1-3.

In model-driven engineering, a developer uses models (e.g., safety models) to analyze and to optimize specific characteristics of systems (e.g., safety). To this end, she or he has to predict the system's runtime context since system properties like safety depend on the system and its interaction with its environment at runtime.

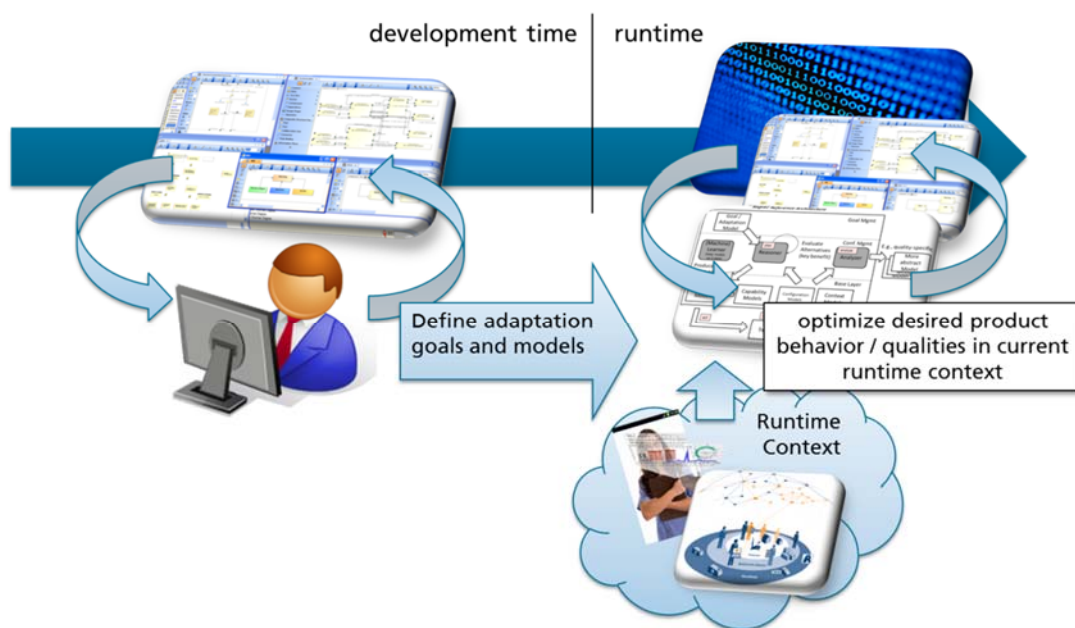


Figure 1-3 Principle idea of models at runtime

For open and adaptive systems, however, the runtime context cannot be sufficiently predicted. Therefore, it is the idea of models at runtime to make important models available at runtime so that the system can perform model based analyses in order to optimize itself autonomously to a given runtime context. This requires a) a more formal and more efficient runtime representation of the models and b) an additional specification

how the system shall interpret the analysis results and how it should adapt to meet its optimization goals. To this end, a so-called adaptation model must be specified, which formalizes the developers' knowledge on how to adapt the system in a given runtime context. Applied to safety assurance this means that safety assurance models have to be made available at runtime. We therefore applied the idea to define safety models at runtime (SM@RT) - so that the system can run simplified safety checks at runtime. In analogy to the adaptation behavior, an additional behavior specification is required that tells the system how it shall react to the runtime analysis results, e.g., how it shall decide on whether or not it is safe in the given context, or how it must adapt in order to assure its safety. Though this approach shifts parts of the safety assurance process to runtime, the usage of models in combination with an explicit specification of how the system will interpret the model-based analysis results minimizes the uncertainty of runtime safety assurance. Nonetheless, this approach leaves sufficient freedom to flexibly adapt the system at runtime. The idea of safety models at runtime is described in chapter 5.

In principle, all safety artifacts mentioned before can be shifted into runtime. But it is reasonable to leave as much responsibility as possible at development time in order to shift only a sufficient minimum of responsibility of safety assurance from a human being to the system. Therefore, we first focused on safety certificates at runtime. Namely, so-called Conditional Safety Certificates (ConSerts) provide a first step towards runtime safety assurance of open systems of systems. Using these safety certificates at runtime as a basic building block, we derived a generic safety assurance framework for open systems of systems. If each system of a system of systems provides a ConSert as black box safety interface, the ConSerts can be used to safely compose the systems. Additional information on how the safety is internally assured in the systems is not required. Therefore, ConSerts facilitate information hiding and enable a modular runtime safety assurance. In consequence, it is easily possible to flexibly integrate different, heterogeneous safety assurance approaches within the single systems. And due to the modularity the framework scales to very complex systems of systems.

Summarizing this means that we advanced safety assurance approaches from existing modular safety analyses to holistic *modular* safety assurance approaches. Then, we integrated these approaches seamlessly into a *model-driven* development setting, in order to increase the efficiency of safety assurance at development time and to provide an essential, sufficiently formal basis for runtime verification. In the final step, we merged the ideas of model-driven safety assurance and models at runtime for deriving a runtime safety assurance framework based on *safety models at runtime* (SM@RT) with a special focus on safety certificates at runtime.

1.2.4. Document Structure

As described in the previous section, the safety assurance of open systems of systems requires a combination of safety assurance approaches, model-driven development approaches, and models at runtime. To this end, the first part of this thesis introduces all

the basics on safety assurance and model-driven development as they are required for the realization of a runtime safety assurance framework. Chapter 2 therefore introduces the required knowledge about safety assurance before Chapter 3 describes the required concepts of model-driven development. Based on these fundamentals, Chapter 4 will then illustrate our research results how safety assurance approaches can be modularized and integrated into a model-driven design process. Finally, Chapter 5 will then present our research results on how the safety assurance concepts can be advanced to a runtime safety assurance framework.

2. Fundamentals of Safety Assurance

Experience in practice and academia shows that there are many misunderstandings regarding the intention and purpose of safety assurance. Therefore, this chapter provides an overview of the basic principles of safety engineering as a fundamental basis of safety assurance for open adaptive systems. To this end, Section 2.1 first introduces the basic terms and definitions before Section 2.2 gives an overview on the principle steps of safety engineering. The subsequent sections describe the single phases and the artifacts used in more detail, since these artifacts built the essential basis for safety models at runtime as they are used for the runtime safety assurance framework for open systems of systems. Finally, section 2.6 introduces fault tolerance mechanisms, which provide a further basis for runtime safety assurance based on error detection and handling.

2.1. Terms and Definitions

Often, there are misinterpretations of important terms that are relevant in the context of safety engineering. Terms like dependability, safety, reliability or availability are mixed up and used as synonyms. Therefore, this section introduces the most important terms that are of relevance for the safety assurance of open systems of systems.

Often, the confusion already starts with basic terms like dependability, availability, reliability, and safety. In fact, there are clear differences between the different terms. However, there are different sources providing different definitions that are inconsistent to each other. In the following, we refer to the widely used definitions of Laprie et. al [3] as well as different safety standards.

The term **dependability** covers a wide range of different aspects of system quality. It is a very generic term, which is used with very different meanings in different contexts. Following [3] it is defined as follows:

Dependability [3]

The *dependability* of a computing system is the ability to deliver a correct service.

The *service* delivered by a system is its behavior as it is perceived by its user(s).

A *user* is another system (physical, human) that interacts with the former at the service interface.

The *function* of a system is what the system is intended for, and is described by the system specification.

Correct service is delivered when the service implements the system function.

Laprie et al. additionally gave a more complex definition of the term dependability by structuring the definition of dependability to different elements as they are shown in Figure 2-1.

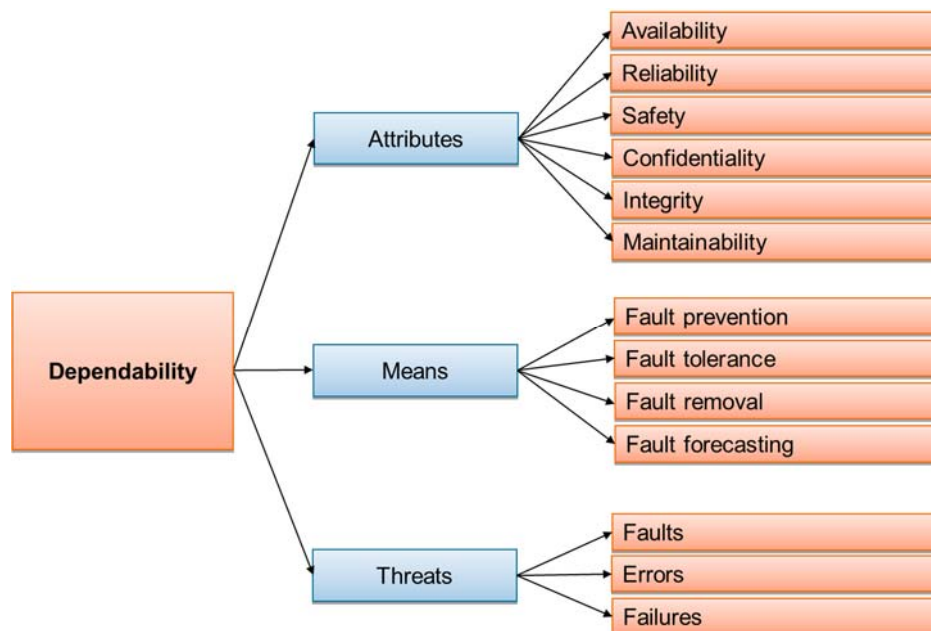


Figure 2-1 Definition elements of the term dependability

They define the term in three dimensions, namely *attributes*, *means*, and *threats*. The *attributes* dimension in some sense refers to different sub properties that must be fulfilled in order to yield dependable systems. Therefore, dependability subsumes the quality properties *availability*, *reliability*, *safety*, *confidentiality*, *integrity*, and *maintainability*. Up to now, however, there is no common agreement about the completeness of this definition. Nonetheless, we agree with the idea that dependability is a term covering different important quality characteristics and that these terms can therefore not be used synonymously. For safety assurance, the terms reliability, availability, and safety are certainly the most important ones and will be regarded in more detail later on.

The *means* dimension includes possible measures that can be applied to ensure dependability. All counter measures available today can be classified to *fault prevention*, *fault tolerance*, *fault removal*, and *fault forecasting*. In fact, this classification also holds for all counter measures available in safety engineering including the counter measures as they are defined in safety standards.

The *threats* dimension actually classifies the possible causes endangering system dependability. In fact, this classification to *faults*, *errors*, and *failures* can be found in similar ways in many safety standards and definitions.

Coming back the first dimension, it is important to understand the meaning of the different terms, since they are often mixed-up and used with a wrong meaning:

Availability [3]

Availability A is the property of a system, to fulfill its purpose at a given point of time.

$$A = \frac{\text{TotalTime} - \text{OffTime}}{\text{TotalTime}} = \frac{MTBF}{MTBF + MTTR}$$

with:

MTBF: Mean Time Between Failure

MTTR: Mean Time To Repair

Obviously, availability does not consider the fact how long a system has been running properly without any interruption. For IT-servers, for example, this metrics is often sufficient since a reboot can be tolerated as long as the server's down time is much lower than its up time. For embedded systems, however, it is often important to have metrics that refers to the time a system is running properly without interruption. Therefore, system reliability is often more important for embedded systems.

Reliability [3]

Reliability R is the property of an entity to fulfill its reliability requirements during or after a given time span under given application conditions. The reliability $R(t)$ is a function over the time t .

Safety is the third important quality property of embedded systems. Safety is defined in several norms. In the following, we use the definitions of the ISO 26262 [4], as it is one of the most recent standards.

Safety [4]

Safety is the freedom from unacceptable risk.

The risk associated with a system is defined as:

Risk [4]

Risk is the combination of the probability of occurrence of harm and the severity of that harm.

Usually, however, it is not possible to directly assess the harm that is potentially caused by a system. Instead, safety managers identify the hazards of a system.

Hazard [4]

Hazard is a potential source of harm.

In many domains, this vague definition is further refined. In the automotive domain, for example, 'hazards shall be defined in the terms of conditions and events that can be observed at the vehicle level' [4].

Regarding the probability of occurrence of harm, it is important to note that harm is only caused when a hazard, a specific environmental situation, and a specific operation mode of the system coincide. This coincidence is called *hazardous event*. The probability of occurrence of harm therefore depends on various different factors. In the automotive industry, the occurrence probability is determined by the probability of the relevant driving situations and the probability that a driver would be able to control the car in spite of a system failure.

In fact, hazards could be anything ranging from choking hazards of small parts to exothermic reactions of batteries in electric vehicles. Particularly for the development of embedded systems, however, the focus lies on hazards caused by malfunctions of the system. This subset is defined as functional safety:

Functional Safety [4]

Functional Safety is the absence of unacceptable risk due to hazards caused by malfunctional behavior of electric/electronic systems

In the context of functional safety, the focus lies on hazards that are caused by failures of an electric / electronic system. Based on these system level failures, safety engineers identify the cause-effect relationships in order to define appropriate counter measures. To this end, it is necessary to have a precise understanding of the different terms describing ‘problems’ in a system. The most relevant terms in this context are *mistake*, *fault*, *error*, and *failure* (cf. [3]). As illustrated in Figure 2-2, these terms can be related in a cause-effect relationship. If a developer makes a mistake, this may lead to a fault in the system, e.g., a non-initialized variable. At runtime, this fault may lead to an *error*, e.g., the variable takes a wrong value thus leading to an erroneous state of the system. An error may in turn lead to *failure*, e.g., a component sends an erroneous value to an actuator.



Figure 2-2 Relationship between Mistake, Fault, Error, and Failure

It is, however, important to note that the terms are sometimes used in exchanged meanings. Whereas, for example, the definition of Laprie et al. (cf. Figure 2-1) and the automotive safety standard ISO 26262 use the definition as described above, the avionics standard DO 178C [5] uses the terms fault and error in exchanged meanings. This means that in the avionics domain, a non-initialized variable would be called an error, whereas its manifestation at runtime would be called a fault.

In the remainder of this thesis, we will use the terms as defined in the ISO 26262:

Failure [4]

Termination of the ability of an entity to perform a function as required.

Error [4]

Discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition.

Fault [4]

Abnormal condition that may cause a reduction in, or loss of the capability of an element to perform a required function.

Although it is not explicitly considered in most safety standards, it is important to include the developers' mistakes as well, in order to yield a complete picture. Therefore, we define a *mistake* as follows:

Mistake

A mistake is made by a human during development time, for example, due to non-attention, and may lead to a fault in the system.

Once safety engineers have identified the hazards and analyzed the cause-effect-relationships leading to these hazards, they have to identify appropriate counter measures. As it was shown in Figure 2-1, Laprie et. al. identified four principle classes of counter measures: *fault forecast*, *fault prevention*, *fault removal*, and *fault tolerance*. Regarding the counter measures used in practice as well as the respective demands in safety standards, these four classes cover all possible measures.

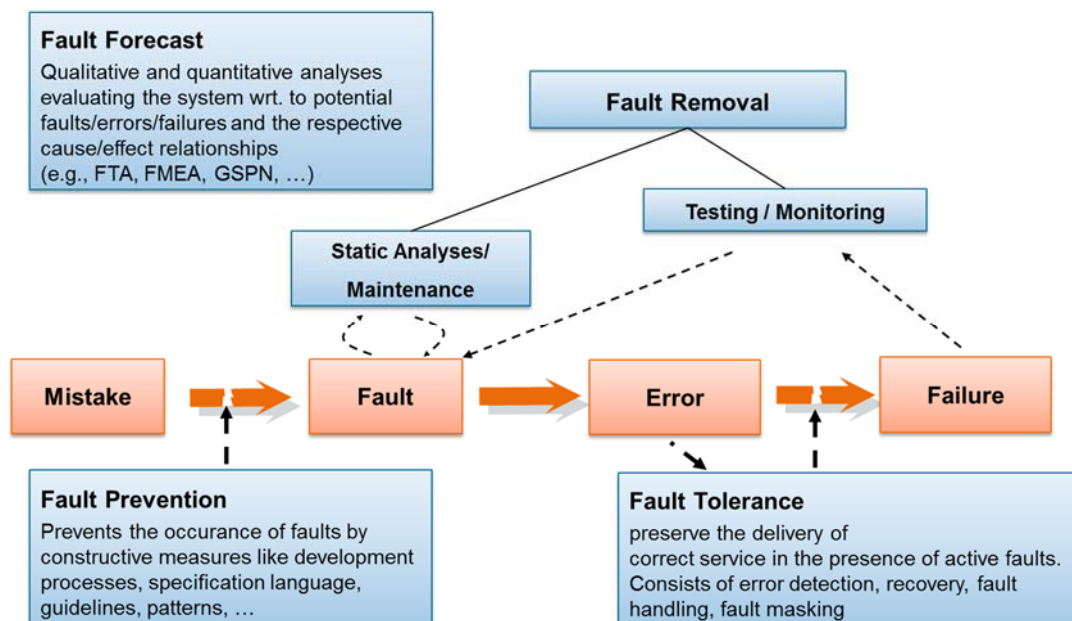


Figure 2-3 Relation between possible Counter Measures

A safety engineer has to select a set of necessary and sufficient measures leading to a sufficiently high probability that these measures will interrupt the fault-error-failure cascade before a hazard can occur. Figure 2-3 provides an overview of the different classes and shows at which point in the fault-error-failure cascade they can be applied.

Fault forecast approaches build the backbone for controlling the overall safety engineering process by supporting the identification of cause-effect-relationships and a quantitative or at least qualitative prediction of the system's safety. Typical examples are fault tree analyses (FTA), failure modes and effects analyses (FMEA), or generalized stochastic petri nets (GSPN). Obviously, such safety analysis techniques are of utmost importance in safety engineering.

Fault Forecast

Fault forecast approaches enable the identification of hazards, their causes, the related cause-effect-relationships as well as a quantitative and/or qualitative estimation of the expected system safety.

In order to increase system safety, it is obviously reasonable to prevent faults before they are created. To this end, the class of *fault prevention* techniques includes rigorous development processes, coding guidelines, modeling guidelines etc. By this means, the developer shall be strictly guided in order to prevent that developers' mistakes result in system faults. For example, if coding rules prohibit the use of pointers, it will be more unlikely to find any pointer-related faults in the system.

Fault Prevention

Fault prevention measures have the purpose to prevent the introduction of system faults through mistakes of developers.

In practice, rigorous development processes and additional means like coding rules are commonly accepted state-of-the-practice. In fact, fault prevention measures play a central role in most safety standards. Obviously, it is however impossible to prevent all possible faults just by the means of fault prevention measures. Therefore, it is essential to identify and remove system faults before the system's commissioning. To this end, the class of *fault removal* measures is used.

Fault Removal

Fault removal measures have the purpose to identify and to remove faults that have been introduced into a system.

Static fault removal measures do not require an execution of the system. They directly analyze the development artifact and identify faults in the analyzed artifact.

Dynamic fault removal measures observe the executed system in order to identify failures. Based on the failures observed, the engineer must identify and remove the causing faults.

Today, it is common practice to use static code analyses like abstract interpretation (static fault removal) as well as different testing approaches (dynamic fault removal). Liggesmeyer gives a very good overview on available approaches in [6].

In spite of the various available fault removal approaches, it is however not possible to completely avoid faults in the system. First, today's systems are often too complex to ensure freedom from faults. Second, also hardware faults like bit flips may cause software errors. Therefore, it is essential to detect and handle errors at runtime in order to avoid the occurrence of hazards. To this end, the class of *fault tolerance* measures can be applied.

Fault Tolerance

Fault tolerance measures have the purpose to identify and to handle errors at runtime.

Following the definition of [3], fault tolerance measures consist of *error detection* and *recovery*.

Error Detection

Error detection originates an error signal or message within the system.

Concurrent error detection takes place during service delivery. *Preemptive error detection* takes place while service delivery is suspended.

Recovery

Recovery transforms a system state that contains one or more errors and (possibly) faults into a state without detected errors and faults that can be activated again. Recovery includes the following elements:

Error handling

eliminates errors from the system state:

- a) rollback: system state is set back to a previously saved checkpoint
- b) roll forward: system is set to a new, error-free state

Fault handling

prevents located faults from being activated again

Fault diagnosis

identifies and records the cause(s) of error(s)

Fault isolation

performs physical or logical exclusion of the faulty components from further participation in service delivery

System reconfiguration

either switches in spare components or reassigns tasks among non-failed components

System reinitialization

checks, updates and records the new configuration and updates system tables and records.

For safety engineering, it is often sufficient if recovery mechanisms bring the system to a safe state, e.g., in the simplest case by switching it off. In this case, it is not necessary to identify and isolate the causing faults.

2.2. Safety Engineering in a Nutshell

While the previous section introduced the definitions of the most important terms, it is very important to understand them in the context of a safety engineering lifecycle. Therefore, this section gives a brief overview on the basic steps of safety engineering from the hazard analysis to the final safety case.

Engineering a safety critical system includes various different aspects. As illustrated in

Figure 2-4, safety standards define the basic demands on the development of safety-critical systems. These requirements can be subdivided into three types:

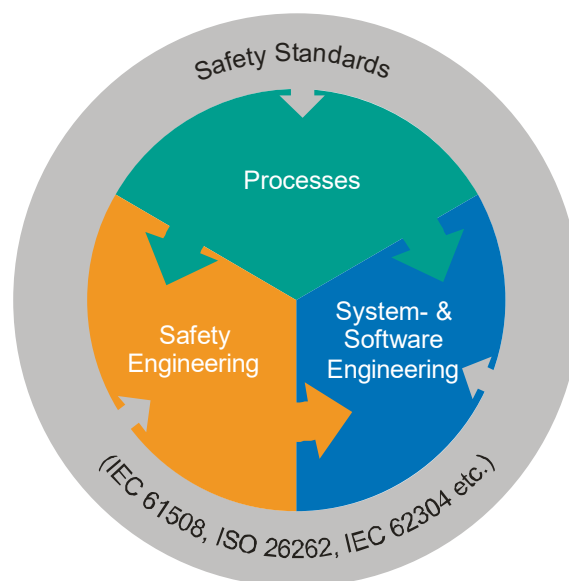


Figure 2-4 Elements of safety-related development

The first type of requirements addresses the development process as one of the most important fault prevention measures.

The second type of requirements addresses the development methodologies and techniques that have to be used. This includes requirements on fault prevention, fault removal, as well as fault tolerance measures. Fault prevention measures could be coding guidelines, limitations in the use of programming and modeling languages, or concrete demands on which aspects have to be defined in a software architecture. Fault removal measures are addressed by defining detailed requirements on testing and static analysis approaches that have to be applied in order to ensure the quality of safety-critical systems. As regards fault tolerance measures, these requirements further define demands on runtime error detection and handling mechanisms that have to be implemented in the system.

The third type of requirements addresses the actual safety assurance activities. This mainly includes all fault forecasting approaches that have to be used during development, i.e. all approaches necessary to identify the hazards, to assess the resulting risk, to analyze the cause effect relationships, to define a safety concept, and to eventually derive a safety case. As described above, these activities build the backbone of every safety-related development. Understanding the cause-effect relationships that lead to hazardous situations is the prerequisite for selecting appropriate counter measures and for proving the fulfillment of safety goals.

If we have a closer look at the safety engineering lifecycle, safety engineers have to identify all relevant hazardous events and to assess the associated risks as the first step of the lifecycle. This step is called ‘hazard analysis and risk assessment (HRA)’ as shown in Figure 2-5. This step is performed during the very early phases of the development process, at the latest when the system requirements are available.

As a result of this step, safety goals are defined as top-level safety requirements, which build an extension of the system requirements. In the subsequent steps, a safety manager must incrementally refine the safety goals. Usually, any safety requirement consists of a functional part and an associated integrity level. The functional part defines what the system must (not) do, whereas the integrity level defines the rigor demanded for the implementation and verification of this requirement. The integrity level depends on the risk associated with the hazardous event, which is addressed by the safety goal. For example, ISO 26262 defines so-called *automotive safety integrity levels (ASIL)*.

Once the safety goals have been defined, the system development continues through different phases like the definition of a network of functions, the system and software architecture, the design, and finally the implementation of the system. The subsequent steps in the safety engineering process should be performed in parallel with the development activities (though this is often not the case in practice). To this end, the available development artifacts are used as input to safety analyses at different abstraction levels in order to identify potential causes of the identified system failures. A

wide range of different safety analysis techniques is available. Failure Modes and Effects Analysis (FMEA) and Fault Tree Analysis (FTA) are certainly the most widely used safety analysis techniques in practice.

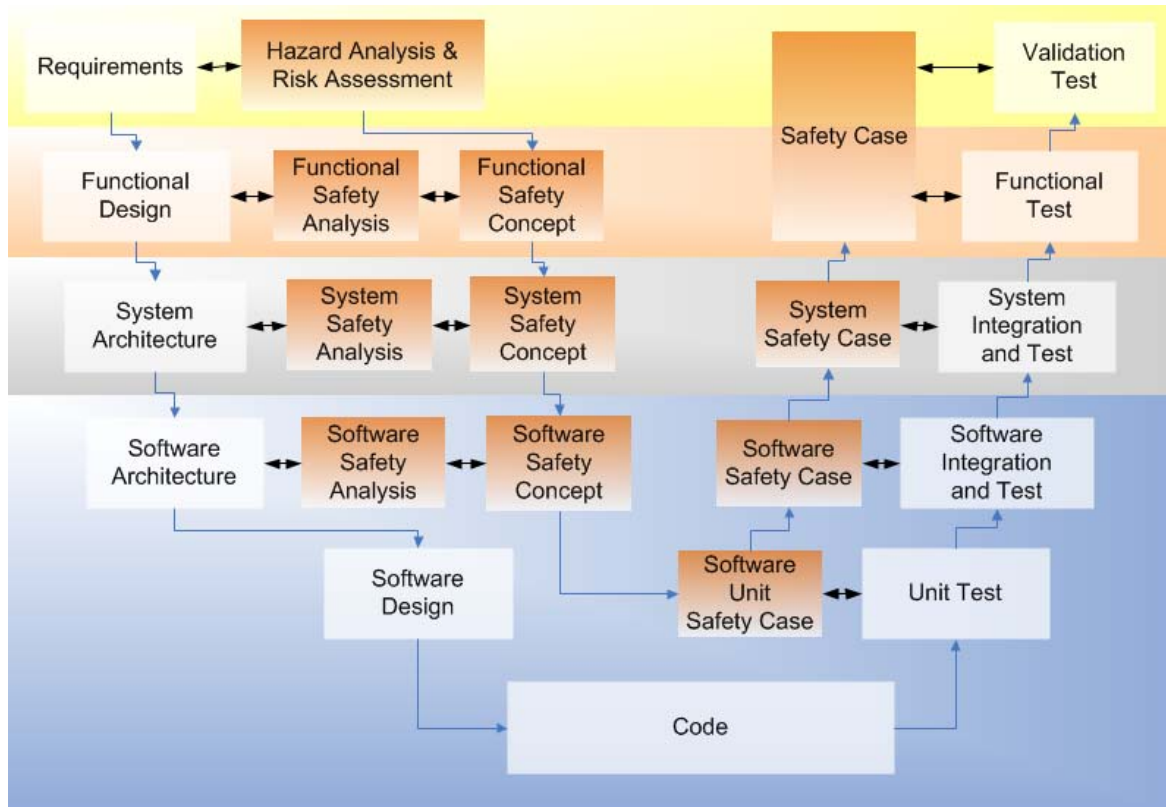


Figure 2-5 Safety Engineering Lifecycle

Based on these results, a safety manager derives a safety concept. Following the idea of ISO 26262, a safety concept can be defined as

Safety Concept [4]

A specification of the safety requirements, their allocation to architectural elements and their interaction necessary to achieve the safety goals, and information associated with these requirements.

In the same way as developers incrementally refine the system over the different development phases, the safety manager analyzes the refined development artifacts step by step and refines the safety concept accordingly. In the beginning, the safety concept is focused on purely functional aspects and will be step wisely refined to a technical concept at system and software level. The safety concept plays a very important role in safety engineering. It defines which countermeasures have to be applied and how the measures in combination shall ensure the fulfillment of the safety goals.

Finally, the safety manager has to define a safety case, which forms the basis for the certification or a final assessment of the systems' safety. A safety case can be defined as

Safety Case [4]

An argument why an item is safe supported by evidence compiled from work products of all safety activities during the whole lifecycle.

To this end, the safety manager uses the safety concept as a basis and tries to prove the fulfillment of each and every safety requirement with so-called evidences. Evidence can be anything supporting an argument in the safety case. The results of validation and verification activities as well as safety analysis results are of particular importance. Since a safety case compiles all evidences that are relevant for proving the system's safety, it is an efficient basis for safety certification or assessment.

Depending on the application domain, today, these steps are performed rather informally. Hazards analyses and risk assessments are usually based on simple spreadsheets and safety concepts as well as safety cases are described as informal text documents or requirements databases. Mainly for safety analyses, dedicated formal approaches and tools for performing FTAs and FMEAs are used. As a further important aspect, most safety approaches used today assume the overall system as a static, monolithic item, i.e. they hardly support modularity and variability.

Such an informal, monolithic approach is used since it seems to be more efficient from a practitioner's point of view. And indeed, using informal documents like spreadsheets and text documents is often faster than obeying to the stricter rules of more formal approaches and the according complex tools. However, embedded systems are not developed as static monolithic systems from the green field. There are many changes already during development. Usually systems are not built from scratch, but by reusing and adapting existing components and systems. Moreover, they are characterized by a very high variability. For example, for an engine control software more than one thousand variants have to be derived per year. And each variant has to undergo a safety assessment. Considering these aspects, the informal approaches used today do not scale and easily become very inefficient.

Regarding open systems of systems, the approaches used today cannot be used. Assuring their safety requires modular concepts in order to be able to modularly assess the safety of the single systems in order to be able to safely compose them to systems of systems at runtime. To this end, the safety artifacts must be described in a more formal way so that they can be interpreted and validated by the systems at runtime.

For these reasons, current safety assurance approaches must be extended. Modular, model driven approaches as they are described in the subsequent chapter are emerging as promising new way of safety assurance – in practice as well as in research. First, such approaches are interesting for increasing the efficiency of safety assurance in the context

of the fast increasing complexity and variability. Second, these approaches build an ideal basis for assuring the safety of open adaptive systems since they are modular, and since the models are sufficiently formal to be used at runtime.

Before the modularization and formalization of the safety artifacts is described in the subsequent chapter, the following sections provide a more detailed introduction to the single artifacts in order to provide a more detailed basic knowledge.

2.3. Hazard Analysis and Risk Assessment

Any safety engineering lifecycle starts with a hazard analysis and risk assessment (HRA). The concrete HRA approach varies from domain to domain, but the core concepts are very similar. In almost all domains, the HRA can be subdivided into two basic steps. The first step is the identification of hazards. In the second step, the associated risk is assessed for each of the identified hazards

2.3.1. Hazard Identification

In the context of functional safety, hazard identification is identical to the identification of system-level failures. This is a very creative process, which can be supported by different safety analysis techniques. In principle there are two main approaches to hazard identification.

The first approach has a focus on the system's interface. This approach is based on the assumption that any hazard can only occur at the direct interface between a system and its environment. To this end, it is however necessary to consider the complete interface. This means that it is not sufficient to consider explicitly defined interfaces like steering or brake actuators. In fact, there are additional implicit interfaces like for example electricity that could cause an electric shock if the power supply of a xenon lighting system is not switched off after an accident. Once the interface is defined, safety managers use approaches like the hazard and operability study (HAZOP, cf. Section 2.4.2). To this end, different guidewords are applied for identifying potential hazards at the interface. If we for example consider the interface *steering*, we could apply the guidewords like *too high* or *too low* leading to potential hazards like 'The *steering* sets a *higher* steering angle than it was desired in the current situation'. Such formulations build the basis for a systematic brainstorming. Relevant hazards are then further refined to yield more precise and thus more useful definitions. For example, in the example it would be necessary to define when exactly a steering angle is considered too high etc. Although it is not possible to anticipate all possible hazards that will occur in the field, applying a predefined list of guidewords to an interface leads to a systematic approach making it more unlikely to omit hazards. Alternatively, often approaches based on a failure modes and effects analysis (FMEA, cf. Section 2.4.1) are used.

However, the focus on the interface might lead to the omission of hazards that are caused by specific interactions of different actuators. As an example let us consider an

electric power train with two separate e-machines driving the rear wheels. One important hazard is that the two e-machines create different torques at the different wheels, which in turn causes an undesired yaw torque. In a strictly interface-based approach, both machines would be analyzed separately making it very unlikely to identify hazards like an undesired yaw torque. Therefore, it is alternatively reasonable to analyze the functions of a system instead or in addition to its interface. This is the typical approach in the avionics domain, where this approach is called Functional Hazard Analysis (FHA). To this end, a complete list of specifications of the system's functions is required. Using approaches like HAZOP or FMEA again, safety managers try to identify possible hazards. In the avionics domain, there are special lists of possible failures modes available that can be applied as guidewords to different functions. If we, for example, consider the function *torque vectoring* of the electric car and apply the guideword *high*, this could lead to the hazard 'the *torque vectoring* creates a *higher* yaw momentum than desired'.

2.3.2. Risk Assessment

Once the potential hazards are identified, safety managers have to identify the associated risks. To this end, different assessment parameters must be defined. Following the definition of risk given in Section 2.1, one parameter specifies the severity of the harm that might be caused by a hazardous event. A set of further parameters specifies the probability that a hazard, i.e. a failure at the system border, actually leads to a hazardous event. Which concrete parameters have to be used depends on the application domain and the relevant safety standards. In most approaches, however, at least two parameters are used: Hazards often only cause harm if they occur in a certain environmental situation. Therefore, the probability of an environmental situation must be defined. In the automotive domain, this is done by specifying the likelihood of driving situations. Moreover, it might be possible that harm can be prevented by persons or external safety measures. In the automotive domain, the controllability parameter is used to define the likelihood that the driver or any other person involved can prevent the hazardous event in spite of a system failure.

In order to derive an integrated risk assessment, safety engineers must integrate all of these parameters to single criticality classes. To this end, most risk assessment approaches used today follow a risk graph approach, as it is for example described in the IEC 61508 [2] (cf. Figure 2-6).

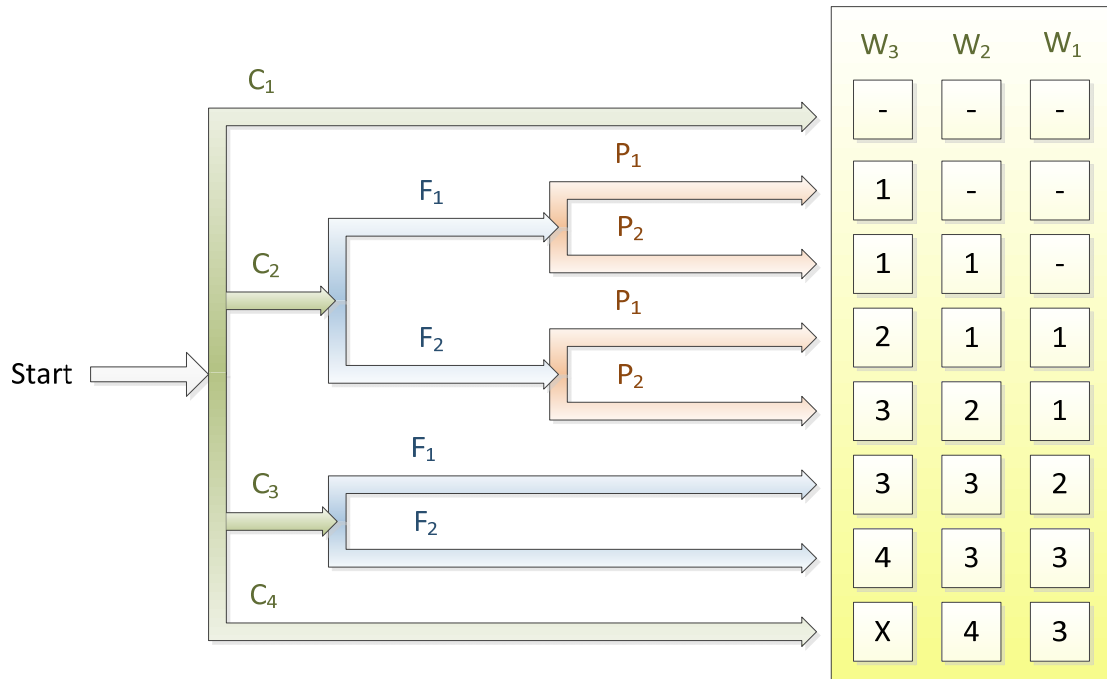


Figure 2-6 Risk graph following IEC 61508-5 [2]

A risk graph is used to determine the risk of a hazardous event based on different parameters as they also have been defined above:

- The *C*-parameter reflects the consequences, i.e. the severity of harm, on an ordinal scale (e.g., *C*₁: minor injuries to *C*₄: very many people killed).
- The *F*-parameter reflects the frequency of exposure in the hazardous zone (e.g., *F*₁: rare exposure to the hazardous zone and *F*₂: frequent to permanent exposure in the hazardous zone).
- The *P*-parameter reflects the possibility of avoiding the hazardous event (e.g., *P*₁: possible under certain conditions and *P*₂: almost impossible)
- The *W*-parameter reflects the probability of the unwanted occurrence. However, this parameter does not reflect the probability of a failure within the embedded system and does not consider safety measures implemented in the system. But it is used to reflect different risk reduction measures like for example covers, cages, or mechanical pressure valves (e.g., *W*₁: A very slight probability that the unwanted occurrences will come to pass and only a few unwanted occurrences are likely to happen. *W*₃: A relatively high probability that the unwanted occurrences will come to pass and frequent unwanted occurrences are likely to happen)

Based on these parameters, it is easily possible to determine the related risk and the related safety integrity level that is required to mitigate the analyzed hazardous event. In contrast to the ISO 26262, as it was described before, the IEC 61508 defines safety integrity levels ranging from SIL1 to SIL4 with SIL4 reflecting the highest possible integrity level.

In the first step, the consequences are assessed. In the case of C_1 , for example, no special risk reduction measures are required as only minor injuries are possible. In case of very critical consequences (C_4), it is not possible to have a lower integrity level than SIL 3 and neither the exposure nor the controllability can reduce the risk. Only the occurrence probability based on external risk reduction measures has an impact on the integrity level. The “X” in this case means that a single safety system is not sufficient to reduce the risk to an acceptable level, i.e. further external risk reduction measures are required (in order to reduce the W-parameter) or redundant safety systems are required. Depending on the path taken based on the evaluation of the C-parameter, the exposure F is regarded in the second step and the controllability P in the third step. Finally, the occurrence probability W is considered.

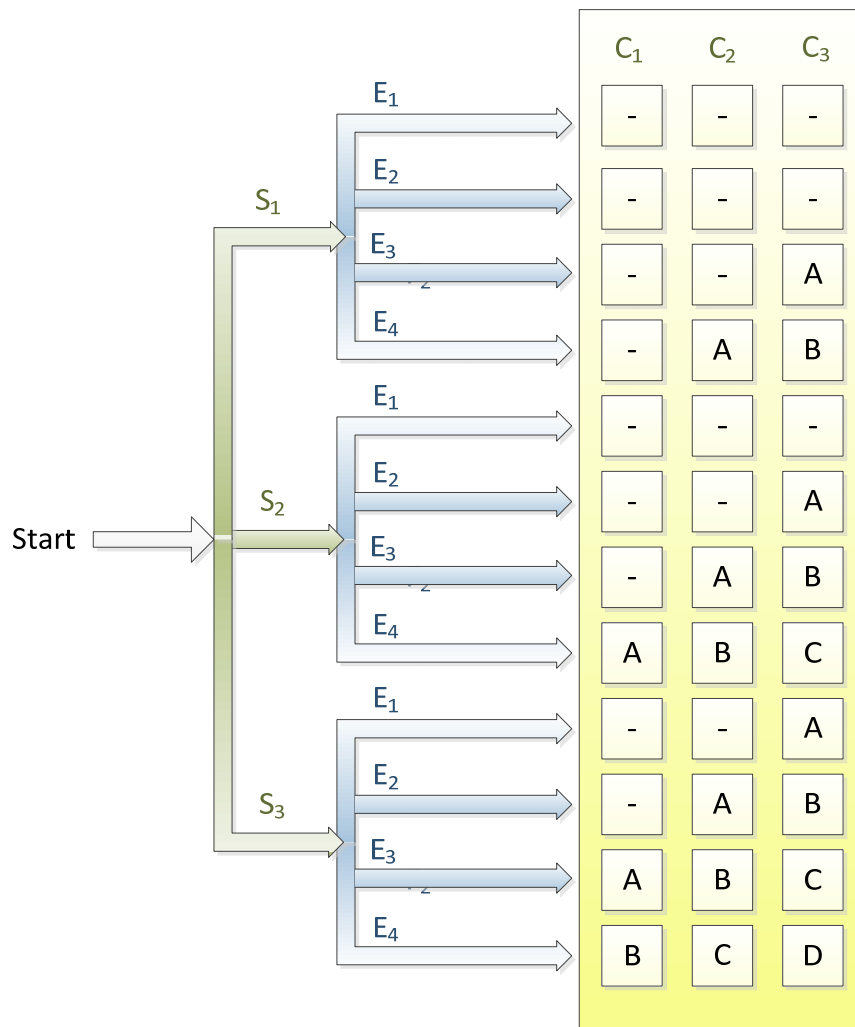


Figure 2-7 Instantiation of the risk graph approach for the automotive industry based on ISO26262

This generic approach has to be tailored to each application domain and possibly to each system. The identification of the acceptable risk and the associated risk graph parameters is a non-trivial step and will not be considered in this document. In order to release developers from this difficult step, the ISO 26262 predefines a simplified risk graph including all parameters for automotive systems. The resulting instantiation of the risk graph for the automotive industry is shown in

Figure 2-7. As mentioned earlier, the ISO 26262 defines the integrity levels as automotive safety integrity levels (ASIL) A to D, with ASIL D being the most critical one. The ISO26262 considers only three risk parameters:

- S: severity, which maps to the consequence parameter C of the IEC 61508
- E: exposure, which maps to the exposure/frequency parameter F of the IEC 61508
- C: controllability, which maps to the controllability parameter P of the IEC 61508
- The W-parameter as it is considered in the IEC 61508 is not considered in the ISO 26262

As shown in detail in Table 2-1, each of the risk parameters can have four different values. The parameter values S0 (no injuries) and C0 (controllable in general) are not explicitly considered in the risk graph, as obviously no harm would be caused in this case.

Class	S0	S1	S2	S3
Description	No injuries	Light and moderate injuries	Severe and life-threatening injuries (survival probable)	Life-threatening injuries (survival uncertain), fatal injuries
Class	E1	E2	E3	E4
Description	Very low probability	Low probability	Medium probability	High probability
Class	C0	C1	C2	C3
Description	Controllable in general	Simply controllable	Normally controllable	Difficult to control or uncontrollable

Table 2-1 Risk parameters defined in ISO 26262 [4]

At first sight, the risk graph of IEC 61508 seems to be more rigorous since for example a high severity immediately leads to at least a SIL 3. However, it must be considered that the ISO 26262 considers different parameter classes. For example, the severity value S_4 in the ISO26262 means “life-threatening and fatal injuries”, which can be seen between the consequence classes C_2 und C_3 of the IEC 61508, which mean “serious permanent injuries to one or more persons; death to one person” and “death to several people”, respectively. As a further aspect, the acceptable risk might be defined differently in different application domains.

Though the parameters and the parameter values used for the risk assessment must be adjusted to the different application domains, the risk graph approach is applied across various different application domains and can therefore be seen as (one of) the major risk assessment techniques.

2.4. Safety Analysis Techniques

Once the risks have been identified and assessed, a safety engineer must understand the potential causes and the cause-effect-relationships. To this end, she or he uses different safety analysis techniques. This section gives a brief overview on the most important analysis techniques used today.

Usually, it is required to combine different analysis techniques. At least, most standards demand the combination of so-called deductive analyses with inductive analyses. Deductive analyses like a fault tree analysis (FTA) use the effect, e.g., the hazard, as starting point and try to deductively derive possible causes that might lead to the analyzed effect. Inductive analyses like the failure modes and effect analysis (FMEA) on the other hand start from potential failure modes of a system element and try to identify potential effects.

2.4.1. Failure Mode and Effects Analysis (FMEA)

The Failure Mode and Effects Analysis (FMEA) was originally developed by NASA for the quality assurance of the Apollo projects in the 1960ies. During the course of the following decades it was increasingly used in a wide range of application domains. In Germany, the FMEA was first standardized in 1980. Today, it is used in almost any application domain.

Since the FMEA is applicable to nearly each problem and at each system level, different specializations of the FMEA have been derived. The *construction*-FMEA which is also named product- or design-FMEA regards the constructive specification of the characteristics of a system part like a capacitor or a microcontroller. The *process*-FMEA is focused on the analysis of processes like for example manufacturing or maintenance processes. The result of a FMEA is usually documented in tables.

Methodology

A FMEA is conducted by a team of all relevant stakeholders and experts, whose knowledge and expertise is required to complete the analysis. A trained FMEA moderator leads the meetings for ensuring a systematic analysis process. A FMEA is applied to single elements of a system at different hierarchy levels. In the first step, the team tries to identify potential failure modes of the component. Then the possible effects of a failure mode have to be identified. Furthermore, the team has to identify potential causes that could lead to the failure mode. After they have identified failure modes, the related effects and possible causes, the team describes already existing counter measures and assesses whether or not further actions are required in order to ensure the element's quality. To

this end they define necessary actions with a deadline and directly assign them to a responsible person.

In order to have a possibility to assess the criticality of a failure mode and to prioritize the usually long list of identified failure modes, the original FMEA was extended to a Failure Modes, Effects and *Criticality* Analysis (FMECA). Today, the term FMEA usually refers to an FMECA. In an FMECA, the team has to define different parameters that classify the criticality of the single findings: First, they have to define an Occurrence Probability (O) of the failure mode. Usually this is done using an ordinal scale ranging from 1 (very unlikely) to 10 (very probable). As a second parameter they have to define the severity (S) of the expected effect – again using an ordinal scale with 1 being the lowest severity. The third parameter describes the probability that the corresponding error can be detected (D) in order to prevent the effect. For this parameter, the ordinal scale ranges again from 1 (very likely to be detected) to 10 (very unlikely to be detected). These three parameters are then combined to a *risk priority number* (RPN), which is usually derived by multiplying the single parameters, so that it can range from 1 to 1000. Based on the RPN, the team can assign a priority to the single entries in an FMEA table.

As a further extension, the FMECA has been extended to the **FMEDA** (Failure Modes, Effects and *Diagnostics* Analysis). The FMEDA aims at defining a quantitative failure rate for a failure mode, a diagnostic coverage (DC) and a safe failure fraction (SFF). The diagnostic coverage indicates the probability that dangerous errors can be detected. The SFF is defined as the ratio between the average rates of safe errors plus the average rate of dangerous, but detectable errors, over the entire average error rate of an element. To this end, a conventional FMECA is conducted first. Then, all identified failure modes are subdivided into safe, detectable-dangerous and non-detectable-dangerous errors. Afterwards the DC and the SFF are calculated.

Advantages and disadvantages

The FMEA is an inductive safety analysis, since it identifies failure modes of system elements and tries to derive the related effects. It can be used on nearly any problem and at any system level in order to completely determine the risks of a system. By its team-oriented approach, it integrates specialized knowledge from different perspectives. Conducting an FMEA usually requires only little knowledge of methods and also a wide range of supporting tools is available. As a disadvantage, the FMEA does not support the analysis of error combinations. Except for some limited capabilities of the FMEDA, FMEAs do not support quantitative analyses.

Further References

In Germany the FMEA is standardized by the DIN 25 448 [12]. The IEC 60812 [21] describes both the FMEA, and the FMECA. The latter is also addressed in detail by the MIL-STD-1629A [34]. The conduction of a FMEDA is standardized in the IEC 61165 [23]. The System-FMEA is described in detail in [46] and in [9]. Moreover, there are two additional standards from the Society of Automotive Engineers, which describe further

adjustments of the FMEA for different applications. The J1739 [43] is a specialization for the automotive domain whereas the ARP5580 [42] should be used for other application domains.

2.4.2. Hazard and Operability Analysis (HAZOP)

The Hazard and Operability Analysis (HAZOP) originates from the chemical industry and was developed during the 1960ies and 1970ies. The goal of a HAZOP is the identification of possible hazards of an industrial plant or industrial process. Today, the HAZOP is applied in a wide range of different application domains.

Methodology

Ideally, the HAZOP is conducted by an interdisciplinary team consisting of five to seven persons, as soon as first drafts of a design of the system or component are available. In order to identify potential hazards, the HAZOP is based on a set of guide words as they are shown in Table 2-2. All components are successively analyzed by applying the guide words for identifying possible deviations from the expected behavior.

The guidewords describe hypothetical deviations from the expected behavior. To this end, they are combined with different parameters that are relevant for the analysis (like for example temperature or pressure) in order to define questions that support the analysis team in identifying potential failure modes. For each identified failure mode, the team tries to identify possible causes and effects and derives required counter measures. In some sense, a HAZOP is therefore similar to an FMEA and the main advantage is given by the guidewords.

Guide Word	Meaning
NO OR NOT	Complete negation of the design intent
MORE	Quantitative increase
LESS	Quantitative decrease
AS WELL AS	Qualitative modification/increase
PART OF	Qualitative modification/decrease
REVERSE	Logical opposite of the design intent
OTHER THAN	Complete substitution
EARLY	Relative to the clock time
LATE	Relative to the clock time
BEFORE	Relating to order or sequence
AFTER	Relating to order or sequence

Table 2-2 HAZOP - guide words

For example, the application of the HAZOP to a vehicle brake system leads to the following breakdown conditions [6]: System does *not* brake, brakes *more* than normal, brakes *less* than normal, accelerates (*reverse*). Not all guidewords lead to reasonable questions. Since a brake system can usually not cause acceleration, the application of the guideword *reverse* does not result in a reasonable question. Nonetheless, the team should consider as many guidewords as possible in order to reduce the likelihood of omitting possible failure modes.

The result of the HAZOP is finally a table including all identified deviations, the identified causes and consequences as well as a list of appropriate counter measures. A possible form of the table is defined in the IEC 61882 [21] as it is shown in Figure 2-8.

Advantages and disadvantages

The main advantage of the HAZOP is given by the guidewords, which systemize the creative process of failure mode identifications. Despite the guidewords, the HAZOP is quite similar to an FMEA. As the FMEA is more widely used in many domains, it is often common practice to use an FMEA as basic analysis technique and to enhance it by applying guidewords as it is described for the HAZOP. This is particularly true, since the guideword idea can be easily adapted.

For example, different variants for the identification of human failures or the analysis of software systems have been defined by Leveson for software safety analyses (Software Hazard Analysis and Resolution in Design - SHARD [71]). As an even more sophisticated extension, we have extended the idea of guide words to guide phrases which further increase the completeness of hazard and failure mode identification in the SafeSpection analysis technique [98].

References

The first manual describing the application of the HAZOP was provided by the Chemical Industries Association [17] in 1977. A short description of the analysis, including the advantages and disadvantages can be found in [31]. Detailed descriptions of the HAZOP can be found [30], [40] and [47]. The SHARD method is described in [71]. More details on SafeSpection can be found in [98].

Study title:		Page: of	
Drawing no.:		Date:	
HAZOP team:			
Part considered:			
Design intent:		Activity:	
Material:		Destination:	
Source:			
No.	Guideworld	Element	Deviation
		Possible causes	Consequences
		Safeguards	Comments
		Actions required	Action allocated to

Figure 2-8 HAZOP-Table from the standard IEC 61882 [21]

2.4.3. Event Tree Analysis (ETA)

The Event Tree Analysis (ETA) is used for the analysis of the consequences of an initial event [31]. In contrast to a fault tree analysis, the ETA is an inductive analysis, as it starts from a failure mode and step wisely identifies different effect scenarios.

Methodology

Event trees are usually defined as binary decision trees. The analysis starts with an initial event that has to be analyzed. The initial event could be a system or component failure or an external event affecting the system (e.g. an operating mistake). In the subsequent steps, possible event sequences are determined, which can occur as reactions to the initial event. For each event, the analysis distinguishes between the cases that the event occurs or that the event does not occur. By this means, a tree is created and each path from the root to the leaves describes one potential effect scenario. As the events often refer to counter measures, the two paths describe success and failure of the measures. The leaves of the tree usually describe the successful avoidance or the occurrence of a hazardous event, respectively.

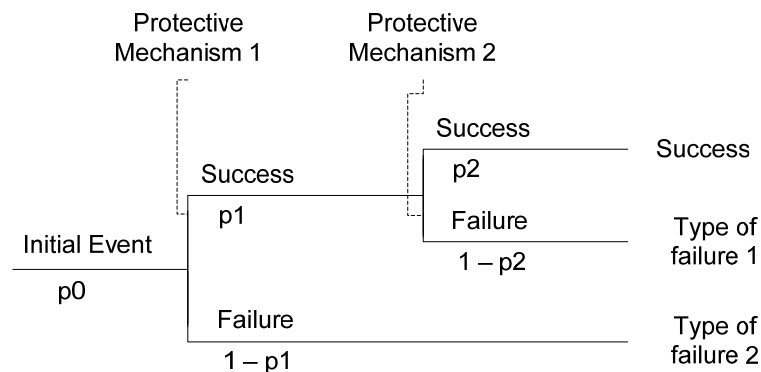


Figure 2-9 Structure of an event tree

For a quantitative analysis, it is additionally possible to assign probabilities to each branch of the tree and to multiply all probabilities along a path. The probabilities of all paths leading to the same hazard type can be combined to an overall probability that the according failure type will occur in spite of the implemented counter measures.

Variants

As described above, event trees are usually represented as binary decision trees with a branch for success and failure at the individual branch points. However, different extensions enable the definition of more than two branches. The methodology, the notation as well as the evaluation of the Event-Tree-Analysis are standardized in the German standard DIN 25419 [11]. As a further extension, the DIN 25419 defines exclusive OR-gates, which enables analysts to join different paths in an event tree for reducing the complexity of larger trees. Moreover, special gates like transmission gates

simplify the modeling of more complex event trees. An example of a DIN 25 419 event tree is shown in Figure 2-10.

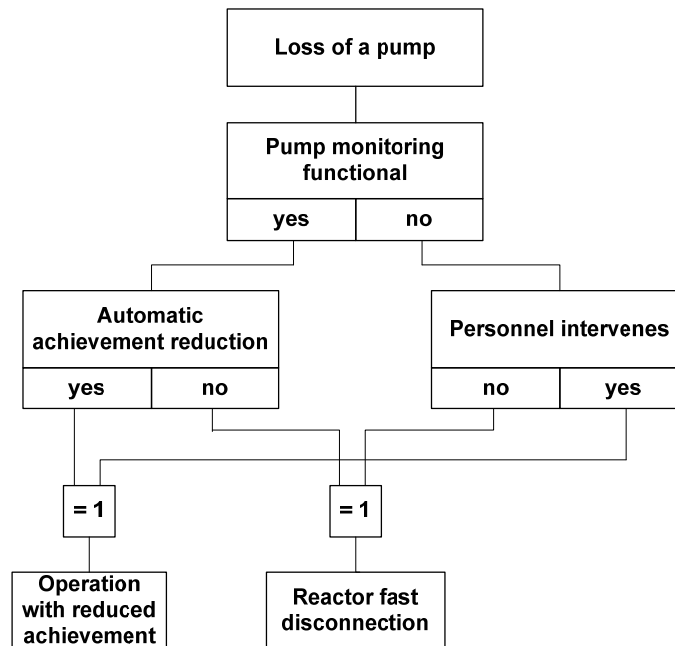


Figure 2-10 Example of an event-tree according to DIN 25 419 [11]

Advantages and disadvantages

Event trees are usually used to analyze the effectiveness of already existing multi-level counter measures. However, this presumes that the protective mechanisms are already known, so that Event-Tree-Analyses are often used relatively late in the development process [31]. In contrast to other methods like for example the HAZOP, event trees are usually conducted by one or two persons, who consult further experts if necessary. The definition of an event tree becomes difficult, if different temporal sequences of events lead to different failure types.

References

A short description of event trees is given in [6]. A more detailed description is given in [31]. The latter describes the advantages and disadvantages of event trees as well as the classification within the development process. Further useful information on the application of event tree analyses can be found in [47]. The DIN 25419 [11] standardizes the event tree analysis notation.

2.4.4. Fault Tree Analysis (FTA)

The Fault Tree Analysis (FTA) was developed in 1961 by the Bell Telephone Laboratories for the ‘Minute Man Launch Control System’, a rocket launching system. It was extended in 1966 by Boeing for computer-assisted applications. Up to now, it has emerged to one of the most widely used safety analysis techniques across all safety-relevant application domains. An example of a fault tree is shown in Figure 2-11.

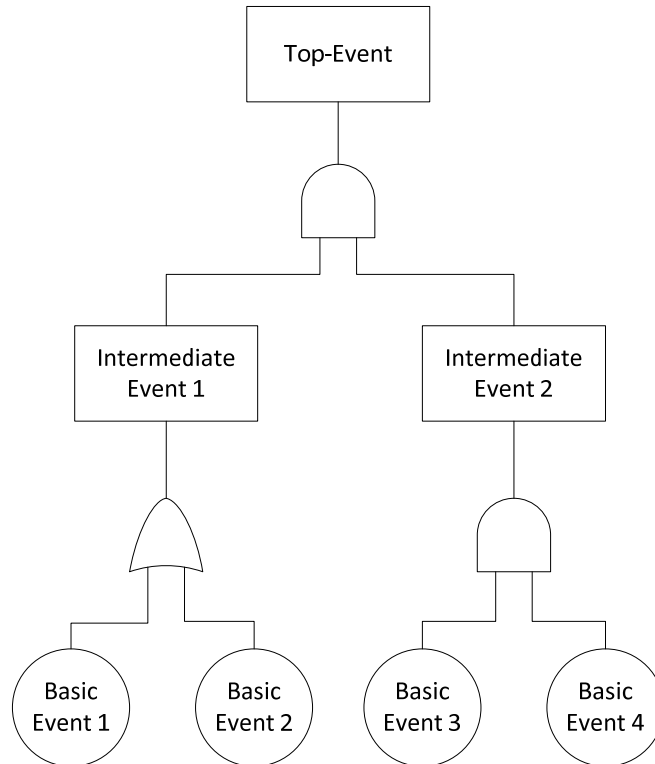


Figure 2-11 Example Fault Tree

Methodology

A fault tree analysis starts with an undesired top event, such as a system failure. In moderated team meetings, the analysts then try to deductively identify necessary and sufficient combinations of potential causes. To this end, *gates* can be used to define the required combinations of causes. The events that represent an elementary cause are called *basic events* and build the leaves of the fault tree.

It is however not the idea of a FTA to identify basic causes in as few steps as possible. It is rather the intention to step wisely derive a traceable and seamless cause-effect-chain. Without such a systematic approach, the discussions within the analysis team often lose their focus – leading to unstructured and mostly incomplete results. The FTA is therefore an efficient means to structure the analysis. In fact, the guided analysis process as such is much more valuable to understand and improve the system's safety than the pure analysis result. Therefore, it is important that the analysis is performed manually. The automatic generation of fault trees is technically possible, but impedes the actual intention of a guided analysis process.

The FTA cannot be used to identify the top events, but the top events must be given as input and are used as starting point of the analysis. The analysis is then first focused on immediate causes of the top-event, only. This means that only faults are considered that are the immediate predecessor of the top-event in a cause-effect-chain. Then the approach is recursively repeated for each identified cause by identifying potential combinations of immediate sub-causes. In principle, a fault tree can be derived in a depth-first or in a

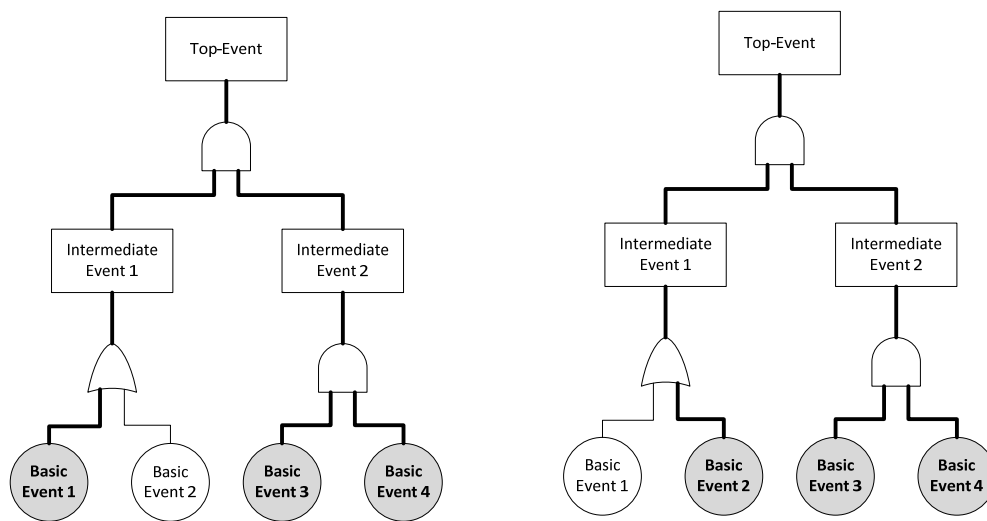
width-first approach. Today, a width-first approach is preferred, i.e. it is common practice to analyze all events at one abstraction levels first, before the identified causes are further refined. It is, however, not necessary to define a balanced tree. This means that it is possible that certain sub trees are refined to a higher level of detail than others. By this means, the effort can be reduced without decreasing the analysis' quality, as usually not all sub trees have the same impact on the top event so that it is not necessary to analyze them in detail. Usually, it is nonetheless an important question at which refinement level the analysis can and should be stopped. As this question cannot be answered in general, it is a rule of thumb to refine the analysis until the analysts are either able to define a concrete, sufficient counter measure to an identified cause or if there is sound evidence that the regarded sub tree has no substantial impact on the top-event.

In fault trees it is possible to define combinations of causes that lead to the event under consideration. To this end, it is possible to use *AND*-Gates and *OR*-Gates. An *AND*-Gate means that all incoming events must occur to cause the outgoing event. An *OR*-Gate means that one of the incoming events is sufficient to cause the outgoing event. The fault tree standards as well as current analysis tools support a variety of additional gates. An overview of basic gates that are used in fault trees is shown in Table 2-3.

Gates	Symbol	
	International	Europe
AND-Gate		
OR-Gate		
NOT-Gate		
XOR-Gate		
n-out-of-m		

Table 2-3 Basic Gates used in Fault Trees

Based on this formal structure, the tree can be analyzed qualitatively as well as quantitatively. An important qualitative analysis is the calculation of so-called *minimal cut sets* (MCS). A minimal cut set is a set of events that are necessary and sufficient to cause the top-event. Sufficient means that the coincidence of the events in a MCS directly causes the top-event. If we, for example, regard the fault tree shown in Figure 2-11, we yield the minimal cut sets highlighted in grey in Figure 2-12. The *AND*-gate's output events only occur if both input events occur, therefore *Intermediate Event 1* as well as *Intermediate Event 2* must occur in order to cause the top event. In order to cause *Intermediate Event 2*, it is necessary that *Basic Event 3* **and** *Basic Event 4* must occur. In order to cause *Intermediate Event 1*, however, it is sufficient if *Basic Event 1* **or** *Basic Event 2* occur. In consequence, there are two possible combinations of basic events whose occurrence leads to the occurrence of the top-event.



Minimal Cutset A = {Basic Event 1, Basic Event 3, Basic Event 4}

Minimal Cutset B = {Basic Event 2, Basic Event 3, Basic Event 4}

Figure 2-12 Minimal Cut sets of a Fault Tree

In order to provide quantitative analysis reports, a probability distribution must be assigned to each basic event. This probability distribution describes the probability that the event occurs over the mission time of the analyzed system or component. To this end, typically exponential distributions and Weibull-distributions are used. Based on the probabilities of the basic events, it is possible to derive the probability that the top event occurs using the following calculation schemes for the different gates:

AND-Gate:

For two inputs: i_1 and i_2 , Output: o

$$P(o) = P(i_1) \cdot P(i_2)$$

For n inputs: i_1, \dots, i_n , Output: o

$$P(o) = \prod_{k=1}^n P(i_k)$$

OR-Gate:

For two inputs: i_1 and i_2 , Output: o

$$P(o) = P(i_1) + P(i_2) - P(i_1) \cdot P(i_2) = 1 - ((1 - P(i_1)) \cdot (1 - P(i_2)))$$

For n inputs: i_1, \dots, i_n , Output: o

$$P(o) = 1 - \prod_{k=1}^n (1 - P(i_k))$$

Extensions and variants

Various extensions to fault trees have been developed in order to increase their expressiveness. The most important extensions are briefly presented in the following.

Dynamic Fault Trees - DFT

A problem of fault trees is that all events must be stochastically independent from each other. In many cases, however, it is necessary to model stochastically dependent events, as well. To this end, *Dynamic Fault Trees (DFT)* [48] have been developed, which extend fault trees by additional gates supporting stochastic dependencies.

The *Functional-Dependency-Gate* can be used to express that a single trigger input event can release a set of dependent entrance events. The *Cold-Spare-Gate* is used to express typical cold stand-by architectures, which consist of several redundant channels. In contrast to conventional redundancy, however, a redundant channel is not activated until the main channel fails. This creates an obvious dependency between failures in the different channels that can be expressed using the cold spare gate. The latter considers that an input event can only occur after the preceding input event occurred. As a further special gate, the *Sequence-Enforcing-Gate* can be used to model that input events must occur in a certain sequence to cause the output event. Therefore, all input events must occur in the order from left to right to trigger the output event. A comparable behavior can also be achieved using a *Priority-AND-Gate*. The latter is however limited to two input events.

Advantages and disadvantages

The FTA offers the possibility to structurally identify a seamless cause-effect chain leading to the analyzed top-event. Fault trees provide a clear notation to specify the cause-effect relationships, but at least as important is the related methodology that enables engineers to analyze even complex systems. Moreover, fault trees support qualitative and quantitative analyses including the analysis of complex fault combinations. In combination with the various existing extensions, fault trees are certainly one of the most important safety analysis techniques available today. As the top-event must already be known before an FTA can be performed, the application of the FTA presumes that the hazards have already been identified. The FTA as such cannot be used to identify hazards.

References

The FTA is standardized by [10] and [24]. A more detailed description can be found in [37] and [36]. Dynamic fault trees are introduced in [48].

2.4.5. Reliability Block Diagrams (RBD)

Reliability Block Diagrams (RBD) is a further important safety analysis technique. It uses blocks to model elements of the system that might fail. The cause effect relationship is modeled by connecting the single blocks. The methodology, the graphical notation as well as the possibilities for the analysis of reliability block diagrams are standardized in the international standard IEC 61078 [18].

Methodology

For defining a reliability block diagram, a first system structure definition must be available. Using this structure as template, the regarded system is subdivided into individual functional blocks that have an impact on system success or failure. Though a reliability block diagram is modeled following the system structure, the RBD structure is not necessarily identical to the system structure, since additional blocks might be required to model the cause-effect relationship. Also it is possible that some system elements are not represented as blocks if they do not contribute to a system failure.

Once the blocks are defined, a failure probability is assigned to each block, defining the likelihood that the according element in the system fails. The probabilities that are assigned to the single blocks must be stochastically independent from each other. Furthermore, it must be considered that a block can only be in one of the two states 'ok' or 'failed'. More detailed failure modes or aspects like operation modes cannot be modeled using RBDs.

By connecting the blocks, these probabilities can be used to calculate the overall failure probability of the system. Basic RBDs support two kinds of connections: a serial connection and a parallel connection. If blocks are modeled in parallel, this means that all parallel blocks must fail to cause a system level failure. If blocks are modeled in a serial connection it is sufficient if one block fails to cause a system failure. An example is shown in Figure 2-13. In this example, the system fails if component A fails or components B and C fail: $A \text{ OR } (B \text{ AND } C)$.

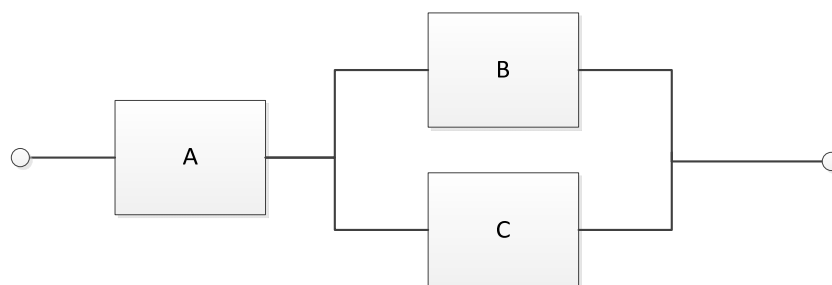


Figure 2-13 Example of a Reliability Block Diagram

In addition to these basic elements, RBDs have been extended by further modeling elements. The most important extension is given by the *k-out-of-n* node as it is shown in Figure 2-14. A *k-out-of-n* node has n edges leading into it. The outgoing path is considered to have failed if more than k of the incoming n paths have failed. For example, the RBD shown in Figure 2-14 specifies a system that fails if component *A* fails *or* at least two of the components *B*, *C*, and *D* fail.

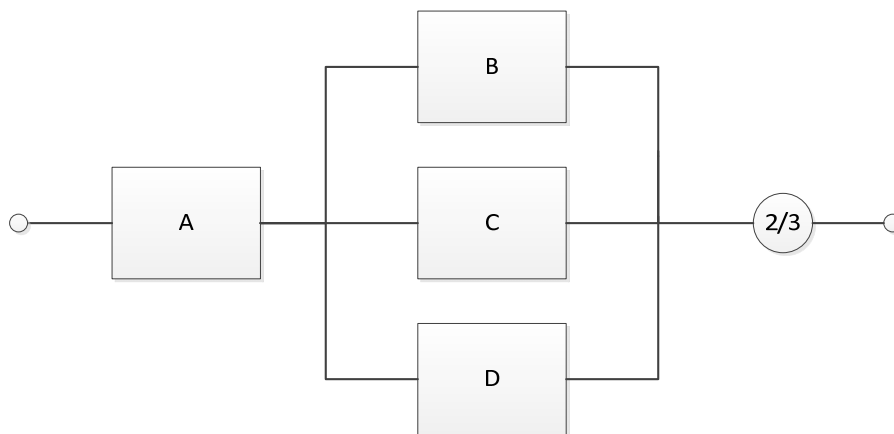


Figure 2-14 Example for a *k-out-of-n* element

References

A short description of reliability block diagrams can be found in [6]. A more detailed explanation is given in [7].

2.5. Safety Concepts and Safety Cases

Once the hazards have been identified and the safety manager has analyzed the possible causes and the resulting error propagation within the system, he or she can define the safety concept. Following the idea of ISO 26262 [4], a safety concept is a ‘specification of the safety requirements, their allocation to architectural elements and their interaction necessary to achieve the safety goals, and information associated with these requirements’. Based on the hazard analysis and risk assessment, the safety manager can define the top level safety goals. Using safety analyses, he gets a better understanding of the underlying cause-effect-relationships potentially leading to the hazards. Using this information, he can step wisely refine the safety goals to more fine grain requirements until he reaches a level, where he is able to define concrete, i.e., implementable, and testable safety requirements, which can then be assigned to the systems’ components. Safety concepts are used to model this systematic break-down of requirements. This includes arguments that ‘prove’ that the fulfillment of all identified refined requirements implies the fulfillment of the superordinate requirement. Usually, this is not done based on techniques like formal proofs. Instead, textual qualitative argumentations are used. For the final safety assessment, the safety manager includes so-called evidences, which prove

that the safety requirements have been fulfilled. Evidences can be anything that provides convincing arguments that the respective requirement has been fulfilled. In most cases, evidences are given by documents like test reports, review protocols, or safety analysis results. By attaching evidences, the safety concept seamlessly evolves to a safety case. Such a safety case includes (a) a systematic decomposition of safety requirements, (b) the arguments why these requirements are sufficient, and (c) the evidences showing that the requirements have been met. Therefore, this meets the ISO 26262 definition of a safety case being an ‘argument why an item is safe supported by evidence compiled from work products of all safety activities during the whole lifecycle.’

Today, safety concepts are defined in a quite informal way. Since they cover safety requirements, they are often integrated into an existing requirements management workflow. Therefore, the safety requirements are defined in typical requirements databases. In addition, safety managers write reports describing the big picture of the safety concept including the rationales that lead to the concept.

As regards the safety case, however, the goal structuring notation (GSN) has evolved as an accepted alternative to simple text documents. Though the GSN is yet not widely used in industry, it is continuously gaining acceptance and importance. It is nonetheless the most important and most widely used dedicated notation for safety cases. Therefore, the following subsection gives a brief overview on the GSN and its fundamental modeling elements.

2.5.1. Goal Structuring Notation

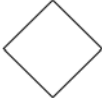
It is the main goal of the goal structuring notation (GSN) to provide an efficient means to describe arguments in safety assurance cases. According to the GSN community standard [49], an assurance case is ‘a reasoned and compelling argument, supported by a body of evidence, that a system, service or organization will operate as intended for a defined application in a defined environment.’ Such an argument is defined as ‘a connected series of claims intended to establish an overall claim.’ This usually requires a hierarchy of claims. It is the aim of the GSN to efficiently support the definition and the graphical notation of such hierarchical claims. To this end, the GSN includes the following core elements:

- Goals
- Strategies
- Solutions
- Contexts
- Assumptions
- Justifications

These elements can be related to each other using the following relationships:

- SupportedBy
- InContextOf

The graphical notation and the principle meaning of these core elements is summarized in Table 2-4.

Graphical Notation	Description
<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: auto;"> <p style="text-align: center;">{Goal Identifier}</p> <p style="text-align: center;"><Goal Statement></p> </div>	<p>A goal, rendered as a rectangle, presents a claim forming a part of the argument.</p>
<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: auto;"> <p style="text-align: center;">{Strategy Identifier}</p> <p style="text-align: center;"><Strategy Statement></p> </div>	<p>A strategy, rendered as a parallelogram, describes the nature of the inference that exists between a goal and its supporting goal(s).</p>
<div style="border: 1px solid black; border-radius: 50%; padding: 10px; width: fit-content; margin: auto;"> <p style="text-align: center;">{Solution Identifier}</p> <p style="text-align: center;"><Solution Statement></p> </div>	<p>A solution, rendered as a circle, presents a reference to an evidence item or items.</p>
<div style="border: 1px solid black; border-radius: 15px; padding: 10px; width: fit-content; margin: auto;"> <p style="text-align: center;">{Context Identifier}</p> <p style="text-align: center;"><Context Statement></p> </div>	<p>A context, rendered as a rounded rectangle, presents a contextual artifact. This can be a reference to contextual information, or a statement.</p>
<div style="border: 1px solid black; border-radius: 15px; padding: 10px; width: fit-content; margin: auto;"> <p style="text-align: center;">{Justification Identifier}</p> <p style="text-align: center;"><Justification Statement></p> <p style="text-align: right; margin-right: 5px;">J</p> </div>	<p>A justification, rendered as an oval with the letter ‘J’ at the bottom-right, presents a statement of rationale.</p>
<div style="border: 1px solid black; border-radius: 15px; padding: 10px; width: fit-content; margin: auto;"> <p style="text-align: center;">{Assumption Identifier}</p> <p style="text-align: center;"><Assumption Statement></p> <p style="text-align: right; margin-right: 5px;">A</p> </div>	<p>An assumption, rendered as an oval with the letter ‘A’ at the bottom-right, presents an intentionally unsubstantiated statement.</p>
<div style="text-align: center; margin: auto;">  </div>	<p>Undeveloped entity, rendered as a hollow diamond applied to the center of an element, indicates that a line of argument has not been developed. It can apply to goals (as below) and strategies.</p>




Graphical Notation	Description
	<p>An undeveloped goal, rendered as a rectangle with the hollow-diamond ‘undeveloped entity’ symbol at the center-bottom, presents a claim which is intentionally left undeveloped in the argument.</p>
	<p>The SupportedBy relationship, rendered as a line with a solid arrowhead, allows inferential or evidential relationships to be documented. Inferential relationships declare that there is an inference between goals in the argument. Evidential relationships declare the link between a goal and the evidence used to substantiate it. Permitted <i>SupportedBy</i> connections are: goal-to-goal, goal-to-strategy, goal-to-solution, strategy to goal.</p>
	<p>The InContextOf relationship, rendered as a line with a hollow arrowhead, declares a contextual relationship. Permitted connections are: goal-to-context, goal-to-assumption, goal-to-justification, strategy-to-context, strategy-to-assumption and strategy-to-justification.</p>

Table 2-4 Core Elements of the GSN (based on [49])

The possibility to define claim hierarchies is a key purpose of the GSN. To this end, a safety manager can refine *Goals* using the *SupportedBy* relationship. In the example shown in Figure 2-15, the top level goal *G1* can be fulfilled if the sub goals *G2* and *G3* are fulfilled.

Often, it is however not sufficient to simply break down a goal into sub goals. Instead, it is necessary to describe the underlying rationale, why the developer believes that this decomposition is valid. To this end, he can define a *strategy*, which describes why and how this breakdown of the goals is valid. To this end, he can use the strategy element of the GSN as shown in Figure 2-16.

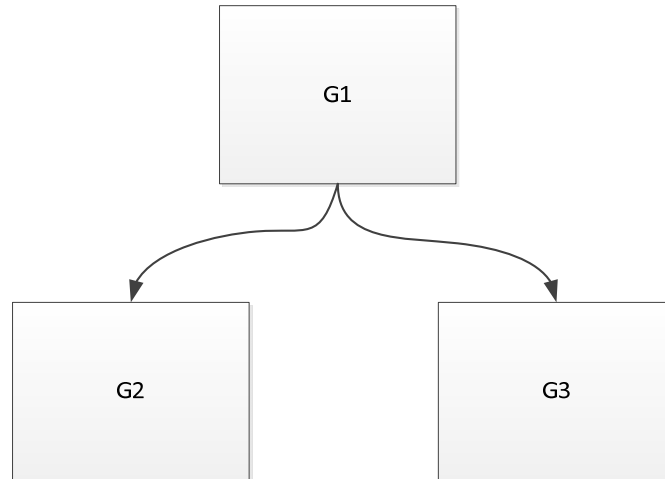


Figure 2-15 Goal refinement in GSN (based on [49])

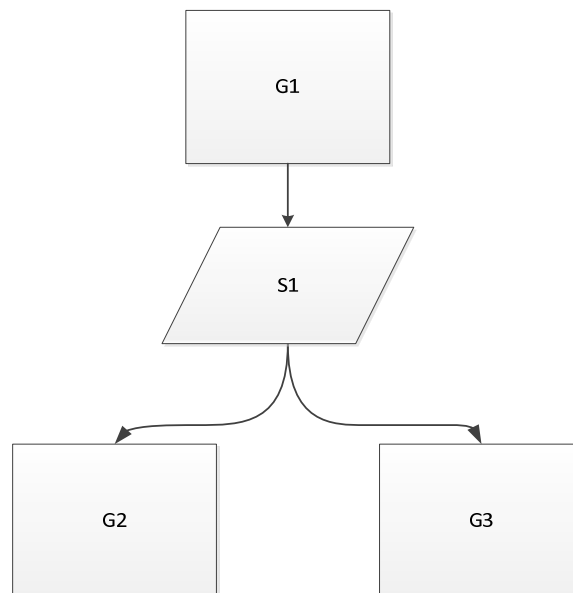


Figure 2-16 Use of strategies in GSN (based on [49])

In order to argue why it is necessary and sufficient to refine goal $G1$ by its sub goals $G2$ and $G3$, the engineer defines strategy $S1$. In principle such strategies do not provide any formal proof and they do not even follow a formal syntax. In fact, strategies are defined as informal text providing a qualitative argument. Nonetheless, it is of course possible to combine the GSN with more formal approaches in order to facilitate more formal proofs of the safety case's correctness and completeness. The GSN as such, however, only supports informal safety cases.

In fact, it is often necessary to refine a goal using various different strategies. Therefore, the GSN supports the refinement of goals using several strategies as it is shown in Figure 2-17. The top-level goal $G1$ is refined by four sub goals $G2$ to $G5$. The refinement to $G2$

and $G3$ is argued using strategy $S1$ whereas the refinement to the goals $G4$ and $G5$ is argued using strategy $S2$.

As a further aspect Figure 2-17 also shows the use of solutions. As mentioned above, solutions provide a link to evidences ‘proving’ that the associated goals have been achieved. In the example, the achievement of goal $G2$ is shown by solution $Sn1$, the achievement of $G3$ by solution $Sn2$ and so on.

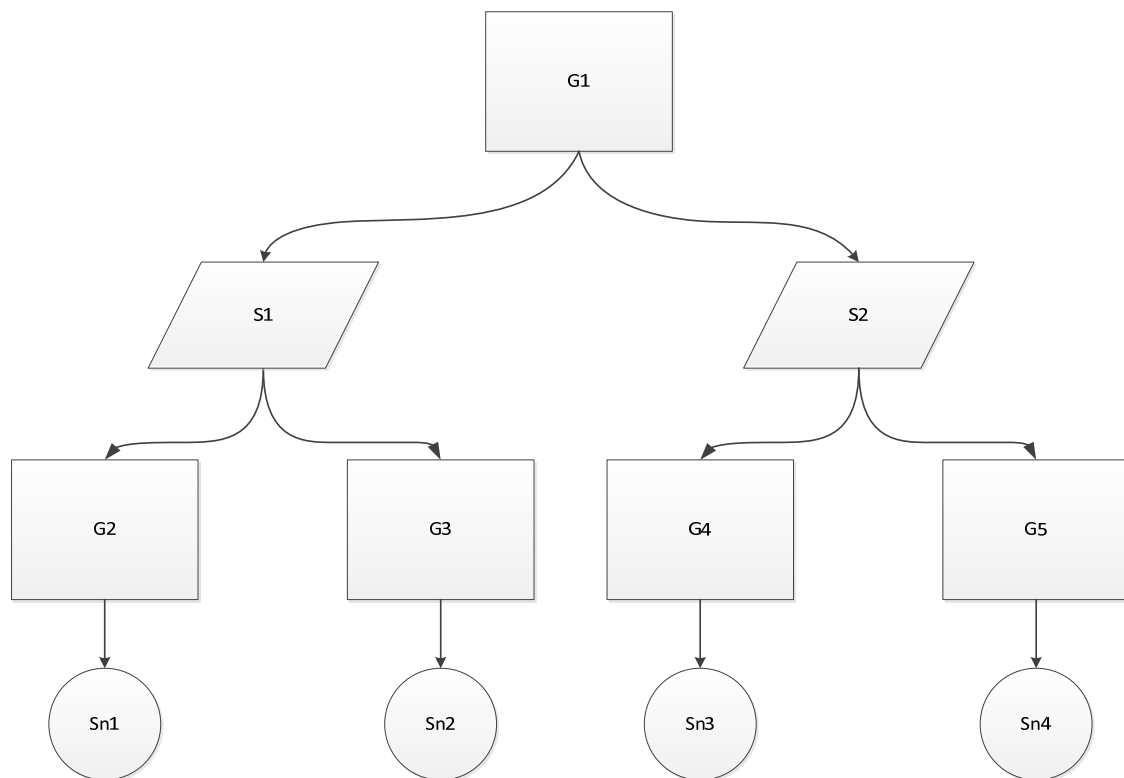


Figure 2-17 Goal refinement using several strategies (based on [49])

Figure 2-18 shows a generic safety case using the GSN as a summarizing example. The generic top level goal defines that the system is demanded to be acceptably safe. In order to refine this goal, the system definition is given as context information. The refinement of this top level goal is based on the strategy that the system is acceptably safe if the occurrence of all identified hazards is sufficiently unlikely.

Obviously, this strategy only holds if an acceptably complete set of hazards has been identified. This limitation is documented as an assumption that is assigned to the strategy. Based on this strategy, a maximum failure rate goal is defined for each hazard. In order to show that these sub goals have been fulfilled, the according fault tree analysis is referenced as evidence. In general, safety analyses like fault trees are important evidences in order to prove that the refinement of requirements is complete.

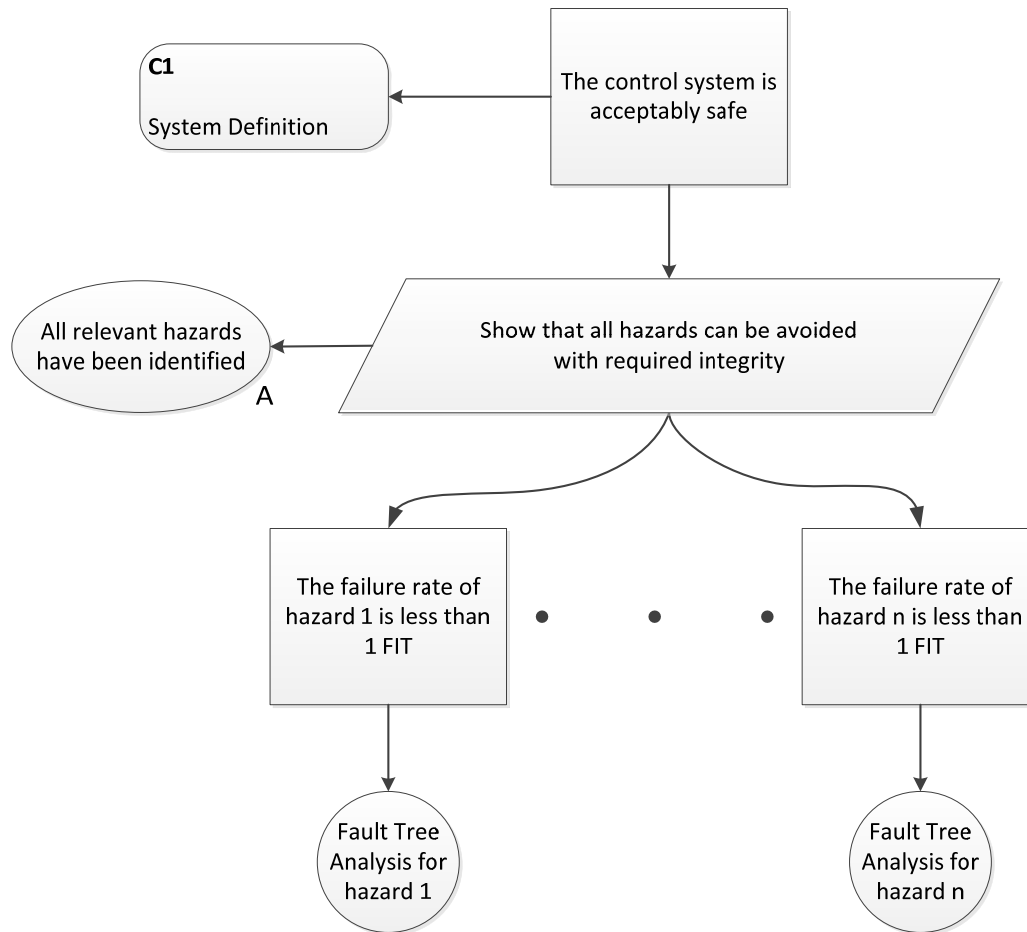


Figure 2-18 Example of a generic safety case using the GSN

Usually, the goals must be further refined since a direct quantitative proof is not possible for rather abstract safety goals. Therefore, the safety case has to be step wisely refined until all derived goals are concrete enough to provide sound evidence that the goals have been fulfilled. This means, for example, that it must be possible to provide sound test results showing that the goals have been fulfilled. If, for example, a plausibility check is defined being an efficient means of error detection, the results of a fault-injection test are required to provide that the error detection provides a sufficiently high diagnostic coverage.

Besides the basic notational elements described above, the GSN provides different extensions in order to support alternative strategies and refinements, or to support modular safety cases. All available extensions as well as detailed guidelines on the usage of the GSN can be found in [49].

2.6. Software Fault Tolerance Mechanisms

As described in section 2.1, fault tolerance is one of the major possibilities to assure safety. Particularly for open systems, fault tolerance mechanisms can be very important. As long as failures can be reliably detected and handled at runtime, the system's safety can be assured. If there is a good fault tolerance monitor in place, the safety demands on

the monitored main functionality can be significantly reduced. This creates some freedom for using more advanced technologies for the development of open adaptive systems.

Therefore, this section provides a brief introduction to the most important fault tolerance mechanisms. To this end, sub section 2.6.1 gives an overview on available mechanisms based on a general classification scheme. The subsequent sub sections will then introduce the single classes of fault tolerance mechanisms.

2.6.1. Overview

Despite various different approaches to avoid or to remove any kind of faults in a system, fault-free systems are very unlikely to become reality. From a safety point of view, however, software faults are not critical as long as the resulting errors are detected and handled, before safety-critical failures can occur. To this end, any safety-critical system must implement fault tolerance techniques. In fact, fault tolerance can be realized at various levels of abstraction. Ranging from hardware mechanisms to redundant system designs. Software implemented fault tolerance (SWIFT) has a very long history and many publications introduce and evaluate a long list of different approaches. All of them follow the basic ideas shown in Figure 2-19.

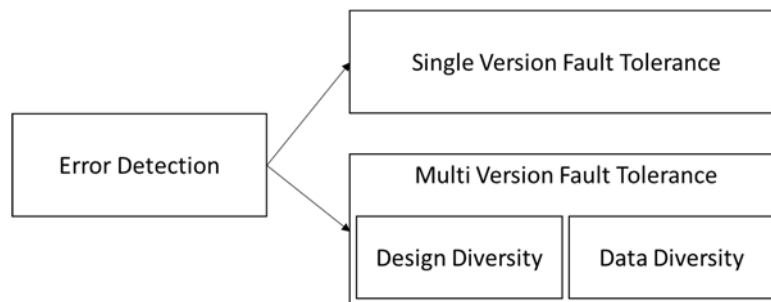


Figure 2-19 Overview on Fault Tolerance Mechanisms

For obvious reasons, error detection is a basic constituent of any fault tolerance mechanism. Therefore, this aspect is dedicatedly described in sub section 2.6.2. Once an error is detected, appropriate counter measures must be taken to avoid a resulting failure. In principle, there are two ways of realizing such handling mechanisms. First, single-version fault tolerance techniques do not implement any kind of redundancy. They can be used to detect errors and to bring the system to a safe state. Therefore, they are important for fail-safe and fail-silent system, which can be switched off in the case of errors. According programming techniques are introduced in sub section 2.6.3.

If the system needs to be fail-operation, i.e. if it must continue operation in spite of errors, it is usually necessary to implement multi version fault tolerance. This means that redundant channels must be implemented. This could also be required in order to realize an error detection by comparing two independently computed result values. Multi-version fault tolerance can be implemented using design diversity and / or data diversity. Design

diversity means that the actual behavior is realized in multiple versions. Data diversity means that data is stored in redundant representations. Approaches belonging to the class of design diversity will be introduced in sub section 2.6.4. Data diversity approaches will be introduced in sub section 2.6.5.

In fact, however, fault tolerance mechanisms can be classified in multiple dimensions and there is no commonly accepted classification available. Therefore, Table 2-5 shows the fault tolerance mechanisms introduced in this section in relation to different classification criteria.

	Single-/Multi-Version	Data-/Design Diversity	Serial/Parallel Execution	Backward / Forward Recovery
Atomic Actions	✓/-	na	✓/-	✓/-
Checkpointing	✓/-	na	na	✓/✓
Consensus Recovery Blocks	-/✓	-/✓	✓/✓	✓/-
Distributed Recovery Blocks	-/✓	-/✓	✓/✓	✓/✓
Exception Handling	✓/-	na	na	✓/✓
N-Copy Programming	✓/-	✓/-	-/✓	-/✓
N-Self-Checking Programming	-/✓	-/✓	-/✓	-/✓
N-Version Programming	-/✓	-/✓	-/✓	-/✓
Recovery Blocks	-/✓	-/✓	✓/-	✓/-
Retry Blocks	✓/-	✓/-	✓/-	✓/-
t/(n-1)- Variant Programming	-/✓	-/✓	-/✓	na
Two-Pass Adjudicators	-/✓	✓/✓	✓/✓	na

Table 2-5 Classification of software implemented fault tolerance techniques

In addition to the criteria described before, it is possible to distinguish between serial and parallel execution of redundant channels. Parallel execution does not necessarily

mean that different channels are executed concurrently, but that all redundant channels are executed and only then the results are compared to each other. Whereas in serial execution approaches only one channel is executed and an error detection checks the result. Redundant channels are only executed if an error of the result is detected. This saves computation power, but then redundancy cannot be used for error detection. And there will be an execution delay in the case of an error, which is sometimes not tolerable in real time systems.

As a further criterion, it is important to distinguish the recovery direction. Backward recovery means that a system is set back a previously stored, correct state if an error is detected. The system is then restarted in this state. In the case of forward recovery, the system is set to a new correct state in case of an error.

2.6.2. Error Detection

Error detection is an indispensable constituent of any fault tolerance mechanism. In the case of multi-version fault tolerance, error detection can be realized by comparing the results of the different channels. In the case of single version fault tolerance, serial execution mechanisms, or if the single channels are additionally monitored, explicit error detection mechanisms are required.

As illustrated in Figure 2-20, error detection mechanisms can be applied in different combinable ways to single version systems. First, it is possible to realize *self-protection* mechanisms. This approach is used to protect a component against erroneous input values, which might corrupt the component's functionality. To this end, only the input values and their semantics can be used to implement a check. Including any information about the algorithms used to calculate the input values or any other white box information about the component creating the input values must not be considered when defining the checks in order to preserve the component's modularity. Otherwise any change in one component might invalidate self-protection checks implemented in other components.

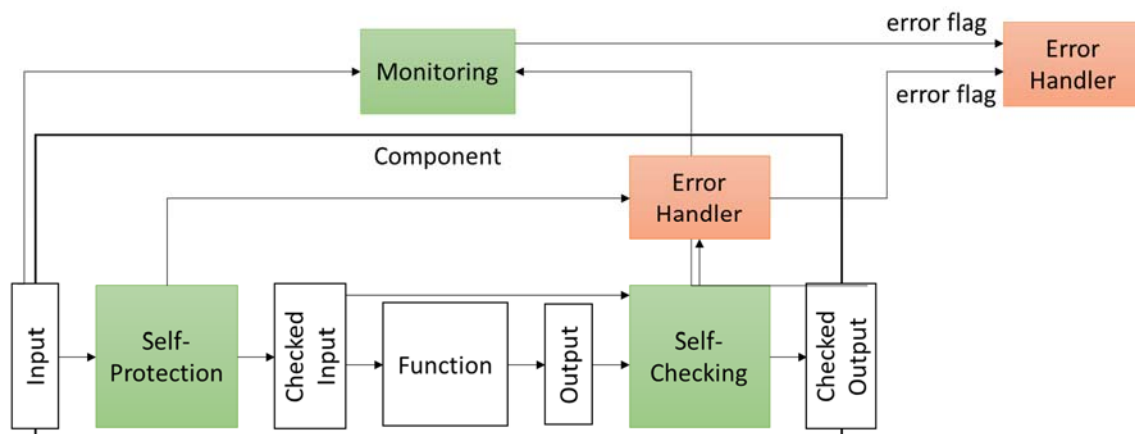


Figure 2-20 Principle Application of Error Detection in Single Version Systems

Alternatively or in addition, error detection mechanisms can be used to realize *self-checking* mechanisms. In this case, input values, output values and the detailed knowledge about the implemented functionality are used to check the correctness of the calculated output values, i.e., the correctness of the monitored function is checked. In a defensive design, self-protection and self-checking are combined in order to provide self-contained fault-tolerance concepts. This means, the component does not assume to get correct input values from the context and avoids to send incorrect output values to the context.

In both cases, error detection only detects runtime errors. In addition, an *error handler* is required to react to detected errors. In some cases, intra-component error handlers might repair the component or they shut down the component to a safe state and set an error flag, which is processed by a superordinate error handler.

In addition or alternatively to error detections within a component, an external monitor can be used to monitor a component or several components at once. In principle, a monitor works like a self-checking mechanism. Often, however, external monitors do not include so much internal white box information, but the detection is usually based upon the black box specification of a component. Monitors have the advantage to be independent from the monitored component. Therefore, an error in the component does not influence the monitor. Moreover, the monitor can be used to monitor unprotected third-party components. As a further advantage, a monitor can cover several components in a more complex data path. First, this leads to more efficient solutions. Second, more global semantic information about the data flow allows for better error detection algorithms.

In any way, error detections must always be tailored to the monitored functionality. Therefore, error detection mechanisms found in literature mainly provide patterns on how to realize a detection, in general. Table 2-6 provides a brief overview on the most important error detection mechanisms and explains widely used terms in the context of error detection. Moreover, it lists references for each of the mechanisms, which can be used for further reading to get more detailed information on the different approaches.

Name	Description	References
Time checks	Time checks are used, if modules have to keep clear time conditions as for example deadlines or response times. An example for the realization is a watchdog timer, which can be realized in hardware or software. In this case, the monitored software has to actively reset the watchdog timer. If this does not happen in time, e.g, because of an erroneous endless loop, the timer runs out and triggers a counter action. Often, an embedded micro controller is reset to reboot the	[103], [114], [121], [126], [127], [197]

Name	Description	References
	system. There are various different extensions of watch dog mechanisms available.	
Inverse functions	Inverse functions are used to calculate the inputs of a module by taking the module's outputs as input values for the inverse function. An error is detected, if the computed inputs do not correspond to the actual inputs. This approach is obviously only applicable if the inverse function of the module can be determined easily.	[102]
Check Codes	In this approach, redundant check codes are used to detect errors. For example, data errors can be detected by cyclic redundancy checks (CRC). Arithmetic codes such as AN-codes or Residue-codes are applied to the input data of an arithmetic operation in such a way that also the result must be correctly coded. With AN-codes, for example, both operands of an addition are multiplied by a constant A. After executing the addition, the result must be divisible by A in whole numbers.	[107], [148]
Plausibility checks	Plausibility checks are used to check the semantic characteristics of data. This includes, for example, checking value ranges, gradients, value sequences etc. In principle, the idea of plausibility checks is very generic, since the check must always be tailored to the monitored function. For example, physical signals like the longitudinal acceleration of a car must follow certain physical constraints, which can be checked in plausibility checks. Therefore, it is rather a general pattern than a concrete error detection mechanism.	[103], [148]
Structural Checks	Structural checks are used to check known, invariant properties of data structures. For example, the number of elements or the correct linking of pointers can be checked. It is also possible to store redundant data within data structures, such as the length of lists, in order to dynamically check them at runtime. By this	[103], [177], [178], [179], [180].

Name	Description	References
	means, it is possible to detect corrupted data structures.	
Hardware implemented error detection	As a further possibility, it is possible to implement error detection checks in hardware. Due to the caused effort and cost, these checks must be independent from the monitored application in order to be reusable for different functionalities. Typical well-known examples are division by zero, overflows, or underflows.	[103], [147].
Control Flow Monitoring	Control flow monitoring is used for detecting errors in the control flow of instructions. To this end, the software is subdivided into basic blocks. This means that a set of instructions without branches is bundled in one basic block. For each of these blocks a deterministic signature is calculated. At runtime, this signature is recalculated and compared to the static signature. By this means, a corruption of the program memory can be detected and undesired jumps can be avoided. Control flow monitoring is a well-established concept, which is often demanded by safety standards. Various different realizations and optimizations of the approach can be found in the references.	[114], [121], [127], [129], [130], [132], [135], [136], [141], [142], [195]
Duplication of data and instructions	A further possibility for error detection is duplicating data as well as the program. To this end, the program can be stored in different memory locations, to execute it concurrently and to compare the results. Alternatively, an orthogonal redundancy can be created. In this case, the operations are executed redundantly by using different hardware units. Regarding the state of the art, the duplication is automated as part of the compiler or as code-to-code translators, which use predefined transformation rules to extend the source code with error detection code. In those approaches, software and hardware tolerance mechanisms are combined in hybrid approaches.	[115], [119], [123], [127], [135], [136], [140], [157], [158], [171], [196].

Name	Description	References
Software-Hardening	Software-Hardening is often used as term to describe approaches applying an automated duplication of data and instructions.	[115], [119], [158], [171]
Acceptance test	The term acceptance test refers to a generic idea on how to check computation results, as it is for example done for recovery-blocks [165]. Therefore, there are no concrete techniques or checks defined. It is rather necessary to define application-specific checks for each particular functionality. The primary problem with acceptance tests is to define them as simple as possible, in order to prevent faults in the test itself and to ensure an efficient execution at runtime. On the other hand, however, the diagnostic coverage must be sufficiently high. To this end, often the fulfillment of specifications is checked (also called self-check). For example, for checking a sorting algorithm it is not necessary to redundantly sort the data. Instead, it is possible to check if the result values are sorted and if the sum of the output values is equal to the sum of the input values. Obviously, some errors cannot be detected by this check, but it is sufficiently unlikely that a hardware or software fault leads to an incorrect result that is not detected by this acceptance test. A methodology for defining acceptance tests and self-checks can be found in [145].	[145]
Assertions	Assertions are provided by programming languages in order to detect arbitrary error conditions during program execution. By this means, it is for example possible to check pre-conditions, post-conditions, and/or invariants of a function. In the same way as most error detection mechanisms, assertions must be specifically defined for a concrete application. [193] introduces an approach for deriving assertions in the context of embedded systems.	[111], [112], [138], [141], [193], [194], [195], [197], [198], [199], [200]

Table 2-6 Overview on Error Detection Mechanisms

2.6.3. Programming Techniques

Before the different fault tolerance approaches based on design and data diversity are introduced in the following sub sections, this section introduces two important programming techniques, which are applied in the development of fault tolerant systems. On the one hand, they directly improve the systems' fault tolerance if single-version systems are developed. On the other hand, they are an important basis for realizing fault tolerance in general, since they are a prerequisite for enabling system recovery in the case of errors.

Checkpointing means that different check points are inserted into the source code, when the current system state (e.g. variables, register values, etc.) is stored. In case of an error, the last stored state can be restored to restart the system. In this case, checkpointing is only applicable to transient errors, since it is assumed that simply repeating the code execution leads to a correct result. This is for example true in the case of bit flips due to transient electromagnetic radiation. In the case of distributed systems as well as process and task duplications, checkpointing can also be used for forward recovery.

The system state can be stored in different ways. Either the complete system state is stored at each check point, or only delta information is stored, i.e. the differences to the last system state. Checkpoints can be stored in fixed or in variable time intervals.

More information on checkpointing can be found in [102], [131], [147], [158], [159], [167], [182], and [191].

Another programming technique is **Atomic Actions**. In this case, all of those commands are encapsulated into one block, if either all or none of these commands must be executed at a time. Atomic Actions are particularly important in multi-threaded-applications in order to coordinate the cooperation between different threads. In the case of an error, it must be possible to roll back all effects of all commands of an atomic action. This is a prerequisite for error handling at runtime. More information on atomic actions can be found in [102], [155], [159], [169], [173], and [183].

2.6.4. Design-Diversity

In order to realize fail-operational systems, it is necessary to provide any kind of diversity. Design diversity is the most commonly used approach for designing redundant systems. In this case, fault tolerance is achieved by having several different versions of a program or a program fragment, respectively. The different versions are developed based on a common specification. However, the versions are developed by different teams who should not know each other. And the different teams should use different technologies like different programming languages or compilers etc. In spite of this independence, it has nonetheless been shown that diverse software is not necessarily free of common cause failures. Therefore, it is reasonable to combine design diversity with other techniques, e.g. by additionally monitoring the single channels and applying rigorous fault avoidance and fault removal approaches.

There are four important design diversity approaches available, which will be briefly described in the following.

N-Version Programming

In the case of N-Version Programming (NVP), N different versions of a program are developed based on a common specification. The single versions are then executed in parallel or sequentially and the different results are compared to each other. If all results are equal to each other, the result is output as a correct result. Otherwise a voting mechanism like a majority voter is required. A majority voter selects that output which has been provided by the majority of the versions. If there is no majority providing the same output value, an error is raised. In any way, an error flag is set in order to inform the superordinate system that at least one of the versions seems to be defect.

Obviously, the number of errors that can be tolerated depends on the number N of different versions. If $N=2$, it is only possible to detect errors since it cannot be decided which of the versions provided the correct result. If $N=3$ one error can be tolerated. If two versions have the same result, this value is assumed to be correct, and the third version is assumed to be erroneous. The number of tolerable errors increases by increasing N. Usually an odd number of versions is realized in order to ensure a majority for the voting mechanism.

More details on N-version-programming can be found in [109], [115], [116], [117], [135], [151], [152], [154], [162], [182], and [187].

t/(n-1)-Variant Programming

$t/(n-1)$ -variant programming assumes that a failure diagnosis happens at system level. Comparable to N-Version-Programming, n versions of a program or program fragment are developed, which are executed in parallel at runtime. Under the assumption, that at most t of these n versions could produce an error at the same time, the erroneous versions can be isolated in a set of the maximal size of $n-1$. This means that there is at least one error-free version left, whose output value can be used.

The identification of the erroneous versions and the construction of the according system architecture is based upon a decision mechanisms based on the $t/(n-1)$ diagnosability theory. More information on this concept can be found in [158], [182], [185], [186], and [188].

N-Self-Checking Programming

N-Self-Checking Programming means that several versions of a program are implemented and usually concurrently executed on different processors. Two versions are combined into a group and the results of the versions within the groups are compared with one another. If the versions of a group provide different results, an error flag is set and the group is ignored in the further steps. The results of the error-free groups are afterwards compared again with one another. If all of the groups provide the same result, the output is used. Otherwise an error flag is set, or if there are enough groups available, a voting mechanism can be applied to select the correct result. More information on N-Self-Checking can be found in [144], [152], [153], [159], and [182].

Recovery Blocks

For recovery blocks, several versions of a program are implemented. In contrast to N-Version-Programming, however, the single versions are executed sequentially. After a version was executed, the result is checked using an error detection mechanism. If no error is detected, the output is used and no further versions are executed. In case of an error, the system state is rolled back to the state it had before the first version was executed. Then the next version is executed and the same procedure is repeated until a valid result is rendered by one of the versions. If none of the versions leads to a valid result, the execution exits with an error message.

Recovery blocks provide a comparably efficient means since only one version needs to be executed in the best case. In the worst case, however, there is a long delay if a version outputs an erroneous result. Nonetheless, recovery blocks are an important fault tolerance approach, which is described in more detail in [109], [145], [146], [151], [152], [154], [162], [164], [166], [182], and [185].

Distributed Recovery Blocks are an extension of recovery blocks. In this case, two versions of a program or program fragment are implemented and they are executed concurrently on two different processors. Comparable to conventional recovery blocks, the results of the single versions are checked using error detection mechanisms, i.e. the results are not compared with each other to detect errors. If the result of the first processor is correct, this output is used. If not, the output of the second processor is used if the error detection found no error. If both processors provide erroneous results, their state is rolled back to the state they had before the versions were executed. Then the versions are exchanged between the processors and executed again. If again no correct solution is provided, the execution of the program or program fragment exits with an error message. More information on distributed recovery blocks can be found in [143], [148], [149], [150], [159], [166], and [186].

As another extension, *consensus recovery blocks* are a mixture of N-Version-Programming and recovery blocks. In this approach, N versions of the same program or program fragment are executed like in N-version-Programming. Using a decision algorithm like a majority voter, it is tried to find a correct result. If this is not successful, the results of the single versions are checked using acceptance tests in order to identify a correct result. Alternatively, the single versions can be executed again following the idea of recovery blocks. More information on consensus recovery blocks can be found in [159], [171], [172], [181], and [182].

2.6.5. Data-Diversity

Besides design diversity, data diversity provides another possibility for realizing redundancy in fault tolerant systems. Data diversity means that data is stored in diverse representations. To this end, the input data of a program or program fragment are modified using data transformation algorithms prior to the program's execution.

N-Copy-Programming

N-Copy-Programming is very similar to N-Version-Programming. N different data transformation algorithms are defined to transform a program's input data into another representation or value. It is important that the result value can be consistently transformed back to the original representation. Then, N copies of the program are executed using the different transformed data representations, the results are transformed back to the original representation and compared to each other like in N-version-programming. Either a correct result can be determined using a decision algorithm like majority voting, or the program exits with an error message. More information can be found in [106], [107], [108], and [159].

Retry-Blocks

Retry Blocks are a similar concept to recovery blocks. The program is called several times in sequence, but before each call, the input data is transformed into a different representation. Just like for N-copy-programming, this requires several different data transformation algorithms. After each execution, the results are checked using an acceptance test. If a program execution leads to a valid result, the result is used as output and no further program calls are required. If no data representation leads to a valid result, the program exits with an error message. More information on retry-blocks can be found in [106], [107], [108], [159], and [162].

Two-Pass Adjudicators

Two Pass Adjudicators are a combination of N-version-programming and N-copy-programming. Therefore, they combine the ideas of design diversity and data diversity. The program is executed in two phases. In the first phase, N versions of a program are executed using the original input data. After that, the results are compared to each other and a result is selected or an error message is raised just like in N-version-programming. If a valid result is available, this is used as output value and the execution is stopped. If no valid result can be found, the second phase is executed. In the second phase, the input data is transformed into N different representations using data transformation algorithms. Then, the N different program versions are executed using the N different data representations. The results are again transformed back and compared to each other. And a result is selected or an error message is raised if no valid result can be found. More information on two-pass-adjudicators can be found in [159], [160], and [161].

3. Fundamentals of Model-Driven Engineering

As described in the introduction, the models@runtime paradigm builds a promising basis for the development of open systems of systems with predictable qualities. Obviously, models play a central role in this context. In some sense, models@runtime can be seen as the next evolution stage of model-driven development. Consequently, understanding the basic idea of model-driven development is a major prerequisite for understanding the concept of models@runtime.

Therefore, this chapter gives an introduction to the basic concepts of model-driven development. To this end, Section 3.1 first introduces the basic idea of model-driven engineering (MDE) before Section 3.2 describes the principles of models and their definition based on meta-models. Section 3.3 introduces the Model Driven Architecture (MDA™) standard, which provides a reference for many MDE approaches available as of today. The MDA, however, has its origin in the development of IT-systems so that approaches for embedded systems have to be adopted as it is described in Section 3.4.

Although model driven engineering approaches are mainly associated with automated model transformations ultimately aiming at the efficient code-generation, models are useful for many other development aspects. First of all, models are used to analyze and to predict system (quality) properties and to optimize the systems already in very early development phases based on model-driven prognostics. This aspect is very important for the idea of models@runtime. Instead of using models at development time, in models@runtime, runtime representations of the models will be used to dynamically optimize systems to a given runtime context. Therefore, the idea of model-driven optimizations will be briefly discussed in Section 3.5.

3.1. The Model-Driven Engineering Paradigm

The complexity of embedded software systems is growing even more rapidly than ever before. Already today, most embedded systems are much too complex to be directly implemented in C code. Many companies had to experience this fact by tremendous quality problems, drastically increasing time-to-market, and exploding development costs.

The main principle that is applied to get this complexity under control is the very same as it has been applied for the shift from assembly code to high level programming languages: Modelling languages introduce more powerful and expressive means for specifying software systems. Using generators, the code is generated automatically, i.e. the models are not an inconsistent documentation but a central development artifact. Already today, such code generators often create very efficient code.

Figure 3-1 illustrates this aspect based on a very simple example. Figure 3-1 a) shows a simplified state chart model specifying the behavior of a DVD player. Obviously, the state chart is not very complex and even without having a deep understanding of state charts, the semantics is quite intuitive. Based on this state chart, it is possible to generate

source code as it is shown in Figure 3-1 b). However, Figure 3-1 b) does not show the complete source code but only the method headers. Both, model and code, specify the very same behaviour, but obviously shifting from source code to state charts tremendously reduces the complexity of the representation developers have to work with.

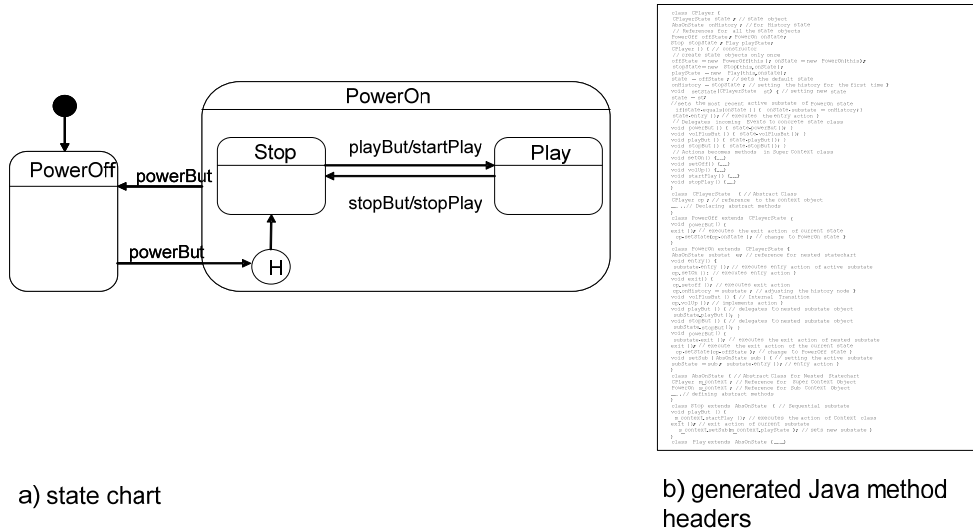


Figure 3-1 Comparison of a state chart and the generated Java method headers

The abstraction that can be achieved by applying model-based development is obviously a key to managing the ever-increasing complexity of embedded software systems. Already today, many systems are developed using model-based tools and languages like Matlab/Simulink®, ASCET, LabView, or UML® - to name only a few examples. The applications range from aerospace to automotive systems to medical devices. In the future, coding will probably be replaced by modelling for the application-level development just like high level programming languages have almost completely replaced assembly code.

The idea of using models instead of code for the development of embedded systems has been around for more than a decade now. The key for the current success of the approach is the combination of abstraction and semi-automated refinement of the models using model-transformations as it is illustrated in Figure 3-2.

Using formal models and automated model transformations, model-driven engineering supports the step-wise refinement of the system. A developer starts with more abstract models abstracting from different implementation details. Once the system is defined at this level, he defines some additional information to refine the model. Based on this information, the model can be automatically transformed to a more detailed model. This process can be repeated step by step until the last transformation generates code.

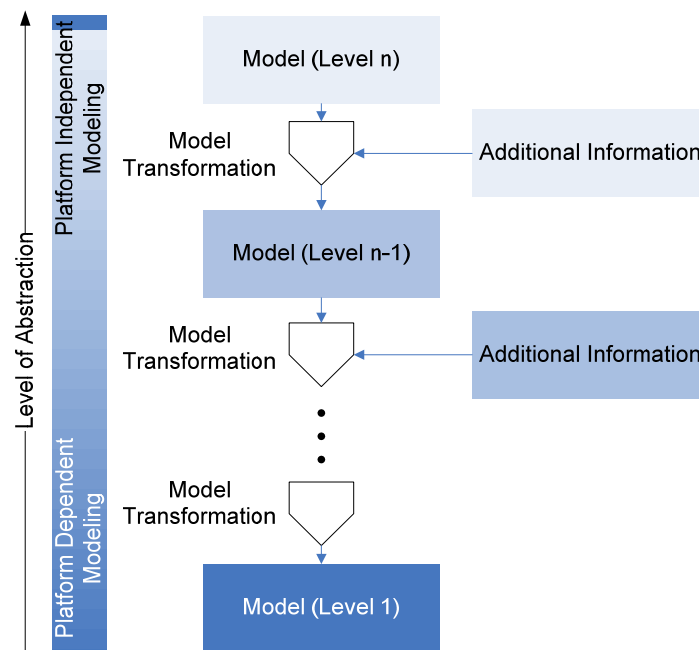


Figure 3-2 General principle of Model-Driven Engineering

This means particularly that models enable the developer to specify the systems using concepts of the problem domain instead of the implementation domain. For example, it is easily possible to specify even complex digital filters by using a single modeling element, which can be used to define the transfer function of the filter. The implementation of this filter, which leads to a complex algorithm, can be generated automatically.

Moreover, it is possible to abstract from the concrete execution platform. For example, aspects like the microcontroller, operating system, or used scheduling strategies need not to be regarded in the early development phases. First, this further reduces the complexity since the developer can concentrate on the functionality of the system through intuitive, graphical modeling languages and does not need to deal with technical details. Second, this enables the platform independent engineering of systems, i.e. it is easily possible to change the execution platform later on without changing the platform-independent models.

In order to further clarify the special characteristics of model-driven engineering, Figure 3-3 illustrates the relation of model-driven engineering to ad-hoc implementation and non-model driven software engineering approaches.

In ad-hoc coding (cf. Figure 3-3 a) the requirements are directly transformed to code. This is obviously a purely manual process performed by the developer. Abstract concepts of the problem domain have to be mapped to a code-based implementation in one step. All technical details have to be considered from the very beginning (e.g., multi-threading, semaphores, data structures, detailed algorithms, etc.). In the domain of IT-systems, many companies had to experience that this approach leads to problems for the development of

complex systems more than two decades ago. With the increasing complexity of embedded systems, however, also many companies in embedded systems domains already have encountered or are currently encountering comparable challenges.

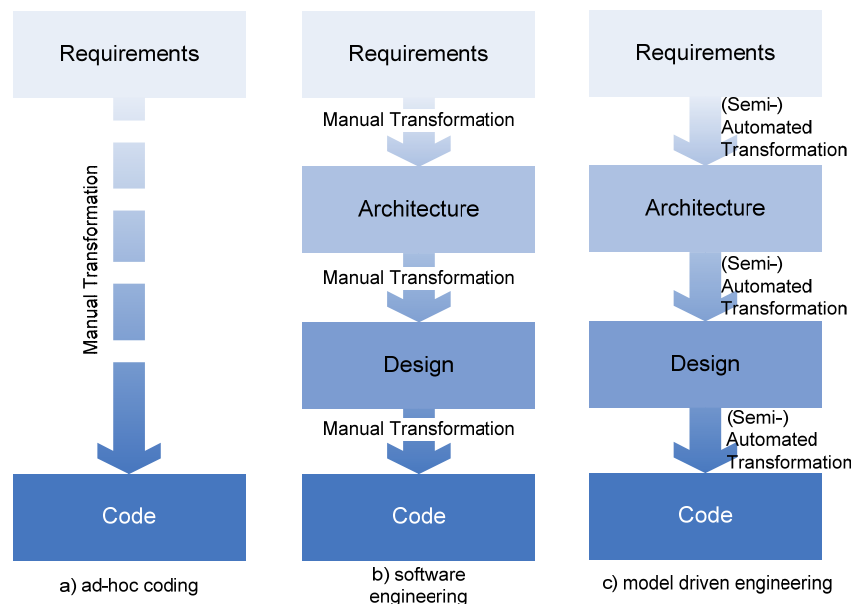


Figure 3-3 Relation between traditional development approaches and model driven engineering

For this reason, it is an established approach to engineer software instead of programming it. The system specification is refined step-wisely from the requirements to architecture to design and finally to code (cf. Figure 3-3 b)¹. Although this approach already decreases the development complexity, the manual transformation of the models requires additional effort and a rigorous development discipline in order to not consider everything above the code as inconsistent documentation and not to focus on the source code, only.

In the model-driven engineering approach, the same development artifacts are created as in conventional software engineering approaches. However, the model transformations are automated. In contrast to non-model-driven approaches, this requires that the specified models follow a formally defined syntax and have an unambiguous semantics. Using semi-automated model transformations, it is possible to tap the full potential of software engineering. The indispensable abstraction and information hiding using different modeling artifacts can be combined with automated transformations in order to shorten the development time and to further increase the quality of the developed systems. Particularly, this means that MDE made models to the central development artifact.

¹ There are various different software engineering approaches with different development phases and artifacts available. The chosen phases are based on a possible derivative of the V-product-model since the main purpose is to illustrate the relation to model-driven engineering.

Therefore, model driven engineering approaches rapidly gained importance and acceptance.

3.2. Meta-Modeling

Models are the key ingredient of model-driven development. Models provide an abstract specification of a system's structure and behavior. Though programs specified using textual programming languages could be considered to be models, as well, model-driven development usually refers to graphical models. The main intention of models is the reduction of complexity by abstraction. As described in the previous section, a single model element can represent a complex behavior, which could only be specified by a complex program if textual programming languages were used. In some sense, a key principle of models is therefore to abstract from the implementation domain to the problem domain. Implementation domain means that programs have to be written using concepts dictated by the underlying hardware and the principles of software programs (e.g., loops, branches, basic commands etc.). In contrast to that, MDE abstracts from these implementation concepts by providing modeling languages that reflect concepts of the problem or application domain (e.g., fast fourier transformations, PID-controllers etc.). The transformation from this problem or application domain to a software-based implementation is then done using automated code generation.

In order to provide a formal means to software development, models have to follow a clear syntax and must have an unambiguous semantics. In the same way as grammars are used to define the syntax of textual programming languages, the syntax of models is defined using meta-models. A meta-model defines all modelling elements that are part of the language as well as the valid interconnections between these elements.

For example, the meta-model shown in Figure 3-4 defines a modeling language for data flow models. It consists of the flow model that contains flow elements (processing nodes, which are comparable to *Simulink* blocks), and flow links between the blocks. A model may contain any number of flow elements and links, each flow element and link is assigned to exactly one flow model. The flow element meta-class is abstract, which means that it may not be instantiated directly. Therefore, there will be no *FlowElement* language element in the resulting modeling language. Language elements are defined by specializations of the flow element meta-class. Two specializations are defined: Ports represent input and output ports, similar to *Simulink* ports. *Add* blocks define blocks that process data. This data is delivered through flow links, which implement data flows, Therefore, each flow link provides one value. *Add* blocks sum up all of these values and provide the result of this calculation as output flow.

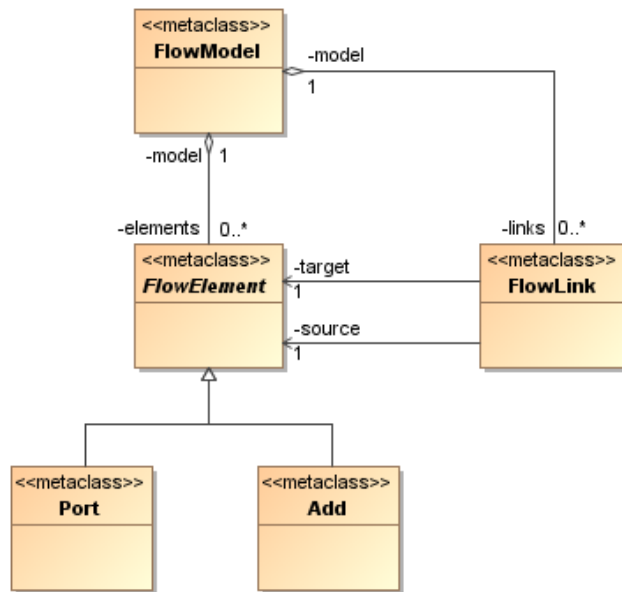


Figure 3-4 Data flow modeling meta model

Meta modeling is supported through different approaches by current tools. Most UML tools do not support true meta-modeling, but UML profiles enable the creation of new modeling languages. Meta models may be transformed into UML profiles. For example, Figure 3-5 shows the meta-model from Figure 3-4 implemented as UML profile. With this profile, the data flow modeling language will be supported by any UML tool that is able to load profiles.

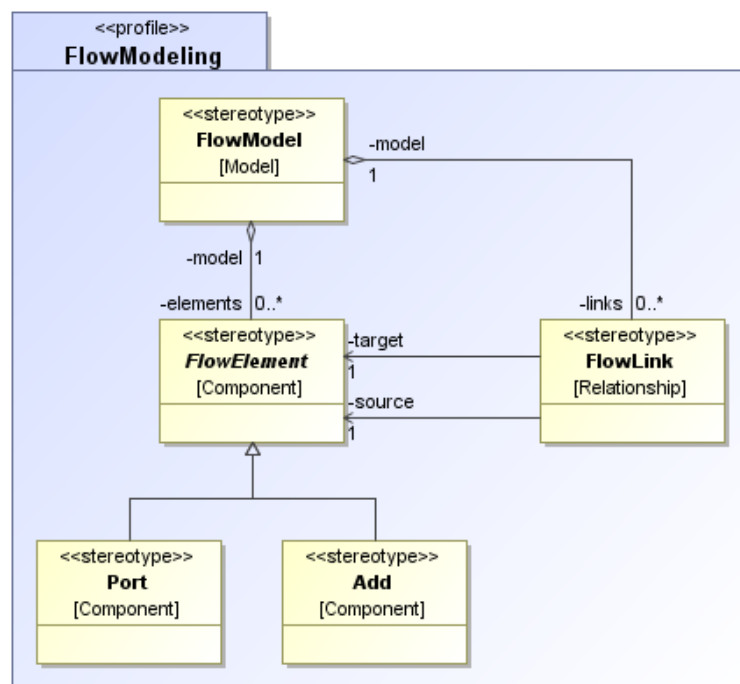


Figure 3-5 Data flow modeling language implemented as UML profile

The difference between a meta-model and an UML profile is that UML profiles define no entirely new meta-classes, but specializations of existing meta classes, which inherit all members of their base classes. Through profiles, it is therefore not possible to create entirely new classes. The base class that will be specialized is shown in brackets, e.g. *[Model]* indicates that a stereotype specializes the meta-class *model*. Available UML meta-classes are defined in the UML meta-model, which is part of the UML language specification. This means that profiles cannot be used to define completely new languages, but only to extend and to refine the UML.

Other tools like Eclipse, on the other hand, support true meta-modeling with the Eclipse Modelling Framework (EMF). EMF defines the following core elements:

- Classes implement meta-classes that will become language elements. EMF classes may be concrete or abstract.
- Specializations define inheritance hierarchies between meta-classes.
- Properties define relations between meta-classes.
- Packages group classes into logical groups.

Figure 3-6 shows the example meta-model from Figure 3-4 realized as EMF model. Based on this model, the Eclipse Graphical Modeling Framework (GMF) provides generated graphical editors that are tailored to this language. Further information regarding EMF and GMF is available in [50].

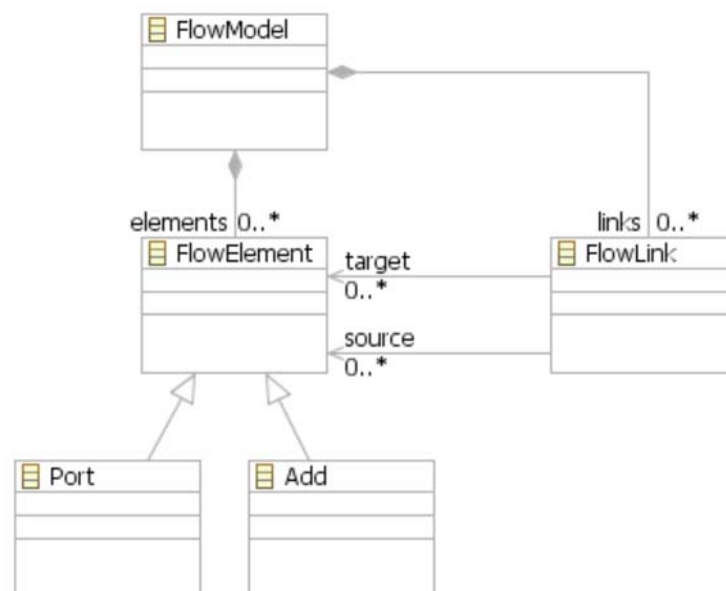


Figure 3-6 Data flow modelling language implemented as EMF model

Using meta models, it is therefore easily possible to define almost arbitrary modeling languages. At the same time, meta models define a formal data structure representing a

system that has been defined using the respective modeling language. By this means, models can be easily processed by automated code generators and analysis algorithms.

A meta model as shown above is usually referred to as the abstract syntax of a modeling language as it only describes the available modeling elements and their interrelationships. In order to enable developers to use this language, a concrete syntax must be defined. This mainly means that the graphical representation of the modeling elements and interconnections must be defined.

Obviously, the definition of meta-models also requires a kind of a modeling language—in the same way as the EBNF is used to define grammars of programming languages. In model-driven engineering, this is called a meta-meta modeling language. In fact, there is an abstraction hierarchy of different meta-models:

- M3: On M3 level, the basic meta-meta modeling language is defined. This language is used for defining modeling languages for developers. For example, the Meta Object Facility (MOF), as it is standardized by the OMG, defines the core meta-meta-model of several modeling languages. It defines primitives like classes, attributes, and packages.
- M2: On M2 level, meta models define modeling languages that are used by developers, for example, the UML. These meta models are based on the MOF that resides on M3 level. UML profiles and their stereotypes therefore do extend the meta model of the UML on the M2 level. Meta models describing non MOF-conforming languages, e.g. the meta model describing Simulink models, also reside on the M2 level. If meta models of several modeling languages are built on top of each other, all of these meta-models do exist on M2 level.
- M1: Models of (software) systems that are created using modeling languages do reside on M1 level.
- M0: Instances of the software system, i.e. the executed system in memory, do reside on M0 level.

3.3. The Model Driven Architecture – MDA®

Although there are different approaches available to realize model-driven software development and the idea has been the vision of many developers and researchers for a very long time, the Model Driven Architecture (MDA® [51]) of the OMG defines a de-facto standard. Although MDA is not focused to a specific domain, one of its main applications is enterprise business systems. It was one of the major original goals of the MDA to separate the business logics and functionalities from fast changing implementation technologies. Nonetheless, MDA is often used as a reference for any model-driven development approach including embedded systems development.

The concept of MDA is illustrated in Figure 3-7. Implementing the general idea of model-driven engineering, as it was introduced above, the MDA considers four different types of models.

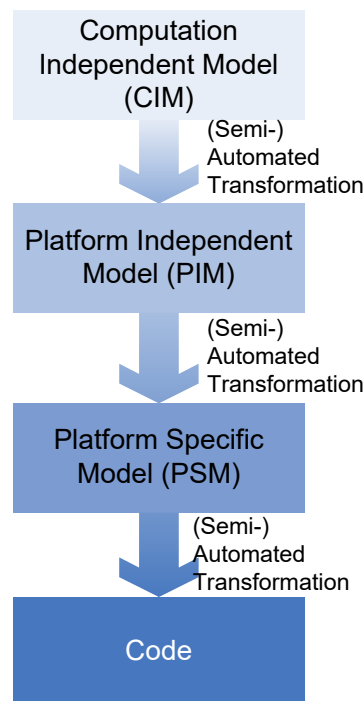


Figure 3-7 Model Driven Architecture (MDA)

The most abstract model is the *Computation Independent Model (CIM)*. These models mainly capture the requirements on a system defining *what* the system has to do and *how well* the system has to do this. At this level, there are usually however no details defined on *how* the system has to realize these requirements. Moreover, the CIM is used to define the environment the system is embedded into.

The *Platform Independent Model (PIM)* is used to define the system independently from the target execution platform. At the PIM level, the actual application is defined, i.e. in the case of the MDA origin this means modeling the business logics and functionalities. Technology specific details are only modeled in the *Platform Specific Model (PSM)*. Different execution platforms in the context of the MDA could be CORBA, .NET, or J2EE to name a few examples. The execution platforms are described in *Platform Models (PM)*, which can be used as an additional input for the transformation of a PIM to a PSM.

3.4. Model-Driven Engineering of Embedded Systems

Regarding the development of embedded systems, the idea of model-driven engineering has originally been driven independently from the MDA®. Many vendors of development tools for embedded systems came up with different model-based design tools like Matlab/Simulink®, Labview, or ASCET. One of the most important objectives of these tools was to shift the focus from programming to modeling embedded systems.

Based on these models, it is therefore possible to specify software using models and to create production code using code generators. It is often necessary to manually modify the code for performance reasons and to integrate it manually into the execution platform. Nonetheless, the code generated by most tools has already a very high quality. After the MDA® has rapidly gained visibility and acceptance, the tool vendors in the embedded systems domain aligned their products to the MDA®. Regarding the origin of the tools, they are obviously focused on the very next level above the implementation. Although sometimes propagated otherwise by tool vendors, it is therefore currently not possible to cover the complete software engineering lifecycle with a single tool.

This tool vendor driven approach in the embedded systems domain, on the one hand, lead to very concrete, but pragmatic and proprietary solutions implemented in professional tools. The MDA® approach, on the other hand, lead to a more systematic and unified understanding and definition of the model driven engineering paradigm, but without providing concrete, professional implementations. The merge of these two approaches in the embedded systems domain therefore lead to a further progress towards the model-driven engineering of embedded systems. Because of this process, most vendors meanwhile agree that it is necessary to provide an appropriate tool chain with an appropriate model exchange. On the one hand, this has pragmatic reasons since the various different tools are already available. Regarding the diversity of the different development tasks in the overall lifecycle on the other hand, it is obviously reasonable to provide a set of tools -each one specialized to a certain lifecycle phase- rather than trying to cover all of these different tasks in a single development tool.

Based on these preliminary remarks, Figure 3-8 illustrates a possible model-driven software development lifecycle. The lifecycle is related to the MDA® on the one hand and available model-based design tools on the other hand.

The requirements (and thus the computation independent model – CIM) are usually captured using special requirements engineering tools such as DOORS™. Depending on the modeling languages used to describe requirements, alternatively or in addition the UML can be used to model requirements.

The transformation from requirements to architecture (and thus from CIM to PIM) is today mainly a manual step since the definition of an architecture is a very creative process. Therefore, the link between requirements and architecture is usually limited to traceability links.

An architecture definition consists of various different views. Supporting various different diagrams and being easily extendable, the UML is an appropriate modeling language for architecture specification. Data-flow oriented modeling tools like Matlab/Simulink® do not provide the required language concepts and flexibility and are therefore in most cases not applicable to the modeling of architectures.

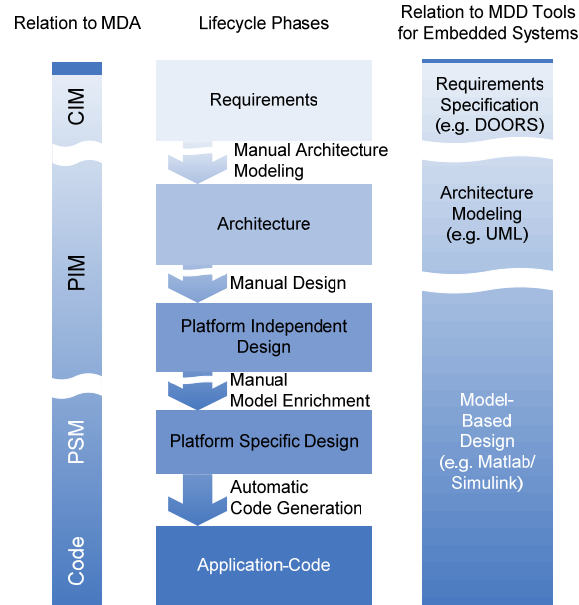


Figure 3-8 Relation of model-based design tools to MDA and the software development lifecycle

Once the architecture of a system is defined, it must be refined in the design. In the first step, the design leads to an executable, but still platform-independent model. Therefore, the platform-independent design also belongs to the platform independent model in the context of the MDA. During this first step of the design, the different components and connections that have been identified during the architecture specification are refined to obtain an executable system. To this end, further sub components must be identified and the actual behavior of the basic components must be modeled. The design can be done using UML-based tools and/or data-flow-oriented modeling tools like Matlab/Simulink®. Although there are various different approaches available, there is a trend to use the UML for modeling the refined structure of the system. Matlab/Simulink® is used to model the behavior of components, particularly if it is necessary to model continuous or hybrid behavior like closed-loop controllers or signal processing. Already today, commercial tools support this combination of UML and Simulink.

In principle, these models are already sufficient to generate executable code. Usually, however, it is necessary to further refine / extend the models in the platform-specific design in order to support the generation of efficient code for the target platform. In relation to the MDA, the platform specific models are created in this phase. Examples for these tasks are the transformation of the behavior models from floating-point to fix point arithmetic or the integration of the generated application code to the platform. This task covers issues like the connection of the application software to the I/O interfaces of the platform (e.g., analog-digital-converters, digital I/O, communication busses etc.) as well as the scheduling of the application code.

Although it is not possible to seamlessly cover the complete development lifecycle using model-driven engineering paradigm, today, tool integration makes rapid progress

and particularly for the design phase the automated transformation from design models to code is possible already today.

3.5. Model Driven Optimization

Often, model-driven engineering is reduced to a series of model-transformation eventually leading to automatically generated code. In fact, however, using models has many further advantages. As one of the most important advantages of models, they enable and facilitate the analysis and simulation of systems already in very early design phases. This approach is comparable to early model-based analyses and simulations as they are used in mechanical engineering. For example, the wings of a new aircraft are modeled, simulated, and optimized before the first wing prototype is actually built. By this means, valuable time and cost can be saved. Comparable to the models used in mechanical engineering, the software system models used for early design evaluations abstract from the actual implementation but preserve the relevant characteristics of the system. Often, early evaluations are based on specifications, i.e. they do not measure the concrete quality properties the systems will finally have, but they rather provide a prognosis on the probable quality the system can have if its implementation will follow the given specification. Therefore, such simulations are not used to verify the systems' implementation but they are used to validate their specification prior to its cost-intensive and time-consuming implementation.

Usually, embedded systems must be optimized with respect to various different, often conflicting quality properties. In the context of a model-driven engineering process, developers do therefore not only define models concerning the actual behavior, but additional models specify the systems' (intended) timing behavior or safety models can be integrated in order to systematically reason about the systems' safety. In combination with the existing design models, this enables the early automated analysis and optimization of systems. As shown in Figure 3-9, the formal syntax of models and the semantics of the underlying models of computation are used to perform automated analyses and simulations in order to determine or to anticipate relevant system properties. Based on these properties, engineers can evaluate the system's quality with respect to different quality properties in order to identify the system's weak spots. Those build the basis for the subsequent optimization step.

In principle, a model-driven optimization makes use of the primary advantage of models: Since models use more abstract specification concepts of the application domain, it is possible to yield formally specified, executable system specifications, which facilitate the early analysis and simulation, very early in the development process. Certainly such models still abstract from many implementation details. For many optimization goals, however, they sufficiently preserve the relevant characteristics. To this end it is however reasonable to choose or extend the modeling language in order to facilitate the modeling of important characteristics.

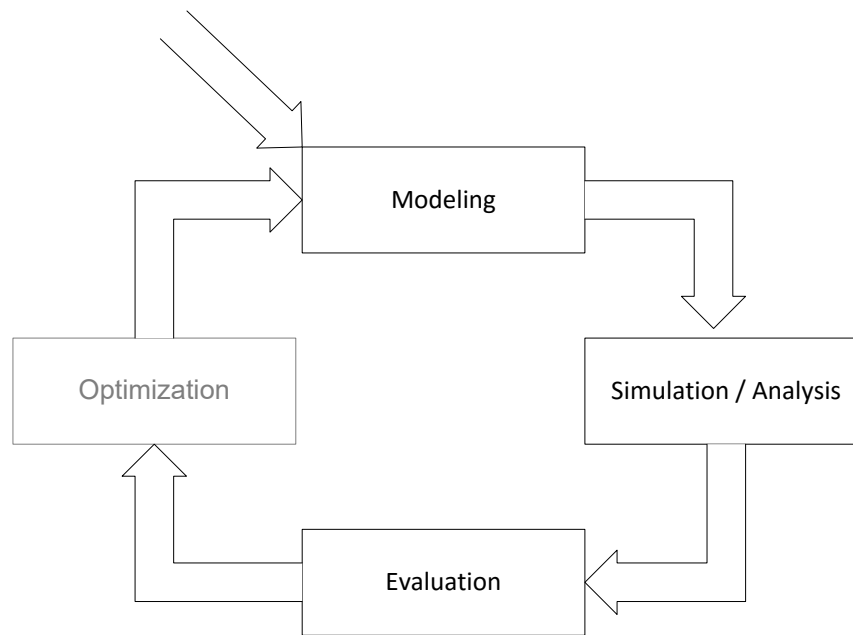


Figure 3-9 Principle Lifecycle of a model-driven optimization process

An example of a model-driven optimization is shown in Figure 3-10. Two models are used. One model defines the E/E-architecture of an aircraft (top-left) including the computers' geometric positions within the aircraft. A second model specifies the available functions that are realized by software including interface requirements (e.g. access to certain sensors, peripherals, memory, CPU-performance etc.). It is now a typical design task to deploy these functions, i.e. to assign them to available computers with the goal to optimize different properties like cost, weight, timing, safety and reliability etc. In order to support this design decision, a model-driven optimization approach takes these two models as input and generates for example an ILP (integer linear programming)-formulation that can be used as input for an ILP-solver in order to identify an optimal deployment.

Usually, however, this is no fully automated step. In fact, the developers define certain constraints (e.g. due to an independence that is required from a safety perspective). Then, the optimizer provides a first solution. Developers take this solution and adopt and extend their models and constraints and start the optimizer again. Using such an iterative approach, the design space exploration can be significantly simplified and important decisions in early development phases can be based upon sound evidence. The developers' experience is very valuable and therefore included in such an iterative process. But due to the immense complexity of such optimization problems, only relying on experience often leads to costly wrong decisions.

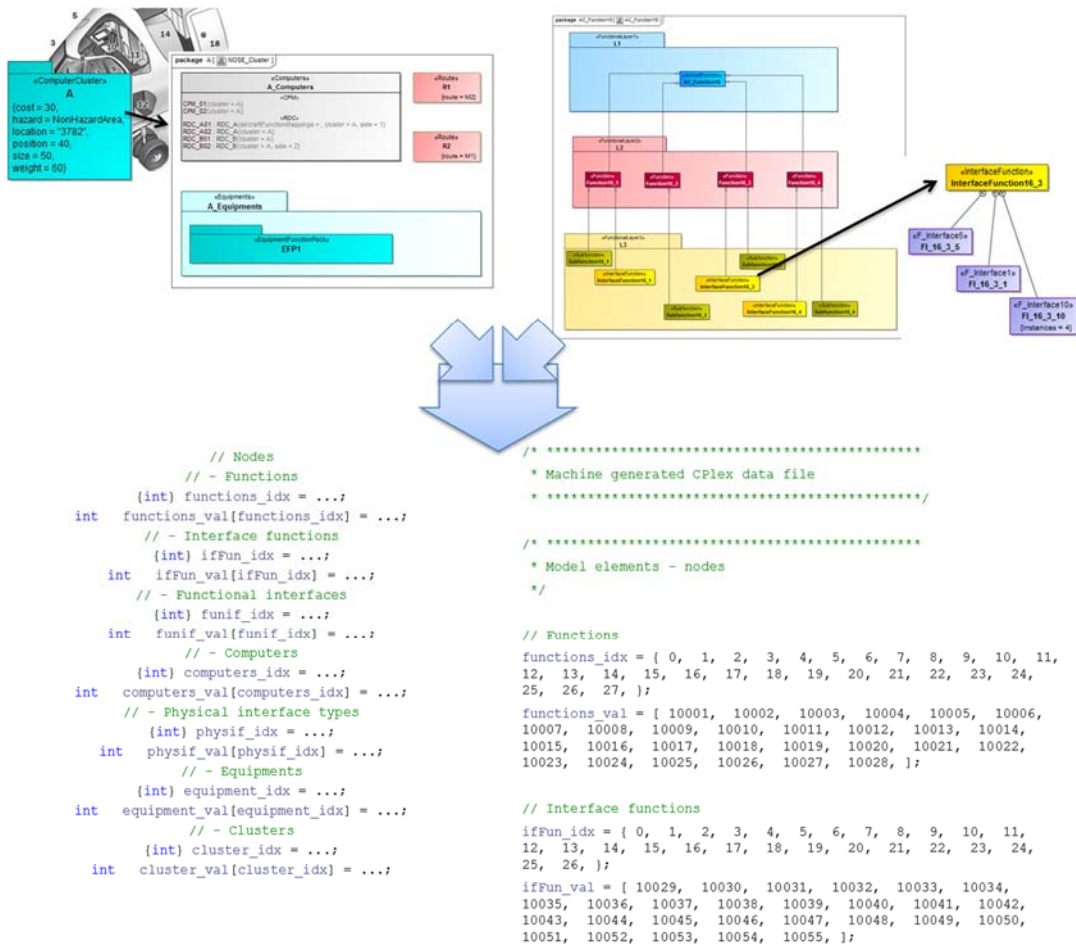


Figure 3-10 Example: Use of models as decision support for deployment optimization

Actually, using models for optimizations is the primary use case of models at runtime as we want to use it for runtime safety assurance: Since system qualities heavily depend on the system’s runtime context, an appropriate anticipation of the latter is indispensable for reasonable optimizations. For open adaptive systems, however, it is almost impossible to predict the runtime context with a sufficient precision. It is therefore the idea of models at runtime to shift models to runtime so that the optimization can take place at runtime when the concrete context is known. The runtime safety assurance framework described in the subsequent chapters applies this idea to the use of safety models at runtime in order to optimize the available system functionality without violating any safety goal.

4. Model-Driven Safety Engineering

The complexity of embedded software systems is rapidly increasing. Since most embedded systems are used to implement safety related functionalities, this trend also affects the safety assurance of these systems. In practice this means that safety engineers have hardly a chance to assure the systems' safety with the required rigor in the time given. In spite of the increasing complexity, the available resources for safety assurance are not increased. In fact, the resources for safety engineering are even step wisely reduced in some application domains. In consequence, the efficiency of safety assurance approaches must be improved.

In order to manage the rapidly increasing development complexity, model-driven engineering has emerged as a new paradigm for embedded systems development. In the same way, by creating model-driven safety engineering approaches, we aim at a reduction of the complexity of safety assurance and therefore to increase the efficiency of safety engineering. It is the idea to profit from the same benefits of model-driven engineering, where

- full modularity,
- more abstract, intuitive, and expressive graphical notations, and
- a sufficient formality to support automation

significantly improved the development efficiency. To this end, safety assurance approaches are set-up based on the principles of model-driven engineering. Though there might be different approaches to improve the efficiency of safety assurance, model-driven safety engineering has the invaluable advantage to be seamlessly integrated into model-driven development approaches, which are more and more used for the development of embedded systems.

As regards the safety assurance of open systems of systems, advancing conventional safety assurance towards model-driven safety assurance has two main advantages. First of all, models provide a sufficiently formal means to represent safety-relevant information. Second, model-driven safety assurance is based up on modular assurance concepts, which are essential for integrating safety assurance with modular development approaches. Therefore, modular safety models provide a sound starting point for the runtime composition of single systems to systems of systems. Model-driven safety assurance does not only improve the efficiency of assuring the safety of conventional systems. But is particularly a very important intermediate step towards the safety assurance of open adaptive systems using safety models at runtime as they will be described in the subsequent chapter.

For these reasons, this chapter introduces model-driven safety engineering. To this end, section 4.1 first gives an overview on the current status quo, analyzes the problems, and shows how we try to use model-driven safety engineering approaches to address these problems. Section 4.2 introduces basic fundamentals of model-driven safety assurance,

before Sections 4.3 and 4.4 introduce our approaches for model-driven safety analyses and safety concepts, respectively.

4.1. Motivation

Before we introduce the main principles of model-driven safety engineering and our concrete approaches, this section describes the big picture of model-driven safety engineering. To this end, we analyze the current challenges and show how model-driven safety engineering can provide appropriate solutions.

4.1.1. Problem Analysis

If we regard the status quo today, the systems' complexity is a major challenge for safety engineers in practice. In order to solve this problem, practitioners see a huge potential in reusability. And indeed, most systems are not developed from the green field but they are based upon already existing systems. Therefore, it is self-evident to think that reusing available safety assurance artifacts from previous projects is a powerful leverage to reduce the safety engineering effort. Reality, however, looks different: For example, the effort that is required for the recertification of an avionics system after a modification may be even higher than the certification effort that was needed for the original system [52]. Considering that about 70% of the total development cost is allocated to verification and certification illustrates the economical dimension of this challenge.

The reasons for this effect are manifold. In general, however, it is not surprising since the approaches used for safety assurance like Fault Tree Analysis (FTA) or Failure Modes and Effects Analysis (FMEA) still follow the basic principles that were developed for non-software systems several decades ago. In consequence, they neither address the specific characteristics of software systems nor do they scale to the rapidly increasing complexity.

We identified four main causes that make safety assurance inefficient and that impede the efficient reuse of safety assurance artifacts:

(1) Inappropriate Architecture

In practice, a main reason for complexity can be found in inappropriate system architectures. Often developers complain that their systems would be too complex to apply safety analysis or quality assurance techniques with the required rigor. Regarding the architecture of real world systems, however, they often look like the example shown in Figure 4-1, which was created by reverse engineering source code of an automotive system. For obvious reasons, it is hardly possible to perform any kind of analysis or modification in such a system. Therefore, the first reason of complexity is rather the system architecture than the complexity of safety assurance approaches. And it is important to note that no safety assurance approach can compensate a bad system architecture.

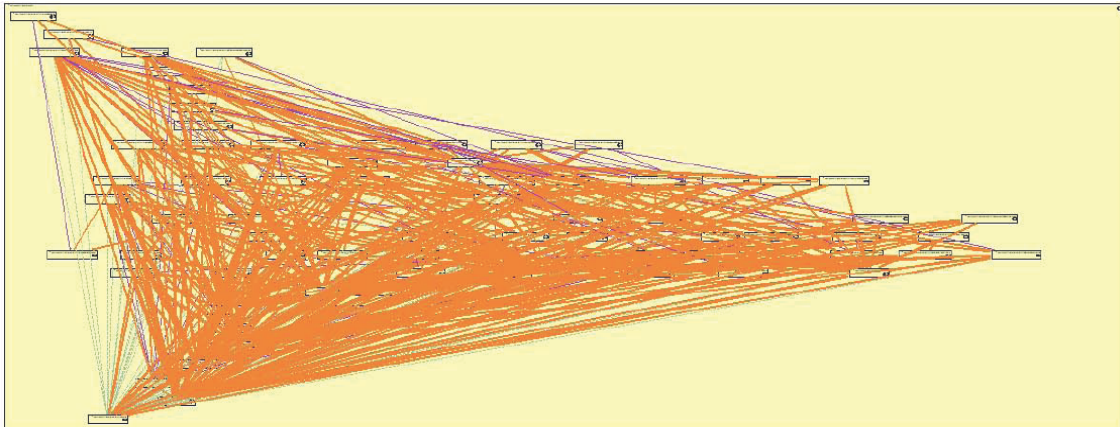


Figure 4-1 Architecture of one out of 20 sub-systems of a real-world embedded system.

(2) Modularity

But even if we assume there was a sound, modifiable and extendable architecture available, the essential modularity of this architecture could not be represented in traditional safety assurance approaches. For example, neither conventional Fault Trees Analyses nor Failure Modes and Effects Analyses support modularity. Traditional safety analyses, for example, only consider one-dimensional error propagation paths, which is a valid assumption for mechanical systems, where a fault of a sub system propagates to the superordinate system. Regarding electronic or software-systems, errors propagate along data paths (e.g., erroneous signals), hardware resources (e.g., shared busses, memory etc.), or physical interactions (e.g., temperature, radiation etc.). The resulting error propagation is multi-dimensional: from sub components to super components, along the signal path between components at the same hierarchy level, between hardware and software, or between the system and its physical environment etc. When this n-dimensional problem is mapped to a one-dimensional safety analysis approach, it is not possible to modularly define all safety aspects that are related to one system component.

(3) Misalignment between Architecture and Safety Artifacts

For the reasons mentioned above, missing modularity leads to a misalignment between system architecture on the one hand and the structure of safety assurance artifacts on the other hand. For example, the faults of a single component can be scattered to several sub trees of a fault tree. In consequence, major parts of a safety analysis can be affected even though only a single component in the system architecture is modified. This problem becomes even worse, since not only safety analyses, but also all other safety artifacts like safety concepts and safety cases are affected. Identifying all required changes of safety artifacts therefore becomes a very time-consuming and error-prone task.

(4) Decoupled Design and Safety Assurance Approaches

This problem is even further aggravated since design and safety assurance are decoupled approaches. Safety engineering artifacts are created in a manual process based on design artifacts like system requirements or architecture. But there is no formal link between them. In consequence, even smallest changes in the design artifacts cause inconsistencies and therefore invalidate the related safety artifacts. In consequence, all of the steps of an impact analysis must be performed manually. And even worse, many inconsistencies are not even detected directly since there is no means to check the consistency automatically. And developers are often not aware that their modifications have an impact on safety so that the safety engineers are not even informed in the case of seemingly minor modifications. The later the inconsistencies are detected in the development lifecycle, the more effort is caused by their correction.

4.1.2. Advantages of Model-Driven Safety Engineering

In order to address these challenges, we applied the idea of the model-driven development paradigm to safety assurance models, namely safety analyses, safety concepts, and safety cases. To this end, all safety artifacts are represented as models in a semi-formal way as it is done in model-driven development. This means that the modeling notation follows a meta-model that formally defines the syntax of the models whereas the semantics is defined using informal text. But nonetheless different analysis and model-transformation algorithms can be implemented to unambiguously realize the semantics in the same way as for example code generators are used to formally implement the informal semantics description in conventional model-driven development approaches.

Generally speaking, using model-driven safety engineering closes the gap between design documents and safety documents in order to improve the alignment and the coupling between design and safety assurance. Therefore, model-driven safety assurance improves consistency, simplifies modifiability and reusability, and furthermore increases the degree of automation by using additional information from the design models for safety analyses.

In principle, model driven safety engineering provides several potential advantages, which build upon each other:

Aligned Modularity

In the first step, it is necessary to provide sufficient concepts of modularity for the safety engineering approaches, before the actual model-driven safety engineering concepts can be used beneficially. It is however important to note that the modularity must be aligned with the concepts of modularity as they are used in the system design in order to improve the modifiability and reusability of the safety models. For example, if the architecture is decomposed into components whose interface is defined by signal ports, the decomposition of the safety models should follow the component structure as

well, and the interfaces of the modular safety models should be related to the components' signal ports. As an example, Figure 4-2 shows how the concepts of modularity of model-driven fault trees seamlessly follows the modularity of the system architecture. A single safety model, in this case a modular fault tree, is seamlessly associated with the component. In model-driven development, modular safety analyses are an integral part of the component's model-based specification. This one-to-one relationship significantly improves the understandability of the relation between safety models and design models, since all faults of the component are defined in its own single safety model.

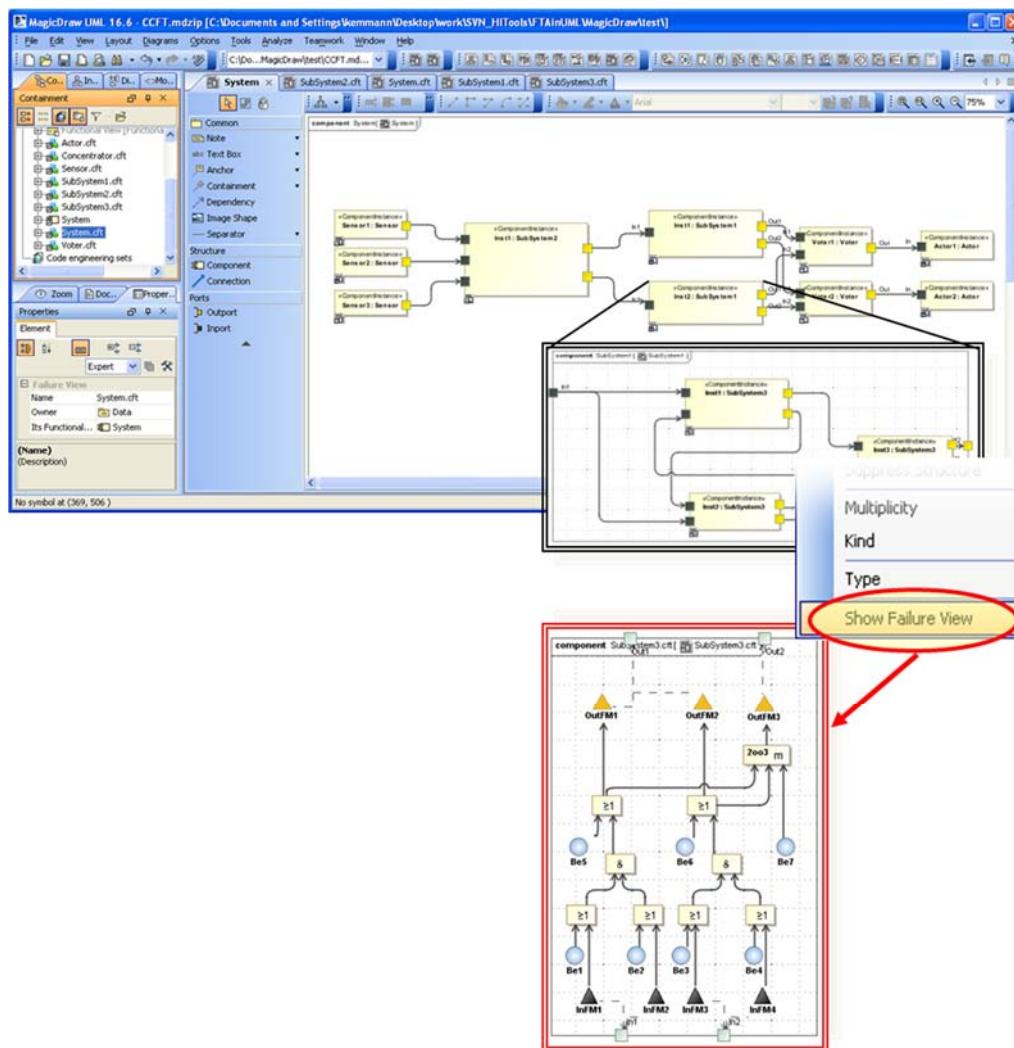


Figure 4-2 Example of integrated safety models based on component fault trees

The self-evident this aspect might seem, the important it is to be considered since different safety approaches support modularity but do not consider the alignment of safety models with the system's architecture. For example, safety case approaches like the goal structuring notation GSN provide their own concepts of modularity [53]. The modularization might however be based on the chain of argumentation. The resulting safety case structure is modular but hardly shows similarities to the system structure.

Therefore, this would again lead to a misalignment between architecture and safety models with all the problems described earlier. Though using modular concepts, the modifiability and reusability would hardly be improved.

Traceability

Aligned modularity is a prerequisite for simplified modifiability and improved reuse. Without the latter, an isolated change in the system may easily impact major parts of the safety assurance models. In order to improve the modifiability of the system, it is additionally important to have an efficient means to identify all parts of the safety models that are impacted by system modifications. This is a major advantage of model-driven safety engineering since a sufficiently formal integration of safety models and design models enables intelligent traceability links. Therefore, model-driven safety engineering as an integrated ingredient of model-driven development provides the ideal basis to efficiently realize complete traceability.

For example, the safety models and design models shown in Figure 4-2 are based upon the same meta-model and refer to the same modeling elements. Thus, they formally refer to the very same parts of a component. Based on this formal link, a change of the component’s interface automatically affects the safety model, as well, and inconsistencies can be detected and partially removed automatically.

As a further example,

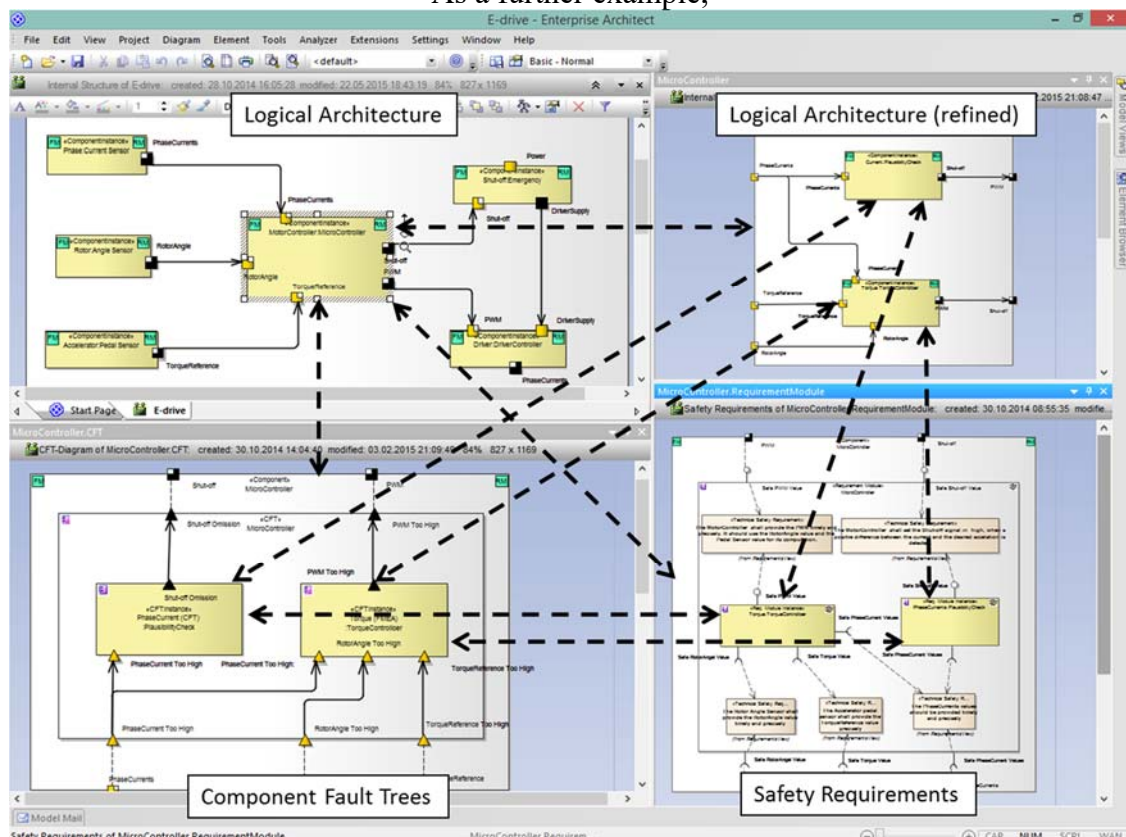


Figure 4-3 shows how the model-driven integration of different safety models and design models enables a seamless traceability based on our model-driven safety analysis

tool iSafe [64]. The safety requirements defined in a safety concept are formally linked to the failure modes that are covered by the requirement as well as to the architectural elements like components, ports, or signals they are referring to. By this means, the impact of changes in the architecture on the safety models can be identified automatically. In the same way, it is possible to identify modifications of the architecture that are required to implement the requirements. Furthermore, it is, for example, much easier to analyze the completeness of the safety requirements with respect to the failure modes that need to be addressed.

Levels of Reusability

Aligned modularity and traceability are prerequisites for reusability. Modularity provides the basis to define self-contained reusable entities. If the safety models are aligned with the architecture, reusing a component means that its safety models are reused automatically, as well. All safety-related information concerning this component is a seamlessly integrated part of the component’s specification and can be reused much more easily than separated safety information that is scattered across various unlinked documents. Additionally, it must be easily possible to check whether or not reused safety artifacts are still valid in a new context. To this end, traceability is required. Simple traces enable checks, whether or not all components or signals that are required to implement the reused safety measure are still available in the new system.

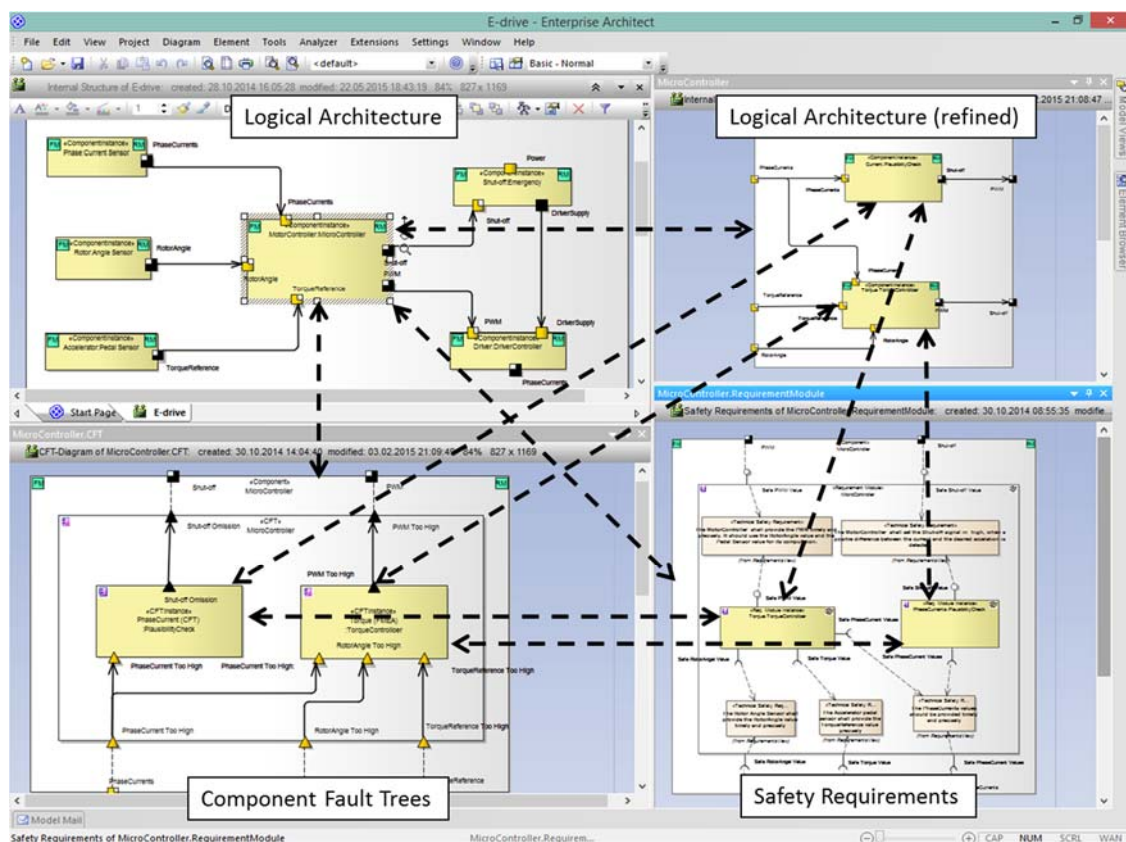


Figure 4-3 Seamless integration between different safety and design models

Reusing components and their associated safety models is however only one of different possible levels of reusability that are supported by model-driven safety engineering. In addition to reusing components, it is also possible to reuse more generic knowledge and parts of the safety assurance as additional advantages of model-driven safety assurance:

For example, the concrete structure of a system often varies from one product to another one, but the relevant signals like the vehicle speed or the lateral acceleration are very stable within a specific domain like vehicle dynamics. Therefore, all the safety-related information concerning these signals is reusable. Such information may include failure modes and possible error detection or handling mechanisms including an argument why those measures are appropriate for the given failure mode. Such information can be reused for semi-automatically generating safety analyses and even safety concepts. Independent from the concrete structure of the component that creates or uses this signal, it is very likely that its safety model has to identify the failure modes of this signal, which can then be reused in order to speed up the process and to enhance the completeness, since the knowledge about potential failure modes increases with each and every product. In addition, it is very likely that the knowledge about possible counter measures for the different failure modes can be reused including potentially available assurance cases for the counter measure.

As a further aspect, counter measures such as the three-layer monitoring architecture ‘e-gas’, as it is often used in the automotive industry, usually affect a series of components. From an architectural point of view, such counter measures can be reused in form of design patterns. With extending model-driven engineering approaches to safety, also the concept of patterns can be extended accordingly. This means that it is not only possible to reuse a typical measure in form of a pattern to modify the architecture, but the pattern can also define the according modifications of related safety models. For example, applying an e-gas pattern would then not only modify the architecture, but it could also modify the corresponding safety analyses in order to include the measure, and it could be used to modify the safety concept / safety case, including already available evidences why the e-gas pattern is appropriate to handle a given set of failure mode types.

As a further extension, so-called *aspects* have gained importance over the last decade. Comparable to patterns, aspects can be used to specify typical, reusable system behavior that is scattered to different components. A typical example is application level communication protocols that are centrally defined as an aspect, which is then automatically *woven* at certain joint points into the system. This means that at every part of the system where signals with specific characteristics are sent or received, the protocol-relevant behavior is integrated automatically. By this means, a cross-cutting aspect like a protocol can be easily modified and reused although its implementation is scattered across different components of the system. Again, this concept can be extended to safety. For example, such mechanisms could be used to model application-level end-to-end communication protocols that ensure a safe communication over untrusted

communication stacks and media as it is done in the AUTOSAR technical safety concept. The aspect will then not only change the system, but it will also be included at all relevant places in the safety analysis models as a counter measure for communication errors. It is then only necessary to model the aspect once in a safety concept as a counter measure for communication problems and it is automatically applied to all untrusted communication channels.

As these examples show, only reusing components including their associated safety models would by far not tap the full potential of model-driven safety engineering. Instead, it is possible to reuse more generic, partially domain-specific but application-independent know-how, which is robust with respect to structural changes of the system. Regarding practice, reusing a lot of knowledge from previous projects does not necessarily lead to similar architectures. As the examples described above show, it is possible to efficiently reuse valuable safety know-how, even though not a single component would be reused unchanged in a new product context. If we, for example, regard the safety concept of a sunroof, the clamping protection has many similarities with that of a power window. Although the architecture of the two systems might look completely different, the failure modes are comparable and can be reused. In the same way the patterns that have been used to assure a clamping protection can be reused and applied to the new architecture. In addition to the knowledge about the pattern as such, for example, all available safety assurance documents can be reused for the safety case, as well.

Automation

All of the advantages described above already increase the efficiency of safety engineering if they are used manually. The main potential, however, lies in the automation of many activities based on the formality of the models. For example, if safety analysis models are given for single components, the resulting analysis model for the overall system can be automatically composed using the design structure as it is defined in the architecture models. Therefore, a manual specification of the system's structure and a network of functions as it has to be done in current FMEA tools, is not required. This information is already defined in design models and can be reused automatically. This example illustrates a main principle of model-driven development: The formality of models in combination with automation ensures that all information that has been defined once is automatically and thus efficiently reused wherever it is required. This approach reduces effort, but also avoids inconsistencies. As another example, consistency checks or impact analyses can be automated in order to increase the efficiency and to improve the model quality.

Particularly regarding the different levels of reuse, automation provides a leverage to significantly improve efficiency. The validity of reusing a component or even the identification of reusable components can be automated. Based on safety-related information about signal types, different parts of safety analyses can be generated automatically. And for obvious reasons, the automation of safety pattern applications or the weaving of safety aspects significantly reduces the required effort.

As regards the safety assurance of open adaptive systems, all of these aspects are of relevance. Of particular importance are the increased degree of formalization enabling an interpretation of the models at runtime as well as the supported modularity, which is an indispensable ingredient enabling runtime safety assurance.

4.2. Safety Modelling

As illustrated in the previous section, model-driven safety engineering provides a series of advantages. The basis for increasing the efficiency lies in the aligned modularity of the safety models as well as the seamless integration into a model-driven development approach. Therefore, this section provides some general information on how we achieve modular safety assurance, and it introduces the fundamental ideas of meta-modeling as they are required for model-driven safety engineering.

4.2.1. Modular Safety Assurance

The idea of modular development is an established key principle in software and systems engineering. Therefore, modular safety assurance can be based upon the basic principles of component based development, which is the most recent and most advanced approach towards modular embedded software and systems engineering.

Regarding component based software development as a basis, a software component can be defined as ‘a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.’ [54].

This definition points out different important characteristics of a component. The first important characteristic is that a component must have contractually specified interfaces. This means that it must be completely and formally specified which services are provided and which services are required by the component. For example, using pre- and post-conditions, a contract defines what is required by a component and what the component provides if all of the defined preconditions are fulfilled. These pre-conditions may include required input data, but also further (non-functional) requirements such as the required quality and timing properties of the data used, required memory, CPU-time, special I/Os, or temperature requirements. In this context, it is very important that a component has explicit context dependencies only. This means that a component must only interact with the usage context using the explicitly defined interfaces. Unspecified side effects, for example caused by communication mechanisms like global variables or shared memory, are not allowed.

As a further important property, a component can be deployed independently. This means a component is a completely independent sub system that can be developed (completely) independently without knowing anything about the usage context except for the information that is given in the interface specification. Consequently, a component can be deployed as a standalone product with all binaries and documents that are required

to integrate the component into a system. Since a component is subject to composition by third parties, it must be defined in such way that it can be integrated into a system without knowing anything about its internal realization except for the information given in the component's interface specification.

In order to enable modular safety assurance, the same principle ideas can be adopted for safety assurance approaches. According to the ideas of component based development, modular safety assurance means that it must be possible to completely assure a components safety without knowing anything else about the usage context than the information that is specified in the component's interface. This characteristic is crucial to preserve the component's modularity since any specific relations on the context that go beyond its interface creates a context dependency, which means a violation of the principle of modularity. Moreover, it must be possible to include the component's safety assurance artifacts in the safety assurance case of a superordinate system without requiring any internal white box information about the component. As soon as it is necessary to understand the component's internals in detail, the intended complexity reduction would be impeded. And it would become difficult for a component supplier to protect its intellectual property.

Interfaces

For these reasons, interfaces are a central element of modularity. In general, an interface should define which types of information are required and provided by a component, respectively. To this end, type systems are used to unambiguously define the components' interface, for example, by defining the service types that are required and provided by a component, respectively. For software components it is not sufficient to define which kind of information is exchanged, but also how this information is exchanged. To this end, it is for example possible to define so-called protocol automata or sequence charts, which specify the protocols, e.g., the sequence of signals, that must be implemented to use the component. As a simple example, such automata could specify that the component requires its input data periodically in specific constant periods.

Applied to safety assurance this means that it is necessary to extend existing type systems with safety specific information. Considering safety analyses and safety requirements as the mostly used safety artifacts, there must be according interface specifications for both of these artifacts.

For safety analyses, we do this by specifying the possible failure modes of the data that is exchanged between components. If, for example, the functional interface of a component is defined using ports that send or receive signals of a certain type, these signals types can be analyzed in order to identify potential failure modes of the signals. By this means, it is for example possible to automatically connect modular safety analyses, if the components' modular safety models are based upon the same failure type system. This is a very efficient approach, since the failure modes have to be defined only

once for a domain when the signal type system is defined.

Usually this can be done using existing safety analysis techniques like HAZOP guidewords. As the types should be valid independently from the implementation of the algorithm providing the signals, the analysis can be performed without knowing anything about the component's implementation. Once the failure modes of the signal types are defined, they provide a very efficient means to specify safety interfaces in a sufficiently formal means. The failure modes can then be used to extend the signal types, facilitating an efficient composition of modular safety models since type checks are quite easy to realize but nonetheless allow for unambiguous compliance checks.

In order to extend this idea to modular safety concepts, i.e. to safety requirements, it is necessary to define a kind of unambiguous safety requirement types in order to specify which safety requirements are required, i.e. demanded by the component, and which safety requirements are provided, i.e. guaranteed by the component. Though this is a difficult challenge for requirements in general, a straight forward solution for safety requirements can be derived from the failure modes: For example, the requirements can be typed as detection or mitigation of specific failure modes. If this is not sufficient, it is possible to define acceptable counter measures for the signals' failure modes. In the same way as the signal failure modes can be derived independently from a component's implementation, it is also possible to identify potential counter measures independently from a component's realization. The signal failure mode types can then be extended by appropriate counter measures. It is then sufficient to reference the according counter measure types in a requirement in order to demand or guarantee its implementation. By this means, even complex requirements can be reduced to types. And checking the compliance of requirements can be reduced to type checks.

In contrast to functional interfaces, it is usually not necessary to extend such a type-based interface with something comparable to protocol automata. Requiring and providing safety requirements is purely static and temporal sequences of requirement fulfilment are not considered in practice. The same is true for safety analyses, since mostly time-independent analyses like fault trees or FMEAs are used in practice, whereas analyses like stochastic petri nets, which can be used to model temporal dependencies, are hardly found in practice.

This basic idea of safety interfaces will be seized again in Sections 4.3 and 4.4 in order to explain its concrete realization for safety analyses and safety concepts, respectively.

Mapping

In addition to the pure interface of a component, it is necessary to have a basic understanding from a black box point of view of how the component's outputs depend on its inputs. In system and software engineering, this is nothing else than a black box behavior specification of the component. For safety engineering this means that we must define a mapping from input failure modes to output failure modes, and from safety requirements that are required, i.e. demanded by the component, to safety requirements

that are provided, i.e. guaranteed by the component. In order to support heterogeneous safety models, this mapping should abstract from concrete safety models that are used internally within the component. To this end, it is important to choose a mapping that is expressive enough so that all relevant safety models can be easily transformed into this general mapping specification.

As regards safety analyses, the mapping specification therefore depends on the required expressiveness. As long as it is not necessary to consider the temporal order of events, quite simple mappings are sufficient. For example, minimal cut sets are an efficient and sufficient means to define which input failure modes and which internal failure modes lead to which output failure modes. Besides the qualitative dependencies, cut sets can also be used to quantitatively calculate the failure probability distributions of the output failure modes based on the probabilities of the input and internal failure modes. All time-independent analysis techniques like FTA and FMEA can be easily transformed into a cut set notation. If temporal dependencies shall be considered, it is necessary to use more expressive notations like generalized stochastic petri nets (GSPNs) [55]. This aspect will be discussed in more detail in Section 4.3.

While the typical mappings are quite well established for safety analyses, there are no established approaches for the mapping of requirements available. Nonetheless, we defined first concepts, which could be understood as ‘positive trees’, i.e. they are comparable to fault trees with the difference that they do not consider faults and their consequences, but they consider the success case. By this means it is possible to model which combination of demands must be fulfilled so that a guarantee will be fulfilled [65][66][67][68]. This aspect will be discussed in more detail in Section 4.4.

4.2.2. Meta-modeling

As described in more detail in the previous chapter, meta models are one of the core foundations of model-driven development. Just like grammars are used to define new programming languages, meta-models are the basis to define new modeling languages. To this end, they define the syntax of the language, i.e. which modeling elements can be defined and how the single modeling elements can be connected. In addition, it is essential to define the language’s semantics. This means that it must be defined how the system will behave based on the model-based specification. The semantics is therefore the indispensable basis for automated analyses and transformations based on the models. Formal languages define the semantics using formal specification techniques, which provide the basis for sophisticated analyses like formal proofs. In practice, so-called semi-formal modeling languages like the UML are more widely used. In this case, the syntax is formally defined, whereas the semantics is unambiguously described but not formally specified. Nonetheless, the semantics is eventually implemented in model transformations (e.g., code generators) and model-based analyses. Usually this leads to a sufficiently formal realization of the semantics from a practical point of view.

Syntax of Safety Models

In order to integrate safety models into a model-driven development approach, it is therefore indispensable to extend the underlying modeling language. To this end, the modeling elements that are required for specifying safety artifacts must be integrated into the underlying meta model.

In order to illustrate this aspect, Figure 4-4 shows exemplarily the integration of model-based fault trees into the meta-model of an UML-based modeling approach. The underlying modeling approach provides modeling elements to model components, to specify the components' interfaces using ports, and to define the interaction of components using connections between ports. In order to be able to model failure modes that can occur at certain ports, this modeling element must be extended in the meta-model as it is shown in Figure 4-4.

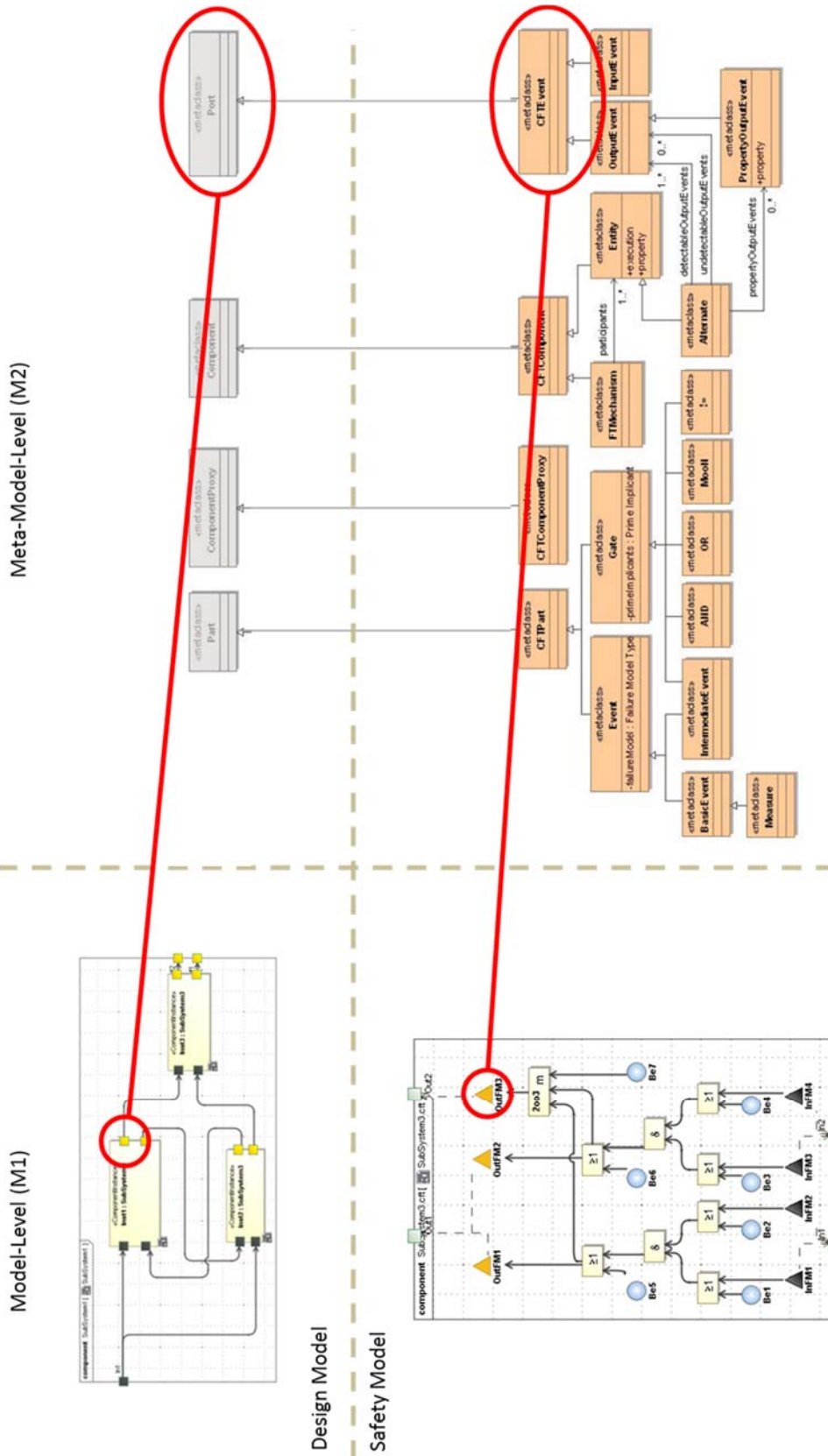


Figure 4-4 Meta-Model Integration of Design Models and Safety Models

Semantics of Safety Models

Additionally, it is important to define the semantics. For safety analyses this is often possible quite easily since model-driven safety analysis approaches are based upon existing analysis techniques like fault trees. Since the semantics of the analysis techniques is usually mathematically and thus formally described, it is sufficient to specify a mapping between the modeling elements and the corresponding concepts of the existing analysis approach. If no new concepts are introduced in the modeling language, this step is usually quite straight forward. For example, the modeling element ‘BasicEvent’ in the meta-model of a model-driven fault tree can be directly mapped to the established (mathematical) meaning of a basic event in fault tree analysis. If new modeling elements are introduced, like for example failure ports in component fault trees, it is necessary to extend this semantics. To this end, it is reasonable to describe the semantics by extending the underlying mathematical analysis model. For example, the mathematical semantics of component fault trees including the port concept is described mathematically in [27]. Knowing the semantics is very important for safety engineers. Otherwise they would not be able to ensure that the safety models actually express the intended meaning of the modeler. For obvious reasons, the semantics is even more important for engineers who would like to extend the model-driven safety engineering approach with their own analyses or generators.

As regards safety concepts, it is a little bit more difficult to express their semantics in a sufficiently formal way, since no established underlying formalisms exist. As described in the previous section, it is however possible to define the semantics of safety concept interfaces based on extended type systems. As regards the mapping, it is often based upon a simple logical mapping between demanded and guaranteed requirements comparable to fault trees. In this case, the semantics of the mappings could be defined using predicates. Obviously, it is often not necessary to use highly sophisticated and thus complex formalisms, but rather simple solutions are often sufficient and even more recommendable.

4.3. Model Driven Safety Analyses

In order to realize model-driven safety analyses, it is first necessary to understand the basic principle of modularizing safety analyses. Today, most of the available modular safety analysis approaches follow the idea of failure logic modeling as it is introduced in sub section 4.3.1. As a concrete implementation, sub section 4.3.2 briefly introduces our model-driven safety analysis approaches.

4.3.1. Failure Logic Modeling

Aligned modularity is a key to model-driven safety analyses. Conventional safety analyses such as fault tree analyses or failure modes and effects analyses are not modular by default. Conventional fault trees, for example, are real trees and therefore only allow

for defining sub trees, but no real modules are supported that could be assigned to architectural components.

In order to ensure the alignment of the modularity used in design and safety, the modularization concepts used for the safety models should be similar to those used in development. Regarding typical modeling approaches for embedded systems, they follow an actor-oriented development approach [56]. Superficially explained this means that different concurrently executed components (called *actors*) are modelled, whose interfaces are defined using ports and who communicate with each other using connections between the ports. In other words, this means that modularity is achieved by port based interfaces, and the composition is realized by connecting the ports.

A similar concept for modularizing safety models can be found in failure logic modeling [69]. Based on failure logic modeling (FLM) we define a general concept for modular safety analyses as it is shown in Figure 4-5. A modular analysis model defines how failure modes at a component’s input interface in combination with failure modes inside the component propagate to failure modes at the component’s output interface. In principle, the interface can include any kind of interaction between a component and its context. Besides the functional interactions such as the exchange of signals, this might also include physical aspects such as heat, radiation, or power consumption.

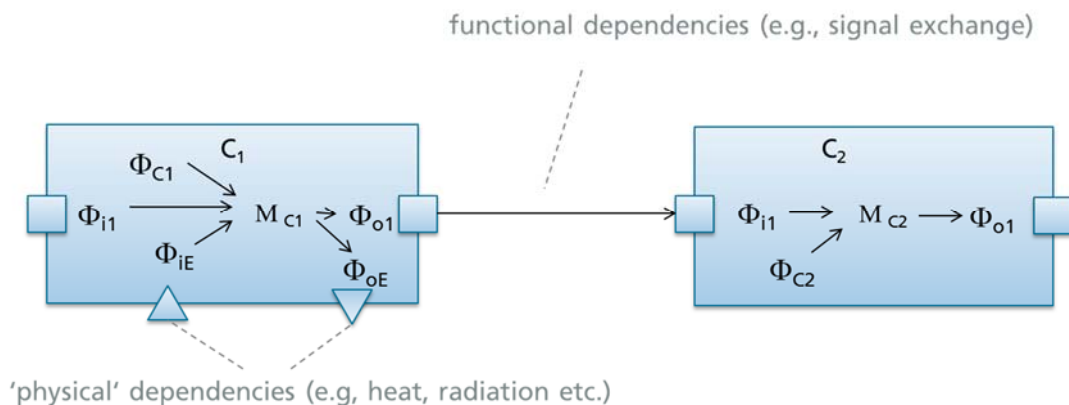


Figure 4-5 Principle Idea of Modular Safety Analysis Modeling

In a simplified understanding, a modular safety analysis model can be seen as a tuple

$$MSAM = \langle \Phi_i, \Phi_c, \Phi_o, M \rangle$$

with

- $\Phi_i \in \Phi$: set of input failure modes
- $\Phi_c \in \Phi$: set of internal failure modes
- $\Phi_o \in \Phi$: set of output failure modes
- Φ : set failure types
- $M: \Phi_i \times \Phi_c \mapsto \Phi_o$

For each component, a set of input failure modes Φ_i , a set of internal failure modes Φ_c , and a set of output failure modes Φ_o must be defined. In order to ensure the composability of two components, it is necessary to type the failure modes. This means that in contrast to traditional analyses, it is not sufficient to simply define failure modes using a string, but all failure modes must have a type. Otherwise, it would not be possible to unambiguously map the output failure modes of one component to the input failure modes of a connected component. In consequence this means that at least the modular safety analysis models of each pair of connected components must be based upon the same set of failure types Φ .

In order to define the failure propagation inside the component, it is necessary to define a mapping M , which specifies how input and internal failure modes are mapped to output failure modes.

In principle, any modular safety analysis approach follows this idea and defines specific realizations for the single elements of the tuple. This means that any modular safety analysis approach must provide a means to specify

- a failure type system
- typed input failure modes
- (typed) internal failure modes
- typed output failure modes
- a mapping between input/internal failure modes and output failures modes.

The first modular approaches such as the *failure propagation and transformation notation* (FPTN) [70] introduced new notations for defining failure modes and the mapping between them. Therefore, they are hardly found in practice, today. Papadopoulos et al. introduced the Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) in [203]. HiP-HOPS have been the basis of the error annex of EAST-ADL. Based on EAST-HDL, HiP-HOPS have also been integrated into model-driven tool chains. HiP-HOPS play an important role in research and industry. A further approach was introduced by Hawkins and McDermid for object oriented systems, which is based on contracts for modular safety analyses [204].

In practice, more established analysis approaches have gained more importance. For example, the fault tree analysis has been advanced to a modular approach by the introduction of component fault trees (CFT) [26], as it will be introduced in more detail in Section 4.3.2. Analogously we extended the failure modes and effects analysis (FMEA) following the idea of modular safety analysis modeling.

4.3.2. Safety Analysis Meta-Models

In order to realize a model-driven modular safety analysis model, it is necessary to define a meta-model, which specifies a modeling language, a safety engineer can use to model the failure mode propagation. The first modular safety analysis approaches used their own notation and have not found acceptance in practice. Established approaches like

fault tree analyses or failure modes and effects analysis on the other hand did not provide sufficient concepts of modularity. With the introduction of component fault trees (CFT) [26] by Kaiser et al., fault trees as an established safety analysis approach had been extended by a sufficient concept of modularity. Component fault trees were further matured and a sound underlying mathematical concept was defined [27]. Component fault trees provided the means to model them following the systems' structure. But they were still defined independently from architecture models. Moreover, CFTs did not support failure type systems. Therefore, we have extended the concept in order to seamlessly integrate it into a component-based development approach: Component-Integrated Component Fault Trees C²FT [57][58][59]. In order to particularly support the analysis of software-based systems, we further extended the concept by a semi-quantitative analysis approach based on second-order-probabilities [60]. As a further step towards model-driven safety analysis, we seamlessly integrated C²FTs as a profile into the UML [61]. We then integrated C²FTs into a standardized approach for the model-based engineering of embedded systems [62].

In practice, however, different analysis techniques are used for safety engineering. Therefore, we defined an approach enabling the combination of different established safety analysis approaches, namely fault tree analyses and failure modes and effects analyses as it will be described in this thesis. This generalized approach will be briefly introduced in the following.

Generic Failure Propagation Meta-Model

Particularly considering the modularization of development, it is quite likely that different analysis techniques are combined within one system. While one component might be analyzed using an FTA, another one might be analyzed using an FMEA. Nonetheless, it should be possible to analyze the resulting overall failure propagation throughout the overall system. Therefore, we defined an analysis framework facilitating the combination of different analysis techniques.

To this end, it is necessary to first define a generic failure propagation meta-model as it is shown in Figure 4-6. This provides a generic abstraction from a concrete safety analysis technique. If the meta-models of different analysis techniques like FTA and FMEA are then derived from this common failure propagation model, they can be easily combined in an integrated safety analysis.

In this meta-model, all model elements are derived from the base class *FpModelElement*. A basic modeling element that is required in any failure propagation model are failure modes, which are represented by *BasicFailureMode*. Moreover, this basic meta-model defines that any *FailurePropagationModel* has to define *InputFailureModes* as well as *OutputFailureModes*, which are specializations of a generic super class *InterfaceFailureMode*.

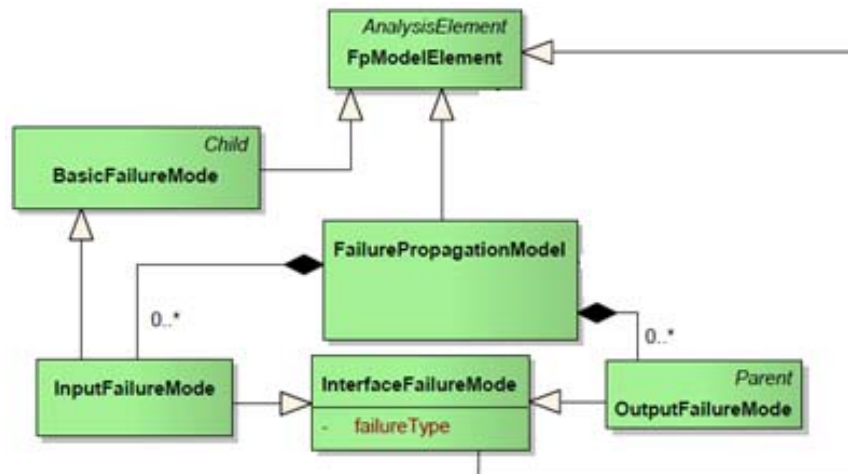


Figure 4-6 Failure Propagation Meta-Model

In fact, the failure propagation model represents a basic modular safety analysis model. Following the idea described in the previous section, it defines a typed interface of input and output failures modes. Basic failure modes provide the basis for internal failure modes. In addition to the elements shown in Figure 4-6, a simple mapping based on minimal cut sets is supported as a common basis for the different heterogeneous safety analysis models.

Failure Type System

In order to connect different failure propagation models, it is necessary to ensure that they have the same meaning of failure modes. To this end, we use a failure type system providing a lightweight mechanism to formally specify failure modes [63]. Although there are different approaches available, which aim at defining a common failure type system including failure modes like omission, commission, too late, too early, etc., practice shows that failure modes are very system, component and signal-specific. Therefore, it is important to enable safety engineers to define a failure mode type system for a system, which lays the common basis for all modular failure propagation models. The meta model of the failure mode type system we have defined is shown in Figure 4-7. A *FailureTypeSystem* consists of a list of different *FailureTypes*, which are related to each other as *subTypes* and *superTypes*.

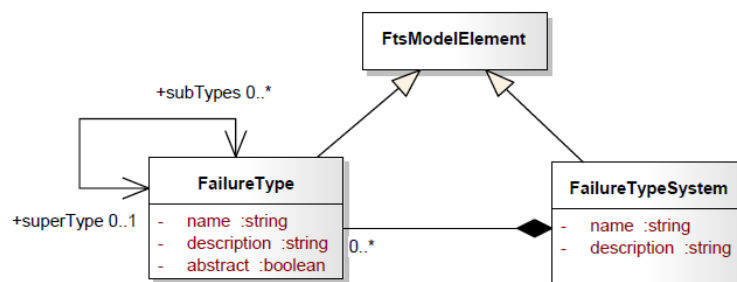


Figure 4-7 Failure Type System Meta-model

An example is shown in Figure 4-8. The example shows a generic failure type classification as it was defined by McDermid et al. [72]. Following this commonly accepted classification, a failure type can be refined to timing failures, value failures, and provision failures. Provision failures are refined to commission failures and omission failures, respectively. As regards value failures, McDermid distinguishes between magnitude failures and logical failures. The former is refined to ‘too high’ and ‘too low’ failure modes, respectively. The latter is refined to ‘false positive’ and ‘false negative’ failure modes, respectively. As regards timing failures, he defines the refined failure modes ‘too early’ and ‘too late’.

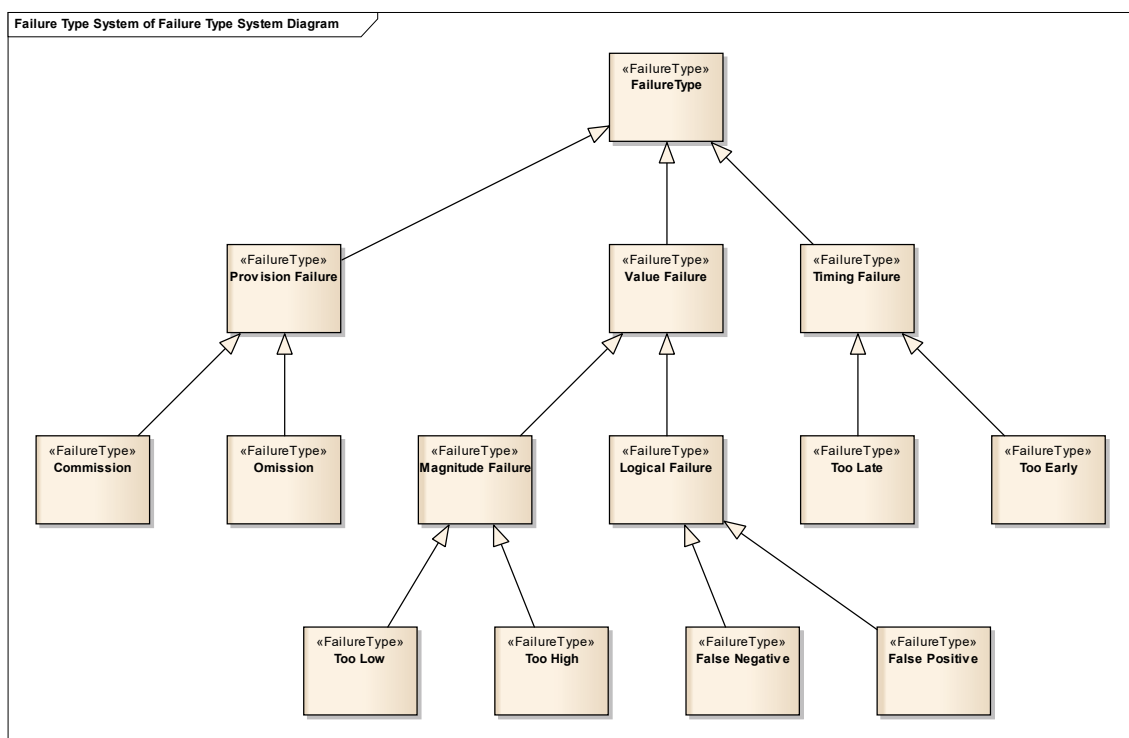


Figure 4-8 Example of a failure type system

As mentioned before, this example is based on a generic failure classification scheme. But many modular analysis approaches only allow to use such generic failure types. Experience in practice, however, has shown that it is reasonable and often indispensable to have more specific failure types. Nonetheless, this generic classification often provides a good starting point for deriving product-specific failure type systems.

In fact, dedicated failures types contain a lot of reusable knowledge. As an example, let us consider a wheel speed signal v_{Wheel} of a car. If we apply the generic failure type system shown in Figure 4-8, the failure mode that the value is higher than it should be would be represented by the failure type ‘too high’. There are, however, various different reasons why the value can be too high, each of which requiring different counter measures and being of a different criticality. For example, the value can be too high due to a sensor failure. But since the wheel speed is measured based on rotation pulses, a wrong

calibration of the wheel's diameter could also lead to too a high value of v_{Wheel} . A further reason could be that the wheels are spinning, i.e. wheel slip, which is not an actual error, but is a nominal result based on the limitations of the principle of measurement.

All of these failure modes require different counter measures. For sensor failures, a sensor self-test can be used to check the sensor value, or a plausibility check using other wheel speed values could be used to implement an error detection. As regards a wrong estimation of the wheel's diameter, however, this sensor check would be useless, since even a correct sensor would nonetheless lead to a wrong value. Therefore, different checks and recalibration algorithms are required to detect and potentially to correct wrong diameter estimations. Lastly, spinning wheels are no real failure mode, but nonetheless this situation causes too high a wheel speed value leading to similar effects in depending components like a faulty signal. In this case, however, the actual functionality, i.e. the nominal behavior must be adjusted instead of defining an error detection and handling mechanism.

Explicitly defining these different failure modes in a type system with an according explanation obviously contains a lot of valuable domain knowledge, which has been gathered over years of experience in a series of safety analyses. It is hardly possible to regain the same maturity of failure mode definitions in a simple safety analysis. Therefore, it is very reasonable to explicitly define those failure modes in a type system. Moreover, obviously completely different counter measures are required. Storing this knowledge about appropriate counter measures is very valuable, as well. More importantly, however, this shows that simply defining a failure type 'too high' is not sufficient since it is not detailed enough to define an adequate counter measure. Nor is it sufficient for implementing a check whether or not an implemented counter measure is appropriate since, for example, a sensor check does not cover a wrong diameter calibration. Moreover, a failure type check based on a simple type such as 'too high' easily leads to wrong results. While, for example, the analyst of the providing components thinks of a calibration problem, when defining a 'too high' failure, the analyst of the consuming component might consider the same failure type being a sensor failure. As a consequence, the safety analysis models would be inconsistent to each other and their composition would lead to wrong analysis results.

Even this very simple example shows, why we believe that defining dedicated type systems is not only reasonable in the sense of storing valuable and reusable domain knowledge. But that it is particularly necessary to avoid wrong analysis results if independent modular safety analysis models are composed based on the failure type systems. For this reason, failure type models play an essential role in modular safety analysis approaches.

Structural Propagation Meta-Model

In order to enable an aligned modularity supporting heterogeneous analysis models, it is necessary to provide some generic representation of the structure as part of the safety meta-model. Usually, the structure follows the structure of the architecture of the

analyzed system. But the interfaces of the structural elements do not represent the data interface, but the failure mode interface of the single components. This is necessary, since a signal usually has more than one failure mode. And in order to compose the modular safety analysis models, it is necessary to have the possibility to unambiguously connect the single failure modes. Figure 4-9 shows the structural propagation meta-model as we defined it for supporting general structural failure propagation models.

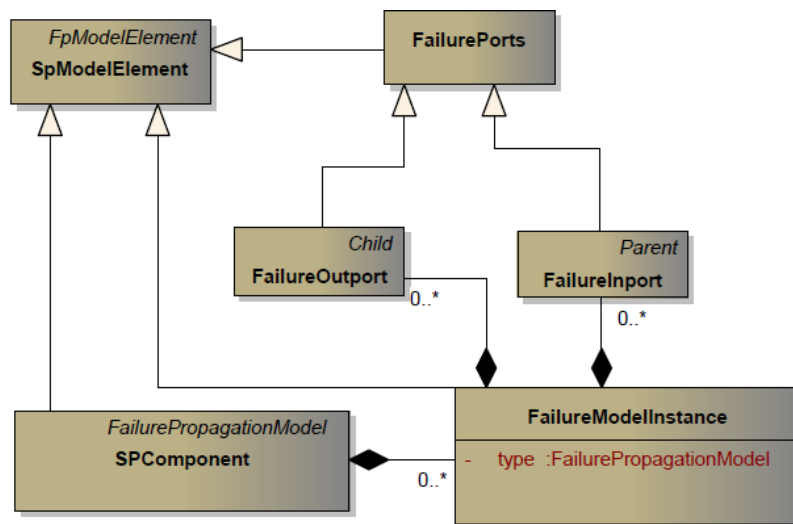


Figure 4-9 Structural Propagation Meta-Model

All elements of a structural propagation model are derived from the basic class *SPModelElement*. A component defined in a structural propagation model is nothing else than a special failure propagation model. Therefore, the meta-model element *SPComponent* is derived from *FailurePropagationModel*, which was defined before. Each *SPComponent* contains an instance of another failure propagation model (*FailureModelInstance*), which could be another structural propagation model in case the component is structurally refined. Or it could be another propagation model like an FTA or an FMEA. The interfaces are defined using *FailurePorts*, which are specialized to *FailureOutputs* and *FailureInports*.

This generic structural failure propagation model is very important. Just like behavior models are often only defined for the atomic components, i.e. components without sub components, safety analyses are often also only defined for atomic components. Nonetheless, the propagation obviously spans across all hierarchy levels. Structural failure propagation models therefore provide the essential means facilitating aligned modularity, i.e. a one-to-one relationship between architectural components on the one hand and the structural elements used in the safety analysis model on the other hand.

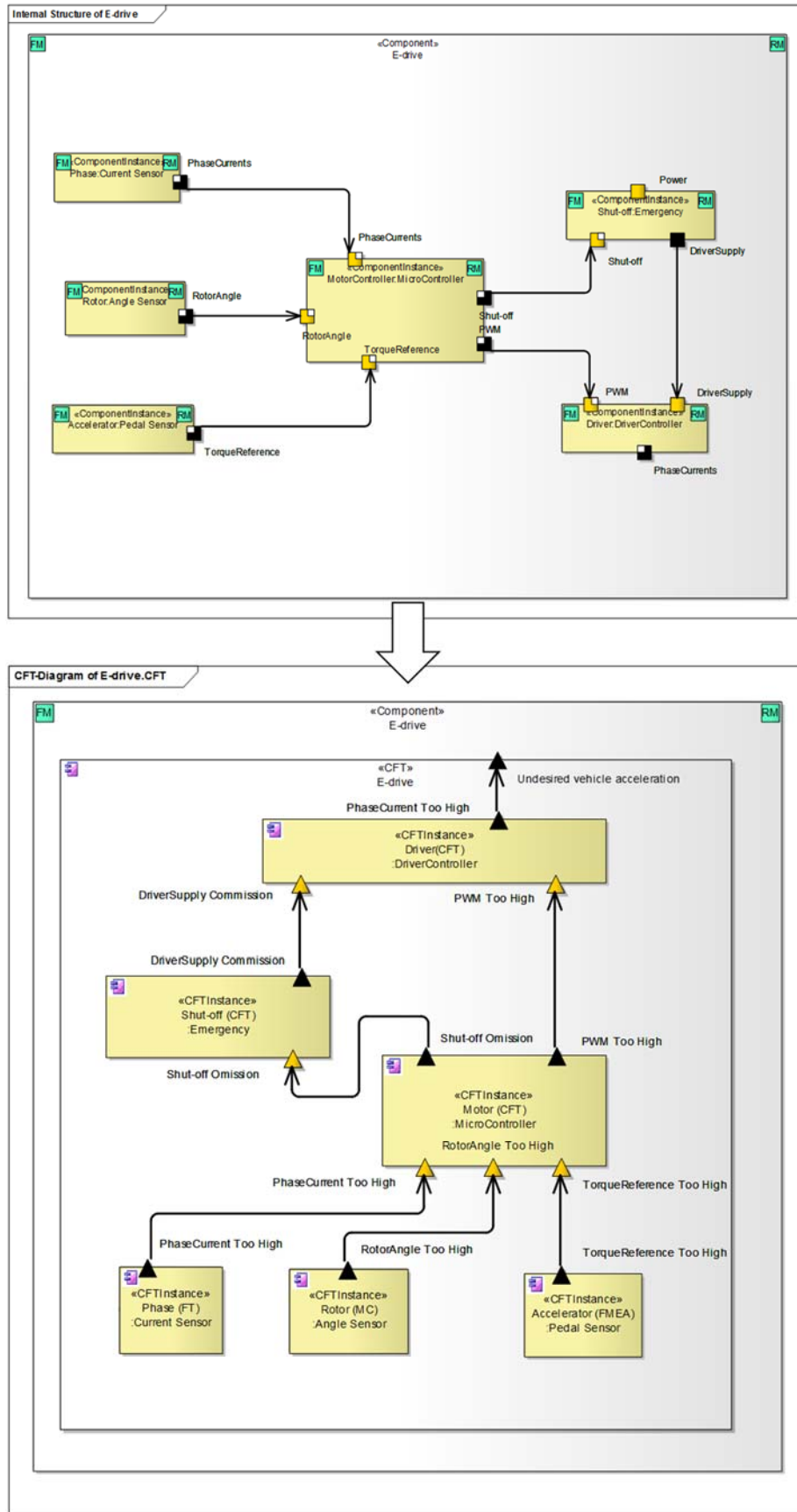


Figure 4-10 Example of a structural failure propagation model.

Figure 4-10 shows an example of a structural failure propagation model. The upper part of Figure 4-10 shows the logical architecture of a control system of an electric drive train consisting of six connected components. The lower part of Figure 4-10 shows the failure propagation model. Each component of the architecture is represented by one structural failure propagation component. Instead of signal ports, however, the interfaces in the structural propagation model are defined by failure types propagating from one component to another one.

At this level of the hierarchy, no failure propagation models like fault trees are defined for the single components. Instead, it is sufficient to model the propagation following the system structure. As a result, the structure of the safety analysis model is completely aligned to the system structure defined in the architecture model.

Component Fault Tree Meta Model

Once the safety analysis has been refined to the lowest level of the hierarchy that ought to be considered, it is necessary to define a concrete modular safety analysis model for modelling the failure propagation. It is, however, often not necessary that the analysis model replicates all hierarchy levels of the architecture. In fact, it is often reasonable to not further refine a component from a safety point view and to directly define the failure propagation model even though the architectural component is further refined to sub components.

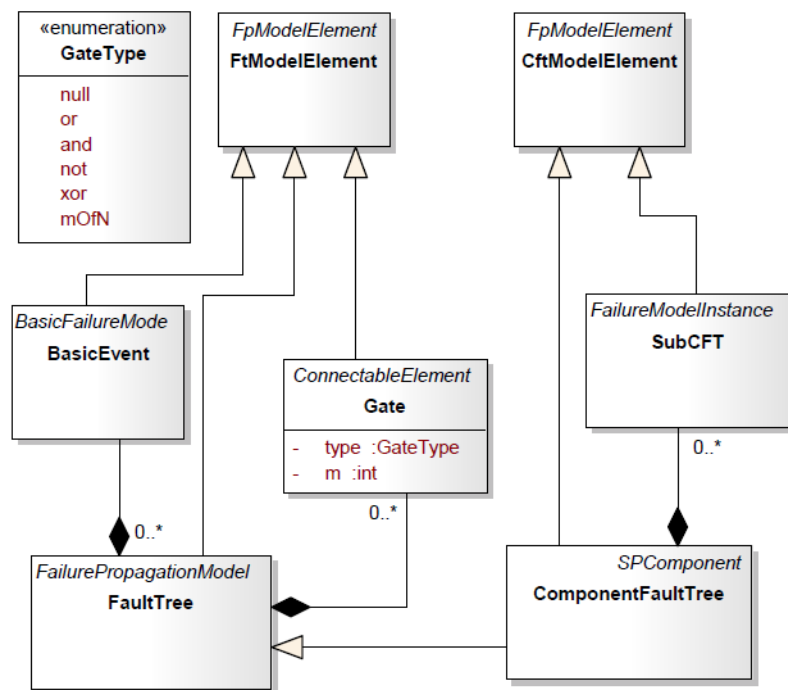


Figure 4-11 Component Fault Trees and Fault trees Meta-Model

A very important concrete modular safety analysis model is component fault trees. Figure 4-11 shows the meta model of component fault trees. In order to support heterogeneous safety analysis models, they are defined as a specialized failure propagation model. In fact, our meta model integrates the concept of basic fault trees,

which are used to model the failure propagation as such, with component fault trees, which extend fault trees by the concept of failure mode interfaces and hierarchy.

The fault tree model as such is quite simple, since a fault tree mainly consists of *BasicEvents* and *Gates*. All other required elements are inherited from the generic failure propagation models. A *ComponentFaultTree* is then a specialization of a fault tree. A component fault tree might contain *SubCFTs* in order to model hierarchical fault trees. Since component fault trees additionally inherit all elements of a structural propagation component (*SPComponent*), they also provide all concepts required to model input and output failure modes.

Defining component fault trees as a specialization of conventional fault trees has several advantages. Most importantly, converting component fault trees to conventional fault trees prior to analysis opens a wide range of possibilities since already existing analysis techniques can be easily adopted to model-driven fault trees without requiring a modification of the analysis algorithm to modular component fault trees. Moreover, it is then possible to export the models to established tools in order to run the analyses provided by these tools. First, this increases the range of possible analyses. Second, and more importantly, this enables the use of qualified analysis tools. Since safety analysis results are used as evidences in safety cases, it is necessary to prove that the analysis provides correct results. Such a proof is very difficult and requires a lot of effort. Using existing proven-in-use tools therefore avoids this need for a proof, which is very important for the practical application of model-driven safety analyses.

As an example, the lower part of Figure 4-12 shows the component fault tree of the architectural component *DriverController* of a drivetrain control system. Component fault trees are very similar to traditional fault trees, but there are two main differences. First, a component fault tree can have several inputs and outputs, i.e. it may have several top events. Second, C²FTs are seamlessly integrated into the architectural component. This means that the fault tree is an inherent part of the component. If the component is reused, its fault tree is reused as well. Furthermore, the input and output failure modes of the C²FT are semantically and syntactically assigned to the according ports of the architectural component. By this means, the failure propagation can be determined automatically by using the system architecture. And it is possible to detect inconsistencies automatically, for example, if ports are modified, added, or removed. Moreover, it facilitates completeness checks, since it is for example possible to detect ports, which are not considered in the failure model.

Even though the example is rather simple, it shows some typical aspects of modular failure propagation models. Failure modes of the outputs depend on internal failures of the component as well as on failures of the input signals. The component calculates a phase current for controlling an electric motor. In this simplified example we use the generic failure type system introduced before. Therefore, the most important failure mode of the according output signal is ‘too_high’.

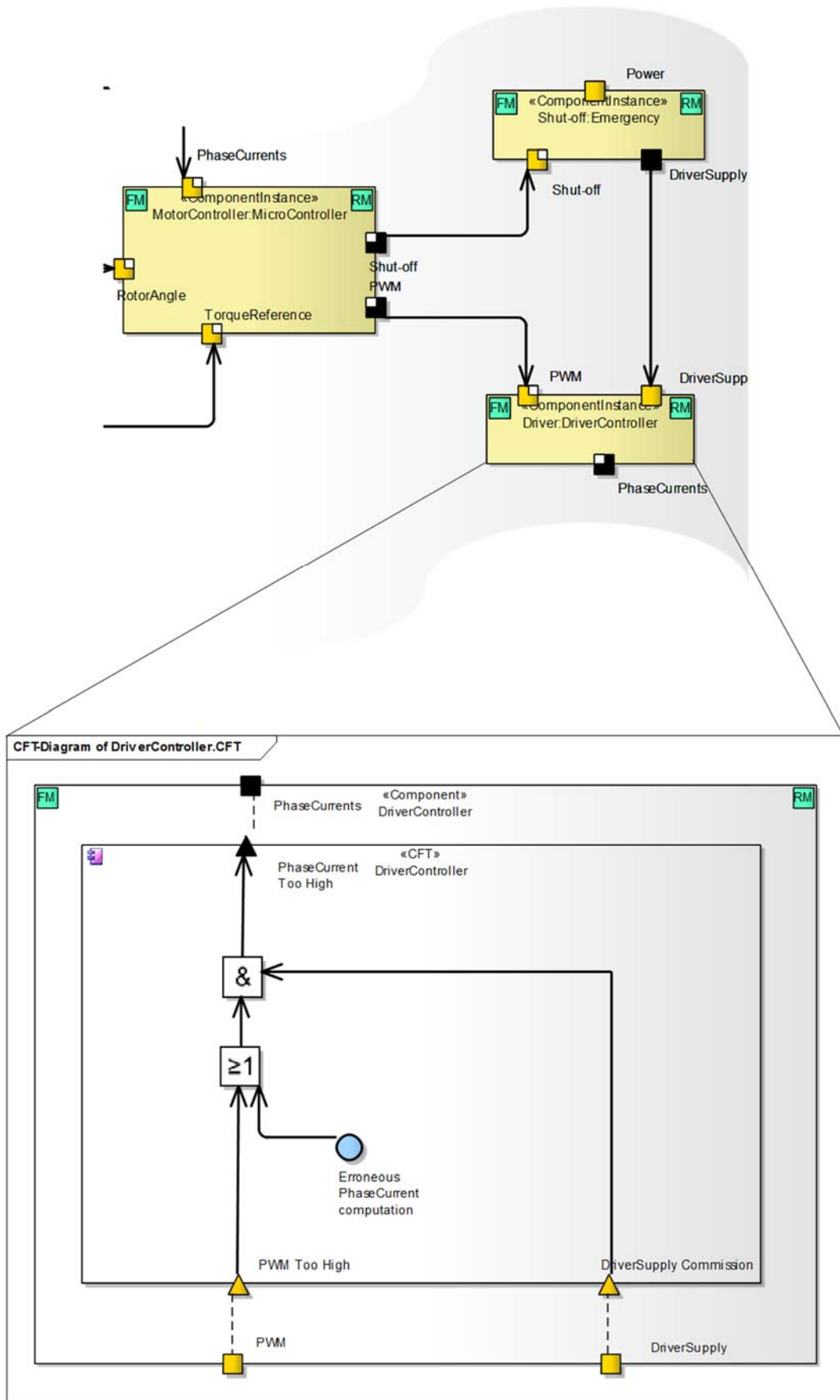


Figure 4-12 Example of a component fault tree

This happens only, if there is a commission failure of the signal ‘DriverSupply’, i.e., the signal is received due to an error although it should not be received. Additionally, the phase current calculation must be erroneous. This could happen if there is an internal calculation error (‘Erroneous phase current computation’), or if the PWM input signal is already ‘too high’.

These two branches of internal problems as well as external problems resulting from faulty input values is a typical pattern found in almost any modular safety analysis models as it was described before.

Failure Modes and Effects Analysis (FMEA) Meta-model

Besides fault trees, the failure modes and effects analysis (FMEA) is one of the most widely used safety analysis techniques in practice. A conventional FMEA is not modular since it is usually applied to a monolithic system design. For example, assume a system with two components *A* and *B* and that *A* sends a signal to *B*. If we now analyze component *A* using a conventional FMEA, we would first define a failure mode. If we now define the effect of this failure mode, we would have a look at component *B*, since *B* depends on a signal from *A*. More precisely, the effect defined for a failure mode in *A*, would be a failure mode defined in component *B*. Vice versa, if we analyze the causes of the failure mode in component *B*, we would identify the failure mode of *A* as one potential cause. Therefore, the FMEA of *A* depends on the FMEA of *B* and vice versa. In consequence, they cannot be modularly defined.

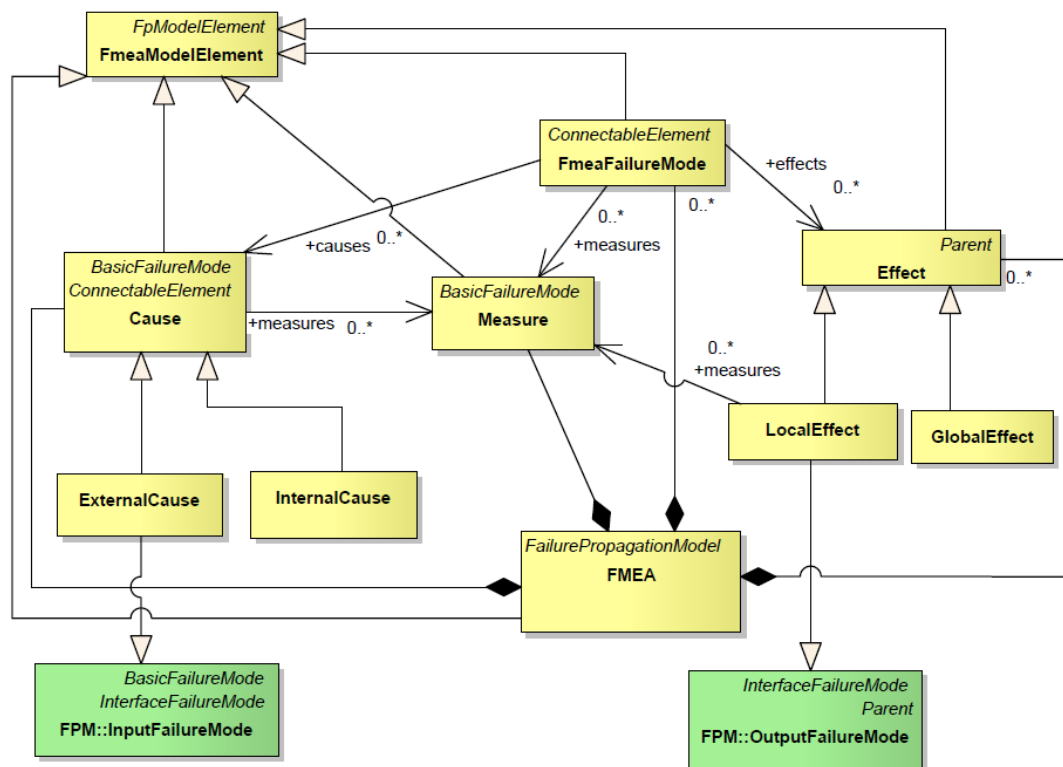


Figure 4-13 FMEA Meta-Model

Therefore, we first needed to modularize the FMEA in order to provide an FMEA notation as modular safety analysis model. This result is shown in Figure 4-13. In the same way as component fault trees, the *FMEA* is also realized as a failure propagation model.

In principle, a modular FMEA contains the same elements as any FMEA, namely *FailureModes*, *Causes*, *Effects*, and *Measures*. In a traditional FMEA, however, the effect is related to a failure mode of a depending component (e.g., the super component) and the cause is related to the failure mode of a used component (e.g., a sub component). As a consequence, a conventional FMEA is not modular, but it strongly depends on the overall system structure and the relationships between the single components. In order to overcome this limitation, a modular FMEA only refers to elements of the analyzed component. To this end, the cause is refined to an *ExternalCause* and an *InternalCause*. The internal cause refers to a fault or error inside the analyzed component. The external cause refers to an input failure mode, e.g., a failure mode of an input signal, and can thus be specified based on the component's interface without any knowledge about the superordinate context. In the same way, we distinguish between a *LocalEffect* and a *GlobalEffect*. The local effect relates to an output failure mode, e.g., a resulting failure mode of an output signal. Again, this effect can be specified based on the component's local interface. The global effect is only informal text, which is not used for any formal analysis, but it helps safety managers to document any ideas on additional effects that go beyond the local interface (e.g. potential effects at system level, which can be cross-checked during the system level analysis).

As an additional aspect, a *Measure* is not informally defined as in a conventional FMEA. Instead, it must be assigned to the element it addresses, i.e., a measure may address a single cause, the failure mode as such, or one of its effects. This is necessary for determining the failure propagation model. By this means, such a modular FMEA can be easily translated into a fault tree following a simple worst case estimation. The effects form the top events of the fault tree. All failure modes of an effect are combined in an or-gate, since a single failure mode is sufficient to cause the effect. In the same way, all causes of a failure mode are connected using an or-gate. Only the measures are connected using an and-gate.

As an example Figure 4-14 shows the graphical representation of a modular, model-driven FMEA. Causes, failure modes, effects, and measures are assigned to different compartments in order to keep a clear structure. As mentioned above, the measures are assigned to the element they address. For example, the *PlausibilityCheck* detects all possible erroneous calculations independently from the cause, whereas the *InputGradientCheck* only detects errors in the input *PhaseCurrents*, i.e. it covers only a single cause of the failure mode. In the same way as component fault trees, the input causes and effects of the FMEA are syntactically and semantically assigned to the according ports of the associated architectural component.

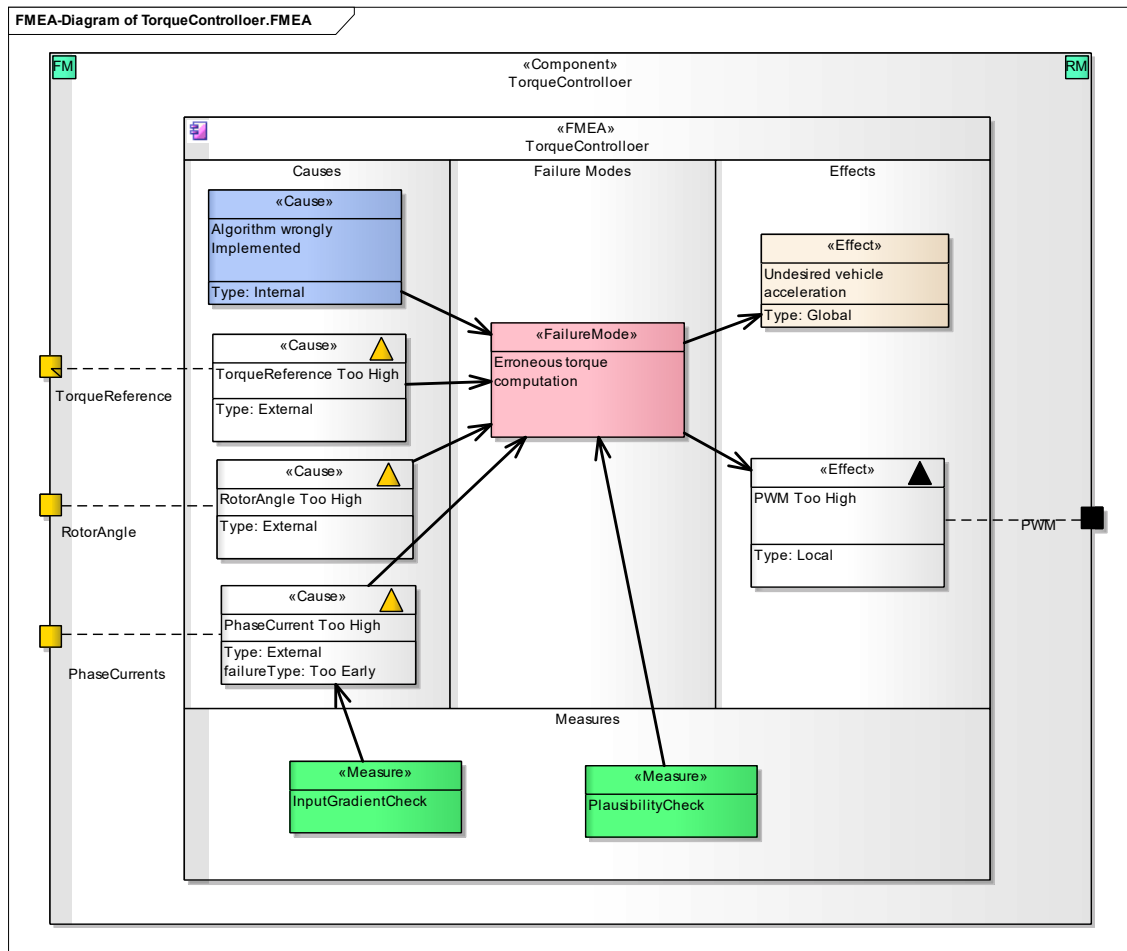


Figure 4-14 Example of a graphical representation of a model-driven FMEA

4.4. Model-Driven Safety Concepts and Safety Cases

In addition to modular safety analyses, the modular definition of safety requirements as modular safety concepts as well as modular safety cases, i.e. the proof that requirements have been fulfilled, play an increasingly important role. Regarding approaches summarized in terms like modular safety certification - as they are also described in safety standards - mainly consider modular safety cases. Defining a safety case is in principle a decoupled process, which can run after or in parallel to other safety assurance activities summarizing the main outcomes in order to compile a qualitative proof that the system meets its safety goals. Just like any other validation and verification activity, however, a safety case should be strongly related to the safety requirements, since eventually it shall proof that the safety requirements are fulfilled by the system.

Therefore, it is reasonable to use similar approaches for safety concepts containing the safety requirements and safety cases. Starting with a safety concept, the latter will be step wisely extended by evidences that the requirements have been fulfilled, like test reports etc. In consequence, a safety concept and a safety case are the same development artifact, which evolves with the progress of the project. For this reason, we provide an integrated model-driven approach for modular safety concepts and modular safety cases.

Before our approach is described in section 4.4.2, section 4.4.1 gives a short overview on currently available modular certification approaches.

4.4.1. Modular Certification Approaches

The international standard ISO 26262 for the functional safety of road vehicles defines the concept of a Safety Element out of Context (SEooC) [4]. A SEooC is defined as a component for which there is no single predestinated application in a specific system. Therefore, the SEooC developer does not know the concrete role the product has to play in the safety concept. Subsystems, hardware components, and software components may be developed as SEooCs. Typical software SEooCs are reusable, application-independent components such as operating systems, libraries, or middleware in general. The standard does not provide any suggestions or methods on how to identify safety requirements or how to increase the chance that demanded and guaranteed requirements match. Neither does the standard provide information on how to perform the verification of the guaranteed requirements during integration of the SEooC.

The DO-297 [73] standardizes the modular certification of components in an Integrated Modular Avionic (IMA) system in the avionics domain. In the avionics domain, the term incremental acceptance is used instead of modular certification. Acceptance is defined as the confirmation of a certification body that a module of an IMA system (a general-purpose execution platform or an application) fulfills its specification. This is important, since a certification can only be given to a final, completely integrated system. An acceptance, however, can be achieved for an IMA system and is one building block of the final certification, with the latter always being in the context of a specific aircraft. The wording incremental has been chosen because the process of the DO-297 allows step-wise acceptance of single modules of a system and because it allows incrementally extending a system with new applications, without having to re-certify all the modules in the system. This is particularly important, since not only the composition of a system out of single components is interesting, but also the re-certification effort after modifying the system should be reduced.

There are different research approaches available, which could be used to instantiate these rather coarse concepts defined in safety standards:

Rushby provides some considerations on the use of modular certification for software components in IMA architectures. The goal is to enable the certification of software components by enabling them to perform their functions in a given (aircraft) context solely based on assumptions about other related software components. He identified three key elements as the potential backbone of a corresponding approach [74]:

1. *Partitioning* creates an environment that enforces the interfaces between components; thus, the only failure modes that need be considered are those in which software components perform their function incorrectly, or deliver incorrect behavior at their interfaces.

2. *Assume-guarantee reasoning* is a technique that allows one component to be verified in the presence of assumptions about another one, and vice versa.
3. *Separation of properties into normal and abnormal properties*. Abnormal properties capture behavior in the presence of failures.

To ensure that the assumptions are fulfilled and that the system is safe, he identified three classes of properties that must be established using assume-guarantee reasoning:

1. *Safe function* ensures that each component performs its function safely under all conditions consistent with its fault hypothesis;
2. *True guarantees* ensure that each component delivers its appropriate guarantees;
3. *Controlled failure* is used to prevent a ‘domino effect’ where the failure of one component causes others to fail, too.

Another possible approach is given by the Goal Structuring Notation (GSN) [49] (cf. Chapter 2), which is a graphical notation for modeling a safety argument, which is the core part of every safety case. A safety case has been defined in the context of the GSN as follows:

‘A safety case communicates a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context.’

Therefore, a safety case serves the purpose of specifying a comprehensive argument to prove the safety of a system. To this end, the GSN allows modeling tree-like arguments beginning with safety goals, and iteratively connecting them through chains of logical argumentation and sub-goals, with the evidences created during system development. Evidences can be performed tests or analysis reports from an FMEA or an FTA that are used for underpinning the fulfillment of the goals.

In order to deal with modular systems and modular certification, there is an extension to the GSN available that allows for modularizing safety cases [53]. The interface of a safety case module is defined by a set of public items that are available to be used in other safety case modules and a set of items that the safety case module at hand demands from other modules. Those items can be goals, evidences, and context. A strategy for the construction of a modular safety case architecture is given in [75].

The DECOS (Dependable Embedded Components and Systems) project [76] provided a generic safety case approach for incremental certification, which improves the efficiency of the certification process and thus shall facilitate significant cost savings during the development of safety-critical systems. Modularity is achieved by separating the certification of core services and architectural services from applications [77] [76].

Following these basic ideas, we developed the VerSaI (*Vertical Safety Interfaces*) method in order to assist the integrator of an integrated architecture in checking whether the application software components are able to run safely on the execution platforms of

the system, and if so, provide assistance in generating appropriate evidence [66]. This is particularly important since applications are developed independently from platforms like, for example, in AUTOSAR [78] or IMA [79].

In order to check the *safety compatibility* between the application and the platform, demands and guarantees have to be specified using a model-based approach. Demands are typically used to express all the properties a platform needs to provide for an application to be executed safely, whereas guarantees represent the safety-related properties the platform possesses. A compatibility check is successful if a sound argument for the fulfillment of the demands with the available guarantees can be established. In order to enable tool-supported integration, the VerSaI approach offers a semi-formal language for modeling these demands and guarantees. The language consists of a number of elements, each representing a certain type of demand or guarantee exchanged by an application and a platform. This implies that there is a finite number of language elements and, therefore, also a finite number of dependencies that can be expressed with the language. First evaluations have shown that this is suitable, because the typical service relationships between an application and a platform are finite and regular, too.

4.4.2. Safety Concept Meta-Models

As mentioned earlier, it is in many cases reasonable to define safety concepts and safety cases in the same artifact. Starting with the top-level safety goals, the safety requirements are step wisely refined in a modular and hierarchical way, in order to derive more detailed requirements for the single components of the system. Along the development lifecycle, evidence will be added to the safety concept in order to proof the fulfillment of the safety requirements. In early phases, the results of safety analyses can be used to proof the adequacy of a refinement of a safety requirement. In later phases, results of verification and validation activities will be added in order to proof the fulfillment of the single requirements.

To this end, safety concepts can be represented as modular requirement trees. Comparable to a fault tree, such safety concept trees (SCT) [65], support Boolean operators, namely *or*- and *and*-operators, for a more precise definition of the refinement of requirements. While an *or*-operator expresses that one of the sub requirements is sufficient to fulfill the superordinate requirements (i.e. a redundancy is expressed), an *and*-operator expresses that all sub requirements must be fulfilled for fulfilling the superordinate requirement.

We use this idea to define safety requirement modules as it is illustrated in Figure 4-15. A *RequirementsModule* consists of a set of *SafetyRequirements*, *InterfaceElements*, and a *Rationale*. The latter is important for the safety case and documents the principle strategy of the safety concept.

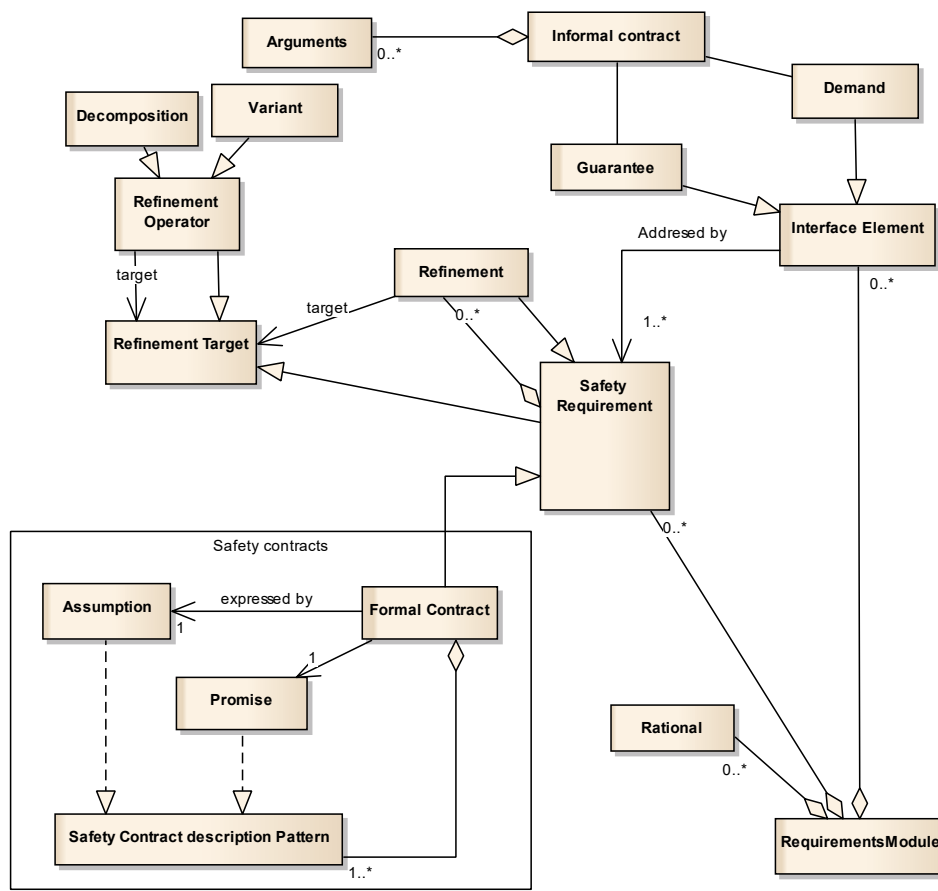


Figure 4-15 Meta-Model of modular safety requirements

The interface specification follows a contract-based approach. To this end, the interface consists of *Guarantees* and *Demands*. Guarantees and demands describe informal or semi-formal contracts, which are usually defined as textual requirements with some additional *Arguments*, which refine the requirements. For example, if the requirements demand a timely signal delivery, an argument can be used to specify acceptable delay times. Alternatively, the interfaces as well as any other safety requirements can be defined as formal contracts [80].

In order to refine safety requirements, it is necessary to use *RefinementOperators*. In case of a *Decomposition*, all sub requirements must be fulfilled in order to fulfill the superordinate requirement. This operator represents the *and*-operator of safety concept trees. In case of a *Variant*, one of the sub requirements is sufficient to fulfill the superordinate requirement. Thus this operator represents the *or*-operator of safety concept trees.

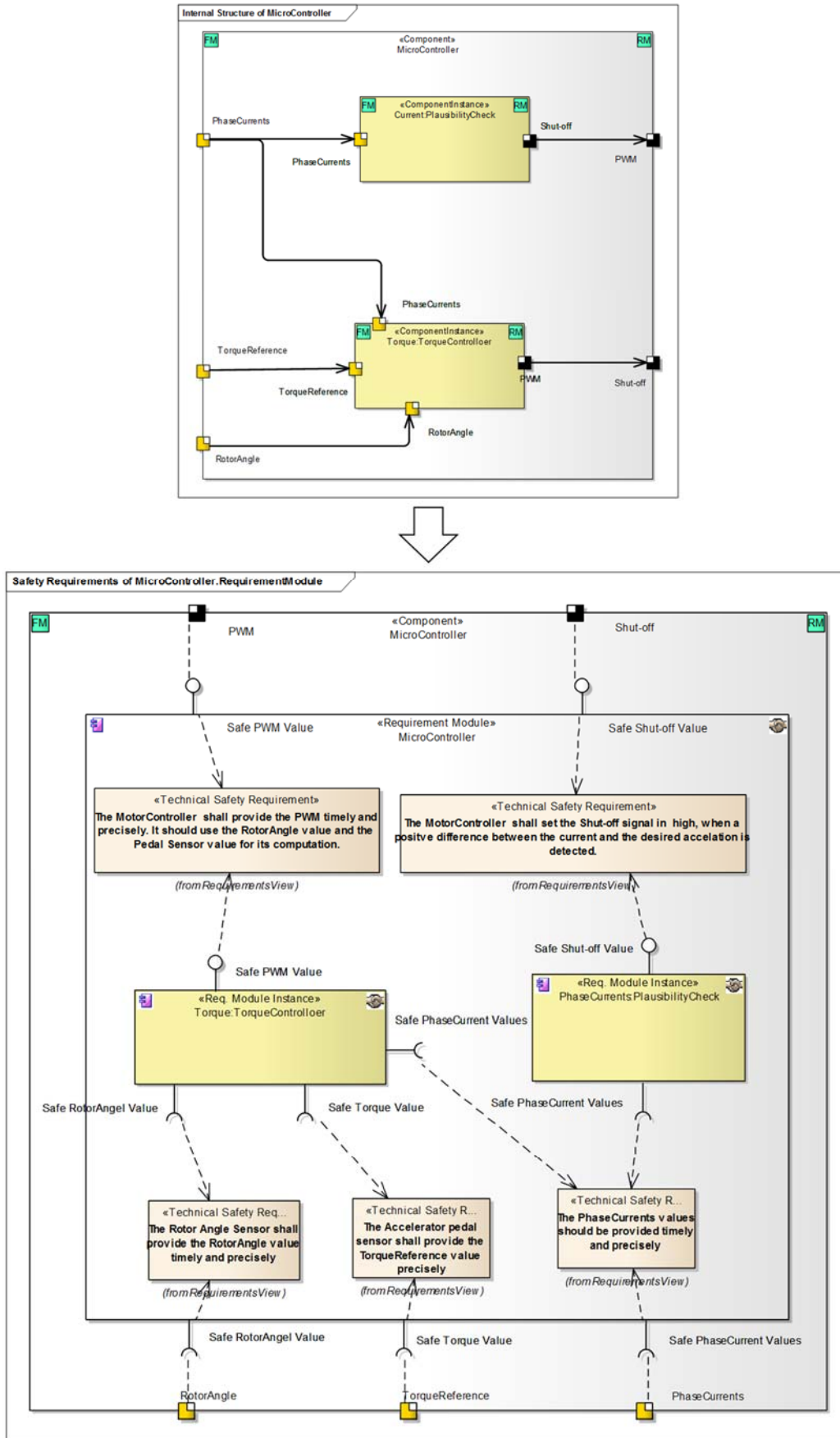


Figure 4-16 Example of a modular safety requirements model

An example is shown in Figure 4-16. The upper part shows an excerpt of the architecture of a control system of an electric drivetrain. The lower part shows the associated safety requirements model. The requirements model contains two sub modules following the architecture. Using guarantees and demands, the safety interfaces of the sub components are defined. By this means, the requirements model follows the structure of the architecture and if a component is reused, its requirements can be reused as well. Considering its guarantees and demands, it can be checked, whether or not it fits into its new usage context or which modifications are necessary to use it.

5. Runtime Safety Assurance

Regarding the concepts of model-driven, modular safety assurance, a safety expert assesses the integrated system. If we want to apply such concepts to open systems of systems, however, there will be no human expert to check the system's safety. Rather, the system must assure its safety autonomously. This leads to a series of new challenges as they were introduced in Chapter 1. Extrapolating the current developments of safety engineering, it would take much too long until urgently required safety assurance approaches for open systems of systems would be available. In the same way as open systems of systems form a new paradigm in system development, there needs to be a change of paradigms in safety assurance, as well.

And in the same way, as the systems become more and more intelligent for realizing future functionalities, it is necessary to increase their *safety intelligence*, as well. Instead of using rather simple fault tolerance mechanisms, the systems must become self-aware with respect to their safety properties. By this means, they are enabled to reason about their safety in a concrete context given at runtime, e.g., if they connect to other 'foreign' systems, or if they are used in environmental situations that have not been anticipated during development.

During the last decade, Models@Runtime [81] have emerged as a new paradigm for maturing the systems' reflection capabilities to self-awareness. In fact, Models@Runtime are an evolution of model-driven development. And even though they enable very flexible runtime mechanisms, using models at runtime leads to a sufficient degree of formalism and determinism. Models@Runtime provide a deterministic reflection of the systems' runtime quality since the models are not unpredictably modified during runtime. In contrast to approaches like artificial intelligence, it is therefore possible to predict and to reproduce how the systems behave.

Applying the idea of Models@Runtime to safety assurance means to make existing safety models like safety cases available at runtime [82]. Considering the idea of model-driven safety assurance as it was introduced in the previous chapter, there is an ideal basis available to create safety models at runtime. To this end, the models need to be available in a machine interpretable form so that they can be evaluated based on runtime information automatically. Additionally, an algorithm is required enabling the system to interpret the results and to react appropriately. Particularly the latter aspect constraints the applicability, since safety assurance usually requires cognitive interpretations of the results by human beings. Therefore, section 5.1 gives an overview on established safety models with respect to their applicability for runtime evaluation, before the subsequent chapters illustrate how the models can be used at runtime. Section 5.7 will then evaluate the different approaches before Section 5.8 derives a runtime safety assurance framework based on the idea of Safety Models@Runtime (SM@RT).

5.1. Overview on safety models for runtime evaluation

Considering the idea of model-driven safety assurance as it was introduced in the previous chapter, all safety artifacts that are created during a safety lifecycle are available as models, which could serve as a basis for safety models at runtime. Figure 5-1 gives an overview on the models created during a safety assurance lifecycle.

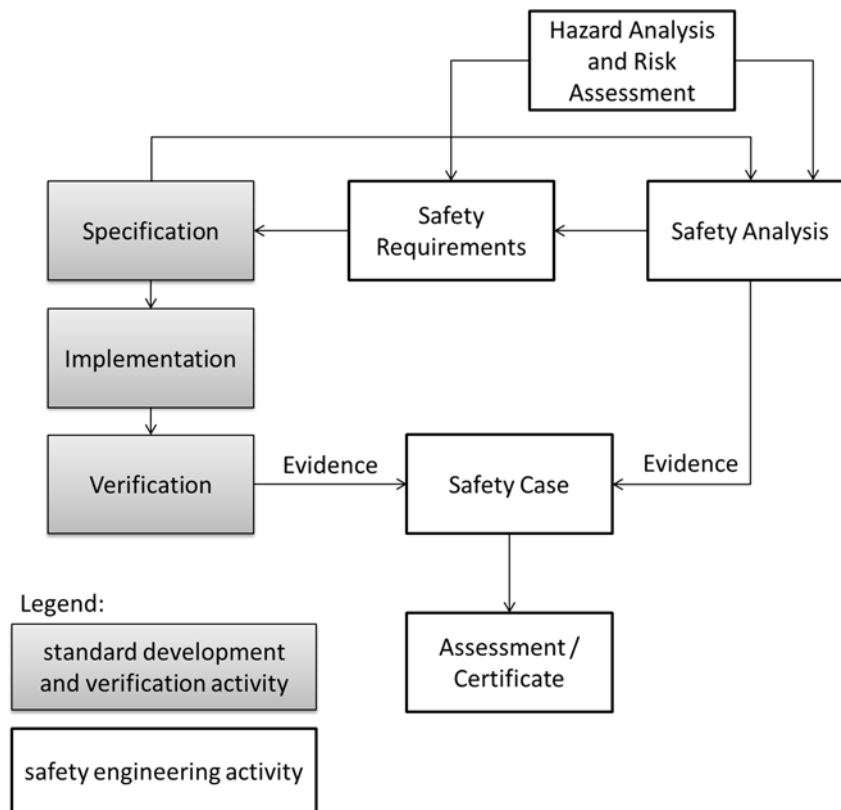


Figure 5-1 Overview on safety models in a safety assurance lifecycle

Any safety assurance starts with a hazard analysis and risk assessment, also leading to the top level safety requirements. Using safety analyses, the safety requirements are refined. Using the analysis results as well as results of the verification and validation activities as evidences, the safety requirements are evolved to a safety case. Finally, the system's safety is assessed based on the safety case. And a 'certificate'² is issued if the assessment comes to a positive result, which includes potential limitations and constraints that must be considered for safely using the system.

As regards the *hazard analysis and risk assessment*, models are also available for this safety artifact. However, deriving and particularly assessing hazards and the associated risks is a creative process and requires a lot of human intelligence. Nonetheless, if, for

² The term certificate often refers to a formal process including official certification bodies so that it is avoided in some domains (e.g., in the automotive domain). But nonetheless some kind of certificate is required as final document confirming the system's safety.

example, system requirements were to be changed at runtime, it would be necessary to update the hazard analysis and risk assessment. In consequence, this also means that potentially the complete subsequent safety assurance process has to be updated, as well. But though providing hazard analyses at runtime is a challenging task, they will be required for supporting more flexible system adaptations at runtime.

Safety analyses at runtime can be very beneficial since any change of the structure or the current state of the system could be reflected in runtime analysis models and the analysis can be dynamically updated at runtime. However, it must be considered that particularly systematic faults are regarded in a qualitative way, only. In contrast to a quantitative evaluation, it is therefore potentially non-trivial for a system to interpret the analysis results at runtime.

As regards *safety requirements*, they evolve to a safety case by adding evidences that a complete set of requirements has been derived and that the requirements have been fulfilled. From a safety point of view, it is therefore more reasonable to directly consider *safety cases* instead of safety requirements at runtime.

Finally, *safety certificates* are the final document stating the systems' safety including potential constraints and limitation. Particularly regarding modular certificates, they are very promising to be used at runtime since they summarize the results of the creative process of safety assurance, so that they can be more easily interpreted at runtime. On the other hand, they are usually not available in a sufficiently formal format and they only support a very restricted flexibility. Nonetheless, having certificates at runtime is the most promising first step towards runtime safety assurance, since it leaves the major responsibility of safety assurance with human experts at development time.

In general, providing safety models at runtime becomes more challenging the earlier in the safety lifecycle they are created. Therefore, the subsequent sections elaborate on the different safety models at runtime in reverse order of their creation in the safety lifecycle.

5.2. SafetyCertificates@Runtime

Safety certificates contain all information that is necessary to identify which safety requirements are fulfilled at which integrity level and which constraints have to be considered to use the system safely. The same information must be available in a machine processable format in SafetyCertificates@Runtime. Just like conventional safety certificates, SafetyCertificates@Runtime do not contain any white-box information on how the system was realized to yield the certification. They are particularly interesting for the safe integration of systems of systems at runtime, since SafetyCertificates@Runtime can be used to retrieve the safety requirements fulfilled by the single systems as well as the constraints that must be fulfilled to safely integrate the systems. This includes possibly required checks that must be executed at runtime to verify the integrated system.

So summarizing *SafetyCertificates@Runtime* are modular certificates that can be interpreted, composed, and adapted at runtime. They are dynamically adapted to represent the safety state of the system at runtime. The certificates of subsystems can be composed at runtime in order to yield an overall safety approval for a given composition.

The idea of how *SafetyCertificates@Runtime* can be used for a dynamic composition of systems of systems is shown in Figure 5-2. The individual subsystems provide a runtime representation of the modular certificates (*SafetyCertificate@Runtime*). In order to assess the safety of the resulting system of systems, the single certificates have to be composed.

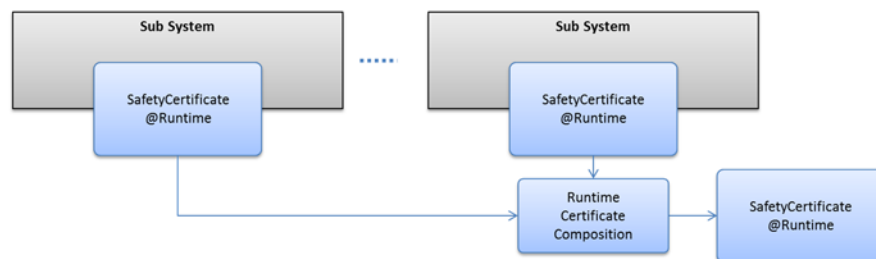


Figure 5-2 *SafetyCertificates@Runtime* enable dynamic system composition.

In order to yield such a *SafetyCertificate@Runtime*, the process is very similar to modular certification. After the subsystem has been developed, it must undergo a manual certification process at design time. Usually, however, the safety assurance of a single subsystem at design time can only yield a conditional certificate, since the certification is based on various assumptions. These assumptions might be concrete demands on other subsystems. For example, there might be a demand that the failure modes of a received signal must be mitigated by another subsystem according to a specific safety integrity level. Other assumptions might consider the integration context in general, such as the maximal number of collaborating subsystems, the type and quality of the communication system used, etc. Consequently, *SafetyCertificates@Runtime* often follow the idea of safety contracts defining a set of safety guarantees provided by the subsystem and a set of safety demands the subsystems require to be fulfilled by the integration context. This means that they provide runtime information on which safety properties can be guaranteed by the system under the precondition that the defined demands are fulfilled. At runtime, the fulfillment of the demands is checked and the resulting guarantees are derived. Usually, however, safety is not a completely modular property, i.e., the composition of safe components does not necessarily lead to a safe composition, even though the safety demands are fulfilled. Therefore, it is often necessary to perform additional checks in the integration context at runtime.

When subsystems are composed at runtime, it is possible to compose the *Certificates@Runtime*, as well. To this end, the conditions defined in the runtime certificates must be checked. In the simplest approach, a system of systems is considered safe if all preconditions of all conditional certificates are true. Otherwise, the system of

system must not be used. In most cases, however, the certificates of the single subsystems are not harmonized with each other. So it is very unlikely that there will be any safe match, at all. In fact, such an approach is only reasonable if it is possible to adapt the `SafetyCertificate@Runtime` to the current integration context.

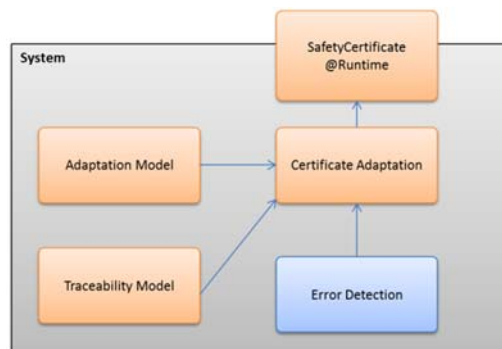


Figure 5-3 Runtime adaptation of certificates provides more flexibility

Actually, the possibilities of how a runtime certificate could adapt are as versatile as adaptation in general. There could be different pre-defined variants that are dynamically selected in a given context. Or there could be more sophisticated and flexible adaptations of the certificate. From a safety point of view, however, predictable, traceable, or even provable solutions are more likely to be accepted. Some ideas are illustrated in Figure 5-3. If we assume, for example, that there is an adaptation model defining how the system adapts in certain situations, this information could also be used to adapt the certificate. Alternatively or additionally, a traceability model could be used to identify those elements that are affected by system adaptations and to derive necessary certificate adaptations. An efficient impact analysis is of utmost importance in traditional safety engineering in order to identify necessary changes and thus reduce the revalidation effort. As long as the effects of adaptations can be traced back to anticipated classes of system changes, even complex system adaptations could be handled by simple variants in the `SafetyCertificate@Runtime`.

As a further extension, it could be possible to use error detection mechanisms to adjust the runtime certificates using up-to-date runtime error information.

Usually, the adaptation goal for the certificates is to provide the best possible guarantees in the given context. The context, in turn, is usually given by the set of externally fulfilled safety demands and the internal state of the system. In the simplest form, they simply provide different alternatives how the safety guarantees can be achieved, i.e. depending on the amount and quality of fulfilled demands, they will provide a different amount and quality of guarantees. We used this basic idea for Conditional Safety Certificates – ConSerts [83][84][85], which have already proven to be very valuable for the safety assurance of open systems. Therefore, ConSerts will be explained as one concrete approach in more detail in Section 5.9.

5.3. SafetyCases@Runtime

The more adaptive a system is, the more difficult it is to consider all the different adaptations in a SafetyCertificate@Runtime. This particularly increases the effort at design time since the complete adaptation space must be considered in the certification process. Alternatively, it could therefore be another option to provide SafetyCases@Runtime. Safety cases are a direct input to certification. In contrast to certificates, however, they include the complete argument why a system is considered safe. A good safety case model includes a complete breakdown of top-level safety goals to the detailed requirements realized in the system. And it particularly includes the evidence proving that the arguments used are sound and that the requirements have been fulfilled.

SafetyCases@Runtime therefore provide more information at runtime and enable more flexible system adaptations. In consequence, however, they are more complex to handle, since there is no pre-certification at design time and all the steps from a safety case to certification have to be shifted to runtime as well. As a further consequence, this will most likely reduce the acceptance of such an approach compared to SafetyCertificates@Runtime.

Summarizing these ideas, *a SafetyCase@Runtime is a formalized, modular safety case that can be interpreted and adapted at runtime. Based on the interpretation, it can be dynamically checked to which extent the safety goals of subsystems are met. Using adaptations, the line of argumentation can be adjusted to system adaptations. In addition, the revalidation of evidences at runtime must be supported in the case that system adaptations lead to the invalidation of evidences.*

As shown in Figure 5-4, SafetyCases@Runtime extend the idea of SafetyCertificates@Runtime. Instead of explicitly defining the adaptation of the certificates, it is possible to describe the adaptation of the safety cases and use the SafetyCases@Runtime to adapt the safety certificates automatically.

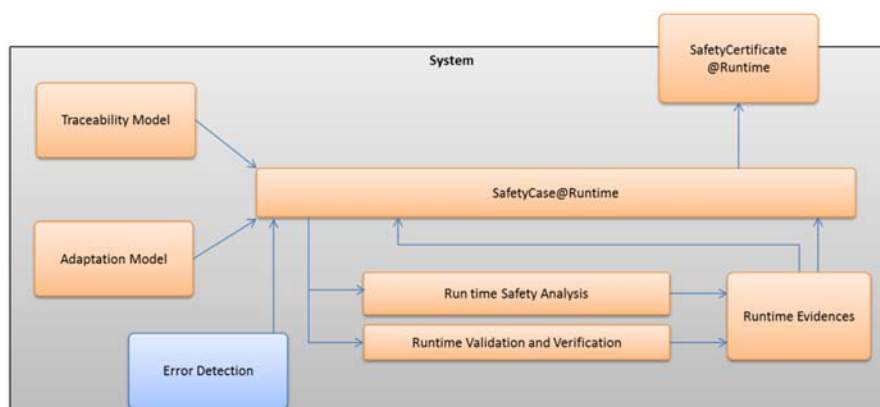


Figure 5-4 Conceptual model of how safety cases could be used at runtime

A certificate certifies that certain safety guarantees are fulfilled. The safety case models the argument why these guarantees are fulfilled. If a safety case is adapted at runtime, the

resulting argumentation should enable the system to conclude autonomously which safety guarantees can still be provided at which integrity level.

By shifting safety cases to runtime, it is possible to adapt (1) the argumentation and/or (2) the evidences to the currently given context. As regards the adaptation of the argumentation, a very straightforward solution would be to include different variants of the argumentation. In more complex versions, more intelligence might be integrated that is able to derive new lines of argumentation.

As regards the evidences, which are, for example, verification and validation results or the results of safety analyses, it is necessary to attach constraints to the evidences used in the safety case. At runtime, it is then possible to evaluate whether or not these constraints are still fulfilled. If not, there are basically two combinable options. First, it is possible to find an alternative argumentation based on the remaining valid evidences – including argumentations that potentially require a reduction of the safety guarantees that can be provided in the given context. A second option would be the revalidation of evidences. This requires the capability to re-perform safety analyses and/or validation and verification activities at runtime. For `SafetyCases@Runtime`, let us assume that this revalidation is limited to repeating the checks defined at design time in order to provide the evidence. This presumes that the system adaptation does not lead to a change of requirements or a change of the system's interface.

If the respective pass-criteria are met, the newly created evidence can replace the invalidated original evidence and can be integrated into a new argumentation. Otherwise, the evidence remains invalid and the system must either find an alternative line of argumentation or invalidate the affected safety goals.

Conventional safety cases are often still compiled in simple text documents or tables instead of (semi-) formal models. Though approaches like the goal structuring notation (GSN) [49] provide a standardized notation for safety cases, which has been realized as part of a model-driven safety assurance lifecycle, the actual semantics of the argumentation still lies in informal text, which can hardly be interpreted at runtime.

Approaches like VersaI [66], which we proposed for the automated modular certification at design time provide a formal basis for runtime safety cases, since they are based on formalized requirements type systems rather than text. This approach, however, is focused on hardware-software-integration with a dedicated type system for the resulting specific types of requirements. Combined with more generic safety requirements type systems as they are also used for ConSerts [85] (cf. Section 5.9), the approach could nonetheless provide a sound basis for safety cases at runtime.

For the time being, however, we limit the concept of `SafetyCases@Runtime` to modeling different variants of lines of argumentation. To this end, we use model driven safety concepts, as they have been defined in Section 4.4.2 as starting point. These safety concepts already define gates for refining the requirements, namely *decomposition* and *variant* gates. The latter already allows defining different variants. Usually these variants,

however, are used to model redundancy. Therefore, we additionally add an *alternative* gate, which can be resolved at runtime. In order to achieve an efficient runtime representation, all argumentation lines between two alternative gates are considered as *argument block*. And all evidences that are assigned to the requirements of one block are collected in an *evidence set*. For all of these evidences, the safety engineer has to decide whether and how these can be checked at runtime. For the time being, this requires the definition of dedicated runtime executable checks.

At runtime, it is then possible, to check the demands, which are the sources of each modular safety case at runtime in the same way as it is done for *SafetyCertificates@Runtime*. And in principle, the variants can be checked in the same way, as well. Additionally, however, it is possible to not only consider predefined certification variants. But it is possible to flexibly adapt the system at runtime. After the adaptation, all evidences are revalidated based on the new system structure and behavior, e.g. by running runtime tests which have been defined by the safety engineers. As long as the evidences are still valid, the safety case is still considered to be valid and the system adaptation is considered safe. Else, the system adaptation has to be rolled back to a previous, safe state.

Obviously, this is only a slight extension of safety certificates and safety cases at runtime provide much more potential. Therefore, further extensions are subject of ongoing research.

5.4. V&V-Models@Runtime

SafetyCases@Runtime already provide a very flexible means for safety assurance at runtime. Some system adaptations, however, might require a new set of verification and validation checks to provide the evidence required for the argument. Moreover, it might be desirable to be able to remove the faults identified during runtime V&V instead of being limited to only checking the pass-criteria.

For the former aspect, it is necessary to additionally enable the system to define verification and validation suites autonomously. Realizing the latter aspect even requires systems that are able to localize the causing faults, and to isolate or even remove them. Considering how difficult this step easily becomes for developers at design time, it is obviously a very challenging task to shift these activities to runtime.

In consequence, a realization of *V&V-Models@Runtime* means that all models that are necessary to perform validation and verification activities (e.g., test cases, pass/fail-criteria etc.) can be interpreted and adapted at runtime in order to create new evidences after system adaptations.

As regards the current state of the art, runtime validation and verification are typically applied in a complementary way together with corresponding development-time activities. On the one hand, there are runtime verification techniques that utilize runtime monitoring to record software execution traces that can then be analyzed [86]. On the

other hand, there are approaches that employ quantitative model checking at runtime as an assurance technique in the context of adaptive systems (e.g., [87], [88], and [89]). In [90], Goldsby et al. present AMOEBA-RT, a run-time monitoring and verification technique that provides assurance (based on dynamic model checking) that ensures that dynamically adaptive software satisfies its requirements. Calinescu and Grunske introduced the QoS MOS (QoS Management and Optimization of Service-based systems) framework for the development of adaptive, service-based systems that are able to manage their QoS adaptively and predictably [91]. QoS MOS utilizes probabilistic model checking at runtime to evaluate whether or not the system satisfies the given QoS requirements. In the traditional development-time versions of these kinds of approaches, the analysis of temporal-logic properties (including probabilities, costs, and rewards) is commonly used to assess relevant non-functional properties of a system. At runtime, such analyses can be performed on a model base that is continually updated as the underlying system evolves. In general, this introduction of runtime measures is particularly promising since traditional development-time techniques do not scale sufficiently well. Moreover, at runtime, detected issues can be addressed directly with adequate adaptations (i.e., counter measures). A short related survey (which is not limited to V&V) considering runtime assurance techniques for adaptive systems has been published by Calinescu [92].

From a safety assurance point of view, however, validation and verification approaches at development time must be qualified. This means that it is necessary to show that, for example, the test tools provide correct results. In the same way, it would be necessary to qualify runtime V&V-approaches in order to create the required confidence to trust in the runtime analysis results. As of today, this is sometimes more difficult than proving the safety of the actual system. Therefore, V&V-Models@Runtime are an interesting research topic for future applications, but they are still far away from a serious practical application.

5.5. SafetyAnalysis@Runtime

A further very important safety artifact is safety analyses. In fact, safety analysis models already have a formal format and can be analyzed automatically. Therefore, they can be easily made available at runtime. By this means, safety analyses can be repeated after any system adaptation or if the composition of a system of system changes. Particularly for software-intensive systems, however, safety analyses are not quantitative but qualitative so that the evaluation of the results usually requires an expert's interpretation.

So a SafetyAnalysis@Runtime means that safety analysis models are made available at runtime and that the system is enabled to evaluate the models using typical analyses like quantitative evaluation or minimal cut set calculations. Moreover, the system needs to be enabled to interpret the analysis results in order to conclude on the system's safety. The safety analysis models are adapted according to system adaptations in order to re-analyze the system's safety after it has adapted itself.

In order to realize the adaptation of safety models, there are various possible ways. Considering modular safety analysis models as they have been described in the previous chapter, exchanging, removing, or adding a component or a system at runtime only requires to exchange the corresponding safety analysis component, as well, in order to re-analyze the system. If single components adapt, the corresponding adaptation of the safety analysis models becomes more difficult. If a system only reconfigures itself, then the possible configurations can be modularly analyzed at design time and the corresponding safety analysis models can be exchanged at runtime. Therefore, we developed an approach that enables the application of safety analyses to reconfigurable systems [93].

If the system adapts in a more flexible way by actually modifying its behavior in previously unknown ways, it will be necessary to modify the safety analysis models, as well. However, creating safety analysis models is a very creative process which is based upon moderated meetings with a group of experts. Current approaches to automatically create safety analysis models, e.g. based on model checking, are still not sufficiently mature – even for design time analyses.

The main application scenario for safety analysis at runtime is therefore a change of a composition of a system of systems. Using modular safety analyses, the emerging system of system can be analyzed automatically. Even though it is difficult to provide quantitative analysis results, it is often sufficient to perform qualitative analyses based on minimal cut sets. To this end, we further extended the semantics of model-driven fault-trees, as they have been introduced in the previous section. Most importantly, it is necessary to provide different types of basic events. Namely, it is reasonable to distinguish between primary faults and a loss of a counter measure. By this means, it is then possible to analyze the resulting cut sets for identifying single point failures, or to ensure that all possible fault combinations leading to a hazard are covered by counter measures.

As a further aspect, it is necessary to type failure modes as well as counter measures. Then, it is possible to specify valid tuples $\langle failure\ type, measure\ type \rangle$ for specifying valid counter measures for the different failure types. For example, timing failures could be covered by a watchdog as counter measure, whereas a wrong data value cannot be detected by a watchdog timer. By this means, it is not only possible to check whether or not all failure modes are addressed by a counter measure, but it is additionally possible to check the appropriateness of the counter measure in place. This requires quite detailed type definitions for enabling valid evaluations at runtime. On the other hand, the more detailed the types are defined, the lower is the supported flexibility of system adaptations. Therefore, it is always necessary to find an appropriate trade-off between the rigor of safety assurance on the one hand and flexibility on the other hand.

5.6. Hazard Analysis and Risk Assessment@Runtime (HRA@Runtime)

In the previous alternatives, we assumed that the requirements and the resulting safety goals are not adapted. As a consequence, it has ‘only’ been necessary to adapt the

argumentation that the safety goals are still met in spite of system adaptations based on the safety case, safety analyses, and the evidences created at runtime. Some adaptation approaches, however, also consider a change of requirements at runtime. If we apply the safety lifecycle to the idea of Models@Runtime, this means that we require a hazard and risk analysis at runtime, i.e. that the system must adapt and extend the hazard and risk analysis and potentially has to adapt and to extend the set of safety goals. By doing so, the complete existing argumentation for a changed safety goal might be invalidated. For new safety goals, an argumentation is completely missing. On the one hand, this type of runtime assurance certainly provides the highest possible flexibility. On the other hand, however, it requires very intelligent mechanisms for defining a safety argumentation and generating the necessary evidence autonomously at runtime.

A HRA@Runtime implies that a hazard and risk analysis model can be interpreted and adapted at runtime. This includes the identification of new hazards and the reassessment of existing hazards after adaptations at the requirement level.

So far, there are no significant research results available for supporting hazard analyses at runtime. One approach including the aspect of hazard analyses has been proposed by Priesterjahn et al. in [94]. The main idea of this approach is to ensure the safety of adaptive systems during runtime by checking whether a reconfiguration is allowed based on associated hazard probabilities and potential damage that would be imminent after the reconfiguration. To this end, adapted hazard and risk analysis techniques are applied during runtime. In this case, however, the hazard and risk analysis and models at runtime are used to monitor the system, but they are not adapted in order to reflect adapted system requirements.

Hazard analyses and risk assessment are usually based on spreadsheets. Accepted model-based, more formal representations are not available at this point in time. However, first approaches integrate hazard analyses and risk assessments into the model-driven design process, as we have done it, for example, as part of the model-driven safety tool iSafe [64]. A major challenge, however, lies in the runtime interpretation of hazard analyses and risk assessments. In order to further formalize hazard analyses, we introduced the SAHARA [95] approach, which provides a formalization based on ontologies. Though this does still not provide a sufficient support for runtime evaluations, it provides a first step towards HRA@Runtime.

5.7. Evaluation of the different approaches

Regarding the approaches described above, they obviously build upon each other. This means that a HRA@Runtime requires V&V-Models@Runtime and SafetyAnalyses@Runtime, which in turn require SafetyCases@Runtime and so on. So it is necessary to decide to which extent we want to shift parts of the safety lifecycle to runtime. This results in a trade-off decision. From a safety point of view, it is certainly preferable to leave as much responsibility as possible with a human expert. Consequently, it would be reasonable to have only SafetyCertificates@Runtime. From an adaptation

point of view, however, it is preferable to have as much flexibility as possible in order to tap the full potential of dynamic adaptation. In consequence, this would require shifting elements of the complete safety lifecycle to runtime.

In order to further illustrate this trade-off, Figure 5-5 shows the relations of the different approaches to their acceptance on the one hand and to their flexibility on the other hand. Acceptance in this case refers to the probability of acceptance by safety authorities and legislation. Since there is no practical experience available, this is a qualitative estimation. First, we assume that acceptance is inversely proportional to the responsibility and intelligence given to the system. Second, the acceptance of an approach is usually inversely proportional to its complexity. Or vice versa: The simpler an approach can be realized, the more probable is its acceptance. For obvious reasons, it is very probable that the required intelligence as well as the resulting complexity will grow with the number of safety assurance steps that are shifted to runtime. Consequently, in our opinion, SafetyCertificates@Runtime have the best chances of being accepted, whereas the acceptance of a HRA@Runtime (i.e., shifting all safety assurance activities to runtime) is quite improbable. As a further aspect, acceptance will be higher if the Safety-Models@Runtime are reconfigured at runtime to predefined variants only, whereas acceptance will rapidly decrease if the safety models themselves are adapted more flexibly at runtime.

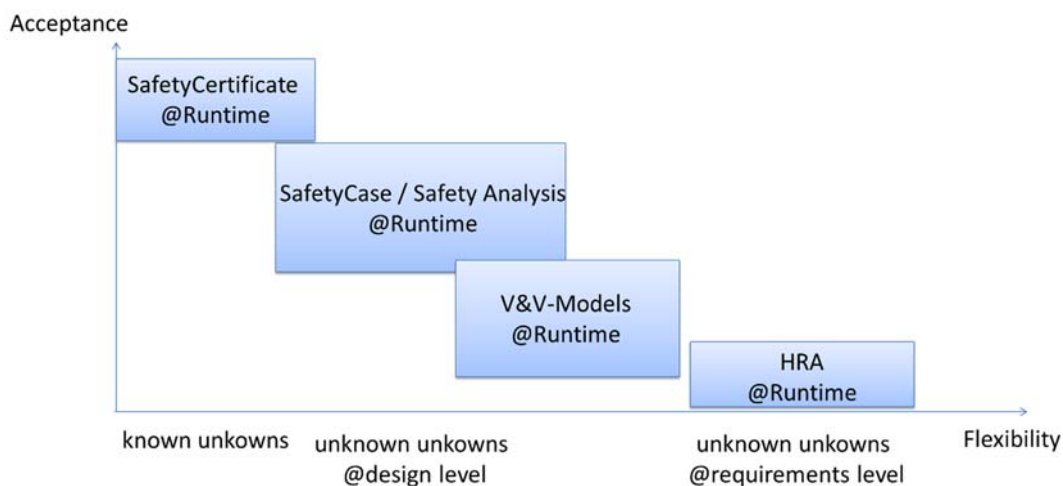


Figure 5-5 Qualitative relations between acceptance and flexibility

Flexibility, on the other hand, represents the degree of which different types of adaptations are supported. More precisely, in this case we refer to the type of adaptation used to adapt the system itself and not to the type of adaptation used to adapt the safety models, since different adaptation approaches might be used for the system itself on the one hand and the safety models on the other hand. In order to classify the supported flexibility of system adaptations, we differentiate between three basic classes. We first differentiate between ‘known unknowns’ and ‘unknown unknowns’. In the former case,

we assume that the system can only adapt to a runtime context that has been anticipated at design time. In the latter case, we assume that the system needs to flexibly adapt to situations not anticipated at design time. In consequence, the system structure or behavior is hard or even impossible to predict. We have further subdivided the ‘unknown unknowns’ into adaptations at the design level on the one hand and at the requirements level on the other hand. In the former case, we assume that the requirements can remain unchanged and an adaptation of the realization (e.g., at the architecture level) is sufficient to adapt to the context given. In the latter case, the adaptation also includes the adaptation of existing and/or the definition of new requirements.

`SafetyCertificates@Runtime` can only be used to address ‘known unknowns’ since an adaptation of certificates to an unpredicted context is not possible without considering the underlying safety case, which forms the indispensable basis for a sound argumentation of a certificate’s validity. But even for ‘known unknowns’ the configuration space might be too large to be covered completely by variants at the certificate level. Therefore, it might be reasonable to use `SafetyCases@Runtime` already to efficiently support ‘known unknowns’.

If we consider ‘unknown unknowns’ at the design level, this means especially that the requirements and thus the safety goals remain unchanged. Depending on the degree of system modifications required for the adaptation, `SafetyCases@Runtime` or `V&V-Models@Runtime` and `SafetyAnalyses@Runtime` are therefore sufficient. While `SafetyCases@Runtime` are limited to running predefined validation and verification activities at runtime, `V&V-Models@Runtime` as well as `SafetyAnalyses@Runtime` additionally support the modification of V&V models, e.g., the modification of test cases or pass/fail criteria, as well as a re-analysis of the system’s safety at runtime. The more flexible the system adaptations must be, the more likely it is that `V&V-Models@Runtime` as well as `SafetyAnalyses@Runtime` approaches will be required in addition to `SafetyCases@Runtime`.

As soon as the adaptation to ‘unknown unknowns’ also requires an adaptation of requirements, it is additionally necessary to adapt the hazard and risk analysis and the resulting safety goals at runtime. As described above, it is not sufficient to identify new hazards or to re-assess the associated risk at runtime. In fact, the system must be able to appropriately create or adapt all affected artifacts along the complete safety lifecycle.

5.8. Conceptual safety assurance framework for Open Adaptive Systems

`Models@Runtime` obviously provide a wide range of possible approaches for the safety assurance of open systems of systems and it is certainly not possible to pick out one particular approach that leads to the best trade-off between flexibility and acceptance in general. In fact, we believe that it will be necessary to integrate different approaches into an assurance framework in order to use the advantages and compensate for the disadvantages of the different approaches.

Therefore, we defined the runtime safety assurance framework shown in Figure 5-6. Learning from traditional safety engineering, we use modularity as the basic ingredient for a safety assurance framework from the very beginning. First, this obviously reduces complexity. Second, this enables us to use different assurance approaches for different modules. In this context, we use the term module very flexibly to express a modularized entity that can range from a complete system in a system of systems to a single software component. Since the required types of adaptation usually differ widely across the different modules, it is reasonable to limit more complex assurance approaches to those modules that actually have to adapt very flexibly.

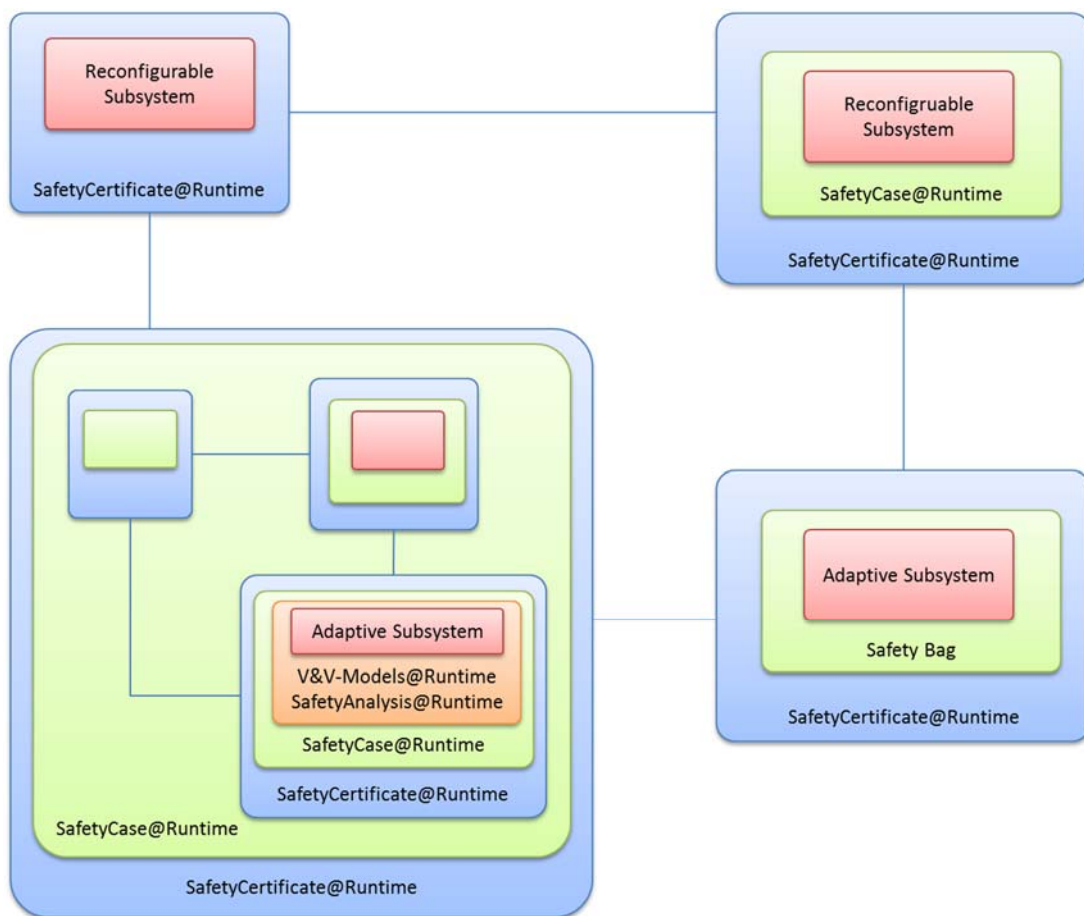


Figure 5-6 Integrated Runtime Safety Assurance Framework

Following the idea of modular certification, we use *SafetyCertificates@Runtime* as the basic building blocks enabling the modularization and runtime integration of different subsystems. In this case, *SafetyCertificates@Runtime* are the common denominator enabling the combination of a wide range of different assurance approaches used for the single modules. As long as each system in a system of systems provides a *SafetyCertificate@Runtime*, the system of system can be safely integrated. And most importantly, it is not necessary to know anything about how the internal safety assurance is realized within the single systems. This means that *SafetyCertificates@Runtime*

provide the generic interface of the runtime safety assurance framework enabling information hiding in the safety assurance process. This is a key for the scalability and the flexibility of the concept.

Assume, for example, that we have a module that reconfigures to ‘known unknowns’ only, as shown in the upper left corner of Figure 5-6. Then it might be sufficient to perform the major safety assurance activities at development time and limit the runtime models to `SafetyCertificates@Runtime`, only. If we have a module that has too large a configuration space or that also adapts to ‘unknown unknowns’, it might be necessary to have `SafetyCases@Runtime` as well, as shown in the upper right corner of Figure 5-6. As described above, `SafetyCases@Runtime` are an extension of `SafetyCertificates@Runtime`, so that a runtime certificate is still available at the module’s interface, facilitating the safe integration of the components. In some cases, a module might adapt so flexibly that we will need `V&V-Models@Runtime`, `SafetyAnalyses@Runtime` or even a `HRA@Runtime`. However, realizing this is very complex, so it is reasonable to keep the complexity of such modules very small. To this end, it is helpful that the modularization of the framework can be applied recursively to achieve hierarchical decomposition, as illustrated in the lower left corner of Figure 5-6. This decomposition additionally illustrates an alternative way of composing `SafetyCertificates@Runtime`. Particularly regarding systems of systems, each providing a `SafetyCertificate@Runtime`, the single systems are usually sufficiently independent from each other that composition at the certificate level is likely to be sufficient. If we assume the runtime integration of different software modules running on the same platform, however, there are usually tight interdependencies. Merely the fact that they share the same resources, for example, creates a safety-relevant dependency. For this reason, it is likely that additional evidences will be required for proving that the integration of the single modules is safe as well. Therefore, it might also be reasonable to have `SafetyCases@Runtime` at the integration level.

The acceptance of sophisticated assurance approaches, in particular, is very low. An alternative way to ensure the safety of highly adaptive systems is given by different traditional approaches, particularly in the field of fault tolerance. So-called safety bags (cf. e.g., IEC 61508 [2]), for example, are a typical concept for monitoring a function to detect anomalies and trigger counter-reactions. Assuming that it would be possible to define a safety bag that can detect and handle any safety-related failure of an adaptive module, it would not be necessary to provide further assurance of that module. Though such approaches are based on traditional mechanisms rather than `Models@Runtime`, they would nonetheless fit into the framework as shown in the lower right corner of Figure 5-6. In general, any type of assurance approach can be easily integrated into the framework, as long as it is possible to provide a valid `SafetyCertificate@Runtime`.

Summarizing, we defined this framework based on a prognostic evolution of state-of-the-practice safety engineering lifecycles using the idea of `Models@Runtime` as a catalyst for building a conceptual bridge between the world of safety engineering on the one hand

and Models@Runtime on the other hand. Being based on safety engineering principles makes acceptance of the approach more likely. Yet it provides sufficient flexibility to integrate various different solution approaches based on Models@Runtime.

5.9. Conditional Safety Certificates – ConSerts

Shifting a kind of safety certificate into run time is obviously a promising first step towards safety assurance of open systems of systems. As a concrete solution approach, we have developed the concept of conditional safety certificates (ConSerts) [85]. ConSerts are predefined modular safety certificates of the single systems (i.e., devices) that are to be integrated at runtime. Following the ideas of modular certification, they use the idea of contracts by defining *safety guarantees* on the one hand that are fulfilled by a component under certain preconditions, and *safety demands* on the other hand, i.e., safety requirements that must be fulfilled by the integration context. In addition, they define *invariants* expressing assumptions made during the safety assurance of the component, which must be fulfilled by the integration context at runtime.

Obviously, ConSerts follow basic principles that have proven to be very valuable for the modular certification at design time. There are, however, two important differences between ConSerts and standard modular certificates:

- (1) A conventional modular certificate certifies that a fixed set of safety guarantees is fulfilled by a system - given that a fixed set of invariants as well as safety demands is fulfilled by the integration context. This is possible since the systems or components are developed independently, but the development nonetheless follows the rules and requirements of a central integrator. In open systems of systems, however, we find heterarchical systems, which –as opposed to hierarchical systems – do not have a central root node with an integrator who takes responsibility for the integrated system. But all systems are independent from each other and have the same rights. Therefore, it is very unlikely that two systems could be integrated based on a statically and independently defined set of guarantees and demands since it is unlikely that the demands of a system would be fulfilled by guarantees of other systems.

Therefore, ConSerts are conditional. They define a number of variants, i.e., a set of alternative safety guarantees at different integrity levels, depending on which invariants and safety demands are fulfilled at which integrity level. Depending on the current integration context and the safety demands it fulfills, the system evaluates whether it can operate safely in this context and, if yes, which guarantees the system can fulfill in this context.

In fact, ConSerts are issued, just like static certificates, based on a static assessment by safety experts at design time. However, the experts must evaluate not only a single set of guarantees and demands, but all possible variants that are defined within the ConSert.

- (2) In consequence this means that ConSerts need to be evaluated at runtime by the systems themselves. Therefore, ConSerts must be available at runtime in an executable representation and the systems need to possess mechanisms for composing and analyzing these runtime models, i.e., the systems must evaluate which safety guarantees are fulfilled. Often, this step requires a negotiation protocol, because different options of guarantees and demands must be negotiated between the involved systems in order to find a configuration providing a maximal level of functionality. Using these means makes it possible to establish and maintain safety contracts at runtime that span all levels of a composition hierarchy through pairs of ConSert-based guarantees and demands. However, safety is not a completely modular property. Therefore, it is usually not sufficient, to evaluate guarantees and demands, only. Instead, it is likely that some evidences must be acquired at the integration level. For example, to check if a shared communication medium provides sufficient bandwidth and availability for all communication partners. As another example consider two systems acting as redundant channels in a redundancy concept. In this case, the independence of those systems must be checked at runtime in order to exclude common cause failures. In order to support these types of checks at the integration level at runtime, we introduced the concept of runtime evidences as part of ConSerts.

Summarizing these aspects, there are five major constituents of ConSerts, which will be described in more detail in the following sub sections. First, sub section 5.9.2 illustrates the basic idea of guarantees and demands, before sub section 5.9.3 introduces the idea of runtime evidences. The evaluation of demands, runtime evidences and their mapping to guarantees at runtime is described in sub section 5.9.4. Finally, sub section 5.9.5 briefly introduces a model-based approach for specifying ConSerts.

Before these aspects are explained, sub section 5.9.1 explains some general assumptions on modeling open systems of systems.

5.9.1. General Assumptions on Modeling Open Systems of Systems

In order to define model-based safety certificates for open systems of systems, it is important to have an understanding, how the systems as such are modelled in order to augment the models with additional safety information. Figure 5-7 shows a possible approach of basic elements that can be used to describe open systems of systems and how those can be extended for supporting ConSerts. This approach has been introduced in [96] as a general basis for modeling open adaptive systems. In principle, ConSerts can be applied to various different approaches of modeling open systems. Therefore, the idea of using this approach for the remainder of this section is only for the purpose of illustrating how ConSerts can be integrated into a modeling framework for open systems. It does not mean that ConSerts cannot be applied to any other kind of modeling approach for open systems of systems.

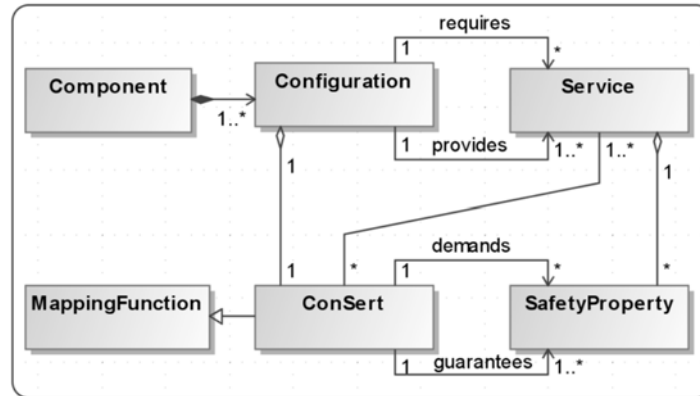


Figure 5-7 - Elements of modeling open systems of Systems [84]

The upper part of Figure 5-7 shows the main elements that are required for modeling open adaptive systems. The approach assumes that components in an open adaptive system are adaptive in the sense that they set themselves to different configurations at runtime, i.e. the freedom of dynamic adaptation is limited to dynamic reconfiguration. The services provided by a component depend on its currently active configuration. Which configurations can be activated at runtime depends on the availability and quality of the required external services in the current runtime context. We assume that all available services in an open system of systems need to be defined in a service type system, which is then used as a formal basis to match required and provided services as it is common practice in service oriented architectures.

Since the behavior of a component strongly depends on its current configuration, ConSerts are assigned to configurations instead of components. In order to express guarantees and demands of a ConSert, they refer to the services required and provided by a component. The services define the logical interface of a component. They are extended by a safety property, e.g., by defining potential failure modes of a service and / or appropriate counter measures.

For example, consider a component of a tractor providing the service ‘tractor acceleration’. In turn, the component requires a service ‘acceleration command’, e.g., provided by an external tractor implement. The latter service could then be extended by a safety property ‘too_high’ expressing the failure mode that too high an acceleration is commanded. A ConSert can then, for example, easily express a demand by referring to the required service ‘acceleration command’ and specifying that the failure mode ‘too_high’ has to be mitigated by the service provider.

By this means, ConSerts and functional interfaces are tightly coupled. On the one hand this means that ConSerts and all related safety assurance approaches can be based upon already existing interfaces. On the other hand, ConSerts can be considered as part of service interface negotiations.

Regarding the tractor example again, the acceleration interface has to be specified from a functional perspective anyway. The ConSerts are based upon this already existing interface. And the tractor component would not only check whether or not the implement

provides the required service. But based on ConSerts, it can additionally check whether or not the implement fulfills the safety demands before it grants access to the acceleration interface of the tractor.

5.9.2. Safety Guarantees and Demands

As described above, the dynamic integration of systems of systems technically relies on a service concept and type systems for all service properties. If one system defines a safety-related demand and another system defines a safety-related guarantee, it must be possible to unambiguously and reliably decide whether or not guarantee and demand match for composing these systems safely.

It is important to find an appropriate level of abstraction supporting automated analyses at runtime. Thus, in order to represent ConSert guarantees and demands at the service level, service specifications are augmented with safety properties (SP). A guarantee is fulfilled if the referred service can be provided in such a way that the assigned safety properties are fulfilled. In the same way a demand is fulfilled if the referred service is available in such a way that the associated safety properties are fulfilled.

In fact, safety properties are comparable to safety requirements, which can be derived similarly to those in a traditional safety engineering approach as described in chapter 2. The main difference is that the components and the structure of an open system of systems cannot be completely predicted at design time, so that a typical top-down approach based on a concrete system would not work. To this end, it is reasonable to work in terms of service types instead of concrete components. ConSerts assume that any functionality supported by an ecosystem of systems of systems, such as agricultural machinery, is documented in an appropriate service type system of the ecosystem and thus known at design time. This is a reasonable assumption since something like a type system is required for establishing functional connections anyway. For example, for connecting tractors and implements in the agricultural domain, the ISOBUS standard [100] specifies all information required to define such a type system for the ecosystem of connected machines for agriculture and forestry.

This knowledge can then be used as a basis for a safety analysis. The service specifications define the to-be-behavior of the services. Using techniques like HAZOP [40], interface FMEA [46], or sophisticated approaches like dynamic guide phrases [97][98], possible deviations from the intended behavior can be derived in order to identify failure modes of the analyzed service type. Based on these failure modes, it is then possible to define safety properties.

As an example, Figure 5-8 shows a generic definition of safety properties based on a common failure type classification defined in [72]. Usually, the safety property definition reflects the domain and service specific failure modes identified during the safety analysis as described above.

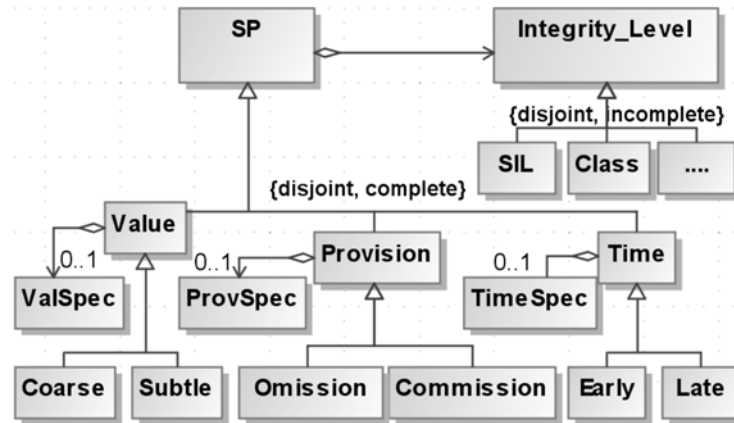


Figure 5-8 Generic type system for safety properties [84]

In this generic example, the principle failure modes *Value*, *Provision*, and *Time* are defined, which are further refined to *Coarse* and *Subtle*, *Omission* and *Commission*, as well as *Early* and *Late*, respectively.

A demand can then be specified by referring to the respective service type. Since the service type specification is extended by safety properties, it is additionally possible to define which of the specified safety properties must hold. In the same way a guarantee can be defined. Using such a type-system-like definition of safety properties is quite intuitive since it reflects failure modes as they are identified in safety analyses already today. And the runtime evaluation can be kept rather simple since it can be reduced to a type check.

In addition to the simple types, it is also possible to provide a refined specification for safety properties. For example, the safety property *Value* as shown in Figure 5-8, can optionally be refined using an additional parameter. As a concrete example, consider the failure mode *value failure* of the acceleration command service required by a tractor. Here it is possible to specify a concrete threshold using the additional parameter in order to refine what a *value failure* means in this particular case. In the concrete case of the example a value of 0.5 m/s^2 could be assigned to this parameter in order to express that the safety property *Value* is fulfilled, if the current acceleration command is not more than 0.5 m/s^2 higher or lower than the intended nominal acceleration value.

Additionally, it is necessary to assign an integrity level to safety properties. Those integrity levels fulfill the same purpose as integrity levels that are assigned to safety requirements in every conventional safety assurance approach. Examples are the Safety Integrity Level (SIL) of the IEC 61508 [2], the Automotive Safety Integrity Level (ASIL) of the ISO 26262 [4], the Design Assurance Level (DAL) defined in the DO-178C [5], and the Software Safety Classes for medical devices of the IEC 62304 [99]. Such an integrity level ultimately represents a level of confidence regarding the fulfillment of a given safety property.

Based on this information, it is possible to completely define guarantees and demands in a sufficiently formal way. By relating the definition to existing interfaces based on

service types simplifies the composition and increases the semantic expressiveness. By extending the type systems with safety properties, it is possible to provide an intuitive, yet sufficiently formal mechanism enabling runtime checks.

5.9.3. Runtime Evidences

When developing a modular safety argument for a service based on guarantees and demands, it might happen that additional evidences are required that cannot be modularized. Consider a device using different external services that must be executed independently from each other for ensuring the absence of common cause failures. In a closed system, requirements such as the independence can be proven at design time using techniques such as fault-tree-based common cause analysis. In open systems of systems, however, the according evidence must be derived at runtime when the required information is available. To this end, ConSerts support so-called *Runtime Evidences* (RtE) as an additional concept. Comparable to demands, it can be specified that different runtime evidences must be fulfilled in order to fulfill a guarantee. In contrast to demands, however, it is necessary to not only check the availability and quality of another service, but dedicated runtime analyses must be executed in order to check whether or not the required conditions are fulfilled in the current runtime context.

Figure 5-9 shows an initial classification of runtime evidences as they are currently supported in the ConSerts approach [85]. They can be classified into two categories, namely *Intra-Device RtE* and *Inter-Device RtE*. The former can be evaluated based on information available within a single device, whereas the latter require distributed evaluation mechanisms across several devices. This requires a collaboration of all devices involved. For this reason, Inter-Device RtEs cannot be defined as ‘freely’ as Intra-Device RtEs and should be part of a domain-specific standard. Runtime evidences are generally part of the safety measures and thus need to be certified according to the integrity level of ‘their’ associated guarantees in the ConSert.

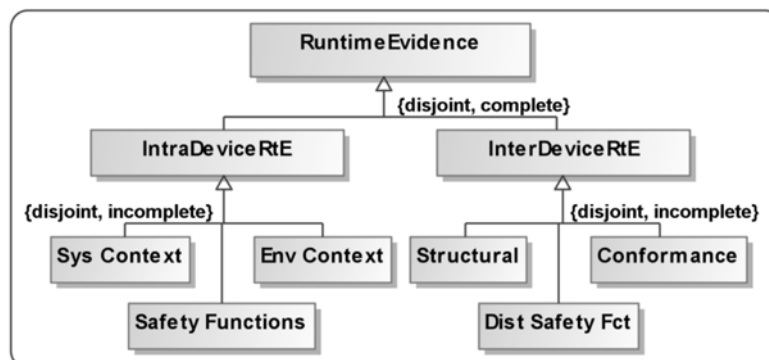


Figure 5-9 Initial classification of runtime evidences [85]

As for Intra-Device-RtEs, ConSerts currently consider three classes of runtime evidences. One class of evidences considers the system context. For example, consider the successful allocation of internal resources that are required to provide a given service.

The second class addresses safety functions, such as error detection, error handling, fault tolerance etc., that must work properly at runtime. Therefore, the corresponding runtime evidences monitor of the correct operation of safety functions. The third class covers evidences based on the environmental context. For example, the maximal acceleration accepted by the tractor could depend on the current awareness of the tractor operator, which could be measured using special sensors at runtime. It is however important to note that all information required to evaluate an intra-device evidence is available within the system itself.

As regards Inter-Device RtEs, there are currently three classes of runtime evidences considered, namely structural evidences, distributed safety functions, and conformance evidences.

Structural evidences are used to check whether or not the system of systems is structured as expected. If, for example, different services are used as redundancy, it must be ensured that they have been realized completely independently from each other in order to avoid common cause failures. To this end, a runtime evidence checks the structure of the system of systems for ensuring that the chains of systems, which are required to implement the services, are independent from each other.

Runtime evidences belonging to the class ‘distributed safety functions’ check whether or not safety functions that are distributed to different systems work correctly. For example, a tractor must not be automatically accelerated from a standstill position. The according safety function can be distributed to the tractor as well as to the implement. The runtime evidence has then to check that the distributed safety function works properly in the current machine setting and runtime context.

Lastly, conformance evidences check that all systems involved comply to the same standards of interoperability, safety etc. For example, it might be necessary to check that the tractor as well as the implement are compliant to the ISOBUS standard. Moreover, this includes checks that the ConSerts of all machines have been issued by accredited authorities.

This is, however, only an initial set of runtime evidences, which can be and must be extended for concrete application scenarios. Referring to the ideas of safety models at runtime, it is for example possible to easily integrate runtime checks of safety cases or runtime verification mechanisms as runtime evidences into the ConSert framework.

5.9.4. Mapping Functions and Runtime Evaluation

With the help of safety properties, we can now specify ConSerts through (sets of) mapping functions between guaranteed safety properties of provided services and demanded safety properties of required services and runtime evidences. ConSerts are generally specified per component, or, in case the component is adaptive and has several configurations, per configuration. Mapping functions are specified for each offered service of a configuration. Therefore, a ConSert might comprise several mapping functions. The domain D of a concrete mapping function $f: D \rightarrow G$ is the union set of sets

of safety demands, i.e., safety properties that are relevant for the required services, and the set of runtime evidences. The co-domain G is the set of safety guarantees, i.e., the safety properties that can be guaranteed by the provided service. Though arbitrary mapping functions are possible, specifying the mapping functions as Boolean functions turned out to be fully sufficient. To this end, the mappings are split up as described in the following.

For each provided guarantee of each provided service, there shall be a separate ConSert sub tree (CST), represented by a Boolean function $f: B^k \rightarrow B$. Inputs of the mapping function are k Boolean variables, each representing a demanded set of safety properties belonging to a required service as well as required runtime evidences, respectively. Such a Boolean variable is true if the demanded safety properties are actually met or if the runtime evidence evaluates to true at runtime, respectively. The Boolean variable of the co-domain represents one concrete set of safety properties that can be guaranteed for the provided service if the Boolean function (i.e., the respective CST) evaluates to true (i.e., all demanded safety properties are met by the required services and all required runtime evidences evaluate to true). So all in all, ConSerts consist of a number of trees specifying Boolean functions, each modeling the condition for its respective guarantee variant and taking demands and runtime evidences as input.

Using guarantees and demands and the according mapping functions, a system's dynamic safety characteristics can be described completely modularly, since it is only based on information available in the local interfaces. Due to the simplicity of the mapping functions and the definition of services and safety properties based on simple type systems, a runtime evaluation is easily possible.

At runtime, the modular ConSerts of the single systems that shall be integrated need to be composed in order to assess the safety of the emerging system of systems. It is desirable to keep the ConSerts and the runtime evaluation as simple as possible and to leave the major part of safety engineering at development time. It is therefore important that ConSerts operate at the certificate level, i.e., by conducting (almost) complete safety engineering and certification at development time and only leaving open some pre-defined certificate variants. These variants can then be resolved at runtime by checking the associated conditions that are bound to the fulfillment of formalized domain-specific demands.

Thus, each component in an open system of systems should at least have one ConSert (several if it has different configurations), specifying its respective pairs of safety guarantees and demands. Technically, there needs to be an adequate runtime representation for ConSerts as well as corresponding mechanisms to enable dynamic composition and evaluation, which will be described below. When a composition structure is build at runtime, a ConSert of a component is always responsible for providing the correct guarantees (i.e., the correct certificate variant) for this component based on the fulfillment of the respective demands. The demands are either directly related to the safety properties of required services or to runtime evidences. With runtime evidences, it

is possible to acquire evidences beyond the required services (such as properties of the compositional structure).

The evaluation of a composition of ConSerts starts from the ‘leaf components’ that do not have any further service dependencies. Their guarantees are propagated up through the composition hierarchy by means of the mapping functions until the top-level component is reached. The evaluation of ConSerts is generally conducted at integration time and whenever the overall composition or configuration changes. Of course, runtime evidences (and also ConSerts as a whole) might be further utilized to monitor systems during runtime. Whenever a runtime evidence is no longer valid, this will have an impact on the current ConSert and the levels of guarantees that can be assured. Through the dependencies between ConSerts within a composition hierarchy, this can result in fewer guarantees or a reduction of integrity levels on the top level, too. When this violates safety requirements, appropriate countermeasures can be taken. Typical measures are switching into a fail-safe state or reconfiguration (graceful degradation). The monitoring frequency could be adjusted according to the respective setting. Services that sporadically send some information could re-evaluate their ConSerts every time and attach the result to their information. Some runtime evidences might even be monitored continuously without big overhead. Others, like independence, would only be evaluated when the application is initialized or when the structure of the system composition actually changes.

The actual component-level evaluation algorithm is identical for each component. It begins with the Invariant and then continues with the CST that offers the best safety guarantees for the service. If the configuration offers several services, this step must be done for each service. At runtime, a CST is represented as BDD at runtime for enabling an efficient evaluation. Therefore, it can be evaluated by traversing its runtime BDD representation, thereby evaluating the BDD nodes that correspond to the Boolean input variables of the CST.

This evaluation is twofold: If the corresponding Boolean variable refers to a demand, its specification has to be matched with the specification of the corresponding guarantee of the consumed service. This comprises matching the different safety properties with their types, specifications, and integrity levels, which are all specified according to the type system. If the Boolean variable refers to a runtime evidence, a specific evaluation (sub-) process is triggered. If a CST (other than the Invariant) evaluates to false, the subsequent CST will be evaluated. As soon as a CST evaluates to true, the corresponding set of guarantees is propagated up in the composition hierarchy to the potential consumer of the evaluated service. If none of the CSTs evaluates to true (or the invariant evaluates to false), only the default guarantee can be assured. Eventually, all components in the composition hierarchy have conducted their evaluation process and the top-level component (i.e., the component that initially started the process) can match the actual possible safety guarantees with its top-level safety demands. Now, the overall composite can either be accepted (i.e., corresponding service contracts can be established) or rejected.

As an example for the evaluation of runtime evidences consider an independence runtime evidence. The latter plays an important role when redundancy is required in order to avoid dependent failures, such as common cause failures or cascading failures. There are two different aspects to this problem that need to be resolved. On the one hand, it must be guaranteed that the respective services do not have any dependencies within single components (i.e., different dependent services that are offered by the same component configuration). On the other hand, it must be ensured that there are no commonly used services on lower levels of the composition hierarchy. The first of these two aspects can be tackled with the help of dependency matrices, which need to be integrated into the components. These matrices make potential dependencies between different provided services of a component explicit. The second aspect requires a dedicated distributed protocol to generate and compare service dependency traces. A prerequisite for such a protocol is a suitable naming scheme for services that yields unique service names. A pragmatic approach is to use a naming scheme that incorporates (unique) device/component names such as $\langle device_name \rangle . \langle service_name \rangle$.

To generate service dependency traces, the first step is to consult the local dependency matrix belonging to the current component configuration. The dependency matrix contains internal as well as external dependencies. For both, the respective service names are added to the trace. The next step is to request the providers of the required external services. This is done via a specific independence request, which is issued directly to the respective components. Again, their dependency matrix is analyzed and the same steps are conducted. Step by step, all dependent services, internal and external ones, are added to the trace. Eventually, when there are no further dependencies, the traces are propagated back to the initial issuer of the request. There, all traces are compared with respect to common elements. If there are common elements, the services are dependent and the runtime check fails. Otherwise, the runtime evidence evaluates to true.

In fact, each runtime evidence might require a separate runtime algorithm and/or protocol. Particularly distributed runtime evidences therefore require a standardization within an ecosystem of systems of systems. Connecting arbitrary systems not following the same standards can therefore not be safely integrated using ConSerts.

5.9.5. Specification Technique

In order to seamlessly integrate the idea of ConSerts into a model-driven design process, it is important to provide a graphical modeling notation for specifying ConSerts. In fact, this mainly requires a possibility to model CSTs. As described earlier, a CST is a Boolean function taking demands and runtime evidences as input. Therefore, a CST can be seen as a tuple (D, R, O, g, E) with:

D: A set of demands

R: A set of runtime evidences

O: A set of Boolean operators, which are represented as gates

g: the provided guarantee, i.e. the guaranteed output value

E: A set edges connecting demands, runtime evidences, operators and the guarantee

In addition to the single CSTs, a ConSert contains an invariant and defines a default guarantee. For each of these constituents a modeling element has been defined as they are shown in Figure 5-10.



Figure 5-10 Graphical Modeling Elements of a ConSert [85]

All of these elements can be connected using directed edges. The name of a ConSert should always follow the following scheme:

<component_name>. <configuration_name>. ConSert

The top-level element of any CST is a guarantee, which has one ingoing edge, representing the evaluation result, and no outgoing edge. It defines the associated provided service, the integrity level, and a set of fulfilled safety properties. A special type of guarantee is the default guarantee, which has no ingoing edges. It defines the guarantees that can be fulfilled in any case, independent of any demands, runtime evidences, or invariants.

In addition to defining guarantees, it is possible to define an invariant, which must be fulfilled for all guarantees except for the default guarantee. In principle, it is defined just like a CST, which will be evaluated at runtime before any CST of the guarantees is evaluated. However, an invariant is not associated to a specific service. Therefore, in contrast to guarantees, there are no further specifications like services or integrity levels required. To underline this difference, invariants can be defined using a dedicated modeling element (cf. Figure 5-10).

In order to express the mapping functions for the normal guarantees, different Boolean gates can be used. A gate has an arbitrary number of ingoing edges and exactly one outgoing edge. Currently, ConSerts support AND-gates as well as OR-gates. Following the standard semantics, the output of an OR-gate is true if one of the ingoing edges is true. The output of the AND-gate is true if all ingoing edges are true. In principle, any kind of Boolean function could be extended as gate to ConSerts.

The leafs of the resulting tree are defined by demands or runtime evidences, respectively. Demands express requirements on external services, namely which safety properties must be fulfilled at which integrity level. Therefore, they have no ingoing edges and may have an unlimited number of outgoing edges leading to different gates or guarantees. In the same way as guarantees they specify the associated required service, the required integrity level, and a set of safety properties that must be fulfilled.

Finally, it is possible to model runtime evidences. They have at least one outgoing edge and no ingoing edges. Since various different runtime evidences are possible, they are typed. Depending on the type, additional specification elements might be required. Considering the independence evidence as example, the additional specification contains a list of all services that need to be regarded in the analysis.

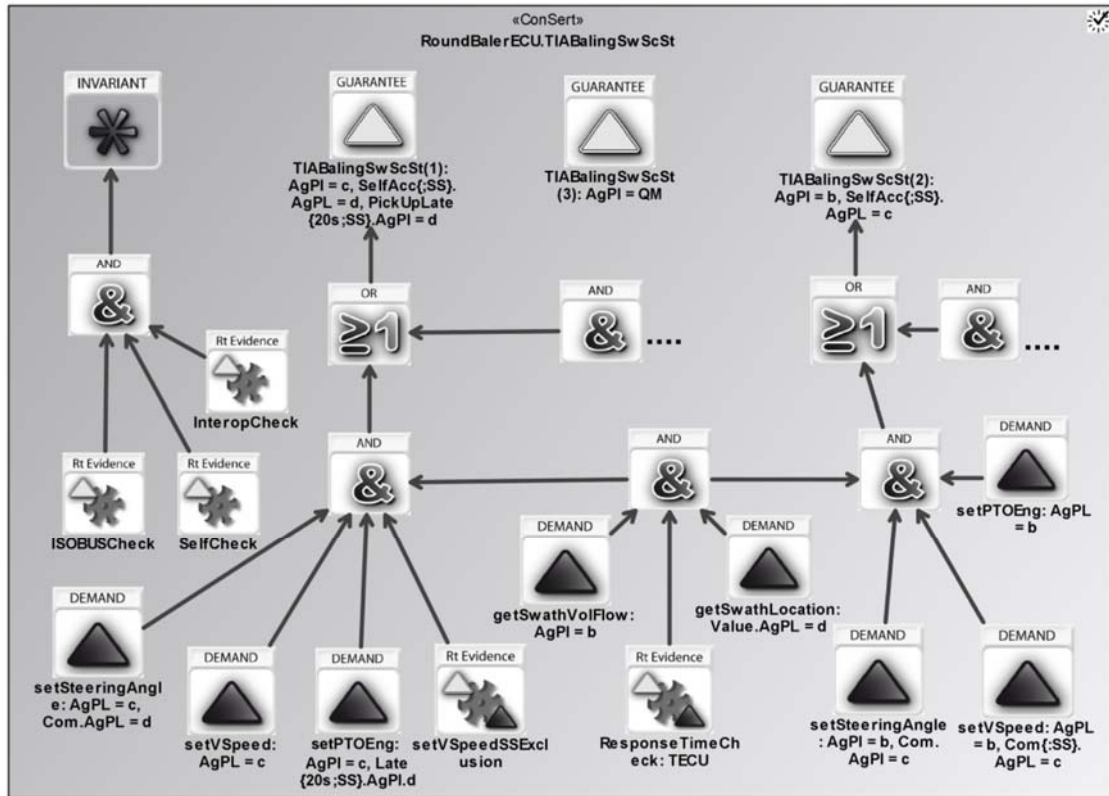


Figure 5-11 Example of a ConSert [85]

In order to further illustrate the concepts, Figure 5-11 shows an example taken from [85]. It shows an excerpt of a ConSert of a round baler, which can control a tractor using a tractor implement automation protocol (TIA). The default guarantee in the middle will not guarantee any integrity level³, i.e. the system must not be used in safety critical setting. The left guarantee is the one with highest priority as it is shown by the priority number in brackets. In general, it provides an integrity level of AgPL³ c and refines certain safety properties. For example, it guarantees that a self-acceleration will not be caused by the implement in a standstill situation by AgPL d (*SelfAcc{SS}.AgPl=d*). There are two options modelled, how this guarantee can be fulfilled. To this end, an OR-gate is used.

One of the options has been omitted in the example. The second option, requires various demands regarding the steering angle commands, speed commands, and power take off (PTO) engagement. Additionally, a runtime evidence is required, which ensures

³ AgPL= Agricultural Performance Level, notion of integrity levels used in the agricultural domain. QM=Quality Management, no safety guarantees are given. (ISO 25119 [101])

that the safety function that prohibits an operation during standstill is still working at runtime.

Furthermore, the ConSert shown in the example models an invariant, which must be fulfilled for all guarantees. In this case, three runtime evidences must be fulfilled to fulfill the invariant. First, it must be ensured that the implement and the connected partner currently work compliant to the ISOBUS-standard⁴ [100]. Second, the implement must first successfully pass a self-check. And third, a special protocol checks the interoperability, i.e., whether or not tractor and implement can work together properly from a functional point of view, which could be realized, e.g., using runtime interoperability tests.

Based on this notation, ConSerts can be easily integrated into a model-driven design-flow. They provide a very intuitive yet sufficiently expressive means to express runtime certificates. Due to the given formality and simplicity, they can be easily evaluated at runtime. Therefore, they provide an ideal basis for runtime safety assurance. This is particularly true, if we do not consider ConSerts for runtime certification only, but as the key element in the broader context of the safety assurance framework that was discussed in the previous section.

⁴ The ISOBUS standard defines all rules for implementing tractor implement automation

6. Summary and Future Work

Assuring safety in open systems of systems poses several challenges. In general, safety assurance approaches and the complete underlying philosophies assume that the system is completely specified prior to a final safety assessment. And that the systems' safety is assessed by a human expert before its commissioning. Open systems of systems, however, cannot be completely specified at design time. They are dynamically composed at runtime, when different systems 'meet' in the field and dynamically interconnect to each other. Therefore, open systems of systems emerge dynamically in the field at runtime. This emergent property is however not limited to the structure, but the system start collaborating as a collective, so that a collective behavior emerges. Moreover, the systems need to dynamically adapt themselves in order to provide the flexibility that is required to operate in such a dynamic setting. In consequence, this leads to uncertainties. This means that there are several properties that cannot be completely anticipated at design time. Thus, traditional safety assurance approaches cannot be used for open systems of systems. Instead, it is indispensable to shift certain parts of the safety assurance process into runtime so that safety assurance can be completed when the concrete context is known.

In order to yield such a runtime safety assurance framework, it is important to use conventional safety assurance lifecycles as starting point, because it is necessary to demonstrate that runtime assurance leads to the system safety integrity as established design time safety assurance approaches. Several steps were then necessary for advancing conventional safety assurance approaches to a runtime safety assurance framework. In order to enable such a framework, we combine the ideas of safety assurance with those of Models@Runtime, which are used for developing open adaptive systems. This means, we created safety models which can be shifted into runtime for enabling the system to reason about its safety at runtime. Using models at runtime enables a sufficient degree of flexibility without losing the required reproducibility. This approach required several steps, which are briefly summarized in the following:

In the first step, it is essential to provide the necessary modularity. While modularity is common practice in engineering, most safety artifacts are not modular by default. In addition to existing modular fault tree analysis approaches, we therefore also modularized the FMEA as a further widely used safety analysis technique. And even more importantly, we also modularized safety concepts and safety cases, as the most important safety artifacts besides safety analyses.

In addition to the modularity, most conventional safety artifacts do not have a model-based representation as it is required for enabling safety models at runtime. Particularly this means that the safety artifacts must provide a minimum level of formality to be interpreted at runtime. Therefore, we advanced modular safety assurance to model-driven safety assurance. This means that we defined a model-based representation of all relevant safety artifacts. By doing so we furthermore seamlessly integrated the safety assurance

lifecycle with the development lifecycle. Besides providing a sound basis for safety models at runtime, this approach has proven to provide many advantages for increasing the efficiency of design time safety assurance as well.

Finally, we used the safety models as a basis to create safety models at runtime. In principle, all of the safety models can be made available at runtime. However, it is reasonable to leave as much responsibility as possible at design time. Therefore, we first focused on safety certificates at runtime, which we implemented as conditional safety certificates – ConSerts. ConSerts already provide a very powerful means of runtime safety assurance for open systems. We have successfully applied the concept in different projects with industry [85]. But even more importantly, ConSerts are the key element of our runtime safety assurance framework. If each system in a system of systems provides a ConSert, it is possible to safely integrate the systems. How the safety assurance happens within the single systems is not relevant for their safe integration. Therefore, ConSerts as safety certificates at runtime provide a means for realizing information hiding in the runtime safety assurance framework. This ensures the required flexibility of the framework to support heterogeneous assurance approaches and it enables the required scalability of the framework to large scale systems of systems.

Using safety certificates at runtime as generic safety interface for a safe system integration at runtime opens a wide range of possibilities to assure the safety of single systems at runtime. For example, we provided concepts to additionally have safety cases or safety analyses at runtime enabling even more flexible dynamic system adaptations. And any other approach for runtime safety assurance can be integrated as long as a runtime safety certificate is provided as safety interface at the system border. This is an essential characteristic of our safety assurance framework, since the systems are developed independently by different companies, each of which needing the freedom to use its own safety assurance approach.

The safety assurance framework has already shown to be a very valuable result for runtime safety assurance of open systems of systems. Nonetheless, it opened a door to a completely new field of safety assurance. Traditional safety assurance research was experiencing a saturation over the last decade and runtime safety assurance was hardly considered. Based on our work on runtime safety assurance, safety research got new impulses and the safety assurance of open systems of systems has gained a lot of importance over the last five years in the safety community. Particularly the idea of SafetyModels@Runtime (SM@RT) has a huge potential for future work. Though we made first steps considering safety cases and analyses at runtime, they bear much more potential. For example, safety analyses could use real time information for better estimations of failure rates, e.g., based on similar approaches as they are used for predictive maintenance. Moreover, the runtime information gathered in analyses and safety cases cannot only be used within a single a system or system of systems. But it is possible to gather the collected knowledge of complete fleets. In consequence, general patterns, issues and solutions could be derived and the systems can be optimized

accordingly. This means that the safety of systems of systems cannot only be monitored, but the collective safety intelligence across a complete fleet could be used to improve the systems' safety. Particularly regarding more complex systems like autonomous systems, whose safety is very difficult to assure, this collective safety knowledge from systems in the field could be invaluable for an effective safety assurance. In the same way as the systems' functional intelligence is increasing based on collective experience collected in the field, it would be possible to use a runtime safety assurance framework as a basis for building up collective safety experience in order to improve the systems' safety intelligence.

Obviously, the runtime safety assurance framework for open adaptive systems is a first step into a new world of safety assurance. It provides a basis for a wide range of completely new possibilities and approaches. And since intelligent, collective systems of systems will be the basis of future innovations, there will be a continuous need for appropriate safety assurance approaches.

Therefore, this thesis does not describe the result of a finished thread of research, but it describes the results, which could be a starting point for a young field of research, which will continuously gain even more importance in the future.

7. References

- [1] agendaCPS – Integrierte Forschungsagenda Cyber-Physical Systems (acatech STUDIE), Heidelberg u.a.: Springer Verlag 2012.
- [2] IEC 61508:2010, SC 65A, Functional safety of electrical/ electronic/ programmable electronic safety-related systems
- [3] A. Avižienis, J. Laprie, B. Randell, C. Landwehr. “Basic concepts and taxonomy of dependable and secure computing.” IEEE Transactions on Dependable and Secure Computing, vol. 1, pp. 11-33, 2004.
- [4] ISO 26262-1:2011 Road vehicles -- Functional safety
- [5] RTCA DO-178C Software Considerations in Airborne Systems and Equipment Certification
- [6] P. Liggesmeyer, Software-Qualität, 2nd ed. Heidelberg, Germany: Spektrum-Verlag, 2009.
- [7] Birolini, A.: Reliability Engineering – Theory and Practice, Third Edition, Springer-Verlag, 1999
- [8] Charpentier, P.: Tools for Software fault avoidance, Standards for Safety Related Complex Electronic Systems, 2000
- [9] DGQ-Band 13-11: FMEA. Fehlermöglichkeits- und Einflussanalyse., Deutsche Gesellschaft für Qualität e.V., 2001
- [10] DIN 25 424: Fehlerbaumanalyse, Deutsches Institut für Normung e.V., 1981
- [11] DIN 25 419: Ereignisablaufanalyse – Verfahren, graphische Symbole und Auswertung, Deutsches Institut für Normung e.V., 1985
- [12] DIN 25 448:Ausfalleffektanalyse (Fehler-Möglichkeits- und -Einfluss-Analyse), Deutsches Institut für Normung e.V., 1990
- [13] Dugan, J.,Bavuso, S.J., Boyd, M.A.: Dynamic Fault Tree Models for Fault-Tolerant Computer Systems, IEEE Transactions on Reliability, Vol. 41, No. 3, 1992
- [14] <http://www.essarel.de/>, Fraunhofer IESE, TU Kaiserslautern, 2006
- [15] System Safety Handbook, Federal Aviation Administration, 2005
http://www.faa.gov/library/manuals/aviation/risk_management/ss_handbook/
- [16] HCi: Cause and Effectdiagram,
<http://www.hci.com.au/hcisite3/toolkit/causeand.htm>
- [17] A Guide to Hazard and Operability Studies, Chemical Industries Associations, 1977
- [18] IEC: Techniken für die Analyse der Zuverlässigkeit – Verfahren mit dem Zuverlässigkeitsblockdiagramm, International ElectrotechnicalCommission, 1993
- [19] IEC 61508-2: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems, International Electrotechnical Commission, 2000

- [20] IEC 61508-6: Functionalsafety of electrical/electronic/programmable electronic safety-related systems – Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3, International Electrotechnical Commission, 2000
- [21] IEC 60812: Analysetechniken für die Funktionsfähigkeit von Systemen - Verfahren für die Fehlzustandsart- und auswirkungsanalyse (FMEA), International Electrotechnical Commission, 2001
- [22] IEC 61882: Gefährdungs- und Betreibbarkeitsuntersuchung (HAZOP) – Leitfaden, International Electrotechnical Commission, 2001
- [23] IEC 61165: Anwendung des Markov-Verfahrens, International Electrotechnical Commission, 2003
- [24] IEC 61025: Fault tree analysis (FTA), International Electrotechnical Commission, 2004
- [25] ISO/TS 16949:2002 Corrected Version, Automotive Industry Action Group(AIAG), 2003
- [26] Kaiser, B., Liggesmeyer, P., Mäckel, O.: A New Component Concept for Fault Trees. In: Lindsay, P; Cant, T. :Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software(SCS'03), Band 33 der Reihe Canberra Conferences in Research and Practice in Information Technology, 2003
- [27] Kaiser, B.: State/Event Fault Trees – A Safety and Reliability Technique for Software-Controlled Systems, Dissertation zur Erlangung des akademischen Grades Doktor der Ingenieurwissenschaften im Fachbereich Informatik - eingereicht am Fachbereich Informatik der Technischen Universität Kaiserslautern, 2005
- [28] Ishikawa, K.: Guide to Quality Control, Asian Productivity Organisation, 1986
- [29] Jesty, P.H., Hopley, K.M., Evans, R., Kendall, Ian, Safety Analysis of Vehicle-Based Systems, Proceedings of the 8th Safety-critical Systems Symposium, 2000
- [30] Kletz, T.: HAZOP and HAZAN – Identifying and Assessing Process Industry Standards, Institution of Chemical Engineers, 1992
- [31] Leveson, N. G.: Safeware, System Safety and Computers, Addison-Wesley, 1995
- [32] Liggesmeyer, P., Rombach, D.: Software-Engineering eingebetteter Systeme, Elsevier Spektrum-Verlag, 2005
- [33] Mauri, G.: Integrating Safety Analysis Techniques, Supporting Identification of Common Cause Failures, University of York, 2000
- [34] MIL-STD-1629A: Procedures for performing a failure mode, effects and criticality analysis, Department of Defense, 1980
- [35] System Safety Handbook, NASA, Dryden Flight Research Center, 1999
- [36] Fault tree handbook for aerospace applications, NASA, 2002
- [37] NUREG 0492: Fault tree handbook, U.S. Nuclear Regulatory Commission, 1981
- [38] Quality System Requirements (QS 9000). Third Edition, Automotive Industry Action Group(AIAG), 1998

- [39] QS 9000 FMEA: Fehler-Möglichkeiten- und Einfluss-Analyse – Referenzhandbuch, Automotive Industrie Action Group(AIAG), 2001
- [40] Redmill, F., Chudleigh, M., Catmur, J.: System Safety – HAZOP and Software HAZOP, John Wiley & Sons, 1999
- [41] The Limitations of Using the MTTF as a Reliability Specification, In Reliability Edge, vol 1, no 1, quarter 2, Tucson, AZ: ReliaSoft Publishing, 2000
<http://www.reliasoft.com/newsletter/2Q2000/mttf.html>
- [42] SAE ARP5580:Recommended Failure Modes and Effects Analysis (FMEA) Practices for Non-Automobile Applications, Society of Automotive Engineers, 2001
- [43] SAE J1739: Potential Failure Mode and Effects Analysis in Design (Design FMEA) and Potential Failure Mode and Effects Analysis in Manufacturing and Assembly Processes (Process FMEA) and Effects Analysis for Machinery (Machinery FMEA), Society of Automotive Engineers, 2002
- [44] Stephans, R. A., Talso, W.: Systems Safety Analysis Handbook, System Safety Society, New Mexico Chapter, 1997
- [45] Trivedi, K. S., Malhotra, M.: Power-Hierarchy of Dependability- Model Types, IEEE Transactions on Reliability, Vol. 43, No. 3, 1994
- [46] VDA: Qualitätsmanagement in der Automobilindustrie, Teil 4. Sicherung der Qualität vor Serieneinsatz, Teil 2.System-FMEA: Verband der Automobilindustrie e.V., 1996
- [47] U.S. Coast Guard: Risk-Based Decision-Making Guidelines, Volume 3, Chapter 8: Hazard and Operability (HAZOP) Analysis, Internetauftritt der U.S. Coast Guard, 2006
<http://www.uscg.mil/hq/g-m/risk/e-guidelines/RBDM/html/vol3/08/v3-08-cont.htm>
- [48] Dugan, J., Bavuso, S.J., Boyd, M.A.: Dynamic Fault Tree Models for Fault-Tolerant Computer Systems, IEEE Transactions on Reliability, Vol. 41, No. 3, 1992
- [49] GSN Community Standard Version 1.0, November 2011. Origin Consulting.
- [50] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. EMF: Eclipse Modeling Framework, Second Edition. Addison Wesley, 2008.
- [51] Object Management Group. Model Driven Architecture (MDA). MDA Guide rev 2.0, 2014
- [52] J.L. Fenn, R.D. Hawkins, P.J. Williams, T.P. Kelly, M.G. Banner, Y. Oakshott (Industrial Avionics Working Group). The Who, Where, How, Why and When of Modular And Incremental Certification. 2nd Institution of Engineering and Technology International Conference on System Safety, 2007.
- [53] T. P. Kelly. Concepts and Principles of Compositional Safety Cases, (COMSA/2001/1/1) - Research Report, commissioned by QinetiQ, (2001).
- [54] C. Szyperski. Component Software: Beyond Object-Oriented Programming. ACM Press. 1997.

- [55] M Ajmone Marsan, G. Balbo, G. Conte. A class of Generalized Stochastic Petri Nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computing Systems* 2 (2) (1984), pp. 93-122
- [56] E. A. Lee, D. Neuendorffer, M.J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers* 12, 3, 2003, pp. 231-260.
- [57] D. Domis, M. Trapp. Integrating safety analyses and component-based design, "Computer Safety, Reliability, and Security",,,58-71,2008,Springer Berlin Heidelberg
- [58] D. Domis, M. Trapp. "Component-based abstraction in fault tree analysis" *Computer Safety, Reliability, and Security* ,,,297-310,2009,Springer Berlin Heidelberg
- [59] D. Domis, K. Hofig, M. Trapp. A consistency check algorithm for component-based refinements of fault trees, *Software Reliability Engineering (ISSRE)*, 2010 *IEEE 21st International Symposium on* ,,,171-180,2010,IEEE
- [60] M. Forster, M. Trapp. Fault tree analysis of software-controlled component systems based on second-order probabilities, *Software Reliability Engineering*, 2009. *ISSRE'09. 20th International Symposium on* ,146-154,2009,IEEE
- [61] R. Adler, D. Domis, K. Höfig, S. Kemmann, T. Kuhn, P. Schwinn, M. Trapp. Integration of component fault trees into the UML Models. In: *Software Engineering* 312-327,2011,Springer Berlin Heidelberg
- [62] K. Höfig, M. Trapp, B. Zimmer, P. Liggesmeyer. *Modeling Quality Aspects: Safety, Model-Based Engineering of Embedded Systems*, 107-118, 2012, Springer Berlin Heidelberg
- [63] J. Reich. *Konzeption und Entwicklung eines Werkzeugs zur Spezifikation und Analyse von Safety-Modellen auf Basis von Enterprise Architect*, Bachelor Thesis, TU Kaiserslautern, 2013
- [64] P. Antonino, S. Velasco Moncada, D. Schneider, M. Trapp, J. Reich: *iSaFe: An integrated Safety Engineering Tool*, To be published in: *Proceedings of 5th IFAC Workshop on Dependable Control of Discrete Systems*, 2015
- [65] M. Trapp, S. Kemmann, D. Domis, M. Förster. *Safety Concept Trees*. In: *Proceedings of RAMS Conference 2009*.
- [66] B. Zimmer, S. Bürklen, M. Knoop, J. Höfflinger, M. Trapp: *Vertical Safety Interfaces – Improving the Efficiency of Modular Certification*. In: *Proc 30th International Conference on Computer Safety, Reliability, and Security (SAFECOMP'11)*, Naples, Italy, LNCS 6849, pp.29-43
- [67] P. Antonino, M. Trapp, P. Barbosa and L.Sousa: *The Parameterized Safety Requirements Templates*. To be published in *Proceedings of The 8th International Symposium on Software and Systems Traceability*, IEEE 2015.
- [68] P. Antonino, M. Trapp, P. Barbosa, E. C. Gurjão, J. Rosário: *The Safety Requirements Decomposition Pattern*. To be published in: *Proc 34th International Conference on Computer Safety, Reliability, and Security (SAFECOMP'15)*

- [69] O. Lisagor, J.A. McDermid, D.J. Pumfrey. “Towards a Practicable Process for Automated Safety Analysis.” In: 24th International System Safety Conference, pp. 596-607, 2006.
- [70] P. Fenelon, et al, “Towards Integrated Safety Analysis and Design.” In ACM Applied Computing Review, 2(1) pp.21-32, 1994.
- [71] D.J. Pumfrey. “The Principled Design of Computer System Safety Analyses.” 1999.
- [72] J.A. McDermid, D.J. Pumfrey. “A development of hazard analysis to aid software design.” in Proc Ninth Annual Conference on Computer Assurance. COMPASS '94 'Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security', 1994.
- [73] DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations, Radio Technical Commission for Aeronautics (RTCA) SC-200, 2005.
- [74] J. Rushby. “Modular Certification.” NASA Contractor Report CR-2002-212130, NASA Langley Research Center, 2002.
- [75] I. Bate, S. Bates, R. Hawkins, T. Kelly, J. McDermid. “Safety case architectures to complement a contract-based approach to designing safe systems.” in Proc 21st International System Safety Conference (ISSC'03): System Safety Society, pp. 182–192, 2003.
- [76] E. Althammer, E. Schoitsch, G. Sonneck, H. Eriksson, J. Vinter. “Modular certification support — the DECOS concept of generic safety cases.” 6th IEEE International Conference on Industrial Informatics, (INDIN), pp. 258–263, 2008.
- [77] H. Kopetz, R. Obermaisser, P. Peti and N. Suri: From a Federated to an Integrated Architecture for Dependable Embedded Real-Time Systems. TU Vienna University of Technology, Austria, and Darmstadt University of Technology, Germany, 2004.
- [78] Website of the AUTOSAR Standard: <http://www.autosar.org>. Last visited: May 2015.
- [79] ARINC 653 P1-2, Avionic Application Software Standard Interface ARINC 653, 2005.
- [80] W. Damm, A. Metzner, T. Peikenkamp, A. Votintseva. “Boosting Re-use of Embedded Automotive Applications Through Rich Components” in Proc. Workshop on Foundations of Interface Technologies 2005 (FIT'05), 2005.
- [81] G. Blair et al., “Models@Run.Time” IEEE Computer Nov. 2010.
- [82] M. Trapp, D. Schneider. Safety Assurance of Open Adaptive Systems – A Survey. Models@run.time. Lecture Notes in Computer Science Volume 8378, 2014, pp 279-318
- [83] D. Schneider, M. Trapp. “A Safety Engineering Framework for Open Adaptive Systems”, in Proc. Fifth IEEE International Conference on Self-Adaptive and Self-Organizing Systems, Ann Arbor, Michigan, USA; October 3 - October 7, 2011.

- [84] D. Schneider, M. Trapp. “Conditional Safety Certification of Open Adaptive Systems”, *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2013.
- [85] D. Schneider. *Conditional Safety Certification for Open Adaptive Systems*. Dissertation, TU Kaiserslautern, 2013
- [86] M. Leucker and C. Schallhart: A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293-303, 2009.
- [87] R. Calinescu and M. Kwiatkowska: CADs*: Computer-aided development of self-* systems. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE 2009)*, volume 5503 of *Lecture Notes in Computer Science*, pages 421-424. Springer, 2009.
- [88] R. Calinescu and M. Kwiatkowska: Using quantitative analysis to implement autonomic IT systems. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 100-110, 2009.
- [89] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli: Model evolution by runtime adaptation. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 111-121, 2009.
- [90] H. J. Goldsby, B. H. Cheng, and J. Zhang: AMOEBA-RT: Run-Time Verification of Adaptive Software, In *Models in Software Engineering*, Holger Giese (Ed.). *Lecture Notes In Computer Science*, Vol. 5002. Springer-Verlag, Berlin, Heidelberg, 2008
- [91] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, G. Tamburrelli: Dynamic QoS Management and Optimization in Service-Based Systems, *IEEE Transactions on Software Engineering*, pp. 387-409, May/June, 2011.
- [92] R. Calinescu: When the requirements for adaptation and high integrity meet. In *Proceedings of the 8th workshop on Assurances for self-adaptive systems (ASAS '11)*. ACM, New York, NY, USA, pages 1-4, 2011.
- [93] R. Adler, M. Förster, M. Trapp.” Determining configuration probabilities of safety-critical adaptive systems.” In *IEEE International Symposium on Ubisafe Computing (UbiSafe'07)*, Niagara Falls, Canada, 2007.
- [94] C. Priesterjahn, C. Heinzemann, W. Schäfer, M. Tichy: Runtime Safety Analysis for Safe Reconfiguration. In *Proceedings of the 3. Workshop „Self-X and Autonomous Control in Engineering Applications’*, 10. *IEEE International Conference on Industrial Informatics*, Beijing, China, 2012.
- [95] S. Kemmann, M. Trapp. SAHARA – A Systematic Approach for Hazard Analysis and Risk Assessment. In: *Proceedings of SAE 2011 World Congress (2011)*, Detroit Mi.
- [96] C. Peper, D. Schneider. Component Engineering for Adaptive Ad-hoc Systems, *Proc. of 30th Intl. Conf. on Software Engineering ICSE'08, SEAMS Workshop*, 2008, Leipzig.
- [97] M. Trapp, S. Kemmann, R. Kalmar, C. Denger. Efficient Safety Analysis of Automotive Software Systems. In: *SAE International Journal on Passenger Cars – Electron. And Electr. Syst.* 2(1) 258-270, 2009.

- [98] C. Denger, M. Trapp, P. Liggesmeyer. SafeSpection - A Systematic Customization Approach for Software Hazard Identification. In: Harrison, Michael D. (Ed.) ; Sujan, Mark-Alexander (Ed.): Computer Safety, Reliability, and Security. 27th International Conference, SAFECOMP 2008 - Proceedings. Berlin : Springer-Verlag, 2008, 44-57 : Ill., Lit. (Lecture Notes in Computer Science 5219).
- [99] International Standard IEC 62304, Medical device software – Software life cycle processes. IEC 62304:2006 (E)
- [100] ISO 11783:2007. Tractors and machinery for agriculture and forestry – Serial control and communication data network. International Standards Organisation, 2007.
- [101] ISO 25119:2010. Tractors and machinery for agriculture and forestry – Safety-related parts of control systems. International Standards Organisation, 2010.
- [102] Anderson, T., Lee (1981). P.A., Fault Tolerance: Principles and Practice, Prentice/Hall
- [103] Russell J. Abbott, Resourceful Systems for Fault Tolerance, Reliability, and Safety, ACM Computing Surveys, Vol. 22, No. 1, March 1990, pp. 35 – 68.
- [104] Dhiraj K. Pradhan, Fault-Tolerant Computer System Design, Prentice-Hall, Inc.,1996
- [105] Lyu, M. (Ed.). (1995). Software Fault Tolerance. New York, NY, USA: John Wiley and Sons.
- [106] Ammann, P., Knight J.C. (1988). Data diversity: An approach to software fault tolerance. Computers, IEEE Transactions on, 37 JA - Computers, IEEE Transactions on (4), pp. 418–425.
- [107] Ammann, P. (1987). Data Diversity: An Approach to Software Fault Tolerance. Proceedings of FTCS-17, Pittsburgh
- [108] Ammann, P. (1988). Data Diversity: An Approach to Software Fault Tolerance. Ph.D. dissertation, University of Virginia
- [109] Arlat, J., Kanoun, K., Laprie, J.C. (1990). Dependability Modeling and Evaluation of Software Fault-Tolerant Systems. 39(4), pp. 504–513.
- [110] Andrews, D.M. (1979).Using Executable Assertions for Testing and Fault Tolerance, Proceedings 9th InternationalSymposium on Fault-Tolerant Computing, pp. 102-105
- [111] Mahmood A., Andrews D.M., McCluskey E.J. (1984). Executable Assertions and Flight Software, Proceedings6th Digital Avionics Systems Conference, pp. 346-351,Baltimore (MD), USA, AIAA/IEEE
- [112] Mahmood, A.; McCluskey, E.J., Concurrent error detection using watchdog processors-a survey,Computers, IEEE Transactions on , vol.37, no.2pp.160-174, Feb 1988
- [113] Arora, A., Kulkarni, S. (1998). Detectors and correctors: a theory of fault-tolerance components. Distributed Computing Systems, 1998. Proceedings. 18th International Conference on

- [114] August, D., Chang, J., Rangan, R., Reis, G., Vachharajani, N. (2005). SWIFT: software implemented fault tolerance. Code Generation and Optimization, 2005. CGO 2005. International Symposium on
- [115] Avizienis, A. (1995). The methodology of N-Version-Programming. Software Fault Tolerance, Trends-in-Software Series: John Wiley & Sons
- [116] Avizienis, A. (1977). On the implementation of N-Version Programming for Software Fault-Tolerance During Execution, COMPSAC, Chicago
- [117] Avizienis, A., Chen, L. (1978) N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. Proceedings of FTCS-8 Toulouse
- [118] Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., August, D.I., Mukherjee, S. S. (2005). Design and Evaluation of Hybrid Fault-Detection Systems. In Proceedings of the 32nd Annual international Symposium on Computer Architecture (June 04 - 08, 2005). International Conference on Computer Architecture. IEEE Computer Society, Washington, DC, 148-159
- [119] B. Ramamurthy, S. Upadhyaya, Watchdog processor-assisted fast recovery in distributed systems, International Conference on Dependable Computing for Critical Applications (DCCA-5), 1995, pp. 125-134
- [120] Oh, N., Shirvani, P.P., McCluskey, E.J. (2002). Error detection by duplicated instructions in super-scalar processors. IEEE Transactions on Reliability, 51(1):63–75
- [121] Oh, N., Shirvani, P.P., McCluskey, E.J. (2002). Control-flow checking by software signatures. Volume 51, pages 111–122
- [122] Oh, N., McCluskey, E.J. (2001). Low energy error detection technique using procedure call duplication. In Proceedings of the 2001 International Symposium on Dependable Systems and Networks
- [123] J. R. Connet, E. J. Pasternak, and B. D. Wagner, Software defenses in real time control systems, in Dig. Znt. Symp. Fault Tolerant Comput.. FTCS-2, Newton, MA, June 19-21, 1972, pp. 94-99.
- [124] J. S. Novak and L. S. Tuomenoksa, Memory mutilation in stored program controlled telephone systems, in Conf. Rec. 1970 Int. Conf. Commun., vol. 2, 1970, pp. 43-32 to 43-45.
- [125] S. M. Ornstein, W. R. Crowther, M. F. Krale, R. D. Bressler, A. Michel, and F. E. Heart, Pluribus-A reliable multiprocessor, in Proc. AFZPS Conf., vol. 44, Anaheim, CA, May 19-22, 1975, pp. 551-559
- [126] Oh, N., Shirvani, P.P., McCluskey, E. J. 2002. ED4I: Error detection by diverse data and duplicated instructions. In IEEE Transactions on Computers. Vol. 51. 180 – 199
- [127] Ohlsson, J., Rimen, M. (1995). Implicit signature checking. In International Conference on Fault-Tolerant Computing
- [128] Miremadi, G., Ohlsson, J., Rimen, M., Karlsson, J. (1995). Use of Time and Address Signatures for Control Flow Checking, International Conference on Dependable Computing for Critical Applications (DCCA-5), pp. 113-124

- [129] Saxena, N.R., McCluskey, E.J. (1990). Control Flow Checking Using Watchdog Assists and Extended-Precision Checksums, IEEE Transactions on Computers, Vol. 39, No. 4, pp. 554-559
- [130] Yung-Yuan Chen, Concurrent detection of control flow errors by hybrid signature monitoring, Computers, IEEE Transactions on , vol.54, no.10pp. 1298-1313, Oct. 2005
- [131] Banerjee, S., Gupta, B., & Liu, B. Design of new roll-forward recovery approach for distributed systems. Computers and Digital Techniques, IEE Proceedings-, 149(3)
- [132] G. Miremadi, J. Karlsson, U. Gunneflo, J. Torin, Two Software Techniques for On-line Error Detection, International Symposium Fault-Tolerant Computing, 1992, pp. 328-335
- [133] Avizienis, A., Magnus, U.V., Laprie, J.C., Randell, B. (2001). Fundamental Concepts of Dependability. On applying coordinated atomic actions and dependable software architectures for developing complex systems. ,Object-Oriented Real-Time Distributed Computing, 2001. ISORC - 2001. Proceedings. Fourth IEEE International Symposium on.
- [134] Cheynet, P., Nicolescu, B., Rebaudengo, M., SonzaReorda, M., Velazco, R., Violante, M. (2000). Experimentally evaluating an Automatic Approach for Generating Safety-Critical Software with Respect to Transient Errors. Nuclear Science, IEEE Transactions on, 47
- [135] Elmendorf, W.R. (1972). Fault-Tolerant Programming. Proceedings of FTCS-2, Newton
- [136] Goloubeva, O., Rebaudengo M., SonzaReorda, M., Violante M. (2003). Soft-error Detection Using Control Flow Assertions. Proceedings on 18th International Symposium on Defect and Fault Tolerance in VLSI Systems
- [137] Rabéjac C., Blanquart J.-P., Queille J.-P., Executable Assertions and Timed Traces for On-Line Software Error Detection, Proceedings 26th International Symposium on Fault-Tolerant Computing, pp.138-147, 1996
- [138] Venkatasubramanian, R., Hayes, J.P., Murray, B.T. (2003). Low-cost on-line fault detection using control flow assertions. In Proceedings of the 9th IEEE International On-Line Testing Symposium, pages 137–143
- [139] Rebaudengo, M., SonzaReorda, M., Torchiano, M., Violante, M. (1999). Soft-error Detection through Software Fault-Tolerance techniques, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 210-218.
- [140] Leveson, N.G., Cha, S.S., Knight, J. C., Shimeall, T.J. (1990). The Use of Self Checks and Voting in Software Error Detection: An Empirical Study, IEEE Trans. Soft. Eng. 16, 4,432–443
- [141] Alkhalifa, Z., Nair, V.S.S., Krishnamurthy, N., Abraham, J.A. (1999). Design and Evaluation of System-level Checks for On-line Control Flow Error Detection, IEEE Trans. On Parallel and Distributed Systems, Vol. 10, No. 6, pp. 627-641
- [142] Yau, S.S., Chen, F.-C. (1980). An Approach to Concurrent Control Flow Checking, IEEE Trans. On Software Engineering, Vol. 6, No. 2, pp. 126-137.

- [143] Yanney, R., Hayes, J. (1986). Distributed Recovery in Fault-Tolerant Multiprocessor Networks. *IEEE Trans. Computers*, 35, pp. 871–879
- [144] Yau, S.S.; Chung, R. C. (1975). Design of self-checking software, in *Proc. 1975 Internatl. Conf. on Reliable Software*, ACM, New York, p. 450.
- [145] Gossens, S., Dal Cin, M. (2004). Structural Analysis of Explicit Fault-Tolerant Programs. *High Assurance Systems Engineering. Proceedings. Eighth IEEE International Symposium on*, pp. 89–96.
- [146] Hecht, H. (1976). Fault-Tolerant Software for Real-Time Applications. *ACM Comput. Surv.*, 8(4), pp. 391–407
- [147] Xu, J., Randell, B. (1996). Roll-forward error recovery in embedded real-time systems, *Proceedings of International Conference on Parallel and Distributed Systems 1996*
- [148] K.H. Kim. (1998). ROAFTS: A Middleware Architecture for Real-Time Object-oriented Adaptive Fault Tolerance Support. *Proceedings on IEEE CS 1998 High-Assurance Systems Engineering (HASE)*
- [149] K.H. Kim (1995). *The Distributed Recovery Block Scheme*. Software Fault Tolerance – Trends-in-Software Series: John Wiley & Sons.
- [150] Kim, K., Goldberg, J., Lawrence, T., Subbaraman, C. (1997). The Adaptable Distributed Recovery Block Scheme and a Modular Implementation Model. *Proceedings of Pacific Rim International Symposium on Fault-Tolerant Systems*, pp. 131–138.
- [151] Kelly, J.P.J., McVittie, T.I., Yamamoto, W.I. (1991). Implementing Design Diversity to Achieve Fault Tolerance. *IEEE Software*, 8(4), pp. 61–71
- [152] Laprie, J.-C., Béounes, C., Kanoun, K. (1990). Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. *Computer*, 23(7), pp. 39–51
- [153] Laprie, J.C., Béounes, C., Kanoun, K., Arlat, J., Hourtolle, C. (1987). Hardware and Software Fault Tolerance: Definition and Analysis of Architectural Solutions. *17th International Symposium on Fault-Tolerant Computing*, Pittsburgh
- [154] Lawrence, J. Dennis. (1993). *Software Reliability and Safety in Nuclear Reactor Protection Systems*.
- [155] Lomet, D.B. (1977). Process Structuring, Synchronization and Recovery using Atomic Actions. *ACM SIGPLAN Notices*, Vol. 12, No. 3
- [156] Nicolescu, B., Rebaudengo, M., Sonza-Reorda, M., Velazco, R., Violante, M. (2002). A software fault tolerance method for safety-critical systems: Effectiveness and Drawbacks. *Proceedings of 15th Symposium on Integrated Circuits and Systems Design*
- [157] Nicolescu, B., Velazco, R., Sonza-Reorda, M. (2001). Effectiveness and limitations of various software techniques for "soft error" detection: A comparative study
- [158] Pradhan, D. (1996). *Fault-Tolerant Computer System Design*: Prentice Hall, Inc.

- [159] Pullum, L. L. (2001). *Software Fault Tolerance: Techniques and Implementation*. Boston: Artech House.
- [160] Pullum, L.L. (1992). *Fault Tolerant Software Decision-Making under the Occurrence of Multiple Correct Results*. Doctoral Dissertation, Southeastern Institute of Technology
- [161] Pullum, L.L. (1993). A new adjudicator for fault tolerant software applications correctly resulting in multiple solutions. *Proceedings of the 12th Digital Avionics Systems Conference*
- [162] Chillarege, R.. (1995). Challenges facing Software Fault-Tolerance. *First Conference on Fault-Tolerant Systems, Madras,*
- [163] Randell, B. (2000). Fault tolerance in decentralized systems. *Autonomous Decentralized Systems, Proceedings of The Fourth International Symposium on Integration of Heterogeneous Systems*
- [164] Randell, B. (1975). System structure for software fault tolerance, *IEEE Transactions on Software Engineering, SE 1*, pp. 220–232
- [165] Randell, B. (1998). Dependability – A unifying concept, *Proceedings of the Conference on Computer Security, Dependability, and Assurance: From Needs to Solutions*, pp. 16–25
- [166] Randell, B., Xu, J. (1995). The Evolution of the Recovery Block Concept. In M. Lyu (Ed.), *Software Fault Tolerance* (pp. 1–21). New York, NY, USA: John Wiley and Sons.
- [167] Randell, B., Xu, J. (1996). Roll-Forward Error Recovery in Embedded Real-Time Systems. *Proceedings of the International Conference on Parallel and Distributed Systems*, pp. 414–421
- [168] Rebaudengo, M., Reorda, M., Torchiano, M., Violante, M. (2001). A source-to-source compiler for generating dependable software. *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*
- [169] Reed, D.P. (1975). Implementing Atomic Actions on Decentralized Data. *ACM TOCS, Vol. 1, No. 1*
- [170] Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I., Mukherjee, S.S. (2005). Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization, 2(4)*, pp. 366–396
- [171] Scott, R., Gault, J., McAllister, D. (1983). The Consensus Recovery Block. *Proceedings of the Total Systems Reliability Symposium*, pp. 74–85.
- [172] Scott, R., Gault, J., McAllister, D. (1987). Fault-Tolerant Software Reliability Modeling. *IEEE Transactions on Software Engineering*,
- [173] Shirvani, P., Oh, N., McCluskey, E., Wood, D., Lovelette, M., Wood, K. (2000). *Software-Implemented Hardware Fault Tolerance Experiments COTS in Space*.
- [174] Shrivastava, S., Wheeler, S. (1990). Implementing fault-tolerant distributed applications using objects and multi-coloured actions. *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 203–210

- [175] Sundresh, T. S. (1998). Software Hardening - Unifying Software Reliability Strategies. 1998 IEEE International Conference on, Systems, Man, and Cybernetics
- [176] David J. Taylor, et al, Redundancy in Data Structures: Improving Software Fault Tolerance, IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980, pp. 585 - 594.
- [177] David J. Taylor, et al, Redundancy in Data Structures: Some Theoretical Results, IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980, pp. 595 - 602.
- [178] J. P. Black, et al, Introduction to Robust Data Structures, Digest of Papers FTCS-10: The Eleventh Annual International Symposium on Fault-Tolerant Computing, October 1 – 3, 1980, pp. 110 - 112.
- [179] J. P. Black, et al, A Compendium of Robust Data Structures, Digest of Papers FTCS-11: The Eleventh Annual International Symposium on Fault-Tolerant Computing, June 24 – 26, 1981, pp. 129 - 131.
- [180] Voas, J. (2001). Fault tolerant Software, IEEE, 18
- [181] Vouk, M., McAllister, D., Eckhardt, D., Kim, K. (1993). An Empirical Evaluation of Consensus Voting and Consensus Recovery Block Reliability in the presence of Failure Correlation. Journal of Computer Software Engineering, pp. 364–388.
- [182] Torres-Pomales, Wilfredo (2000). Software Fault Tolerance: A Tutorial.
- [183] Xu, J., Randell, B., Romanovsky, A. (2002). A generic approach to structuring and implementing complex fault-tolerant software. Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. (pp. 207–214).
- [184] Xu, J., Randell, B., Romanovsky, A., Stroud, R.J., Zorzo, A.F., Canver, E., Henke, F. von. (2002). Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. IEEE Transactions on Computers, Vol. 51, No. 2, pp. 164–179.
- [185] Xu, J., Randell, B., Rubira-Casavara, C.M.F., Stroud, R.J. (1994). Toward an object-oriented approach to software fault tolerance. Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems
- [186] Xu, J., & Randell, B. (1997). Software fault tolerance: $t/(n-1)$ -variant programming. Reliability, IEEE Transactions on, 46, pp. 60–68.
- [187] Xu, J., & Randell, B. (1993). Object-Oriented Construction of Fault-Tolerant Software. Technical Report, 444.
- [188] Xu, J. (1991). The $t/(n-1)$ -diagnosibility and its application to fault tolerance. Proc. International Symposium Fault-tolerant Computing, pp. 496–503.
- [189] Xu, J., & Randell, B. (1994). The evolution of the recovery block concept. Software Fault Tolerance, Trends-in-Software Series: John Wiley & Sons.
- [190] Yeung, W., Schneider, S., Tam, F. (1998). Design and verification of distributed recovery blocks in CSP (Technical Report CSD-TR-98-08, Royal Holloway, University).

- [191] Zhang, Y., Chakrabarty, K. (2004). Dynamic adaptation for fault tolerance and power management in Embedded Real Time Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 3
- [192] Hiller, M. (2000). Error Recovery Using Forced Validity Assisted by Executable Assertions. *EUROMICRO Conference, 1999. Proceedings. 25th, 2*
- [193] Stroph, R., Clarke T. (1998). Dynamic Acceptance Tests for Complex Controllers, *Proceedings 24th EuromicroConference*, pp.411-417
- [194] Stuck, L.G., Foshee, G.L. (1975). New assertion concepts for self-metric software validation. *in Proc. 1975 Internatl. Conf. on Reliable Software*, IEEE, ACM, New York, pp. 59-71.
- [195] Leaphart, E.G., Czerny, B.J., D'Ambrosio, J.G., Denlinger, C.L., Littlejohn, D. Survey of Software Failsafe Techniques for Safety-Critical Automotive Applications. *SAE Technical Paper Series, 2005-01-0779*, 2005.
- [196] Barnett, M., Rustan, K., Leino, M., Schulte, W. The Spec# Programming System: An Overview. *Manuscript KRML 136, Microsoft Research. In Proceedings of CASSI 2004. 2004.*
- [197] Meyer, B., "Eiffel: The Language", Prentice Hall, 1992.
- [198] Meyer, B., *Object-Oriented Software Construction, Second Edition*, Prentice Hall, Upper Saddle River, NJ, 1997, pp. 3–19, 39–64.
- [199] Bolstad, M., Design by Contract: A Simple Technique for Improving the Quality of Software. *In Proceedings of the Users Group Conference (DOD_UGC'04)*, IEEE, 2004.
- [200] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", in *Communications of the ACM*, vol. 12, no.10, October 1969.
- [201] R. Adler, D. Schneider, M. Trapp. "Engineering Dynamic Adaptation for Achieving Cost-Efficient Resilience in Software-Intensive Embedded Systems." *in Proc 2010 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '10)*. IEEE Computer Society, Washington, DC, USA, pp. 21-30, 2010.
- [202] R. Adler, D. Schneider, M. Trapp, "Development of Safe and Reliable Embedded Systems using Dynamic Adaptation." *In: G. Blair et al.; Technische Universität Berlin. Fakultät für Elektrotechnik und Informatik: First Workshop on Model-driven Software Adaptation M-ADAPT'07 at ECOOP 2007, Pages 9-14, Berlin, 2007.*
- [203] Y. Papadopoulos, J. McDermid. "Hierarchically Performed Hazard Origin and Propagation Studies." *in Proc 18th International Conference on Computer Safety, Reliability and Security, LNCS 1608 pp. 139-152, 1999.*
- [204] R. Hawkins, J.A. McDermid. "Performing Hazard and Safety Analysis of Object oriented Systems." *in Proc ISSC 2002, System Safety Society, Denver, 2002.*
- [205] Website: <http://www.reliasoft.com> Last visited: May 2015.