TECHNISCHE
UNIVERSITÄT
MÜNCHEN

Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Entwurfsautomatisierung
Univ. Prof. Dr.-Ing. Ulf Schlichtmann

PhD-Thesis

Formalization and Model-Driven Support of Functional Safety Analysis

Moomen Chaari

Lehrstuhl für Entwurfsautomatisierung
der Technischen Universität München

Formalization and Model-Driven Support of Functional Safety Analysis

Moomen Chaari

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender:	Prof. Dr.-Ing. Gerhard Rigoll
Prüfer der Dissertation:	
	1. Hon.-Prof. Dr.-Ing. Wolfgang Ecker
	2. Prof. Dr.-Ing. Georg Sigl

Die Dissertation wurde am 05.03.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 25.11.2019 angenommen.

PhD-Thesis

Institute of Electronic Design Automation
Univ. Prof. Dr. -Ing. Ulf Schlichtmann
Department of Electrical and Computer Engineering
Technische Universität München

in Cooperation with

Infineon Technologies AG
IFAG DES DMF SVT SLM
Dr.-Ing. Matthias Bauer
Dipl.-Inf. Thomas Kruse
Prof. Dr.-Ing. Wolfgang Ecker

Author: Moomen Chaari

Abstract

Functional safety is a crucial aspect of system design and manufacturing, particularly when human life is at stake. During the last two decades, evaluating systems with respect to their safety integrity level has become a major topic among the research community and within the industry. However, several challenges still persist in the two essential contexts of functional safety evaluation, namely analysis on the one hand and simulation on the other hand.

The focus of this thesis is functional safety analysis. It is addressed from three different perspectives: formalization, model-driven automation, and link to simulation.

First, to tackle informality and disorganization issues, traditional functional safety analysis procedures such as Failure Modes, Effects, and Diagnostic Analysis (FMEDA) and Fault Tree Analysis (FTA) are extensively studied and consequently formalized through so-called metamodels. These metamodels are structured descriptions of the analysis artefacts and the relationships between them.

Then, practical inconveniences encountered in the context of functional safety analysis are addressed. In fact, analysis data size is increasing in correlation with system complexity. Subsequently, many analysis tasks, which are traditionally performed manually by functional safety engineers, are becoming significantly cumbersome, so that related human mistakes are occurring more and more frequently. With this respect, metamodel-based formalization of safety analysis methods represents a robust foundation to develop proper platforms and tools enhancing the automation, interoperability, and reuse levels. That is why, a manifold environment for model-driven support of functional safety analysis is conceived, implemented, and applied as the second pillar of this work.

Finally, reducing the gap between safety analysis on the one hand and fault injection and simulation on the other hand is the third central topic in this thesis. The divergence between those two safety evaluation contexts is a major difficulty in the safety lifecycle, as it prevents developers from ensuring data consistency and traceability in a systematic and simple manner. To overcome this divergence, equivalences and/or correspondences between data artefacts of both contexts are investigated in this work. The perceptions and outcomes of this investigation are then used to develop a semi-automated data mapping tool dealing with the syntactic and semantic discrepancies between analysis and simulation.

Zusammenfassung

Funktionale Sicherheit ist ein entscheidender Aspekt beim Systementwurf, vor allem, wenn das menschliche Leben beeinträchtigt werden kann. In den letzten zwei Jahrzehnten ist die Auswertung von Systemen in Bezug auf ihre Sicherheitsintegrität zu einem Hauptthema in der Forschungsgemeinschaft und der Industrie geworden. Trotz der vielfältigen Bemühungen innerhalb dieses langen Zeitraums, gibt es weiterhin Herausforderungen in den zwei wesentlichen Kontexten der funktionalen Sicherheitsbewertung, nämlich der Analyse auf der einen Seite und der Simulation auf der anderen Seite.

Der Schwerpunkt dieser Arbeit ist die Analyse funktionaler Sicherheit. Sie wird aus drei verschiedenen Perspektiven adressiert: die Formalisierung, die modellgetriebene Automatisierung und ihre Verknüpfung zur Simulation.

Zuerst wurden im Rahmen dieser Arbeit traditionelle Methoden der Sicherheitsanalyse, wie zum Beispiel "Failure Modes, Effects, and Diagnostic Analysis" (FMEDA) und "Fault Tree Analysis" (FTA), umfassend untersucht. Um die vorhandenen Formalisierungs- und Strukturierungsprobleme bei der Analyse zu beseitigen, wurden diese Methoden durch so genannte *Metamodelle* beschrieben. Diese Metamodelle dienen zur formalen Strukturierung der Analyseartefakte und der Beziehungen zwischen ihnen.

Als zweites wurden technische Einschränkungen im Zusammenhang mit funktionaler Sicherheitsanalyse untersucht. In der Praxis steigt die Analyse-Datengröße in Korrelation mit der Systemkomplexität. Dadurch werden viele Analyseaufgaben, die in der Regel von Safety-Ingenieuren manuell durchgeführt werden, wesentlich aufwändiger, so dass menschliche Fehler immer häufiger auftreten. In dieser Hinsicht stellt die metamodellbasierte Formalisierung von Sicherheitsanalysemethoden eine robuste Grundlage dar, um geeignete Plattformen und Werkzeuge zu entwickeln, die die Automatisierung, die Interoperabilität und die Wiederverwendung verbessern. Deswegen wurde eine umfangreiche Umgebung für die modellgetriebene Unterstützung funktionaler Sicherheitsanalysen konzipiert und umgesetzt.

Schließlich ist die Verringerung der Lücke zwischen Sicherheitsanalyse einerseits und Fehlereffektsimulation andererseits das dritte zentrale Thema in dieser Arbeit. Die Divergenz zwischen diesen beiden Aspekten ist eine bedeutende Problematik im Sicherheitslebenszyklus, da sie die Systementwickler daran hindert, die Datenkonsistenz und -rückverfolgbarkeit systematisch und einfach zu gewährleisten. Um diese Divergenz zu überwinden, wurden in dieser Arbeit Äquivalenzen und Beziehungen zwischen Datenartefakten beider Kontexte untersucht. Die Ergebnisse dieser Untersuchung wurden dann verwendet, um ein halb-automatisches Werkzeug zur Datenabbildung zu entwickeln, das sich mit den syntaktischen und semantischen Diskrepanzen zwischen Analyse und Simulation beschäftigt.

Acknowledgment

I would like to thank, first and foremost, Prof. Dr.-Ing. Ulf Schlichtmann and Prof. Dr.-Ing. Wolfgang Ecker for giving me the opportunity to accomplish my PhD thesis at the Institute for Electronic Design Automation at the Technical University of Munich in cooperation with the R&D department of Infineon Technologies AG in Munich.

I owe my deepest gratitude to Prof. Dr.-Ing. Wolfgang Ecker for his continuous support and his valuable guidance. This thesis would not have been possible without his encouragement, assistance, and feedback throughout the last five years.

I also thank Prof. Dr.-Ing. Georg Sigl for being the second examiner and providing me with valuable feedback about the thesis.

Furthermore, I am indebted to Thomas Kruse, Bogdan-Andrei Tabacaru, Cristiano Novello, Leily Zafari, Michael Velten, Adalbert Perbandt, Christian Lück, Christian Wiechert, Daniel Müller-Gritschneder, and Matthias Bauer for their highly appreciated support. Their hints and suggestions considerably helped me to fulfill the purposes of my work.

My thanks are extended to all colleagues of the research project EffektiV from whom I learned a lot about functional safety and with whom I shared an enjoyable time.

Finally, I dedicate this thesis to all members of my family, especially to my dear parents Abdelmajid and Mounira, to my beloved wife Amal, to my precious son Adam, and to my cherished sisters Manel and Maissa.

Moomen Chaari
Munich, February 10, 2020

Contents

List of Tables

List of Figures

1 Introduction

"*Whenever the reliability of a product affects human life, the reliability problem becomes part of a larger issue called safety*" [20]. This statement has already been made in the late 1970s. However, even today, it still summarizes the difference between *reliability* and *safety* in a very concise way.

Reliability is a non-functional property qualifying the continuity and stability of the correct service for a certain product. Along with *availability*, *security*, *safety*, and other properties, reliability is one of the so-called *dependability* attributes. In a nutshell, products are considered dependable if the services they deliver can be justifiably trusted [21].

It goes without saying that almost no customer would buy a product if the required service is prone to undesired interruptions, or worse if it is expected to become completely dysfunctional after a short usage time. Nevertheless, it is obviously unacceptable that the product continues to deliver its intended service when the user(s) or the environment may be endangered. In such cases, *functional safety* has priority. In other words, interrupting the primary product functionality and/or transitioning to a *safe state* for all those affected is mandatory.

The question whether and to which extent a product needs to be qualified as *functionally safe* depends closely on the application areas where it is deployed. Several domains, where human life is significantly exposed, such as medical electronics, avionics, and automotive are considered *safety-critical*. For such domains, stringent norms and standards prescribe the guidelines which must be followed to ensure functional safety. It is actually required to conduct a range of assessment, analysis, and verification procedures throughout the production stages, from the early conceptual design to the late physical testing.

In this work, functional safety of electrical/electronic systems is considered. Thus, the remainder of this introductory chapter is organized as follows. First, the omnipresence of electronics in everyday life is emphasized in Section 1.1. Then, a brief introduction into *dependability* terms and concepts with respect to E/E systems is given in Section 1.2, with a clear focus on safety aspects. Although the theoretical basis of this thesis is not limited to a specific application domain, many of the examinations, perceptions, and approaches it contains are motivated by the deployment of E/E systems in automotive applications. Therefore, Section 1.3 provides a general overview of functional safety topics in the automotive context. In Section 1.4, the motivation behind this work as well as its main objectives are elaborated. Finally, the organization of the thesis is given in Section 1.5.

1.1 Our Life in the Age of Electronics

Nowadays, our daily life is marked by the omnipresence of electrical/electronic devices and systems. We need them to access information, communicate with others, and conveniently perform ordinary household chores. We can also use them to make easier payments, flexibly check our health status, or simply occupy our free time with new forms of entertainment. Moreover, mobility is a key aspect which is considerably affected by electronics. Modern cars, trains, planes, etc., rely to a large extent on E/E systems. Hence, it is undeniable that electronics changed our lifestyle and simplified it in multiple ways.

However, this ubiquity of electronics raises many questions about potential negative effects they might have. Can we really trust such devices and to which extent should we allow ourselves to rely on them? How may they affect our health conditions, our safety, and our security?

Such concerns have been extensively addressed in the past, but they still represent a persistent preoccupation for designers and manufacturers of E/E systems. In fact, on top of traditional issues related to functionality, efficiency, and convenience, extra properties must be guaranteed. These are mostly referred to a as *dependability* or *robustness* properties.

1.2 Shifting from Functionality to Dependability

As already briefly mentioned above, proving that a system is *operational*, in the sense that it delivers the *service* which is expected from it, is not enough anymore. In fact, the delivered service, which is commonly defined as the "system behavior as it is perceived by other systems, the user, or the environment" [21], must be *trustworthy* with respect to continuous correctness over time (reliability), absence of catastrophic hazards (safety), ability to resist external attacks (security), etc. This set of non-functional system properties (availability, reliability, safety, security, etc.) is referred to in the literature and also within the industry as *system dependability* [21].

Dependability topics have been gaining more importance in the academia since the 1990s, particularly in research fields related to E/E (electrical/electronic) systems. They also became progressively a major concern for the multiple industries relying on E/E systems such as telecommunications, robotics, avionics, and automotive. The semi-conductor industry, being a key supplier for these industries, is subsequently encountering dependability-related challenges and investigating the most appropriate approaches to overcome them. Such challenges are even bigger, when the manufactured chip is part of a safety-critical product, where failures have an imminent harmful effect on human lives: a scenario becoming more and more frequent. In fact, it is nowadays hard to find a safety-critical product which does not include integrated circuits. Cars, planes, surgical robots, and countless other sorts of products rely on the functionality of the chips they contain.

Beyond the widespread deployment of integrated circuits in consumer products, the evolution of chip technologies over the years and their new physical properties bring dependability issues into focus too.

Due to the increasing integration density of modern chips, a miniaturization trend of electronic components has been witnessed throughout the last decades. As a result, electronic-based products became smaller and subsequently much more accessible and affordable for a large customer range. Obviously, these technology trends represent considerable benefits for chip manufacturers in terms of market opportunities, revenue, and profit margins. Nevertheless, several issues emerged as a result of the miniaturization trend.

In fact, smaller technology nodes bring considerable challenges with them. Conquering the increasing complexity while keeping development and implementation costs at a low level is a difficult dilemma for which appropriate trade-offs must be identified. Furthermore, there are undeniable physical drawbacks of higher integration densities. A small electronic device consumes certainly less power. However, the exponential growth of device count per mm^2 leads to a drastic increase of power density and subsequently to rising temperatures [22]. A common mitigation measure for this issue is the usage of reduced voltages, which leads though to another problem from a robustness perspective. Indeed, voltage scaling is correlated with the susceptibility to radiation, which has an impact on circuit integrity, along with other degradation effects such as negative-bias temperature instability (NBTI) and hot-carrier induced degradation (HCID) [22].

Hence, chip manufacturers are nowadays confronted with the evident fact that highly-integrated circuits suffer from an increasing vulnerability to faults which might be induced by the surrounding environment or by aging effects. So it is necessary to thoroughly analyze the occurrence of such faults, assess their impact on the overall system dependability, and develop the appropriate counter-measures to detect and/or correct them.

To summarize, technology advancements and diversified applications of integrated circuits forced manufacturers to shift from focusing only on traditional concerns such as functionality and cost to a more comprehensive mindset which includes (i) identifying dependability properties that need to be fulfilled, (ii) understanding dependability threats and reasons behind them, and (iii) applying necessary activities to ensure and/or prove dependability.

The dependability scope is obviously too large to be covered by a single work, that is why the primary focus of this thesis is one specific dependability attribute, namely *safety*. Moreover, automotive is one of the largely affected application domains by the importance of safety evaluation, therefore, a brief introduction of functional safety topics in the automotive context is given in Section 1.3.

1.3 Functional Safety in the Automotive Context

Section 1.2 gave an overview about dependability and emphasized its importance with respect to E/E systems in a generic way. The major concrete reflections of dependability concerns in the automotive context are related to one specific dependability attribute: *safety*, commonly defined as "the absence of harmful failures for the human users and the environment of a product". Safety shall not be confounded with *security* which is rather the "absence of unauthorized access and/or handling of data" [21]. There is in fact a simple way to distinguish between both aspects. On the one hand, safety deals with *protecting the environment from the system*, more precisely from the system failures. On the other hand, security deals with *protecting the system from the environment*, more concretely from the malicious attacks coming from the outside.

So, how to create an E/E system which is trustworthy with respect to safety? In other words, how to create a product which is considered *functionally safe*? Probably, the first answer that comes to mind is: "through hardening the system with safety measures". This is however just one aspect of the solution. Perceptions from the industrial practice lead to a much more generic answer: "through tailoring and enhancing the complete manufacturing flow".

In fact, system manufacturers, as well as all relevant suppliers, are responsible for including state-of-the-art safety-related activities into the conception, design, development, and production flow, applying them to a sufficient extent in compliance with the relevant standards, and providing adequate evidence about that. Furthermore, the outcomes of safety-related activities must be documented in order to demonstrate whether and/or how the product meets all *safety requirements* and ensures the intended *safety level*. Only then, the system can be qualified as *safety compliant*.

In the automotive context, ISO 26262 ("Road vehicles - Functional safety") [2] is the dedicated standard which (i) describes the *safety lifecycle* of an automotive system, (ii) prescribes the set of activities and guidelines that shall be followed to ensure safety compliance, and (iii) addresses analytical and experimental approaches to evaluate safety integrity levels and provide evidence to substantiate them.

The remainder of this section is organized as follows. First, Subsection 1.3.1 gives a general overview of the ISO 26262 standard. Then, the different safety-related activities, which are prescribed by the standard, are enumerated in Subsection 1.3.2. As this thesis focuses essentially on functional safety analysis, Subsection 1.3.3 provides a brief presentation of the different analysis approaches that the ISO 26262 standard addresses. Subsection 1.3.4 gives a general insight about *fault injection*, another important aspect of safety evaluation that is addressed by the standard and also relevant for this work. Finally, Subsection 1.3.5 explains the correlations between safety analysis, fault injection, and traditional functional verification with respect to the ISO 26262 standard.

1.3.1 General Overview of ISO 26262

ISO 26262 is an international functional safety standard addressing electrical and/or electronic (E/E) systems which are contained in production vehicles. It is defined by the International Organization for Standardization (ISO). The first edition of ISO 26262 has been published in 2011 and its second edition in 2018.

The complete lifecycle of automotive E/E systems is covered by ISO 26262 across the ten parts it contains. The composition of the standard is illustrated in Figure 1.1.

ISO 26262 comprises a number of rather generic parts, namely:

- Part 1 (“Vocabulary”) defining the terminology used across the standard,
- Part 2 (“Management of functional safety”) giving guidelines on how to manage safety-related activities during the product lifecycle,
- Part 8 (“Supporting processes”) addressing specific tasks such as documentation and tool qualification,
- Part 9 (“ASIL-oriented and safety-oriented analyses) explaining Safety Integrity Levels (ASILs) ranging from A to D and describing common safety analysis techniques such as Failure Modes and Effects Analysis (FMEA) and Fault Tree Analysis (FTA), and
- Part 10 (“Guideline on ISO 26262”) including some illustration examples on the different procedures.

On top of the generic parts mentioned above, ISO 26262 addresses (i) the “Concept phase” in Part 3 where the safety requirements are elicited and the safety lifecycle is

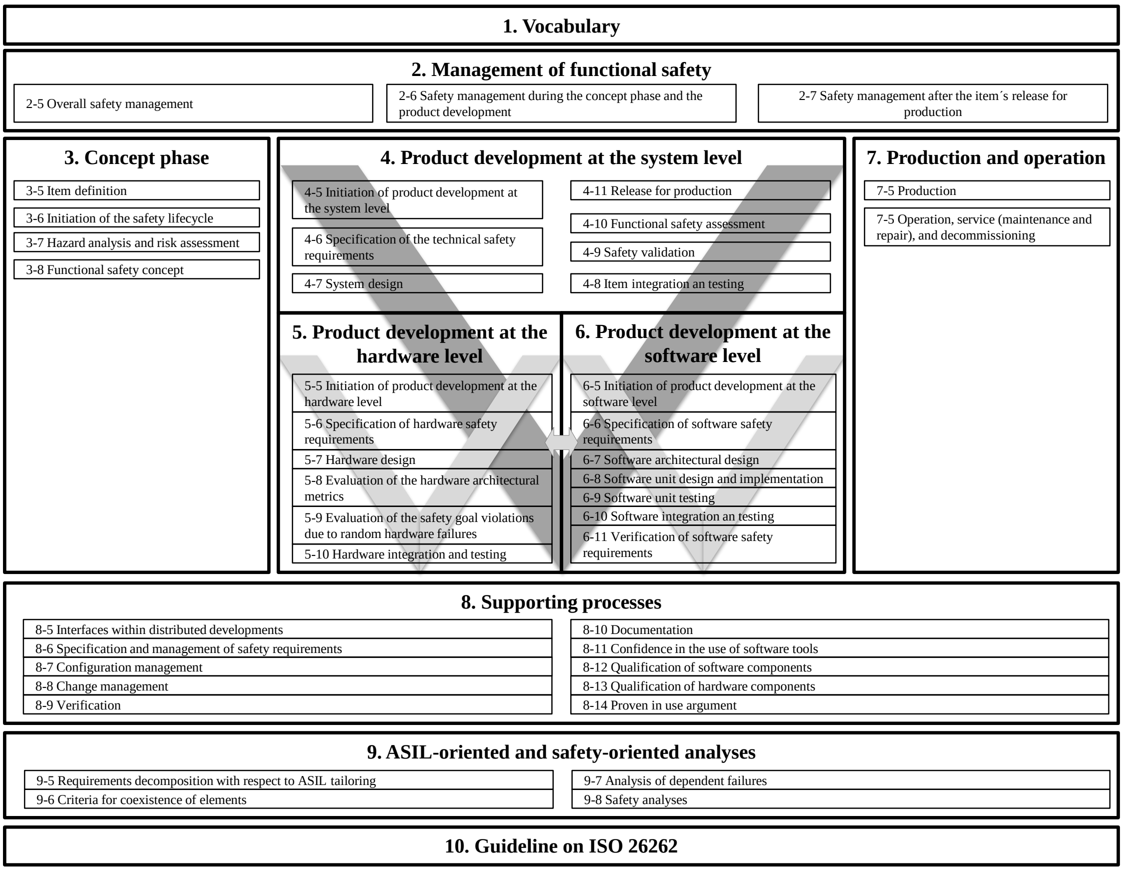

1. Vocabulary
2. Management of functional safety
2-5 Overall safety management
2-6 Safety management during the concept phase and the product development
2-7 Safety management after the item´s release for production
3. Concept phase
3-5 Item definition
3-6 Initiation of the safety lifecycle
3-7 Hazard analysis and risk assessment
3-8 Functional safety concept
4. Product development at the system level
4-5 Initiation of product development at the system level
4-6 Specification of the technical safety requirements
4-7 System design
4-11 Release for production
4-10 Functional safety assessment
4-9 Safety validation
4-8 Item integration an testing
7. Production and operation
7-5 Production
7-5 Operation, service (maintenance and repair), and decommissioning
5. Product development at the hardware level
5-5 Initiation of product development at the hardware level
5-6 Specification of hardware safety requirements
5-7 Hardware design
5-8 Evaluation of the hardware architectural metrics
5-9 Evaluation of the safety goal violations due to random hardware failures
5-10 Hardware integration and testing
6. Product development at the software level
6-5 Initiation of product development at the software level
6-6 Specification of software safety requirements
6-7 Software architectural design
6-8 Software unit design and implementation
6-9 Software unit testing
6-10 Software integration an testing
6-11 Verification of software safety requirements
8. Supporting processes
8-5 Interfaces within distributed developments
8-6 Specification and management of safety requirements
8-7 Configuration management
8-8 Change management
8-9 Verification
8-10 Documentation
8-11 Confidence in the use of software tools
8-12 Qualification of software components
8-13 Qualification of hardware components
8-14 Proven in use argument
9. ASIL-oriented and safety-oriented analyses
9-5 Requirements decomposition with respect to ASIL tailoring
9-6 Criteria for coexistence of elements
9-7 Analysis of dependent failures
9-8 Safety analyses
10. Guideline on ISO 26262

Figure 1.1: General Overview of ISO 26262 [2]

initiated, (ii) the "Product development" in Parts 4, 5, and 6, and (iii) the "Production and operation" phase in Part 7 including service and decommissioning too.

Most emphasis in the ISO 26262 is on the product development phases, first at system-level in Part 4, then at hardware level in Part 5, and finally at software level in Part 6. It should be noted that the standard uses a V-model as reference for product development and that this model does not only apply to the complete system level in a holistic way but also separately for the hardware and software levels [2, 23].

1.3.2 Safety-Related Activities

The primary purpose of safety-related activities throughout the product lifecycle is to ensure the "absence of unreasonable *risk*", where a *risk* is the combination of (i) the occurrence probability of a physical injury or damage to human health, referred to as *harm*, with (ii) the *severity* of that *risk* [2]. In other words, the *residual risk* that the final product would endanger human life must be sufficiently negligible.

Figure 1.2 gives an overview of some safety-related activities described in the ISO 26262 standard and particularly relevant for the scope of this thesis. Figure 1.2 also illustrates the distribution of safety-related activities over the requirements, design, and test phases of the product lifecycle and the interconnections between them.

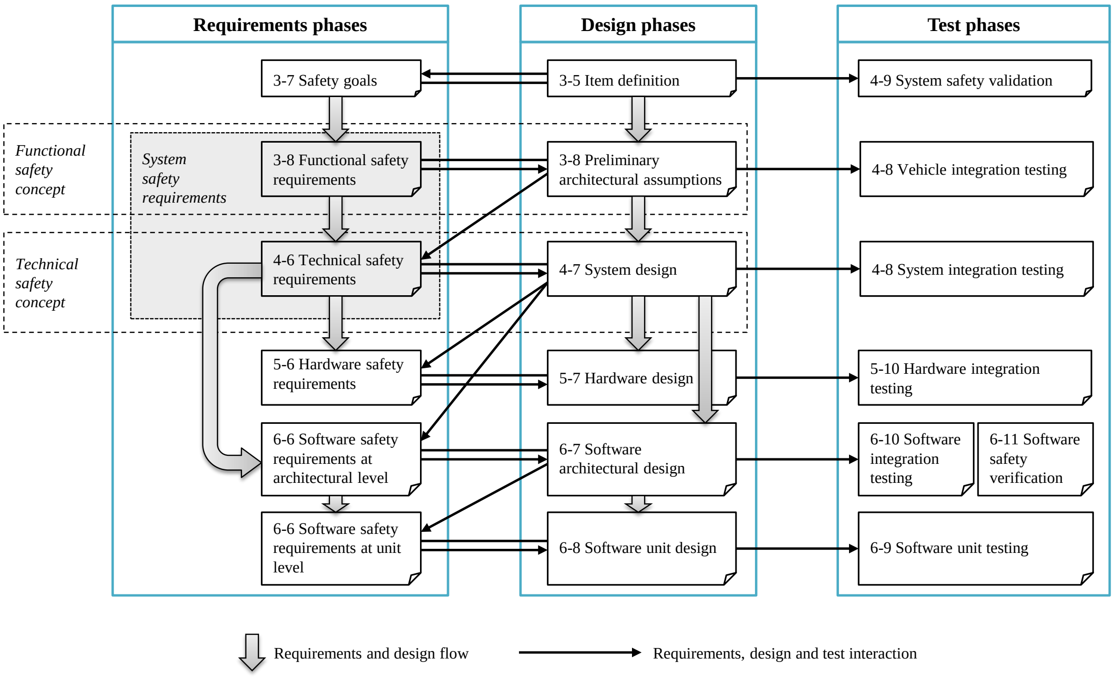

Requirements phases
Design phases
Test phases
3-7 Safety goals
3-5 Item definition
4-9 System safety validation
Functional safety concept
System safety requirements
3-8 Functional safety requirements
3-8 Preliminary architectural assumptions
4-8 Vehicle integration testing
Technical safety concept
4-6 Technical safety requirements
4-7 System design
4-8 System integration testing
5-6 Hardware safety requirements
5-7 Hardware design
5-10 Hardware integration testing
6-6 Software safety requirements at architectural level
6-7 Software architectural design
6-10 Software integration testing
6-11 Software safety verification
6-6 Software safety requirements at unit level
6-8 Software unit design
6-9 Software unit testing
Requirements and design flow
Requirements, design and test interaction

Figure 1.2: Overview of Safety-Related Activities in ISO 26262: Concept, System, Hardware, and Software Levels [2]

The starting point is a so-called *hazard analysis and risk assessment* procedure which identifies *hazards* and *hazardous events* that need to be prevented, mitigated, or controlled (more details in Subsection 2.2.2). For each hazardous event, a *safety goal* is formulated and assigned an Automotive Safety Integrity Level (ASIL) based on specific criteria (see Subsections 2.2.3 and 2.2.4 for more details).

After that, the *functional safety concept*, which allocates *functional safety requirements* to *preliminary architectural assumptions*, is derived as a statement of the safety functionality needed and/or expected to achieve the safety goals (see Subsection 2.2.5.1).

Then, the *technical safety concept*, which maps *technical safety requirements* to system design elements, evolves during the product development phase as a statement of how the safety functionality is implemented on the system level by hardware and software (see Subsection 2.2.5.3).

Similar activities are performed later on at hardware and software levels to state the specific hardware and software safety requirements which will be concretely implemented within the *hardware parts* or the *software units*.

Defining safety requirements and ensuring that they are concretely implemented within the product is necessary but not sufficient. Similarly to traditional design flows where functionality features must be verified, the safety lifecycle prescribed by the ISO 26262 standard includes several dedicated activities to *assess*, *verify*, and *test* the fulfillment of safety requirements.

Within the design phases at system, hardware, and software levels, a set of *safety evaluation* activities shall be applied. In the context of this thesis, safety evaluation is the generic term used to denote both analysis-based activities (see Subsection 1.3.3) and simulation-based activities (see Subsection 1.3.4) which are commonly applied in the industry to provide evidence about the safety level of a certain product.

In the ISO 26262 context, additional safety-related activities are applied during the test phases. Once the integration and testing tasks are completely performed at hardware, software, and system levels, the final vehicle integration and testing is executed. Before moving forwards with the mass-series production, an extra activity called *safety validation* must be conducted to prove the correctness and sufficiency of the safety requirements and of the safety measures that have been implemented at the underlying levels (more details in Subsection 2.2.6).

1.3.3 Safety Analysis Approaches

As already mentioned above in Subsection 1.3.2, safety analysis represents one of two key aspects of safety evaluation activities, which are applied during the product development phases and more precisely at design stages.

Safety analysis is an extensive procedure which starts by listing all potential risks, whose causes and effects are subsequently identified. Evaluating the severity of all failure effects with respect to the overall safety level is also part of the analysis.

Depending on the stage at which the analysis is performed, counter-measures to the identified dangerous failure effects might be completely absent (e.g., at early concept level). In this case, the primary responsibility of the analyst is to raise a flag about the necessity of undertaking action by the designers. If the analyst disposes over sufficient design knowledge, he or she can potentially identify appropriate safety measures and give a recommendation to include them in the design.

When the analysis is performed at a late stage of the safety lifecycle, then it is expected that adequate safety measures are already in place. The analysis has then a quantitative character in the sense that safety metrics (overall failure rate, diagnostic coverage values, safeness, etc.) are derived as an outcome of the procedure.

There are different analysis methods and techniques which are classified in the standard according to multiple criteria. For example, it is commonly differentiated between deductive and inductive approaches. A deductive analysis is performed top-down starting from problematic situations on system-level, and looking for the corresponding causes in the components. On the contrary, inductive analyses start bottom-up from local component malfunctions moving towards the corresponding effects on the complete system.

Many methodologies are addressed by the standard, for example Failure Modes and Effects Analysis (FMEA), Fault Tree Analysis (FTA), Reliability Block Diagrams (RBDs), and Dependent Failure Analysis (DFA).

Being the core topic of the thesis, safety analysis and its state of the art are extensively studied in a separate Subsection (2.2.7).

1.3.4 Fault Injection Techniques

Subsection 1.3.3 introduced safety analysis methodologies, where incorrect behaviors and undesired malfunctions and failures are inspected and analyzed based on more or less abstract representations of the considered system without any alterations. In addition to safety analysis, safety evaluation activities have a second key aspect at design levels. It is simulation-based and consists in the so-called *fault injection*.

Fault injection, an approach that has been widely applied during the last decades, is commonly defined as the deliberate insertion of faults into a system. The system in question is then monitored to determine its behavior in response to the introduced faults. In the ISO 26262 context, it is highly recommended for the two upper Automotive Safety Integrity Levels ASIL C and D to perform fault injection.

Technically, fault injection requires the (i) determination of appropriate injection spots in so-called *sensitivity zones*, where deficiencies are likely to have an impact on system safety, (ii) the specification of corresponding *observation* and *diagnostic* points, and (iii) the deployment of appropriate *workloads*.

Simulating system models which have been altered through fault injection campaigns makes it possible to experimentally provide evidence with respect to the safety level. In fact, system reactions are monitored through the observation points to identify the effects of the respectively injected faults on the overall safety and consequently confirm or refute analysis assumptions about their severity. Furthermore, faults detection and/or correction mechanisms are evaluated by measuring the *diagnostic coverage* values, which correspond to the percentage of captured faults at the diagnostic points out of all injected faults. The measured values are then either a validation of analysis assumptions or a trigger for necessary design hardening measures.

More details about fault injection and the different techniques that are applied to perform it are given in Subsection 2.2.8.

1.3.5 Correlations with Functional Verification

Subsections 1.3.3 and 1.3.4 respectively introduced the two major aspects of functional safety evaluation in the ISO 26262 context, namely safety analysis on the one hand and fault injection and simulation on the other hand.

Certainly, the ISO 26262 standard addresses these activities within the safety lifecycle and defines the work products which are expected from them quite independently from other activities. However, it does not prescribe their separation from the traditional development activities. On the contrary, it is common to find references to well-known development work products such as the *design specification*, the *testing plan*, or the *verification report*.

That is why, this subsection, investigates potential relationships of functional safety evaluation to functional verification.

It is necessary to acknowledge that there is no clear agreement within the ISO 26262 community and across the affected industries on whether those safety evaluation activities (analysis-based and simulation-based) can be aligned with traditional functional verification activities and/or integrated with them. This is actually the reason to use the term “evaluation” rather than “verification” when referring to the combination of both aspects.

Traditional functional verification is used to ensure that requirements have been met [24]. This holds obviously for functional safety evaluation too, as it is used to qualify the fulfillment of safety requirements. Nevertheless, when it comes to safety, the requirements that must be fulfilled go beyond the basic functionality of the considered system. They include the ability to monitor and control safety hazards, so that the safety measures implemented on top of the nominal functionality must be verified. Furthermore, while functional verification deals with inspecting functionality features within the normal conditions and constraints for which the system has been developed, functional safety evaluation requires to consider abnormal and off-nominal conditions too. Hence, the process of traditional functional verification is not sufficient to cover safety requirements and needs several enhancements to be so.

When it comes to safety analysis outcomes, a potential assimilation to functional verification plans might seem reasonable. In fact, to create a verification plan, knowledge about the functionality requirements and the design specification is needed. Based on this knowledge, appropriate regression data is created in order to be used during the verification itself. This regression data is a set of input values and corresponding reference output values reflecting the intended functionality.

Certainly, similar knowledge about requirements and design is expected from the safety analyst. However, it must be extended with an understanding of safety threats, underlying physical root causes, and appropriate counter-measures. Moreover, the set of data that the analyst creates, putting together potential failure modes, resulting failure effects, and accordingly assigned safety measures is just an intermediate step to a work product that can be later on assimilated to the regression data of the functional verification plan, namely a fault list to be injected into the design.

Thus, although the safety evaluation and the functional verification contexts are intrinsically related, it is not easy to mix them from a procedural perspective. It might however be beneficial to progressively bring safety evaluation activities into the traditionally used platforms for functional verification. This will certainly simplify the data exchange and reuse across both contexts. Some Electronic Design Automation (EDA) vendors are already working on such an approach, such as Cadence whose tool called “IFSS: Incisive Functional Safety Simulator” is aimed to enable the usage of sophisticated test benches to control the fault injection campaigns, support the debug process, and tackle efficiency challenges caused by using a modified Device Under Test (DUT) or a different simulation engine [25].

1.4 Motivation and Main Objectives

1.4.1 Motivation

As already mentioned in the previous subsections, the focus of this thesis is functional safety of E/E systems. Its particular significance in industrial applications, including automotive, is one of the key motivations of this work. Automotive suppliers, such as system and chip manufacturers, are also concerned by functional safety topics and play an important role in ensuring the targeted safety level.

In Subsections 1.3.2, 1.3.3, and 1.3.4, the two major functional safety evaluation aspects in the automotive context are explained. On the one hand, safety analysis is always required. On the other hand, fault injection and simulation is highly recommended for the most stringent safety integrity levels.

Appropriate procedures for functional safety evaluation are already in use across the development flow in compliance with the ISO 26262 standard guidelines. However, practical experience shows that there are still opportunities for further enhancements with respect to the efficiency of those procedures. Investigating those opportunities, especially with respect to the analytical aspect of the evaluation, is a key motivation point for this work.

Indeed, three essential challenges related to safety analysis are addressed in this thesis. First, analysis procedures and documentation structures are described in an informal way, making their understanding and deployment rather difficult (see Subsection 1.4.1.1). Then, several manual tasks are still required for safety analysis resulting in high costly and time-consuming efforts (more details in Subsection 1.4.1.2). Finally, an organizational and procedural gap persists between safety analysis on the one hand and fault injection and simulation on the other hand (see Subsection 1.4.1.3).

1.4.1.1 Informality and Subjectivity in Safety Analysis

One of the key issues in safety analysis is the frequently used informal description style for process steps, involved data artefacts, and underlying documentation structure. Such inconveniences make the analysis tedious and error-prone.

In fact, despite the considerable amount of research papers, practice reports, and even industry standards dealing with safety analysis, there is no concise and binding formal description of its scope, flow, and documentation structure. There is of course a common understanding about how the analysis shall be conducted and which data it must include. But the details remain open for interpretation resulting into the fact that safety analysis is often considered as a subjective human assessment. Consequently, safety analysts are tempted, even among the same organization, to adopt different analysis styles and data management schemes, as long as they remain compliant with the overall guidelines.

As a result, safety analysis work flows and products suffer from serious deficiencies with respect to structure, formality, and clearness. Moreover, systematic data exchange and reuse in the context of safety analysis remains limited because of the overhead efforts of transferring knowledge and understanding from one analyst or organization to another. In addition to that, the lack of formalization represents a major obstacle to implementing dedicated software tools for safety analysis with automation and systematic data processing capabilities. Thus, the costly manual character of the analysis procedure persists.

1.4.1.2 Costly Manual Tasks in Safety Analysis

Safety analysis is certainly a complex procedure which requires (i) comprehensive prior knowledge about the field, the design practices, and the application scenarios,

(ii) extensive brainstorming to identify safety threats, predict their impact, and assess their severity, as well as (iii) accurate calculation of safety-related metrics.

Traditionally, all safety analysis tasks are performed manually, leading to high effort and time costs. With the increasing complexity of the systems to be analyzed, the lengthy process becomes very inefficient and might lead to delivery and tape out delays. In today's industrial applications, fault trees consist of hundreds of nodes and failure modes and effects tables extend over thousands of lines. Such data sizes make the creation, exploration, and maintenance of safety analysis artefacts particularly hard.

The manual character of safety analysis has another negative aspect with respect to the data consistency and accuracy. Despite the careful and intensive work of safety engineers and analysts, mistakes cannot be excluded. Until now, a mandatory review has been the mitigation for such risks. However, because of the already mentioned complexity levels, the review is another delay factor and still cannot definitely guarantee the correctness and consistency of the outcome.

1.4.1.3 Gap Between Safety Analysis and Fault Injection

Safety analysis and fault injection are both required for functional safety evaluation of E/E systems whose safety integrity levels are expected to be particularly high. Nevertheless, there is still no established technique for systematic data exchange between them.

The differences between safety analysis and fault injection in terms of requirements, work styles, and purposes are the root causes of the diagnosed gap.

On the one hand, safety analysis aims at gathering safety evidence at different abstraction levels and design stages. Such safety evidence is achieved by (i) inspecting more or less detailed system descriptions with respect to their vulnerability to safety threats, (ii) compiling application-related data, (iii) assessing the fulfillment of safety requirements, and (iv) performing statistical calculations. The gathered safety evidence is then documented, commonly in form of graphical or tabular data representations, and subsequently provided to all relevant parties involved in the safety lifecycle (e.g., designers and/or managers within the same organization, qualification responsible, customers, etc.).

On the other hand, fault injection aims at validating system models and/or implementations with respect to safety at rather late design stages. This validation is achieved by (i) extending nominal design models/implementations with faulty configurations, (ii) observing their effects through simulation and monitoring, and (iii) finding out whether and/or to which extent the system is protected against those effects through safety measures. Hence, the outcome of fault injection and simulation

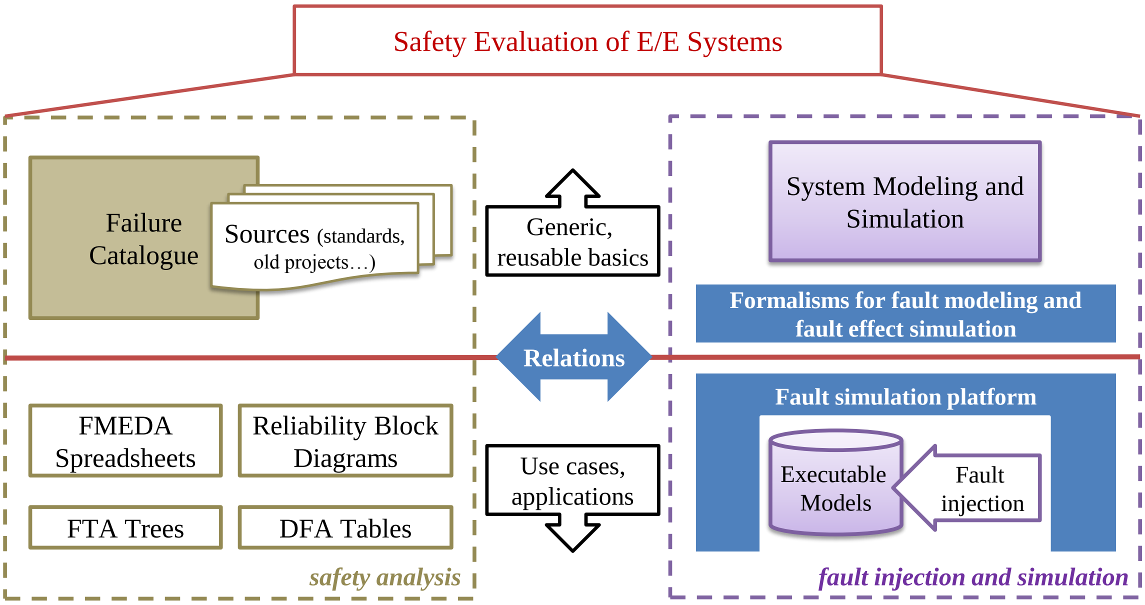

Safety Evaluation of E/E Systems
Failure Catalogue
Sources (standards, old projects…)
Generic, reusable basics
System Modeling and Simulation
Formalisms for fault modeling and fault effect simulation
Relations
Fault simulation platform
FMEDA Spreadsheets
Reliability Block Diagrams
Use cases, applications
Executable Models
Fault injection
FTA Trees
DFA Tables
safety analysis
fault injection and simulation

Figure 1.3: The Gap Between the Different Perspectives of Safety Evaluation

represents an indication to the developers about the design readiness with respect to safety and/or to the functional safety engineers about the accuracy and consistency of their analysis.

Figure 1.3, which gives an overview about both safety evaluation perspectives, differentiates between reusable generic basics and application-dependent use cases for each aspect. On the one hand, safety analysis in the industry relies on a so-called *failure catalogue* representing a database for all known failure modes, failure rates, and other data derived from relevant reliability handbooks or older projects. Fully constructed fault trees and filled FMEDA spreadsheets figure among the tangible reflections of the failure catalogue. On the other hand, fault injection and simulation relies on system modeling guidelines, extended through formalisms for fault modeling and fault effect simulation. These formalisms are applied on appropriate simulation platforms through fault injection campaigns affecting executable models.

The procedural divergence between safety analysis and fault injection as described above results often into an organizational separation between the respectively responsible teams. And as there is no seamless linking mechanism between both contexts, several challenges emerge with respect to data traceability, exchange, and transformation leading to long and error-prone safety evaluation cycles.

1.4.2 Main Objectives

The goals of this thesis consist essentially in tackling the challenges explained in Subsection 1.4.1. They consist in (i) developing a formalization and generalization approach for functional safety analysis (Subsection 1.4.2.1), (ii) offering tool-based support with enhanced automation capabilities in the analysis context (Subsection 1.4.2.2), and (iii) establishing a link between safety analysis and fault injection (Subsection 1.4.2.3).

1.4.2.1 Formalization and Generalization

To overcome the informality and subjectivity issues of safety analysis (see Subsection 1.4.1.1), formalization approaches are investigated in the context of this thesis.

Starting by a state of the art study of the main safety analysis techniques applied in the industry, an iterative approach is developed to formally describe respective procedures, involved data artefacts, and documentation structures.

The main objective of this approach is to instate a general, clear, and well-structured description of functional safety analysis, which serves as a valuable asset for organizing, handling, and re-using large amounts of safety-related data.

A key milestone in the intended approach is to substitute the traditionally used analysis formats (tables, spreadsheets, fault trees, etc.) by more robust and interchangeable assets. Nevertheless, the conventional inputs, outputs, and intermediate steps of the safety analysis procedure shall remain unchanged.

More detailed requirements for the formalization approach are elicited in Subsections 4.1.1 and 4.1.2. The adopted solution concepts to develop the approach are then investigated in Section 4.2.

1.4.2.2 Tool-Based Support and Automation Enhancements

To mitigate the drawbacks caused by the manual character of safety analysis, as described in 1.4.1.2, the feasibility and applicability of supporting tools are addressed in the context of this thesis.

Taking the formalization outcomes (see Subsection 1.4.2.1) as a starting point, software-based frameworks and tools are to be either programmed or generated.

Automated data extraction, processing, and generation are the main features that must be supported through these frameworks and tools. Thereby, cumbersome manual tasks are reduced and the quality and consistency of the resulting outcomes are improved.

More details about the requirements for the tool-based support of safety analysis activities are given in Subsections 4.1.3 and 4.1.5. The adopted solution concepts to achieve the targeted support features are investigated in Section 4.2.

1.4.2.3 Link Between Safety Analysis and Fault Injection

To address the problematic divergence between safety analysis and fault injection (see Subsection 1.4.1.3), syntactic and semantic similarities between them are investigated in this thesis. Based on those similarities, a systematic linking approach is developed.

The ultimate goal of the intended approach is to establish a balanced mixture of both scopes which takes advantage of the benefits and minimizes the drawbacks of each one of them.

The approach shall be oriented towards seamless data transfer, mapping, and/or transformation from safety analysis platforms to fault injection and simulation environments, as well as in the other way around. Bridging the gap is not only convenient, but also necessary to ensure a more reliable double-sided assessment of safety requirements satisfaction.

More detailed requirements for the linking approach are elicited in Subsection 4.1.4. The appropriate solution concepts for the approach development are addressed in Section 4.2.

1.5 Outline

The remainder of this thesis comprises seven chapters which are organized as follows.

Chapter 2 studies the state of the art of functional safety, first generically and then particularly in the automotive context.

Chapter 3 gives an overview of related academic and industrial contributions in the areas of safety analysis formalization, automation, and linking to fault injection and simulation.

Chapter 4 contains an elicitation of the overall requirements for the intended approaches of this thesis as well as a presentation of the solution concepts which are adopted to achieve them.

Chapter 5 is dedicated to the formalization approach which is developed in this work to address functional safety analysis.

Chapter 6 highlights the different frameworks which are developed in the context of this thesis to support functional safety analysis and link it to fault injection and simulation.

To illustrate the relevance of the developed approaches and frameworks, Chapter 7 presents the different case studies conducted during this work.

Finally, the contribution of the thesis is summarized in Chapter 8 which also gives an outlook on potential directions for future work.

2 State of the Art

This chapter presents the state of the art of safety evaluation. First, a general overview of related concepts, terms, application areas, and relevant standards is given in Section 2.1. Afterwards, detailed functional safety guidelines for automotive, the prevailing application domain of this thesis, are addressed in Section 2.2. They range from early risk assessment and requirements definition to elaborated safety analysis, verification, and validation procedures.

2.1 Functional Safety: A General Overview

A system is characterized as *safe* if it does not endanger human life or cause harm to the surrounding environment. Safety is often defined in the literature as the *"freedom of system operation from the occurrence of catastrophic failures"* [26].

In [27], three different aspects of system safety are identified: (i) *'primary safety'*, (ii) *'functional safety'*, and (iii) *'indirect safety'*. First, primary safety is related to risks that hardware operation might directly cause such as electric shocks and burns. Second, functional safety covers the safe operation of *equipment under control*. It depends on the risk-reduction measures and to what extent they correctly and efficiently behave. Finally, indirect safety concerns the consequences that an incorrect system operation could possibly have such as inducing poor medical judgment by an erroneously operated medical database [27, 28, 26].

Over the last three decades, functional safety has become a major topic in many industrial domains. Though, studying the early history of the topic shows that it initially emerged within the chemical process industry of the 1970s. In fact, the disastrous chemical plant explosion of June 1974 in Flixborough, UK is one of the initiating events for the constantly rising importance of functional safety. This explosion caused 28 deaths and 36 serious injuries. Furthermore, the sad incident which happened on July 10th, 1976 in Seveso, northern Italy brought the functional safety topic into focus again. The Seveso accident resulted in releasing 2 kg of the highly acid and toxic TCDD dioxin (complete chemical name: *2,3,7,8-Tetrachlorodibenzo-p-dioxin*) causing serious skin diseases, the death of about 70,000 animals, and serious environmental damage. The reason of the accident was an overheating reaction which caused the destruction of a safeguard by excessive pressure. The absence of automated cooling equipments, warning systems, and alarms made the incident particularly harmful.

The immediate governmental reaction was to revise the laws addressing the safety of such sites containing large amounts of hazardous substances. As a consequence, the so-called *Seveso Directive EC* was established in 1982 [29, 30]. Later on during the 1980s and 1990s, multiple related laws and regulations were successively adopted such as the CIMAH (Control of Industrial Major Accident Hazards) regulations in 1984, the EU Machinery Directive in 1989, and the Process safety management of highly hazardous chemicals regulation by the US Occupational Safety and Health Administration in 1992. Even though these regulations reflected the importance of functional safety in the industrial areas they addressed, they did not provide any specific requirements or formal guidelines for the assessment.

Nevertheless, progressive advancements with respect to the safety evaluation procedures were then achieved, mainly through the EN 1050 (Principles of risk assessment) which was released in 1996 and the machinery control EN 954-1 (safety-related parts of control system) in 1997. These norms provided basic guidance on risk assessment processes and gave some generic advice for risk reduction, but they were still very limited and insufficient with respect to the rising complexity levels.

Real-time and control systems were also increasingly relying on software and *programmable electronic systems* emerged within safety-related applications. Therefore, the need for a more formalized and detailed approach considering both hardware and software failure rates was acknowledged. Subsequently, appropriate activities took place during the 1990s within the IEC: International Electrotechnical Commission leading to the IEC 61508 standard for *Functional Safety of Electrical, Electronic, and Programmable Electronic Safety-related Systems* [1], whose first version was published in 1997. Even if the standard had a voluntary character at the beginning, its usage became more and more widespread during the last two decades, so that its application is nowadays considered as common practice and state of the art in the industry, especially after the publication of its second version in 2010 [30, 4].

The scope of functional safety in IEC 61508 includes a big range of equipment types. It covers for example industrial control systems, automotive systems, and medical equipment. A more detailed description of the IEC 61508 application areas as well as the corresponding standard derivatives is given in 2.1.2. But in general, every hardwired or programmable system whose failure is correlated, either singly or in combination with other failures, with a potential harm to human life (death or injury) or to the surrounding environment falls within the scope of IEC 61508 and is considered *safety-related*, *safety-relevant*, or *safety-critical*. It might be confusing that these three terms are frequently used within the research and industrial community without paying attention to the fine distinctions between them. However, it is common to use the *safety-critical* denomination for equipments whose failure alone endangers exposed persons, while *safety-related* and *safety-relevant* refer more generically to equipments whose failures increase the overall risk only when they concur with other failures affecting different items [1, 30, 26].

Concretely, the two major tasks of safety evaluation in the IEC 61508 context are (i) the identification of specific hazards which have serious consequences on human life and environment integrity and (ii) the assessment of the occurrence frequencies of the different potential equipment failures which may lead to those hazards. These frequencies must be within specific intervals, so that the related risk does not exceed the maximum tolerable level. In other words, it is not enough to say that the considered safety-related system contains dedicated protection measures to minimize potential risk. Beyond that merely informal statement, a formal assessment is required to show *evidence* of the capability of those measures to efficiently reduce the overall risk and reliably offer an adequate protection level [30].

Certainly, the ideal system, that every engineer aspires to create, would be protected against all possible risks, so that it would be qualified as a "zero risk" product. Nevertheless, in reality, there is always a *residual risk* that must be taken into account. A hardware component might be robust enough to avoid most malfunctions or recover from them in time before causing a critical failure with potential harm. However, it remains vulnerable to a subset of malfunctions so that the according *failure rate* is still greater than zero. Similarly, a piece of software can not predict all possible scenarios or contain handling routines for every erroneous situation that may occur during system lifetime. Furthermore, human mistakes can not be completely avoided during system conception, implementation, and verification because of the numerous tasks that are still performed manually. Therefore, instead of targeting that infeasible *zero risk*, safety evaluation is more about defining the *tolerable* or *acceptable risk* for the different system activities and providing evidence of the system capability to remain within the reasonable limits of that risk throughout the complete lifetime.

Perceiving a risk as acceptable depends on multiple factors, such as the deployment area, the number of persons that may be involved in the case of a hazardous event, and the severity or seriousness of the potential consequences. It should be noted here that risk categorization based on tolerability is correlated with a certain level of subjectivity. In fact, a risk deemed tolerable by an individual might be definitely unacceptable for another. The same also holds for different societies having dissimilar economic standards or divergent sets of ethical and moral values. Such differences are reflected in the legislation and regulation systems of the respective countries. For example, there are tangible divergences in laws related to the automotive, railway, and space sectors between Europe, the United States, and China [30, 28, 31].

In Subsection 2.1.1, an overview of the main concepts and terms used in the functional safety context with focus on the IEC 61508 is given. The application areas and the corresponding IEC 61508 derivatives are then presented in Subsection 2.1.2.

2.1.1 Basic Concepts and Terms – IEC 61508

The international standard IEC 61508 provides generic guidelines for the specification, design, and operation of safety-related E/E/PE systems. The standard is qualified as *performance-based* in the sense that it defines particular objectives and outcomes to be achieved with respect to safety evaluation. It is also a *risk-based standard*, i.e., it defines preventive requirements to reduce the risk of failure and keep it within a tolerable range. It should be noted that IEC 61508 has been developed in a comprehensive way: it defines general requirements and procedures and serves as basis for developing derived standards for different industry sectors [4].

Among the key characteristics of IEC 61508, the *safety lifecycle* approach provides a sequence of safety management stages during the system lifetime and forms the structuring basis for the standard itself. Furthermore, the dedicated safety lifecycle ensures addressing safety evaluation apart from classic functional assessment to avoid the assumption that a correct system functionality systematically means that the system is safe. In fact, unlike the traditional functionality requirements which address the *"good-case"*, i.e., the operating mode of the system, the additional safety requirements are separately defined, and verified with respect to the *"bad-cases"*, i.e., the failure conditions of the system. Though, safety evaluation activities are not completely disconnected from the classic design and manufacturing tasks producing an operational system. It is actually crucial to integrate all activities, including those related to safety evaluation, in a comprehensive development flow with globally defined rules, strategies, and perspectives [28].

The standard introduces a total of 16 phases in the overall safety lifecycle (see Figure 2.1). They range from the early *concept* (phase 1) and *scope definition* (phase 2) where a basic understanding of the *Equipment Under Control (EUC)* and its boundaries within its environment is aimed to the final *decommissioning or disposal* (phase 16). Among the intermediate phases, the *overall safety requirements* specification and *allocation* are particularly important, as they define adequate safety functions and appropriately assign them to specific parts of the designated E/E/PE system. During these phases, the targeted *safety integrity levels* are also defined and allocated to the specified safety functions. The safety lifecycle also addresses the realization of the specified safety requirements, i.e., the concrete implementation of the targeted safety functionalities (e.g., phase 10). Safety validation. i.e., assess whether the implemented E/E systems effectively meet the safety requirements within the targeted safety integrity levels is covered by phase 13. Moreover, operation, maintenance, repair, and modification activities are addressed by phases 14 and 15.

In [4], the IEC 61508 safety lifecycle phases are split into five main stages: (i) *Risk assessment*, (ii) *Planning*, (iii) *Design and construction*, (iv) *Operation and maintenance*, and (v) *Disposal*. The stage of each phase is marked in Figure 2.1.

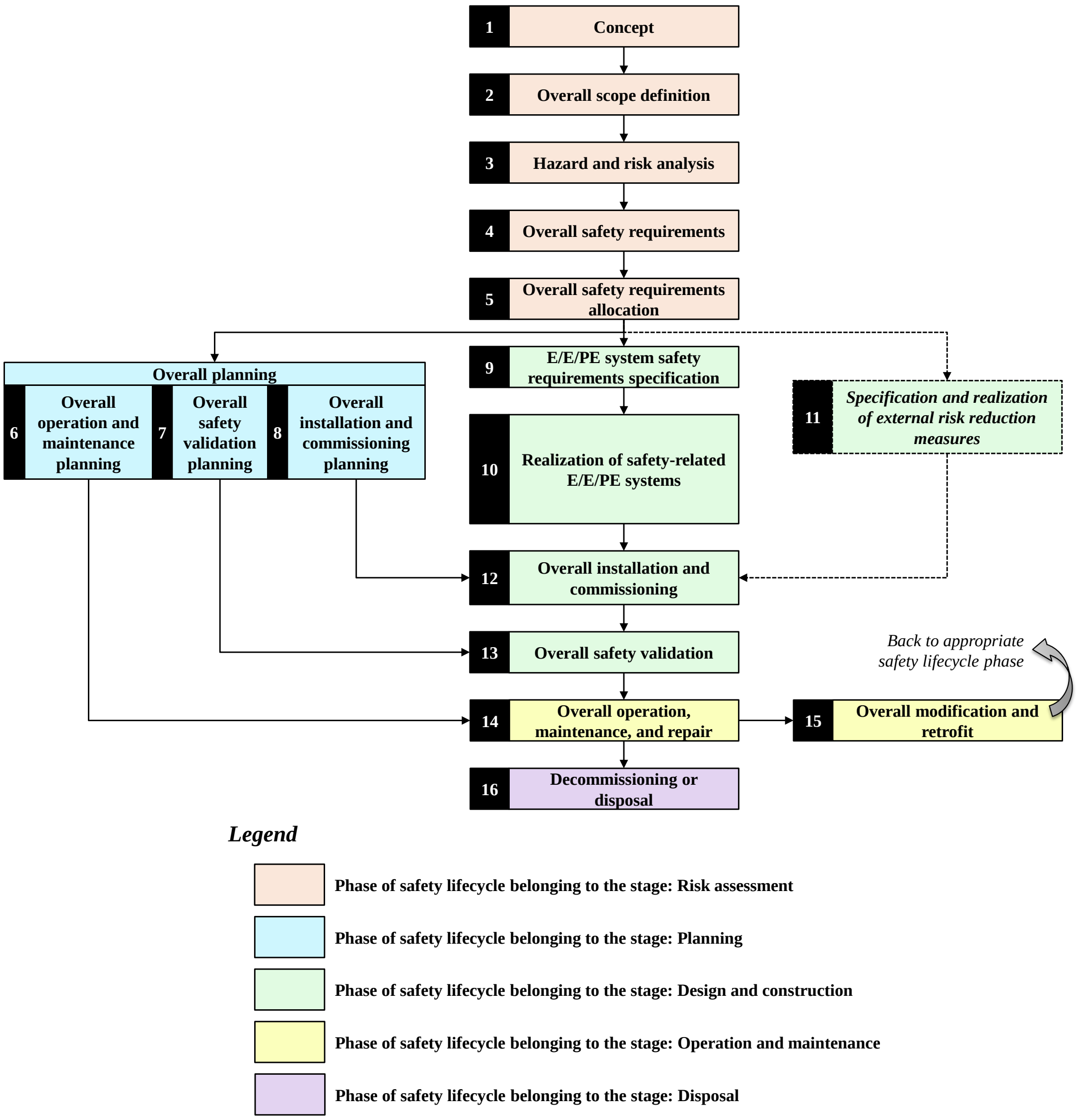

1 Concept
2 Overall scope definition
3 Hazard and risk analysis
4 Overall safety requirements
5 Overall safety requirements allocation
Overall planning
6 Overall operation and maintenance planning
7 Overall safety validation planning
8 Overall installation and commissioning planning
9 E/E/PE system safety requirements specification
10 Realization of safety-related E/E/PE systems
11 Specification and realization of external risk reduction measures
12 Overall installation and commissioning
13 Overall safety validation
Back to appropriate safety lifecycle phase
14 Overall operation, maintenance, and repair
15 Overall modification and retrofit
16 Decommissioning or disposal
Legend
Phase of safety lifecycle belonging to the stage: Risk assessment
Phase of safety lifecycle belonging to the stage: Planning
Phase of safety lifecycle belonging to the stage: Design and construction
Phase of safety lifecycle belonging to the stage: Operation and maintenance
Phase of safety lifecycle belonging to the stage: Disposal

Figure 2.1: The Overall Safety Lifecycle in IEC 61508 [1, 4]

Among the seven parts of the IEC 61508 standard [1], Part 1 to Part 4 are normative while the three remaining parts are informative. Part 1 (*General requirements*) defines the overall safety lifecycle. Part 2 (*Requirements for electrical/electronic/programmable electronic safety-related systems*) defines the development objectives for functionally safe E/E/PE systems with respect to hardware as well as software, which is further on addressed in Part 3 (*Software requirements*). A recapitulation of all terms used in the context of safety evaluation in the standard is given in Part 4 (*Definitions and abbreviations*).

The definitions are grouped under eight *general headings* in Part 4: 1) *Safety terms*, 2) *Equipment and devices*, 3) *Systems – general aspects*, 4) *Systems – safety-related aspects*, 5) *Safety functions and safety integrity*, 6) *Fault, failure, and error*, 7) *Lifecycle activities*, and 8) *Confirmation of safety measures.* Selections of the most important terms related to those groups are respectively given in Tables 2.1 to 2.8 along with their definitions as given by the IEC 61508. A more extensive list of terms is given in the Glossary (see Chapter 9).

Harm	damage to human health, to property, or to the environment
Hazard	potential source of harm
Risk	combined occurrence probability of harm with its severity
Safety	freedom of unacceptable risk
Safe State	state of the equipment under control when safety is achieved

Table 2.1: Selected Definitions of IEC 61508 – Safety terms [1]

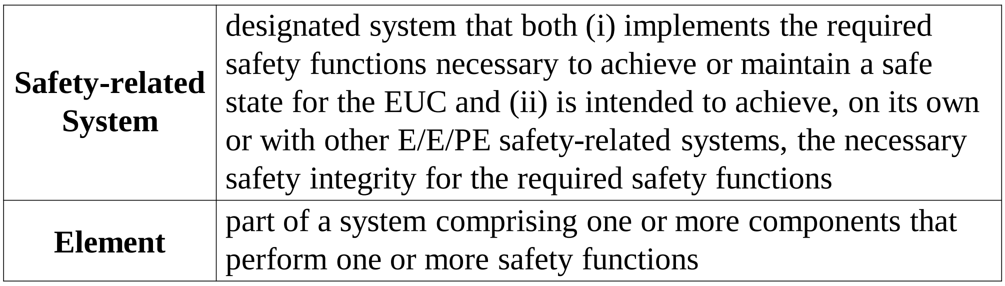
Equipment Under Control (EUC)	equipment, machinery, apparatus or plant used for manufacturing, process, transportation, medical activities, etc.
Functional Unit	entity of hardware or software, or both, which is capable to accomplish a specified purpose
Application	task related to the EUC

Table 2.2: Selected Definitions of IEC 61508 – Equipment and devices [1]

Electrical / Electronic / Programmable Electronic System (E/E/PE System)	system for control, protection or monitoring based on one or more E/E/PE devices. It includes power supplies, sensors, communication paths, and actuators
Architecture	configuration of HW/SW elements in a system

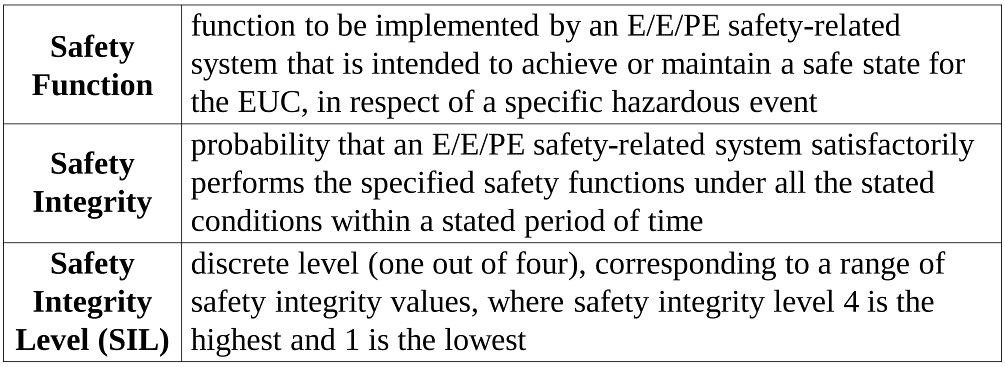
Table 2.3: Selected Definitions of IEC 61508 – Systems (general aspects) [1]

The three informative parts of the IEC 61508 give general guidelines about the procedures and methodologies to be followed in order to satisfy the requirements of the normative parts. For example, Part 5 (*Examples of Methods for the Determination of safety integrity levels*) provides examples of risk analysis techniques and illustrates how to allocate SILs (safety integrity levels).



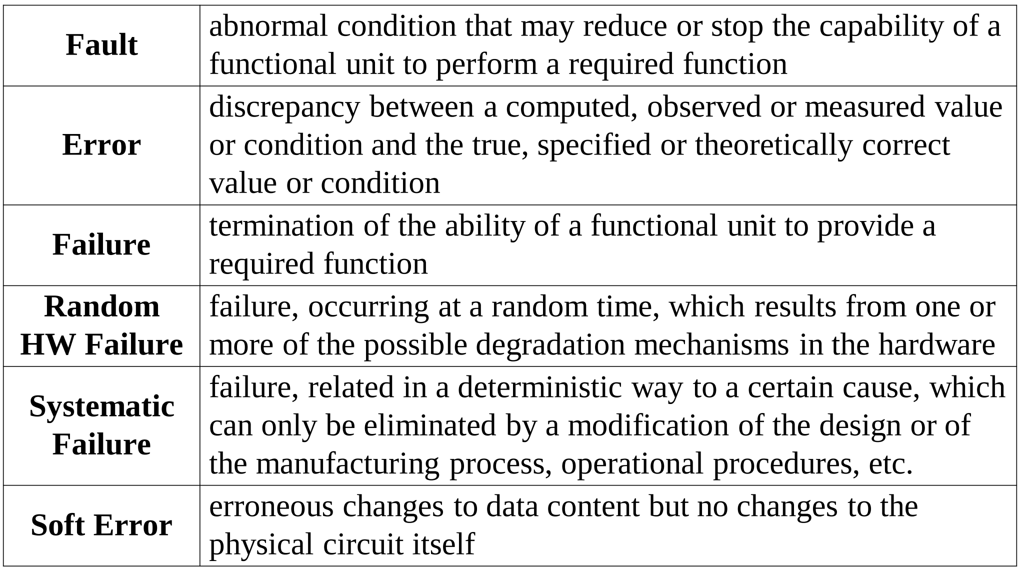
Safety-related System	designated system that both (i) implements the required safety functions necessary to achieve or maintain a safe state for the EUC and (ii) is intended to achieve, on its own or with other E/E/PE safety-related systems, the necessary safety integrity for the required safety functions
Element	part of a system comprising one or more components that perform one or more safety functions

Table 2.4: Selected Definitions of IEC 61508 – Systems (safety-related aspects) [1]



Safety Function	function to be implemented by an E/E/PE safety-related system that is intended to achieve or maintain a safe state for the EUC, in respect of a specific hazardous event
Safety Integrity	probability that an E/E/PE safety-related system satisfactorily performs the specified safety functions under all the stated conditions within a stated period of time
Safety Integrity Level (SIL)	discrete level (one out of four), corresponding to a range of safety integrity values, where safety integrity level 4 is the highest and 1 is the lowest

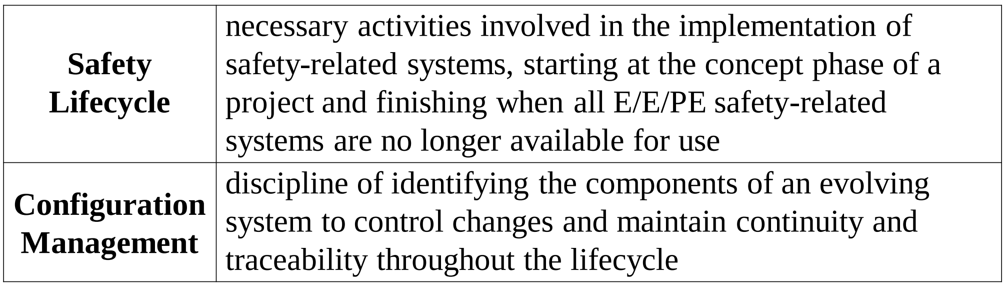
Table 2.5: Selected Definitions of IEC 61508 – Safety functions and safety integrity [1]



Fault	abnormal condition that may reduce or stop the capability of a functional unit to perform a required function
Error	discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition
Failure	termination of the ability of a functional unit to provide a required function
Random HW Failure	failure, occurring at a random time, which results from one or more of the possible degradation mechanisms in the hardware
Systematic Failure	failure, related in a deterministic way to a certain cause, which can only be eliminated by a modification of the design or of the manufacturing process, operational procedures, etc.
Soft Error	erroneous changes to data content but no changes to the physical circuit itself

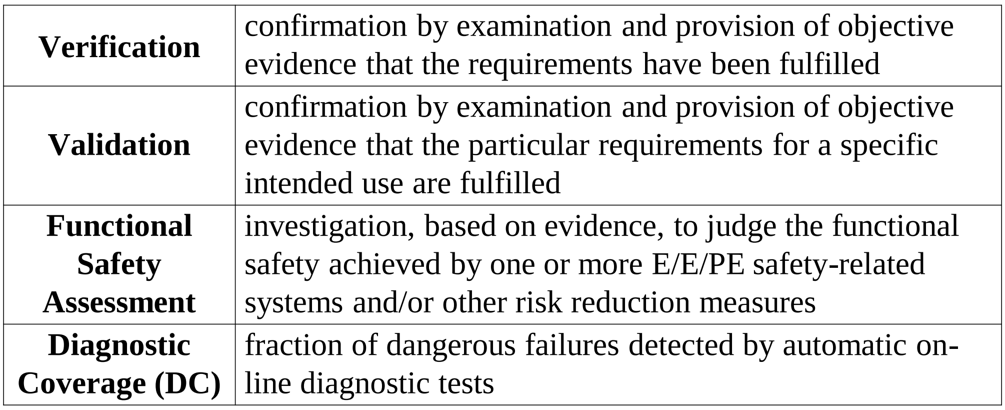
Table 2.6: Selected Definitions of IEC 61508 – Fault, failure, and error [1]

To summarize, the IEC 61508 standard is a basic guide for designers and manufacturers dealing with safety-related systems. It outlines the overall workflow that should be fulfilled to claim acceptable safety integrity levels for the intended use cases. The risk-based approach provided by the IEC 61508 in the late 1990s represented an alternative to the state of the art design methodology back then, which basically consisted in investing all efforts in optimizing the system design to the maximum extent possible and assuming that it would be then automatically safe. Instead of that, IEC 61508 calls for acknowledging that inevitable risks are still likely to happen, mainly because of random failures affecting hardware or systematic failures caused by poor design choices or imperfect software behavior. By studying and understanding these risks, appropriate safety requirements are specified. The fulfillment of these requirements is subsequently verified and validated throughout the safety lifecycle and a functional safety assessment is conducted to prove the targeted risk reduction.



Safety Lifecycle	necessary activities involved in the implementation of safety-related systems, starting at the concept phase of a project and finishing when all E/E/PE safety-related systems are no longer available for use
Configuration Management	discipline of identifying the components of an evolving system to control changes and maintain continuity and traceability throughout the lifecycle

Table 2.7: Selected Definitions of IEC 61508 – Lifecycle activities [1]



Verification	confirmation by examination and provision of objective evidence that the requirements have been fulfilled
Validation	confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled
Functional Safety Assessment	investigation, based on evidence, to judge the functional safety achieved by one or more E/E/PE safety-related systems and/or other risk reduction measures
Diagnostic Coverage (DC)	fraction of dangerous failures detected by automatic on-line diagnostic tests

Table 2.8: Selected Definitions of IEC 61508 – Confirmation of safety measures [1]

Considering functional safety as a central concept which is applicable to all industry sectors, IEC 61508 has been developed in a generic way, so that it represented the basis to create multiple sector-specific standards. An overview of these IEC 61508 derivatives is given in the following Subsection 2.1.2.

2.1.2 Application Areas and Related Standards

In compliance to the generic functional safety guidelines given by the IEC 61508, several sector-specific standards have been drafted during the last two decades. They are tailored to the respective industry sectors and take specific technical conventions and customs into account. Figure 2.2 gives an overview of functional safety standards related to IEC 61508 and applied in different industry sectors.

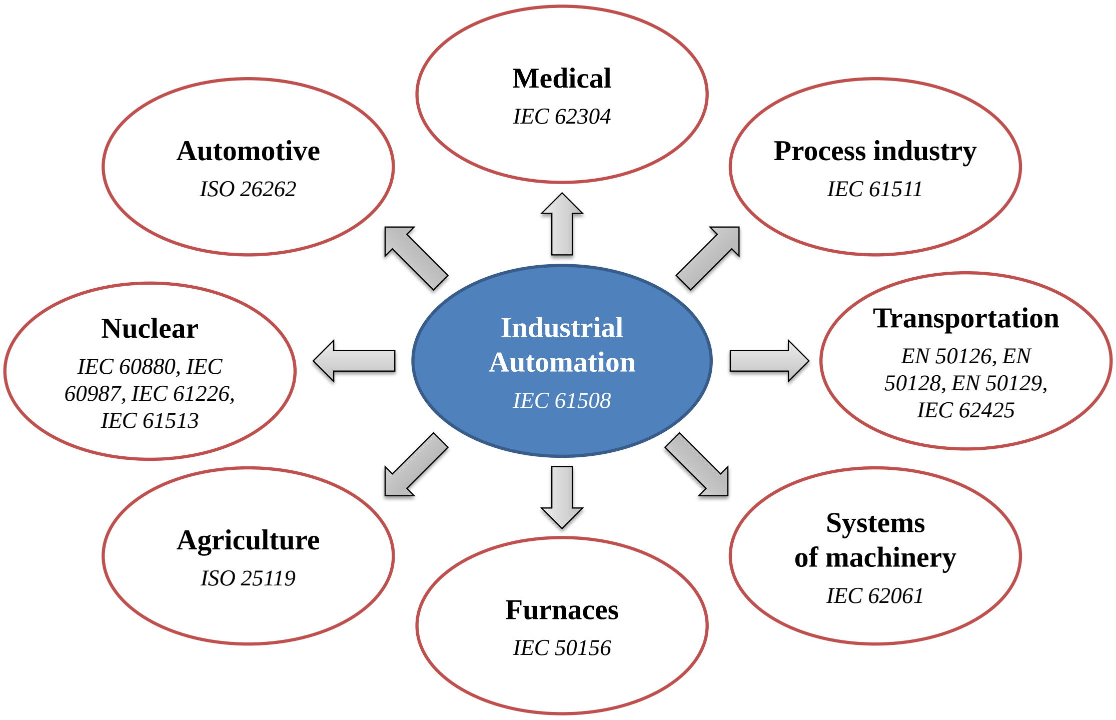


Figure 2.2: Overview of Functional Safety Standards

First, the IEC 61508 principles are reflected in the IEC 61511 standard: *Functional Safety – Safety instrumented systems in the process industry sector* (2003) which addresses so-called *safety-instrumented systems*, including sensors, logic solvers, and actuator elements, applied in multiple kinds of manufacturing processes, such as chemical, oil, and gas industries.

Second, for systems of machinery, IEC 62061: *Safety of machinery – Functional safety of safety-related electrical, electronic and programmable electronic control systems* (2005) is derived from IEC 61508 and is harmonized with ISO 13849 – *Safety of machinery – safety-related parts of control system* (2006), also dealing with E/E/PE based control systems in machinery [4].

For the nuclear industry, IEC 61513: *Nuclear power plants – Instrumentation and control important to safety* (2005) addresses so-called *instrumentation and control I&C* systems. These are based on E/E/PE equipment and ensure critical service and monitoring functions to the operation of the nuclear power plant.

Then, for railway transportation, there are IEC 61508 related standards such as IEC 62278 (EN 50126): *Railway applications - The specification and demonstration of reliability, availability, maintainability and safety* (1999) and IEC 62425 (EN 50129): *Railway applications – Communication, signalling and processing systems – Safety related electronic systems for signalling.* Although these two standards are not directly derived from IEC 61508, their application is considered as a sufficient measure to fulfilling IEC 61508 requirements [4].

In the agricultural sector, IS0 25119: *Tractors and machinery for agriculture and forestry – safety-related parts of control systems* (2010) is applicable to E/E/PE systems used in the field. Through a prescriptive risk-based mitigation approach, the standard specifically addresses agriculture and off-road systems, such as industry transport and mining [32].

Furthermore, for the medical sector, IEC 62304: *Medical device software - Software life cycle processes* (2006) addresses lifecycle requirements for the development of medical software and software within medical devices and is also concerned with functional safety issues. For example, IEC 62304 defines three classes of medical software safety: (i) Class A: No injury or damage to health is possible, (ii) Class B: Non-serious injury is possible, and (iii) Class C: Death or serious injury is possible.

Finally, the sector-specific interpretation of the IEC 61508 for road vehicles is the ISO 26262 standard: *Road vehicles – Functional safety* (2011). The standard addresses series production passenger cars up to 3 500 kilograms and represents the state of art guide to avoid systematic and/or random failures in automotive E/E/PE equipment by deriving appropriate requirements and conducting adequate processes.

Divergent types of target systems is one of the key differences between IEC 61508 and ISO 26262. In fact, IEC 61508 rather targets systems produced in low volumes while ISO 26262 deals with the high-volume mass-market automotive industry. In the context of IEC 61508, the safety-related system is built, tested, installed on the plant, then validated with respect to functional safety. However, in compliance to ISO 26262, safety validation must be performed *before* series production because of the high volume. Furthermore, instead of the Safety Integrity Level (SIL 1 – SIL 4) defined by the IEC 61508 standard as a measure for the goodness of implemented safety functions, ISO 26262 introduces *Automotive Safety Integrity Level* (ASIL A – ASIL D) with respect to safety goals and their respective violations [33, 34]. It should be noted that SIL 1 is commonly mapped to ASIL A, SIL 2 to ASIL B or C, and SIL 3 and 4 to ASIL D. In this thesis, the focus is mainly on the ISO 26262 standard and its relevance for the automotive industry and its suppliers, including system and semiconductor manufacturers. Therefore, a detailed study of the standard is conducted in Section 2.2.

2.2 Automotive Functional Safety: ISO 26262

The ISO 26262 standard is the adaptation of IEC 61508 for the specific needs of the automotive industry. Its first version has been in use since 2011 and its revised second edition is available since the end of 2018. In the standard, all activities introduced in the safety lifecycle of IEC 61508 (see Subsection 2.1.1) are addressed with more automotive-specific technical details and an appropriately adapted nomenclature. Indeed, in comparison to IEC 61508 where the focus is on safety functions and their safety integrity levels, ISO 26262 focuses on potential *violations of safety goals* and how they might harm car passengers. In terms of organization and vocabulary, ISO 26262 is tailored to fit the automotive sector and build upon the applied state-of-the-art development flow within the car manufacturing industry and its numerous supplier industries. For example, the terms *item, system, element, component, hardware part, and software unit* are used in the ISO 26262 scope to depict the several levels of composition that an automotive product has and to accordingly organize and distribute the required safety-related activities across all involved parties.

The framework provided by ISO 26262 is most importantly concerned with functional safety of E/E/PE systems, even if it can be further on applied for safety-related systems based on other technologies [2]. It defines the automotive safety lifecycle including the (i) management, (ii) development, (iii) production, (iv) operation, (v) service, and (vi) decommissioning phases with the ultimate purpose of determining integrity levels and demonstrating the elimination of unreasonable residual risk.

The standard contains ten parts covering all required activities to ensure functional safety in the automotive sector. In Part 1: *Vocabulary*, the terminology of the standard is provided. Many terms have the same definitions as in IEC 61508 (see Subsection 2.1.1), but there is also a considerable amount of new sector-specific definitions (e.g., *safety goal*, *functional/technical safety concept*, *functional/technical safety requirement*, etc.). Part 2: *Management of functional safety* gives the requirements that must be fulfilled from a management perspective. These requirements are classified in *project-independent* requirements concerning the organizations involved and *project-specific* requirements addressing the concrete management activities during the safety lifecycle. Part 3: *Concept phase* specifies the activities to be performed during the concept phase such as the *item definition*, the *Hazard Analysis and Risk Assessment (HARA)*, and the *functional safety concept.* Parts 4, 5, and 6 deal with the development phases. In Part 4: *Product development at the system level*, the specification of technical safety requirements at system level, as well as the system design, integration and testing, and safety validation are thoroughly described. It should be noted here that a *system* consists of at least one sensor, one controller, and one actuator according to the ISO 26262 terminology [2]. Part 5: *Product development at the hardware level* and Part 6: *Product development at the software level* go further on in detail respectively for hardware and software elements of the system. In Part 7: *Production and operation*, service and decommissioning are also considered in addition to production and operation as the part name already insinuates. Among the activities addressed by Part 8: *Supporting processes*, configuration management, verification, documentation, and qualification are mentioned. Part 9: *Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses* is concerned with issues like ASIL decomposition and tailoring, failure dependencies, and recommended techniques for safety analysis. Finally, Part 10: *Guideline on ISO 26262* provides a summarizing overview of the standard, gives additional explanations, and enhances the understanding of the principles stated by the other parts through a set of examples.

2.2.1 Overall Safety Lifecycle

The ISO 26262 standard prescribes the safety lifecycle represented in Figure 2.3. The key safety activities required during the complete lifecycle of a safety-related automotive product, ranging from concept to decommissioning, are illustrated in Figure 2.3. In general, the safety lifecycle can be divided in three stages: (i) *concept phase*, (ii) *product development*, and (iii) *after the release for production.* It also indirectly reflects the three key parts of the overall functional safety flow (see 1.3.2),: (i) requirements specification (e.g., functional safety concept), (ii) design (e.g., item definition), and (iii) test (e.g., safety validation).

It should be noted that safety management tasks are distributed through all lifecycle stages. These tasks are repetitively emphasized in the standard because of their central role in planning, coordinating, tracking, and documenting all functional safety activities [2].

Another important concept in the ISO 26262 with respect to the safety lifecycle is *safety culture*. Defined in Part 1 as the *"policy and strategy used within an organization to support the development, production and operation of safety-related systems"*, safety culture is one of the basic requirements that must be fulfilled by organizations involved in the safety lifecycle in order to claim compliance with ISO 26262. Concretely, appropriate trainings, rules, processes, and reporting mechanisms must be established to affectively support the functional safety achievement and efficiently communicate any potential anomalies or deficiencies to the responsible persons [2].

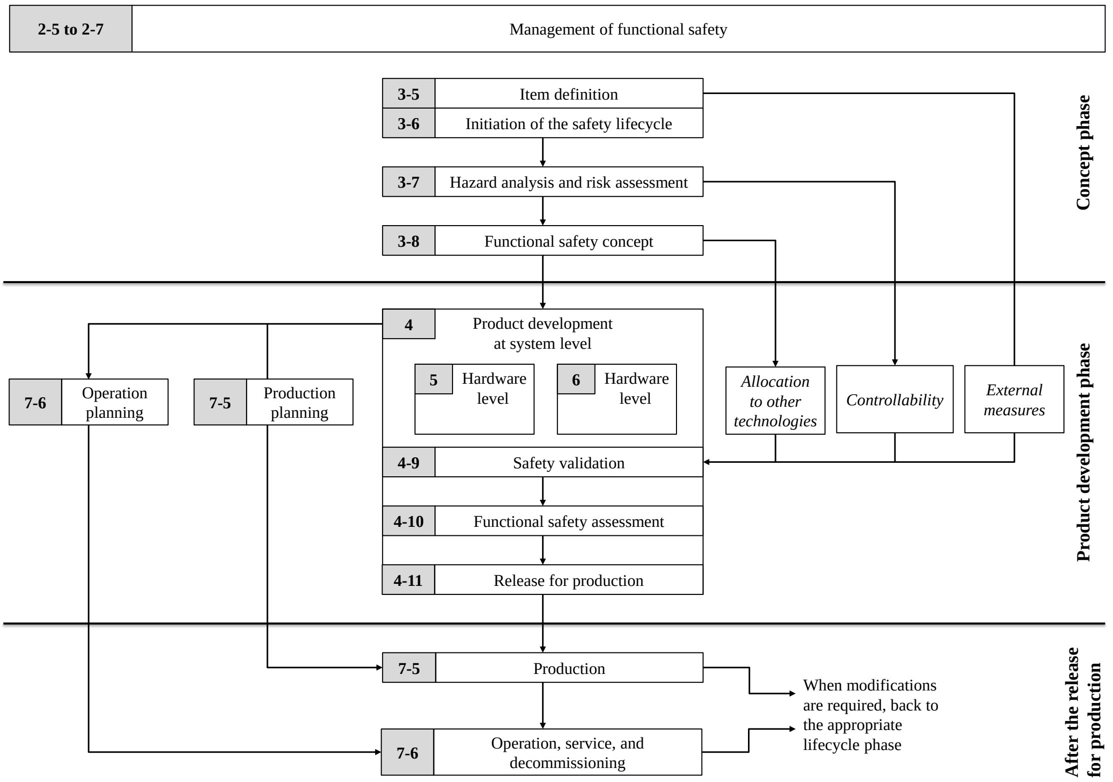

2-5 to 2-7
Management of functional safety
3-5
Item definition
3-6
Initiation of the safety lifecycle
3-7
Hazard analysis and risk assessment
3-8
Functional safety concept
Concept phase
4
Product development at system level
5
Hardware level
6
Hardware level
7-6
Operation planning
7-5
Production planning
Allocation to other technologies
Controllability
External measures
4-9
Safety validation
4-10
Functional safety assessment
4-11
Release for production
Product development phase
7-5
Production
7-6
Operation, service, and decommissioning
When modifications are required, back to the appropriate lifecycle phase
After the release for production

Figure 2.3: Safety Lifecycle in ISO 26262 [2]

Here are selected lifecycle phases which are particularly relevant for this thesis:

- **Item definition**: Initiates the safety lifecycle, it is mandatory to define the *item*. An item description including its functionality, interfaces, environmental conditions, and known hazards is required.
- **Hazard Analysis and Risk Assessment (HARA)**: hazards and hazardous events that must be prevented, mitigated, or controlled are identified in this phase (more details in 2.2.2). Appropriate Automotive Safety Integrity Levels (ASILs) are assigned to the identified hazards, depending on their probabilities of exposure, their controllability, and their severity (see 2.2.3). Additionally, associated *safety goals* (to be detailed in 2.2.4) are formulated and characterized through the adequate ASILs.
- **Functional safety concept**: It is based on the identified safety goals. Considering so-called *preliminary architectural assumptions*, the functional safety concept is created as a statement of the required functionality to achieve the safety goal(s) (see 2.2.5.1). It contains the *functional safety requirements* and

their allocation to the item elements (more details in 2.2.5.2).

- **Product development at system level**: Once the functional safety concept is available, the item development from at system level is performed following a V-model. On the left branch, *technical safety requirements* (explained in 2.2.5.4), a general system architecture, a *technical safety concept* (see 2.2.5.1) including the allocation of technical safety requirements to the system architecture elements, and the system design specification and implementation are consecutively performed. On the right branch, integration, verification, validation (see 2.2.6), and functional safety assessment tasks are required.

- **Product development at hardware level**: Hardware parts are also developed in a V-model process, using the system design specification resulting from the previous phase. This includes the specification of *hardware safety requirements* as well as the hardware design and implementation on the left hand branch. On the right hand branch, the hardware integration and testing are performed.

- **Product development at software level**: Similarly to hardware, software units are developed following a V-model process. This includes the specification of *software safety requirements* as well as the software architectural design and implementation on the left hand branch. On the right hand branch, software integration and testing as well as verification of software requirements are performed.

- **Safety validation**: This concept is defined in the standard as the *"assurance, based on examination and tests, that the safety goals are sufficient and have been achieved"* [2]. More details are given in 2.2.6.

Functional safety evaluation is addressed by the ISO 26262 standard throughout the different parts and during the several stages of the safety lifecycle. Multiple techniques for safety analysis, verification, and assessment are recommended by the standard. Therefore, two subsections are dedicated in this chapter to:

1. *safety-oriented analyses* which examine and estimate the extent of functional safety achievement either qualitatively or quantitatively at different phases of the product development (system level, hardware level, and software level) (see 2.2.7), and to

2. *fault injection and simulation* which represent recognized approaches to qualify the fulfillment of safety requirements using executable system models or even hardware prototypes (more details in 2.2.8).

2.2.2 Hazard Analysis and Risk Assessment

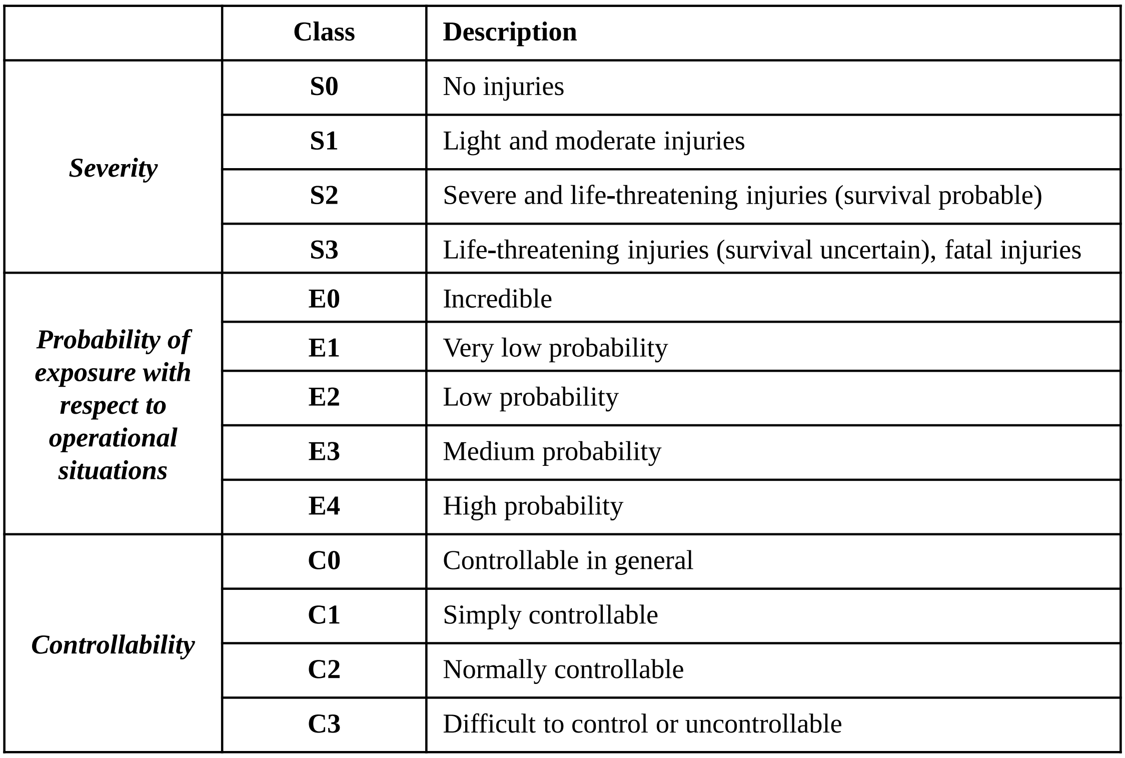
According to the ISO 26262 standard, Hazard Analysis and Risk Assessment (HARA) is an important activity which is performed during the concept phase of the safety lifecycle. It is defined as a *"method to identify and categorize hazardous events of items and to specify safety goals and ASILs related to the prevention or mitigation of the associated hazards in order to avoid unreasonable risk"* [2].

In other words, the HARA is aimed to identify and classify the hazards that can be triggered by item malfunctions. All potential *hazardous events* are determined and appropriately rated using the three so-called *impact factors*: (i) severity, (ii) probability of exposure, and (iii) controllability. Therefore, in addition to the item definition, an *impact analysis* may also be used as a further supporting input to perform the HARA. It should be noted that the item must be considered without any internal safety mechanisms for the HARA. Intended or already implemented safety mechanisms must be ignored at this point of the safety lifecycle in order to accurately evaluate the possible dangerous situations that might happen during the product lifetime and to efficiently specify and implement the appropriate safety mechanisms later on during the development stages.

From a procedural point of view, the HARA starts with a *situation analysis* where the operational situations and operating modes of the item are described. It should be noted that unreasonable driving behaviors (e.g., traveling cross-country at high speed) are not within the limits of those operational situations in which guaranteeing functional safety of the item is mandatory [2].

After situation analysis, *hazard identification* is required. Many techniques such as checklists, field studies, and quality history examinations are applied to determine the hazards and clearly describe them in terms of misbehaviors at the vehicle level. The root causes behind those hazards are not within the scope of the HARA as they will be evaluated in later steps using other analysis techniques. Relevant combinations of operational situations and hazards represent the *hazardous events* mentioned earlier. The consequences of such events must be identified and as long as they are within the scope of the ISO 226262 standard (related to E/E/PE equipment), they have to be classified regarding the impact factors. This is precisely the risk assessment part of the activity which focuses on potential harm to all involved persons who may be at risk at a given situation (driver, vehicle passenger, pedestrians, persons driving other vehicles, etc.).

Each hazard is assigned a severity class ranging between S0 (no injuries) and S3 (life-threatening or fatal injuries). Example of S0 hazards are bumps with roadside infrastructure, light collisions, and leaving the road without collision or rollover, while a side collision with a passenger car with medium speed is classified as S3 because more than 10% of such accidents lead to critical or extremely critical (including fatal) injuries [2]. The list of all classes of severity and their respective descriptions is given in Table 2.9.



	Class	Description
Severity	**S0**	No injuries
	S1	Light and moderate injuries
	S2	Severe and life-threatening injuries (survival probable)
	S3	Life-threatening injuries (survival uncertain), fatal injuries
Probability of exposure with respect to operational situations	**E0**	Incredible
	E1	Very low probability
	E2	Low probability
	E3	Medium probability
	E4	High probability
Controllability	**C0**	Controllable in general
	C1	Simply controllable
	C2	Normally controllable
	C3	Difficult to control or uncontrollable

Table 2.9: Classes of Severity, Classes of Probability of Exposure, and Classes of Controllability in ISO 26262 [2]

A similar categorization is required for operational situations based on the probability of exposure. The standard distinguishes five classes of probability of exposure (E0 – E4). For example, extremely unusual situations are categorized as E0: *incredible* (e.g., facing an obstacle in the highway lane or driving downhill with engine off), while situations that are very likely to happen to any car and for any driver are categorized as E4: *high probability* (e.g., acceleration, deceleration, steering, lane changing, etc.,) [2]. Table 2.9 gives the complete list of exposure classes.

The third impact factor, which is controllability, is used to categorize the hazardous event and assign one of the classes C0, C1, C2, and C3 (see Table 2.9). It should be noted that a hazardous event is deemed controllable only if the driver or another person potentially at risk are able to sufficiently gain control of the event, so that the harm is avoided [2]. An example of C0 hazardous event would be an unexpected increase of the radio volume or a warning message about the gas level. Such events are actually generally controllable by almost every driver. C3 hazardous events are those which are very difficult to control (i.e., less than 90% of drivers or other people at risk are able to avoid the potential harm). Examples are: a failure of the brakes, an incorrect steering angle at high speed, and faulty airbag releases.

After these classifications, the ASIL level of each hazardous event is determined (see 2.2.3) and the associated safety goals are formulated (see 2.2.4). The standard also prescribes a verification step within the HARA, which is concretely a review check with respect to several requirements including (i) the completeness of situations and hazards, (ii) the compliance with the item definition, and (iii) the consistency of the ASIL assignments. Different person(s) from another department or organization than the item developers must perform the review to obtain a trustworthy confirmation or rejection of the HARA outcomes.

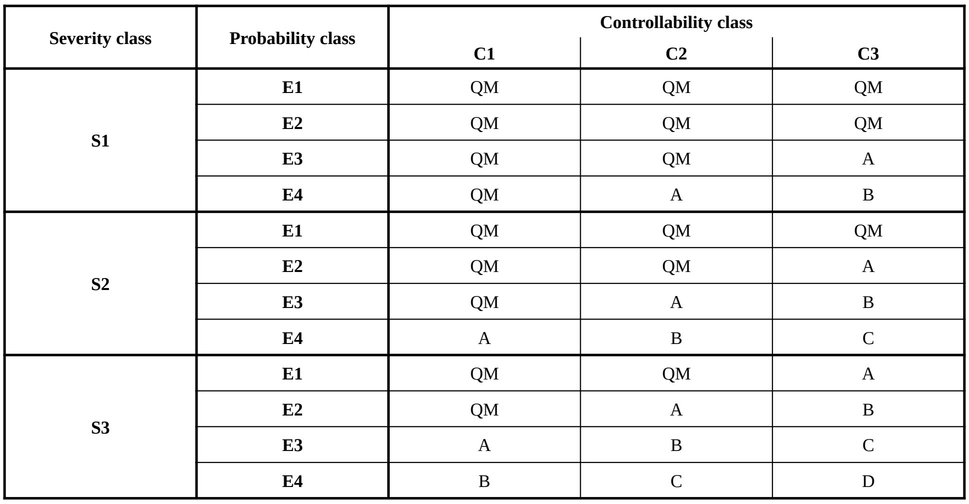
2.2.3 ASIL: Automotive Safety Integrity Level

For hazardous events determined by the Hazard Analysis and Risk Assessment (HARA) presented in 2.2.2, respective ASILs are determined using the parameters severity, probability of exposure, and controllability. ASIL stands for "Automotive Safety Integrity Level" and is defined by the standard as *"one of four levels (A – D) to specify the item's or element's necessary requirements of ISO 26262 and safety measures to apply for avoiding an unreasonable residual risk, with D representing the most stringent and A the least stringent level"* [2].

The classes of severity, probability of exposure, and controllability (see Table 2.9) are considered for ASIL assignment according to Table 2.10. For S0 hazards, E0 operational situations, and C0 hazardous events, no ASIL assignment is required by ISO 26262, as no serious harm can be caused. Moreover, certain combinations lead to the categorization of a given hazardous event as QM (quality management). For such events, no further action is required by the ISO 26262 safety guidelines and subsequently no safety mechanisms are expected. The avoidance of these situations is actually addressed by traditional quality measures.

The ASIL assigned to a hazardous event is also a characteristic of the associated safety goal. For the derived functional safety requirements, and later on the technical, hardware, and software safety requirements, the ASIL is accordingly inherited.

Another ASIL-related concept, which is worth mentioning here, is the so-called *ASIL decomposition* (also referred to as *ASIL tailoring*). Defined in the standard as the *"apportioning of safety requirements redundantly to sufficiently independent elements, with the objective of reducing the ASIL of the redundant safety requirements that are allocated to the corresponding elements"* [2], ASIL decomposition is aimed at a trade-off between redundancy and safety integrity level stringency and is inherently related to the allocation of safety requirements to architectural elements.



Severity class	Probability class	Controllability class		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Table 2.10: ASIL Determination Table in ISO 26262 [2]

In fact, with respect to the hierarchy of safety requirements, the ASIL is first, as already mentioned, an attribute of the safety goal. Then, it is inherited by the subsequent safety requirements which are allocated to architectural elements ranging from preliminary architectural assumptions at the initial steps of the product development at system level to the final hardware and software elements considered at later design and implementation stages. During this allocation and in the cases of sufficient independences between architectural elements, ASIL decomposition can be applied by redundantly implementing the considered safety requirements by the associated independent elements and subsequently assigning a potentially lower ASIL to the decomposed safety requirements. The precise conditions of decomposition applicability and the corresponding decomposition schemes are detailed in Part 9 of the ISO 26262 standard [2]. To give an example, two redundant ASIL-C CPUs form together an ASIL-D computer platform.

2.2.4 Safety Goals

The exact definition is given by the ISO 26262 standard as follows: a safety goal is a *"top-level safety requirement as a result of the Hazard Analysis and Risk Assessment (HARA)"*. In other words, the list of safety goals complies to the item and is a work product of the HARA performed at the concept phase (see 2.2.2). It should be noted that the relationship between safety goals and hazards is *one to many* in both directions, i.e., one specific safety goal can be associated with multiple hazards and multiple safety goals can be related to one specific hazard.

The determination procedure of safety goals relies first of all on the list of identified hazardous events. For each hazardous event which has been assigned an ASIL from A to D (QM-rated events are not considered any more – see 2.2.3), an appropriate safety goal is formulated. When inherently similar safety goals are defined for different hazardous events, they are appropriately combined into a single safety goal, which will be assigned the highest ASIL amongst the related events. The expression of safety goals should not depend on technological solutions or design details, but rather on functional objectives of the item under consideration.

In addition to the Automotive Safety Integrity Level (ASIL), a given safety goal has several properties, such as:

- **Safe state**: If the safety goal is achieved through transitioning to or maintaining one or more safe states, then a corresponding specification of this or these safe state(s) is required. It should be noted that a safe state is defined as an *"operating mode of an item without an unreasonable level of risk"* (e.g., the normal operating mode, a degraded but non-harmfully degraded operating mode, the stationary vehicle state, and the switch-off mode) [2].

- **Fault tolerant time interval**: Defined as the *"time-span in which a fault or faults can be present in a system before a hazardous event occurs"*, the fault tolerant time interval depicts the responsiveness time of the system to the root cause of a specific hazard. This time is proportional to the potential harm of the hazardous event which may occur if no sufficient counter-measures are undertaken. So, by making it an attribute of the safety goal, it will be later on passed through to the derived functional, technical, and eventually hardware or software safety requirements. Consequently, the implementation of the safety mechanisms related to those requirements will take the fault tolerant time interval into account and ensure a timely transition to the safe state.

- **Physical properties**: These are physical characteristics such as the maximum level of unwanted acceleration or steering torque that could still be tolerated.

Throughout the safety lifecycle, and particularly in the development stages at system, hardware, and software levels, the so-called *safety goal violations* are evaluated. For example, in Part 5 (Product development at hardware level), a clause is dedicated to the *evaluation of the safety goal violations due to random hardware failures.* It addresses the quantification of random hardware failures and gives recommendations to provide a rationale about the sufficiently low rates of the related residual risks that are claimed to hold the targeted safety integrity levels. FMEDA (see 2.2.7.5) is one the most established techniques used in the industry to perform the required evaluation of the safety goal violations due to random hardware failures.

2.2.5 Safety Concept and Safety Requirements

Safety requirements play a central role in the ISO 26262 standard and have as primary purpose to ensure the targeted ASILs. In the different phases and sub-phases of the safety lifecycle, requirements addressing functional safety and its fulfillment by the system are specified, refined, and verified. The association of such requirements with the system elements is mostly referred to as *safety concept*.

Safety requirements comply with a hierarchical structure prescribed by ISO 26262. In fact, the standard distinguishes between (i) *functional safety requirements* which are derived from the safety goals (see 2.2.4) and allocated within the *functional safety concept* at the item level, (ii) *technical safety requirements* which are more concrete requirements defined at the system level and allocated in the *technical safety concept* to the architectural elements of the system design, and (iii) *hardware/software* safety requirements which are the most detailed and implementation-specific requirements to be fulfilled by hardware parts or software units. Figure 2.4 gives an overview of safety requirements in the ISO 26262 standard.

Safety requirements are so important in ISO 26262 such that Part 8 (Supporting processes) dedicates a separate clause for their specification and management. Beyond the hierarchical organization, several characteristics must be valid for safety requirements to claim compliance with ISO 26262, such as completeness, unambiguity, atomicity, feasibility, and verifiability.

2.2.5.1 Functional Safety Concept

Defined in ISO 26262 as the *"specification of the functional safety requirements, with associated information, their allocation to architectural elements, and their interaction necessary to achieve the safety goals"* [2], the functional safety concept focuses on the safety measures to be provided by the item but can also include references to external measures and interfaces to other technologies outside the scope of the item. It should be noted here that these safety measures are described in a functional and implementation-independent way. The realization details of the measures and their transformation into safety mechanisms will be covered by later technical and/or hardware/software safety requirements.

During the development of the functional safety concept, several aspects are taken into account, such as:

- **Fault detection**: Dedicated mechanisms for the diagnosis of malfunctions within the item and the production of appropriate warning messages to the driver are required in order to reduce the risk exposure.
- **Fault tolerance**: The number of faults leading directly to a safety goal vi-

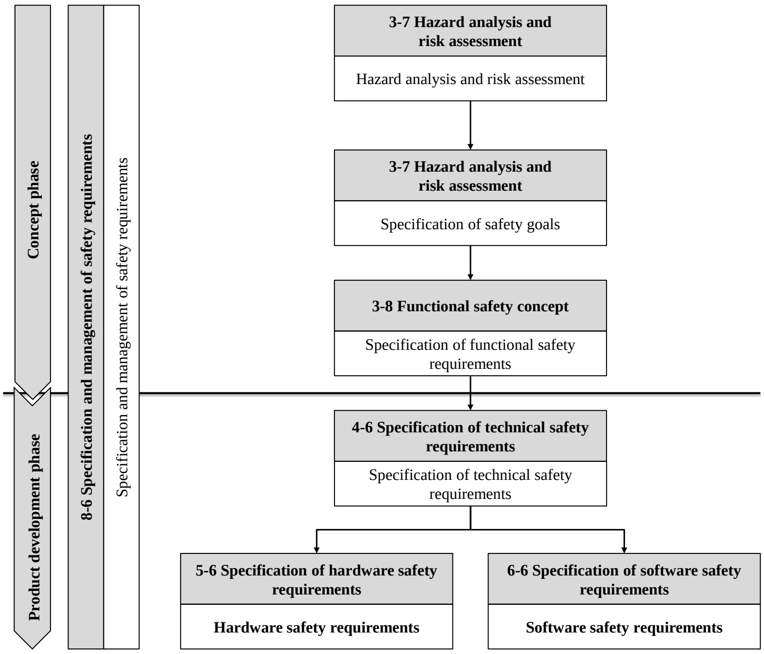

Concept phase
Product development phase
8-6 Specification and management of safety requirements
Specification and management of safety requirements
3-7 Hazard analysis and risk assessment
Hazard analysis and risk assessment
3-7 Hazard analysis and risk assessment
Specification of safety goals
3-8 Functional safety concept
Specification of functional safety requirements
4-6 Specification of technical safety requirements
Specification of technical safety requirements
5-6 Specification of hardware safety requirements
Hardware safety requirements
6-6 Specification of software safety requirements
Software safety requirements

Figure 2.4: Overview of Safety Requirements in ISO 26262 [2]

olation can be reduced. For example, redundancy of architectural elements increases the overall resiliency against random hardware failures.

- **Transitioning to safe state**: Once a fault is detected, and there is no sufficient correction or tolerance mechanisms to ensure the normal operation any longer, an immediate switching to a safe state is required to mitigate the imminent failure and prevent the harm associated with it.

2.2.5.2 Functional Safety Requirements

An important step in the development of the functional safety concept (see 2.2.5.1) is the derivation of the *functional safety requirements*, which are defined as the *"specification of implementation-independent safety behavior, or implementation-independent safety measure, including its safety-related attributes"* [2].

The functional safety requirements are derived from the safety goals taking into account the targeted safe states and the preliminary architectural assumptions. It should be noted (i) that for each safety goal (see 2.2.4), one or more functional safety requirements are specified and (ii) that a given functional safety requirement can be associated with many safety goals.

Potential attributes of a functional safety requirement are e.g.,:

- the operating modes for which it is valid,
- the fault tolerant time intervals inherited from the associated safety goals,
- the safe state to which the system has to switch, and
- the functional redundancies ensuring fault tolerance.

2.2.5.3 Technical Safety Concept

Once the functional safety concept (2.2.5.1) including all functional safety requirements (2.2.5.2) is available, product development at system level is initiated. An essential prerequisite for the system design is the specification of the *technical safety requirements* (see 2.2.5.4).

During the initial steps of the system design phase, the architecture of the system including hardware parts, software units, and the interfaces between them is specified. For this, the technical safety requirements are taken into account and they are appropriately allocated to the system design architectural elements leading to a further work product called *technical safety concept*. The exact definition given by the ISO 26262 standard is: *"specification of the technical safety requirements and their allocation to system elements for implementation by the system design"* [2].

So, it is necessary to have a look at the explanation of *technical safety requirements* (see 2.2.5.4) in order to understand what the technical safety concept is exactly meant for. However, it can be generally understood as a statement of how the intended safety functionality is implemented on system level by hardware and software.

2.2.5.4 Technical Safety Requirements

Defined very generically in the vocabulary part of the ISO 26262 as *"requirement(s) derived for implementation of associated functional safety requirements"* [2], technical safety requirements are derived using the functional safety concept and the preliminary architectural assumptions as inputs. The primary objective is then to refine the item-level safety functionality resulting from the concept phase and start closing the gap towards the final implementation.

Several system properties are considered during the specification of the technical safety requirements. For example, external interfaces such as communication with other systems and user interactions. These interfaces give valuable hints about how a safety functionality could be efficiently realized with respect to the system environment and the perception of the system state by the driver or other persons potentially at risk in failure cases. In addition to that, the system configuration and the usage constraints originating from the system functionality itself or from the environmental conditions are also relevant for the technical safety requirements.

It should be noted (i) that the technical safety requirements also address safety-related dependencies between system elements or between the system and its environment and (ii) that a thorough verification of the technical safety requirements with respect to their compliance and consistency with the functional safety requirements (2.2.5.2) and with the preliminary architectural assumptions of the system [2] is mandatory.

After the specification of the technical safety requirements (2.2.5.4), the specification of the system design architecture, and the creation of the corresponding technical safety concept (2.2.5.3), the product development at hardware and software levels is started. During those phases, detailed hardware and software safety requirements are derived and the final hardware parts and software units are designed and implemented. It should be noted that the fulfillment of the safety requirements is evaluated and verified at different levels using so-called *safety-oriented analyses* (more details in 2.2.7). Afterwards, integration and testing tasks are performed at hardware and software level. Eventually, the final *item integration and testing* at system level is executed. It would be now expected that the production can be initiated, similarly to the IEC 61508 safety lifecycle (see 2.1.1 and 2.1.2). In fact, for the singular or low volume systems aimed by IEC 61508, it is common to build the system, test it, install it on the plant, and only then the *safety validation* is performed. This is however not suited to road vehicles representing mass-market systems, because of the considerable financial loss associated with potential functional safety deficiencies detected after system deployment. That is why, before releasing the system for series production, ISO 26262 prescribes a safety validation (detailed in 2.2.6) and a functional safety assessment phase. Thereby, several management tasks related to the gathering and reviewing of safety documents and audits are performed. Furthermore, the functional safety assessment phase includes the compilation and/or refinement of the *safety case* defined as an *“argument that the safety requirements for an item are complete and satisfied by evidence compiled from work products of the safety activities during development”* [2].

2.2.6 Safety Validation

Defined as the *“assurance, based on examination and tests, that the safety goals are sufficient and have been achieved”* [2], safety validation is a separate phase of the product development at system level. It has two primary objectives:

1. to provide evidence about the appropriateness of the functional safety concept

and about its compliance with the safety goals, and

2. to demonstrate the safety goals themselves are correct, complete, and fully achieved at the vehicle level.

In other words, based on examination and tests, safety validation confirms or rejects the claim that the intended safety measures are adequate for the targeted vehicles, that the safety goals are sufficient to avoid harm, and that they are achieved by the safety measures with a high assurance level.

It should be noted that the safety validation step has to be properly planned from the beginning of the product development at system level. In fact, the *validation plan* is a work product of the very first initiation step (see Figure 1.1) and it gets accordingly refined during the specification of the technical safety requirements, so that the validation criteria are not only dependent on the functional safety requirements (2.2.5.2) but also on the technical safety requirements (2.2.5.4).

The key outcome of the safety validation phase is the *validation report* which provides an evaluation of:

- the controllability assumptions made during the Hazard Analysis and Risk Assessment (HARA) (see 2.2.2),
- the effectiveness of the safety measures to control systematic failures
- the effectiveness of the safety measures to control random hardware failures
- the effectiveness of external measures or other technologies outside the scope of the item.

It should be noted that the effectiveness grading with respect to systematic failures is substantiated through analysis results (e.g., FTA or FMEA results) which have been gathered during the system design phase, and that the effectiveness grading with respect to random hardware failures is substantiated through analysis results (e.g., FMEDA results) which have been gathered during the hardware and software design phases (see 2.2.7.5).

2.2.7 Safety-Oriented Analyses

Functional safety analysis is a key activity in the safety lifecycle prescribed by ISO 26262. Although it is not a separate phase or sub-phase of the production flow, it is reiterated from different perspectives, at several stages of the cycle, and at many levels of detail to evaluate the fulfillment of safety requirements. The importance of the so-called *safety-oriented analyses* (also simply referred to as *safety analyses*) is reflected by the dedication of a separate clause in Part 9 of the standard (ASIL-oriented and safety-oriented analyses) to the requirements which shall be met during safety analysis and to the recommended guidelines to fulfill them.

2.2.7.1 Goals

Generally, safety analyses are aimed at the examination of fault and failure consequences (i) on the system design, (ii) on the functions of the different hardware or software elements, and (iii) on the overall behavior from the item perspective.

At system level, safety analysis is used to identify the causes of systematic failures which are defined as *"failure(s) related in a deterministic way to a certain cause, that can only be eliminated by a change of the design or of the manufacturing process, operational procedures, documentation or other relevant factors"* [2]. In other words, the analysis is concerned with design mistakes that can be committed at this level. Therefore it is considered as an assistance to system designers and it is mostly performed qualitatively without getting into the quantitative assessment of the failure probabilities for example. In order to enable the detection and the exclusion of systematic faults (i.e., the root causes of systematic failures) or at least the mitigation of their effects, the ISO 26262 recommends either a *deductive analysis* (e.g., Fault Tree Analysis (FTA) – 2.2.7.6 and Reliability Block Diagram (RBD) – 2.2.7.8) or an *inductive analysis* (e.g., Failure Modes and Effects Analysis (FMEA) – 2.2.7.5 and Event Tree Analysis (ETA) – 2.2.7.8). Furthermore, the standard recommends the application of so-called *well-trusted automotive systems design principles*, including re-use and standardization, to reduce the likelihood of systematic failures [2].

At hardware and software levels, detailed safety analysis is performed focusing on random hardware failures and on dependencies and interferences in software.

On the one hand, for all safety-related hardware parts, safety analysis is prescribed to classify faults into *safe faults*, *single-point faults*, *residual faults*, *multiple-point faults*, and *latent faults*. Safety mechanisms shall be evaluated with respect to their *diagnostic coverage* values in order to determine the exact portion of residual faults, used later on for the calculation of dedicated safety metrics at the element or system level. The rationale of the analysis at hardware level is structured using an FTA (2.2.7.6) or more commonly an FMEDA (2.2.7.5) where further hardware architectural metrics regarding area and technology as well as failure rates are taken into account.

On the other hand, safety analysis is required for the software architectural level. The three major objectives are the (i) identifying all safety-related software units, (ii) supporting the specification of the appropriate safety mechanisms, and (iii) verifying the efficiency of those mechanisms. When specific software safety requirements address the *freedom of interference* or more generically the sufficient ***independence*** between software components, then an appropriate *Dependent Failure Analysis (DFA)* is required (see 2.2.7.7).

All the previously mentioned safety analyses at system, hardware, and software levels enable indirectly the evaluation of safety goal violations at item level. They actually help the designers and the safety analysts to identify human mistakes, design vulnerabilities, and other conditions possibly leading to a violation of a safety requirement and/or eventually of a safety goal. Furthermore, during such analyses, it is also possible to identify new functional or non-functional hazards that have not been considered during the Hazard Analysis and Risk Assessment (HARA) (see 2.2.2) [2]. That is why it is also recommended to support the derivation of functional safety requirements (2.2.5.2) out of the safety goals (2.2.4) with qualitative safety analyses (e.g., FMEA, qualitative FTA, and HAZOP) to increase effectiveness and completeness.

2.2.7.2 Application Scope

According to the ISO 26262 standard and from the detailed explanation of the multiple purposes of the analysis at the different levels of concept and development, the scope of safety analyses can be summarized as follows:

- verification and validation of safety concepts and of safety requirements (including safety goals)
- identification of conditions and causes (including systematic design faults, random hardware failures, and dependent failures in software) which potentially affect the correct implementation of a hardware, software, technical, or functional safety requirement and eventually lead to a safety goal violation
- identification of additional requirements for fault detection and/or failure mitigation, i.e., proposal of refinements for available safety mechanisms in deficiency cases or definition of new mechanisms in case of absence or complete inadequacy
- identification and/or qualification of responses (actions and/or measures) to detected faults and/or failures

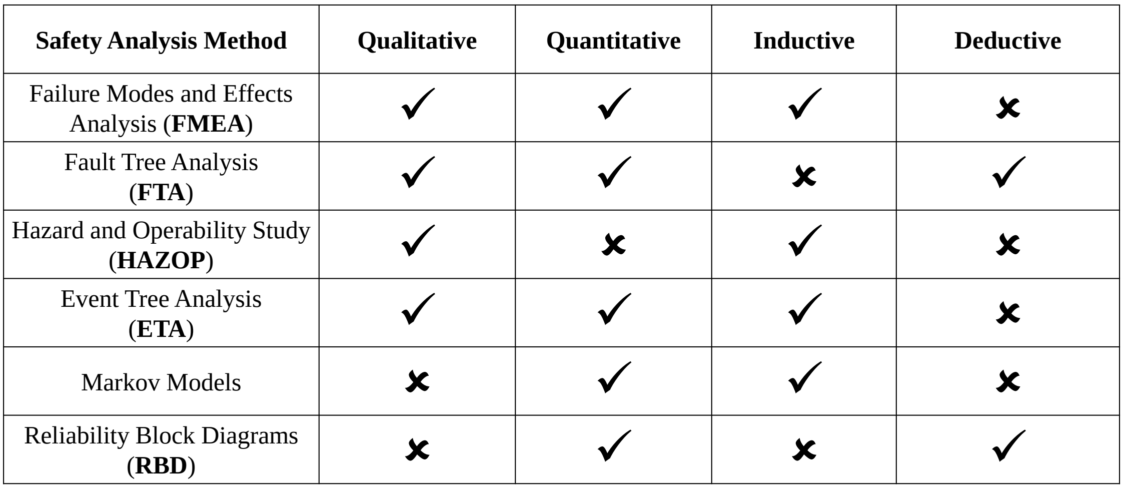
2.2.7.3 Classification Criteria

As already mentioned in 2.2.7.2, safety analyses are performed at different levels of abstraction during the concept and product development phases. The ISO 26262 standard provides a big range of analyses and recommends their application respectively at the different levels. Many of those analyses, such as FMEA (see 2.2.7.5) and FTA (see 2.2.7.6) appear in the concept phase as well as in the development phases (system and hardware levels). Their application methodology depends however on the artifacts to be analyzed (functional safety concept of the item, system architecture, hardware design, etc.,) and on the purpose of the analysis (identification or refinement of safety requirements, assessment of systematic failures, evaluation of safety goal violations due to random hardware failures, etc.,). Despite the considerable number of safety analysis techniques recommended by the standard and their different application cases, they can still be classified according to two major classification criteria: (i) the direction of the cause-to-effect exploration during the analysis (see 2.2.7.3.1) and (ii) its level of detail (see 2.2.7.3.2).

2.2.7.3.1 Cause-to-Effect Exploration The cause-to-effect exploration direction characterizes the way safety analyses are conducted. Based on it, safety analyses are classified into *deductive* and *inductive* approaches.

On the one hand, deductive safety analysis is performed *top-down* starting from problematic situations (i.e., failures) and aiming at the determination of the corresponding causes. Fault Tree Analysis (FTA) (2.2.7.6) and Reliability Block Diagram (RBD) (2.2.7.8) represent popular deductive safety analysis techniques.

On the other hand, inductive analysis methods are *bottom-up* methods that start from given causes or conditions and accordingly forecast potential effects. In other words, they identify local malfunctions and subsequently determine the corresponding effects on the whole element, system, or item. Failure Modes and Effects Analysis (FMEA) (2.2.7.5), HAZard and OPerability study (HAZOP) (2.2.7.4), Event Tree Analysis (ETA) (2.2.7.8), and Markov models (2.2.7.8) are inductive analysis methods which are commonly used both academically and industrially.



Safety Analysis Method	**Qualitative**	**Quantitative**	**Inductive**	**Deductive**
Failure Modes and Effects Analysis (**FMEA**)	✓	✓	✓	✗
Fault Tree Analysis (**FTA**)	✓	✓	✗	✓
Hazard and Operability Study (**HAZOP**)	✓	✗	✓	✗
Event Tree Analysis (**ETA**)	✓	✓	✓	✗
Markov Models	✗	✓	✓	✗
Reliability Block Diagrams (**RBD**)	✗	✓	✗	✓

Table 2.11: Classification of Safety Analysis Methods [3]

2.2.7.3.2 Level of Detail With respect to the level of detail, safety analysis approaches are commonly categorized in *qualitative* and *quantitative* techniques. At certain levels (e.g., concept and system development), qualitative analysis is sufficient. Nevertheless, it must be complemented at other levels (mostly in hardware development) by a more detailed quantitative analysis.

In fact, qualitative safety analysis is limited to the identification, classification, and ranking of the different conditions which potentially lead to system failures. Beyond those tasks, quantitative safety analysis also determines the extent of safety requirements satisfaction in terms of probabilities using several measures such as the *failure rate* (λ) and the *diagnostic coverage* values of the safety mechanisms. That is why, quantitative analysis is most suited to hardware, in the sense that it allows the verification of the hardware design safety using additional knowledge about the failure rates of the hardware parts and taking defined targets for hardware architectural and safety metrics into account. It should be noted that quantitative analysis methods are not required for the evaluation of systematic failures in ISO 26262 and that software safety analysis is also mostly qualitative in the scope of the standard.

Certain analysis methods such as FMEA, FTA, and ETA are used both qualitatively and quantitatively. That is why, it is commonly distinguished between qualitative FMEA (at system, design, or process level) and quantitative FMEA (at hardware level – commonly referred to in the industry as FMEDA). A similar classification holds for FTA and ETA (qualitative FTA/ETA versus quantitative FTA/ETA). HAZOP is used as a qualitative approach while Markov models and reliability block diagrams (RBDs) are applied for quantitative assessment in ISO 26262.

A recapitulating overview of classified safety analysis approaches is given in Table 2.11.

2.2.7.4 HAZOP: Hazard and Operability Study

HAZard and OPerability study (HAZOP) is a technique for hazards identification which emerged in the process industry during the 1970s after the Flixborough disaster (already mentioned in Section 2.1). Since then, it has been widely used for the design of new plants or the extension of existing ones and also found its way to other industries such as nuclear power plants and recently automotive applications. Indeed, in the context of the ISO 26262, HAZOP is used as a qualitative safety-oriented analysis method assisting the derivation of functional safety requirement (2.2.5.2) during the development of the functional safety concept (2.2.5.1).

The HAZOP approach consists basically in a systematic walk-through of the process or operation to be designed or modified. It aims at determining all possible deviations, i.e., hazards or operating problems, which may arise. The study is performed by a multi-disciplinary team with sufficient experience to provide a trustworthy evaluation and a rational expert judgment. For the identified hazardous deviations which are classified as unacceptable, appropriate changes are proposed and reinforced by a justification with respect to risk reduction and cost benefit argumentation. Associated actions are subsequently defined, followed with respect to implementation and efficiency, and properly reported to the responsible persons [35].

Because of its strong subjective character and the lack of quantification capabilities, HAZOP remains a limited approach and it is not suitable for detailed safety analysis at hardware level.

2.2.7.5 FMEA: Failure Modes and Effects Analysis

Failure Modes and Effects Analysis (FMEA) is a *bottom-up* (*inductive*) approach starting from local malfunctions and aiming at the determination of corresponding effects on the overall behavior. It focuses on the individual parts of the system, their potential failure modes and the impact of these failures on the system. The term *failure mode* characterizes the way in which a system can fail and can also be defined as the physical or functional manifestation of a failure such as slow operation, incorrect outputs or complete execution termination [21, 2].

The approach is referred to as *Failure Modes, Effects, and Criticality Analysis (FMECA)* when it is enhanced by a *criticality* analysis which consists in ranking the failures with respect to their likeliness and severity. FMECA is performed either during the design, development, or use phases and it usually requires knowledge about the functional structure of the complete system [36, 37].

Furthermore, two additional aspects can be considered in the FMEA process: (i) *quantitative failure data* consisting in failure rates and the distribution of failure modes and (ii) the capability of a (sub)system to use *automatic on-line diagnostics* to detect internal failures. In this case, the process is referred to as *Failure Modes, Effects, and Diagnostic Analysis (FMEDA)* [38].

FMEDA, which has been developed in the 1990s within a company called Exida [39, 40], is nowadays widely used in the industry as a quantitative safety analysis approach. With respect to ISO 26262, FMEDA is applied for the evaluation of hardware functional safety. It should be noted that the term *FMEDA* does not explicitly appear in the ISO 26262 standard, which only mentions *quantitative FMEA*. However, FMEDA is the frequently used denomination within the functional safety community. In compliance to ISO 26262, Part 5 (Product development at hardware level), FMEDA is a comprehensive procedure which addresses (i) the hardware *safety analyses* (5-7.4.3) of the hardware design sub-phase (5-7), (ii) the *evaluation of the hardware architectural metrics analysis* (5-8), and (iii) the *evaluation of safety goal* *violations due to random hardware failures* (5-9).

In this thesis, formalization and model-driven support for FMEDA are addressed. That is why, the FMEDA flow is covered later in more details in Subsection 5.3.1.

2.2.7.6 FTA: Fault Tree Analysis

Fault Tree Analysis (FTA) is a *top-down* (*deductive*) approach starting from problematic situations and aiming at the determination of the corresponding causes. It addresses unwanted events at system level, considers the related potential failures, and subsequently analyzes the possible causes behind them.

The basis of the FTA is the *fault tree* which is an acyclic graphical representation using gates and logical connectors to structure successive levels of events. The tree root is the top-level event depicting the uppermost system failure. It represents the starting point of the analysis which is performed through a systematic *backward-stepping* process leading eventually to so-called *elementary events* [41, 42, 43].

The predominant combinatorial aspect of FTA, which makes it inappropriate to advanced system descriptions including timing and sequencing, has been extended by different approaches such as Temporal Fault Trees (TFTs) [44] and Dynamic Fault Trees (DFTs) [45]. The TFT approach allows capturing of timing dependencies between faults and events [44]. Furthermore, in the DFT methodology, the FTA syntax is extended with additional fault tree gates (e.g., functional dependency and priority AND) to model dynamic behavior of fault-tolerant systems such as sequence-dependent failures and fault-and-error recovery [45].

Further details about the FTA procedure are given in Subsection 5.3.2 along with its metamodel-based formalization used as basis for model-driven support and automated fault tree synthesis.

2.2.7.7 DFA: Dependent Failure Analysis

In ISO 26262, *dependent failures* are defined as *"failures whose probability of simultaneous or successive occurrence cannot be expressed as the simple product of the unconditional probabilities of each of them"* [2]. They are classified in *common cause failures* and *cascading failures* [2].

Part 8 of the standard (ASIL-oriented and safety-oriented analyses) dedicates a separate clause to Dependent Failure Analysis (DFA) where *independence* and *freedom of interference* are considered. In fact, the invalidation of one of these two characteristics may result in the violation of a safety requirement or a safety goal. That is why, the events causing such invalidation must be identified through performing the DFA.

It should be noted that according to the ISO 26262 definitions, both common cause failures and cascading failures threaten the independence feature. However, freedom of interference may only be affected by cascading failures .

Several architectural features are taken into account during the DFA, such as element redundancies, the physical placement of hardware elements, and shared resources. Potentially dependent failures, which may be either systematic or random failures, are determined within the scope of the DFA by taking outcomes of deductive and inductive analyses into consideration. Indeed, the failure logic represented by fault trees (see 2.2.7.6) and failure mode similarities in the FMEA (see 2.2.7.5) provide useful hints about potential for dependent failures.

Further details about the DFA procedure are given in Subsection 5.3.3 along with its metamodel-based formalization.

2.2.7.8 Other Analysis Approaches

- **Event Tree Analysis (ETA)**: Being an *inductive* (bottom-up) technique for logical modeling of the system behavior in both *success and failure cases*, ETA has been first used in risk assessment for nuclear power plants. Nowadays, it is used in several industries (e.g., process, oil and gas, transportation, etc.,) [46]. The event-tree is constructed in a forward-stepping way, so that starting from a given initiating event, the different possible paths are built leading either to a functionally correct outcome or to a hazardous situation at system level. Qualitative ETA assesses the probability of such hazards and subsequently triggers necessary refinement or change actions for the underlying design [46].
- **Markov models**: These stochastic models, which are represented as state diagrams, are used to model changing systems over time under the assumption that the future state only depends on the current state. The sequence of the previous states is not relevant for the determination of the new state. In the context of functional safety analysis, Markov models address failure states and their occurrence probabilities, dependencies, diagnostic coverage, repair times (or the time needed to reach a safe state), etc., [47].
- **Reliability Block Diagrams (RBDs)**: depicting graphical representations of the system's components and the connections between them, RBDs are a further mean to analyze functional safety. The components can be interconnected either in series or in parallel and the arrangement logic of the diagram shows the combinations of component failures which lead to system failures. Assuming that the connectors do not fail and that every component is considered as a switch which is closed in the case of correctly operational state and open in failure condition, a successful operational system necessitates at least one

complete path between system input and system output [48, 49, 50]. Through inspection of the RBD, the failure sets resulting into a system failure can be identified. Using Boolean logic, the so-called *minimal cut sets*, causing system failure with the smallest numbers of component failures, can be determined.

2.2.8 Fault Injection and Simulation

In Subsection 2.2.7, different safety analysis methodologies have been described. Thereby, incorrect behaviors and undesired malfunctions are inspected based on more or less abstract representations of the considered system without any alterations. Beyond those analyses and with respect to executable system models, a further approach has been widely applied during the last decades for safety evaluation: *fault injection*. In this subsection, the fundamental concepts of fault injection, as well as its most relevant application features and patterns are presented.

In the context of the ISO 26262 standard, fault injection and simulation are recommended during the development at hardware level in order to verify the implementation of safety mechanisms with respect to completeness, and correctness, and fulfillment of hardware safety requirements [2].

In general, fault injection consists in the deliberate insertion of faults into a system which is afterwards monitored to determine its behavior in response to the introduced faults. Several fault injection techniques have already been introduced and experimented in the past. Some of them deal with system prototypes, while others address system models. Most fault injection techniques can be classified into the following categories [51, 52, 53]:

- *Hardware-based fault injection*: This injection technique is accomplished at the physical level using extra components which affect the original hardware by (i) disturbing the hardware with environment parameters (heavy ion radiation, electromagnetic interferences...), (ii) modifying the value of the circuit pins, and (iii) disturbing the power supply (power rails manipulation with voltage sags).
- *Software-based fault injection*: This injection approach is applied to reproduce at the software level the errors that would have been caused by faults occurring in hardware. Software-based fault injection does not require any additional expensive hardware and enables fault introduction into applications or operating systems. It addresses implementation details, program state, communications, and interactions. Related methods are categorized based on the phase during which the fault injection is performed. On the one hand, *compile-time injection* requires the modification of the program instructions before the according loading and execution. So, instead of physically inducing *faults* in the hardware of the target system, the corresponding *errors* are injected into the source or

assembly code. On the other hand, *runtime injection* requires the usage of an appropriate triggering mechanism such as (i) a *time-out* which generates an interrupt invoking the fault injection after a certain time, or (ii) an *exception or trap* which invokes the fault injection when specific events or conditions occur.

- *Simulation-based fault injection*: This injection approach is aimed at the early system evaluation by introducing faults in high-level models such as RTL models. For this purpose, faults are introduced into the model and subsequently, the effects are observed. Two major categories of the simulation-based fault injection are distinguished. First, in *code modification*, the original target system description is modified by adding so-called *saboteurs*, which are only active when a fault is being injected altering the value or the timing characteristics of specified signals, or by applying so-called *mutants*, which depict altered versions replacing given system components and causing a non-compliant behavior with the system specification. Second, using *simulator built-in commands*, the original simulation tools are modified to support the injection of faults and the monitoring of the resulting impact (i.e., errors and failures) on the simulated system. In the context of VHDL simulation-based fault injection, *signal manipulation* through forcing and *variable manipulation* during dedicated simulation runs are the most relevant altering mechanisms accomplished using simulator built-in commands.
- *Emulation-based fault injection*: In this context, hardware prototyping with Field-Programmable Gate Arrays (FPGAs) is used to improve the effectiveness in comparison to simulation-based fault injection campaigns with respect to time and efforts overhead.
- *Hybrid fault injection*: To achieve more speed and efficiency, different techniques may be combined in the context of the hybrid fault injection. For example, the flexibility and the versatility of software-based fault injection can be joined with the monitoring accuracy of hardware-based fault injection.

3 Related Work

This chapter studies the related work with respect to the major thesis topics. First, existing formalization approaches for dependability and safety are presented in Section 3.1. Then, Section 3.2 gives an overview of relevant contributions towards the model-based support of safety analysis. Finally, known academic and industrial alternatives for linking the safety analysis context on the one hand to the fault injection and simulation context on the other hand are depicted in Section 3.3.

3.1 Formalization Approaches for Dependability and Safety

As already introduced in Section 1.2, *dependability* denotes the ability of a system to deliver a trustworthy service. It covers multiple aspects including *safety*, which is the focus of this thesis. During the last two decades, dependability and safety assessment have been extensively addressed in the literature, particularly with respect to the major formality, reuse, and interoperability challenges presented in Section 1.4.

To tackle the significant challenges of ensuring the robustness of increasingly complex devices, many academic contributions addressed the so-called *cross-layer dependability* topic, particularly for System-on-Chips (SoCs). By developing a modeling concept for faults, errors, and failures which can be applied at different levels of abstraction such as the *Resilience Articulation Point (RAP)* [54, 55, 56], a relationship between lower technology levels and higher system implementation levels is established and thereby considerable analysis efforts and design hardening costs are saved. Indeed, through the comprehensive Resilience Articulation Point (RAP) framework which enables probabilistic fault abstraction and error propagation across all hardware and software layers of the SoC [55], a trade-off between the effectiveness and the cost of system resilience is achieved.

Furthermore, several approaches have been proposed aiming at a unified conceptualization of dependability and safety analysis. A brief summary of these proposals is given in Subsection 3.1.1. Furthermore, Subsection 3.1.2 addresses a set of model-driven methodologies for dependability and safety assessment which are generally denoted as *failure logic modeling* techniques. These methods represent a first step towards the automation support of safety analysis covered in Section 3.2.

3.1.1 Proposals for Generic Dependability Modeling

As a part of this thesis, existing methodologies for generic dependability modeling have been comprehensively reviewed. The major perceptions of this review, which have been published in [57], are summarized below.

In [58], the *Unified Model of Dependability (UMD)* is introduced with the primary purpose of creating a common framework for dependability definitions and measurements. By combining invariant concepts across all applications on the one hand and customizable concepts correlated with the usage context on the other hand, UMD provides a common dependability ontology, contributes to a generalized conceptualization, and makes it easier to build customized dependability models which are tailored to the specificities of different systems [59]. However, UMD has many limitations. Indeed, it does not offer a methodology for the systematic derivation of context-related dependability models. Moreover, the integration of UMD in more comprehensive model-driven frameworks is limited, so that the automation opportunities of the dependability evaluation remain restricted.

A further proposal for generic dependability modeling is given in [60] relying on UML[1]. Through an extension of the traditional UML usage, which consists in the the specification and the automated checking of functional system properties, a new profile is provided to cover non-functional properties including dependability attributes. Thereby, new elements are added to enable (i) the description of faults and fault effects, (ii) the mapping of correspondingly constructed system models to a mathematical analysis domain, and (iii) the subsequent automated derivation of transformation rules used later for the translation of abstract representations into concrete views. This UML-based dependability modeling allows an earlier initiation of requirements verification in the development cycle and leads consequently to an enhanced design exploration and to a considerable cost reduction [61]. However, the approach in [60] does not address the cross-domain aspect, so that the tool-interoperability and standard-conformity problems in the dependability context remain unsolved.

In [62], a fault and failure metamodel is proposed as a part of an approach called *SynDia* addressing the synthesis of online diagnostic techniques for embedded systems. The metamodel is IEC 61508 compliant and it is based on an abstract superclass which characterizes the component faulty behavior in hardware architectures whose functional safety is assessed according to the guidelines of the IEC 61508 standard. Specific subclasses are then derived for the multiple component types. This meta-

[1] **UML**: Unified Modeling Language

model helps to generate diagnostic techniques when a further metamodel describing dependability requirements is taken into consideration [62]. Nevertheless, it remains restricted to scope of the IEC 61508 standard, so that considerable extensions or changes are required before reusing it in other contexts.

SafetyMet [63] is a unified metamodel for safety standards which has been developed in the context of the OPENCOSS[2] project aiming to establish a common safety certification framework for automotive, avionics, and railway. A primary characteristic of SafetyMet is the separation between standard-specific and project-specific aspects with respect to safety analysis tasks and artefacts. Thereby, it enables the reuse of safety models and assets originating either from old or current projects. Though, among the limitations of SafetyMet which are stated in [63], most important to mention are the certification risks that may remain even when the metamodel is applied, the persistence of several human aspects of the safety compliance, and the questionable feasibility of safety models generation from very large textual specification files.

In summary, the contributions described above support the generalization of dependability concepts and the integration of model-driven development techniques in the analysis methodology to a certain extent. Nevertheless, many challenges persist. In [57], a generic approach is developed to overcome such issues. It covers different dependability attributes and can be adapted to several domains through a customization workflow.

The model-driven generalization and formalization approach presented in [57] relies on applying metamodeling and code generation techniques (see Subsection 4.2.2). A generic metamodel is constructed as a formalism to describe dependability terms and to capture syntactic and semantic relationships between various domains. Domain-specific dependability metamodels are derived from this generic metamodel using an XMI[3]-based customization tool.

The approach presented in [57] accelerates the construction and configuration of dependability analysis platforms. It offers a model-driven framework to support dependability analysts and engineers by enabling the creation of customizable and reusable domain and project-specific assessment tools.

3.1.2 FLM: Failure Logic Modeling

Failure logic modeling (FLM) is a concept introduced in [64] to depict a group of model-driven dependability analysis techniques based on a compositional modeling of failure behavior across the different design stages. These techniques address two major problems encountered in classical dependability assessment approaches: (i) the

[2] **OPENCOSS**: Open Platform for EvolutioNary Certification of Safety-critical Systems [3] **XMI**: XML Metadata Interchange

extensive knowledge of the whole system's architecture and behavior needed by the analysts and (ii) the limited reuse opportunities caused by the lack of modularization. Through particular notations, the design specification of the individual components of the system architecture is extended with the corresponding failure behavior to enhance the manageability of the complete system's dependability assessment [64, 65]. One important aspect of FLM is the automated extraction of classical evaluation artefacts (e.g., FMEA table segments or fault trees) from the constructed models, which leads to considerable time and effort savings [66].

The common points of all FLM techniques (e.g., *Failure Propagation and Transformation Notation (FPTN)* – 3.1.2.1, *Failure Propagation and Transformation Calculus (FPTC)* – 3.1.2.2, and *Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS)* – 3.1.2.3) are (i) the component-oriented system description on which the analysis is based, (ii) the specification of the failures affecting every component either on the inputs, on the internal elements, or on the outputs, and (iii) the definition of the cause-to-effect relationships between those failures.

3.1.2.1 Failure Propagation and Transformation Notation

The Failure Propagation and Transformation Notation (FPTN) was the first modular and semi-graphical approach for failure behavior specification of architectural elements. It represented a strong driving factor for the development of later models [67, 64, 65].

In FPTN, system components, which are referred to as modules, can either be elementary or decomposable in further modules, and are represented graphically as blocks with certain numbers of *input failure modes*, *internal failure modes*, and *output failure modes*. In addition to some standard attributes (e.g., name), each module is characterized by a set of *failure propagation and transformation equations* where every output failure mode is specified through at most one Boolean formula over input failures [67, 64]. Connections between outgoing (output) failure modes of a given module with incoming (input) failure modes of further blocks is also supported by FPTN as well as hierarchical module nesting. Beyond the logical description of the failure behavior of a given system, FPTN is considered as a more concise and abstract representation of complex fault trees and/or FMEA tables. Thus, it can be used as assistance during the identification of safety threats and hazards [68].

3.1.2.2 Failure Propagation and Transformation Calculus

Failure Propagation and Transformation Calculus (FPTC) [69] is a more recent failure logic method which is closely related to FPTN. Among the extensions offered by FPTC, a more sophisticated syntax to specify propagation equations in a more concise way is noteworthy [69, 66]. Most importantly, FPTC addresses the cyclic dependencies issue in failure logic models. In fact, handling closed feedback loops and similar cyclic flows in the system architecture, which is not supported by FPTN, is possible through fixed-point evaluation techniques in FPTC [69]. The practicability and the concrete application of FPTC in the industry is however limited [65].

3.1.2.3 Hierarchically Performed Hazard Origin and Propagation Studies

A more publicized and experimented failure logic modeling approach is the Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) methodology [70, 71, 72, 73]. The major syntactical particularity of HiP-HOPS is the use of the Tabular Failure Annotation (TFA) format to specify the failure behavior of the different components instead of the graphical representation of FPTN. The specification is done in commercial tool environments such as Matlab-Simulink using a new FMEA variant called Interface Focused-FMEA (IF-FMEA) [72]. Offering fault tree generation and minimal-cut set analysis facilities [71, 73], HiP-HOPS found its way to industrial applications, e.g., at Daimler Chrysler for the assessment of a brake-by-wire system prototype [72].

3.1.3 Other Contributions

In [61], a thorough survey study of dependability modeling and analysis approaches is performed. It addresses software systems whose specification is based on UML and which must be assessed with respect to specific dependability attributes. The survey shows that reliability and safety are more considered in the academic research than the other dependability attributes such as availability and maintainability. Another perception of the study is related to the application timing of these modeling approaches during the development cycle. In fact, the provided support is mainly limited to the early lifecycle phases (e.g., requirements definition, high-level design, etc.,) while the later phases of the dependability evaluation and testing remain uncovered. Most surveyed methodologies support a transformation of the formally captured aspects in the enhanced UML models into the dependability analysis domain. Nevertheless, only few approaches take the necessity of a *feedback loop* into account. In other words, the back-annotation of the dependability analysis results into the original UML model are neglected by most of the surveyed approaches. Finally, the authors of [61] highlight the high potential for automation support in the UML-based dependability modeling approaches that they considered in the survey. However, they emphasize the urgent need for a common platform for UML-based dependability modeling and analysis and for an improved tool implementation and validation flow in this context. Indeed, most of the contributions are evaluated with respect to feasibility and practicability through a few use cases. Several approaches are studied in [61], such as the methodology introduced in [74] focusing on failure severity evaluation from UML specifications, the one of [75] addressing the design of safety-critical distributed embedded real-time systems along with the automated code derivation from UML models, and the approach in [76] defining a UML profile to specify safety concepts of aerospace software systems and correspondingly generate certification-related information.

3.2 Safety Analysis Automation

To overcome the largely manual state-of-practice in safety analysis, first works and experiences towards Model-Based Safety Analysis (MBSA) have been presented in [77], [78], and [79], based on a tight integration of system and safety engineering procedures by using the same central formal model to describe the correct system configuration as well as its dysfunctional behavior in the presence of faults. The approach is able to automate parts of the classical safety analysis through fault tree generation or automated search for failure scenarios leading to hazardous events. However, it has different limitations such as the questionable scalability of the used tools to industrial applications and the additional difficulties caused by "cluttering" the nominal and the faulty system behaviors in the same model [78]. In fact, the supplementary information related to the safety threats makes the exploration and the refinement of the initial system functionality more challenging and more costly for design engineers. In [79], the authors suggest to specify fault models separately from the nominal component behaviors and defining the interactions between them. The clear separation of the nominal model from the fault model and alternative extension and merging capabilities have also been addressed in the context of the ESACS/ISAAC methodology [80] which provides a tool for automated fault tree generation based on symbolic model checking with NuSMV models [81]. However, only basic component failure modes can be specified and injected into the nominal system model. Furthermore, important aspects such as failure multiplicity and propagation are not covered.

In correlation to the safety formalization and modeling strategies presented in Section 3.1, especially to FLM techniques (see 3.1.2), several automation approaches have been proposed in the literature. The following Subsections 3.2.1 and 3.2.2 focus respectively on automation techniques for FTA (2.2.7.6) and for FMEA (2.2.7.5).

3.2.1 Fault Tree Synthesis

Since the 1980s, several strategies have been investigated to automate the FTA procedure and systematically synthesize fault trees. In general, the initial step towards the automated fault tree synthesis is the construction of proper formal models for the systems to be analyzed. These models are used later within algorithmic processes in order to build the intended fault trees. Therefore, they must fulfill certain requirements, such as being machine-readable, containing the necessary details about the system architecture and operation, and enabling easy extensions and changes [82].

Several approaches are commonly used for fault tree synthesis [82], such as:

- **Digraphs**: in [83] and [84], the usage of directed graphs (graphical representations used to trace the local and global causality of disturbances and equipment failures) for fault tree synthesis is described. It relies on an algorithm for the traversal of the digraph, the subsequent derivation of the fault trees depicting the logic of the safety goal violations, and the calculation of the minimal cut-sets of those generated trees.
- **Decision tables**: after creating a tabular model representing the failure behavior of the system, an analysis tool or script is executed to extract the necessary information to construct the corresponding fault tree [85].
- **State diagrams**: based on finite state machines, called either *state diagrams* or *mode automata*, the system failure behavior is graphically described in a formal way and provided as input to dedicated tools transforming it into fault trees [86, 87] decomposition.
- **Matlab-Simulink models**: in association with the HiP-HOPS methodology (see 3.1.2.3), Papadopoulos et al. applied Matlab-Simulink models for model-based synthesis of fault trees [73, 88].

3.2.2 FMEA Automation

FMEA (2.2.7.5) efficiency is limited by several organizational and operational constraints facing safety engineers while performing the analysis. Getting all involved participants together, gathering all required information for the analysis, and linking the results with corrective procedures are examples of those constraints [89]. To handle these challenges, several projects focused on the conception and development of appropriate frameworks to manage FMEA complexity through software tools.

In [72], Papadopoulos et al. proposed a methodology for synthesizing FMEA parts. This approach is based on augmented compositional system diagrams with special characterizations of failure logic for each component. These characterizations are referred to as *failure annotations*. The theoretical background for this approach is the HiP-HOPS safety analysis methodology [70] which aims at performing different safety studies on a consistent hierarchical system model (see 3.1.2.3).

In [90], an integration of formal and informal aspects is suggested to support FMEA automation. Based on high-level graphical notations called *behavioral trees*, formal model checking is applied to find out whether initially specified safety properties are satisfied. The proposed method requires the preliminary construction of the behavioral trees and the formalization of the normal and the failure states as linear temporal logic formulas. The key benefit of this contribution is the facilitated building of relationships between component malfunctions and failures on the system-level.

Another automation approach for FMEA is presented in [91]. It promotes incremental FMEA of component-based HW and SW systems. The approach focuses on deriving component and system FMEA tables in a progressive way using so-called *safety interfaces* [92]. The consistency of the FMEA artefacts with the design model is an important feature which enables the automatic identification of critical effects in the case of design extensions. Although this approach reduces the manual work of the safety engineers and allows reuse of partial analysis results when components are integrated in different systems, it remains a qualitative method [91] which has to be extended with quantitative capabilities to be suitable for industrial contexts.

In [93], a framework for preliminary safety assessment in the software domain is introduced. A comprehensive safety flow is described from the early requirements definition to the final evaluation of the safety integrity level, offering different capabilities such as a safety modeling language, model transformations, as well as generation and automation tools. Among these tools, an UML-based FMECA model creation tool is implemented to transform the software architectural model extended with safety-related information (e.g., as the failure modes, their corresponding failure rates, and the associated severity levels) into an FMECA model which can be further on displayed in tabular form. The details of the transformation rules are given in [93]. This approach has several benefits such as (i) the early engagement of the safety engineers in the architecture design cycle resulting into quality and cost gains, (ii) the enhanced consistency and availability of safety-related data due to the model-based support, and (iii) the process speed-up achieved through automation. However, the application of the methodology is limited to the software domain, its practicability has only been evaluated for air navigation systems, and there is no link between the analysis context and the fault simulation context.

In this thesis, the limitations of the automation strategies for safety analysis mentioned above are addressed. The key objective is to develop a more comprehensive methodology, create a more reusable framework, and evaluate the practicability in the automotive context with respect to functional safety guidelines in compliance with the ISO 26262 standard.

3.3 Linking Analysis and Simulation for Safety Evaluation Purposes

As already mentioned in Subsection 2.2.8, the ISO 26262 standard prescribes the application of fault injection and simulation during the development at hardware level in addition to safety analysis when the targeted Safety Integrity Level (SIL) for the considered system is particularly high. The objective is to verify the completeness and correctness of safety mechanisms, as well as the fulfillment of the associated safety requirements.

One of the major challenges facing the industry in this context consists in the gap between the safety analysis environment on the one hand and the fault injection and simulation environment on the other hand. The tasks are respectively performed by different teams with dissimilar work styles and strategies (see Subsection 1.4.1.3).

For many years, both the academy and the industry seemed satisfied with the procedural separation between analysis and simulation. The data exchange between both contexts has been very limited (e.g., reporting and/or informal communication between the teams), if not completely absent. But recently, because of the increasing complexity of the considered systems and the raising importance of functional safety, linking both aspects of safety evaluation, i.e., analysis and simulation, has become a challenging research topic within the functional safety community.

Among the research works addressing the link between safety analysis and fault injection, an approach to assist fault definition and insertion with FMEA is described in [94] (more details in 3.3.1). In the industry, there is no standard solution to this problem. However, Yogitech [95] has proposed a method to bring the two contexts together through a software tool-chain (a brief overview of Yogitech's apprach is given in Subsection 3.3.2).

Furthermore, in [96, 97], the authors explore how beneficial safety analysis results can be for the experimental system validation. They investigate the integration of fault injection in early design stages and introduce the concept of *pre-implementation fault injection*, which is very similar to the concept of fault injection in virtual prototypes. By establishing an analogy between this pre-implementation fault injection and the traditional safety analyses (FMEA, FMECA, FTA, etc.,), some capabilities with respect to systematic data exchange and transformation are achieved. The proposal mentions the possibility of guiding fault injection experiments at later development phases, but does not address an important aspect for the industry, namely the quantitative assessment of diagnostic coverage values of safety mechanisms. A feedback loop from the fault injection to the safety analysis is not considered. These limitations are addressed in this thesis with respect to the link between analysis and simulation for functional safety evaluation purposes.

3.3.1 Assisting Fault Insertion with FMEA

In [94], the FMEA process is reviewed from three different perspectives. First, the correlation of FMEA with the requirements definition and with the initial architectural design of the system is emphasized. Second, a set of requirements is given with respect to the definition of the failure mode taxonomy and the execution of the analysis itself. For example, the flexibility of the FMEA data and its understandability by the customer are highlighted. Third, the usage of the FMEA outcome to assist the fault insertion testing is described.

Thereby, the failure modes taxonomy defined in the FMEA is used as a starting point to verify the system behavior through simulation. Moreover, the results obtained by the fault insertion testing are helpful to validate the FMEA, especially with respect to the assumed failure effects.

In general, the approach proposed in [94] represents an initial step towards the linking between safety analysis and fault injection. It is however limited to qualitative FMEA. Other analysis techniques that are relevant for automotive functional safety, such as FTA (2.2.7.6) and DFA (2.2.7.7) are not addressed. In addition to that, the safety mechanisms, their diagnostic coverage values, and the ISO 26262 metrics are completely beyond the scope of the methodology in [94]. That is why, the idea has to be thoroughly refined and extended in this thesis to achieve the intended objectives.

3.3.2 Yogitech's Approach

Since 2006, Yogitech [95] has been working on a platform-based solution for safety verification at System-on-Chip (SoC) level, first according to the IEC 61508 standard and later in compliance with the ISO 26262 automotive standard. The so-called *fRMethodology*, which is documented in several papers [98, 99, 100] and patents [101, 102, 103], relies on a heterogeneous safety evaluation environment, including the FMEA on the one hand and the fault injection and simulation on the other hand.

The first key point of the methodology is the derivation of the *sensible zones*, *observation points*, and *diagnostic points* in the design out of the FMEA. These derived elements are then used as potential locations for fault injection, effect monitoring, and diagnostic mechanisms evaluation. The second key point of the methodology is the possible back-annotation of fault injection results into the FMEA for comparison and verification goals.

The Yogitech methodology covers many aspects of the link between safety analysis and fault injection, which represents a key topic in this work. The approach followed in this thesis has multiple intersection points with Yogitech's fRMethodology such as (i) the concept of data exchange between safety analysis and fault injection, the classification of design elements in sensible zones, observation points, and diagnostic points, and (iii) the feedback loop from simulation to analysis through data back-annotation. These concepts are not claimed to be a new perception in this thesis. However, there are many differences. First, the scope of the thesis is not limited to FMEA, but also covers FMEDA, FTA, and DFA. So, Yogitech's tool-chain does not implement the link between FTA/DFA and fault injection yet. Second, the solution methodology in this thesis is based on metamodeling, code generation, and meta-synthesis (see Section 4.2), which offers more flexibility in comparison to Yogitech's methodology.

Finally, this work addresses conceptual and technical limitations that still persist in the available tool-chains ensuring a linking between analysis and simulation for safety evaluation purposes. For example, the data mapping concept between both contexts, which remains manual in Yogitech's tool-chain for example, is further on investigated in this thesis to have a semi-automated support and subsequently improve the consistency and reduce the efforts. In addition to that, the compatibility with already used formats and styles for safety analysis, particularly FMEDA is targeted in this work through the flexibility and interoperability aspects of metamodeling and metasynthesis. Furthermore, there are more capabilities in the solution developed in the context of this thesis with respect to extendability and reuse, so that a translation to new domains and standards is feasible.

4 Overall Requirements and Solution Fundamentals

To tackle the challenges around functional safety evaluation which have been summarized in the State of the Art Chapter 2 and in the Related Work Chapter 3, comprehensive formalization and model-driven automation approaches for functional safety analysis are conceived and implemented in this context of this thesis.

In Section 4.1, the overall requirements to be satisfied by these approaches are presented. Then, in Section 4.2, the general solution fundamentals which are applied in this thesis to develop the intended approaches are outlined.

4.1 Overall Requirements

In Section 1.4, the different problems and challenges which are addressed in this thesis have been presented. Basically, they consist in:

- the informality and subjectivity issue in the context of functional safety analysis (see Subsection 1.4.1.1),
- the manual character of safety analysis tasks (See Subsection 1.4.1.2), and
- the gap between safety analysis on the one hand and fault injection and simulation on the other hand (see Subsection 1.4.1.3).

To overcome these issues, dedicated approaches, methods, and frameworks are developed in this thesis. The first step in the development process is to define clear and concise requirements which have to be accordingly achieved by the provided solutions. Five different categories of requirements are considered in this section: (i) *structure and formalism* (Subsection 4.1.1), (ii) *flexibility and extendability* (Subsection 4.1.2), (iii) *automation support* (Subsection 4.1.3), (iv) *interoperability and data exchange* (Subsection 4.1.4), and (v) *enhanced usability* (Subsection 4.1.5). Furthermore, all requirements are assigned a unique ID (e.g., **REQ 1)** to simplify further references.

4.1.1 Structure and Formalism

REQ 1 : ***Structured flows***

The procedures of the considered safety analysis approaches, particularly FMEDA (2.2.7.5), FTA (2.2.7.6), and DFA (2.2.7.7) shall be covered and organized in structured flows depicting the multiple steps and the connections between them. Dedicated flowcharts shall be created to illustrate them.

REQ 2 : ***Formal description of safety analysis artefacts***

The data elements which are relevant to the functional safety analysis and the relationships between them shall be formally described. The description must be easily understood by safety engineers, analysts, and other actors involved in the safety evaluation flow. Moreover, it has to be machine-readable to enable an associated automation support.

REQ 3 : ***Suitability of representation formats***

The representation formats of the structured flows already mentioned in **REQ 1** and of the formalized data addressed in **REQ 2** shall be clear, concise, and portable. On the one hand, the steps and the interactions within the analysis flows (**REQ 1**) and on the other hand, the attributes of the data elements and their interdependencies (**REQ 2**) must fulfill these requirements. Hence, they can be easily and consistently handled within the already existing safety evaluation environments or by the new frameworks and/or tools developed in the context of this work.

REQ 4 : ***Compliance with safety evaluation frameworks***

The structured flows (**REQ 1**) and the formalized safety analysis artefacts (**REQ 2**) shall be reflected in the already existing or in the new developed frameworks for safety evaluation. All flow steps must be accordingly supported and a compliant implementation for automation support based on the formalized artefacts must be integrated in a comprehensive environment for safety evaluation including already existing solutions and the additional frameworks and/or tools resulting from this thesis.

4.1.2 Flexibility and Extendability

REQ 5 : ***Feasibility of modifications and extensions***

The safety analysis flows (**REQ 1**) and the formalized safety analysis data (**REQ 2**) shall be easily modifiable and extendable. Specific details in the work flow may differ from one team to another and/or from one project to another. Therefore, it must be possible to easily customize the flow of a certain safety analysis, without altering the core logic of the analysis procedure itself. The same applies for the safety analysis data artefacts. In fact, some specificities may change with respect to the documentation style or the data storage format. That is why, it is necessary to have a mechanism to enable a certain flexibility of the formalized artefacts without losing the generic and standardized core offered by the formalization itself. The potential changes and/or extensions with respect to structure, attributes, and/or relationships must be supported in the form of dedicated versions in compliance with **REQ 6** and **REQ 7**.

REQ 6 : ***Traceability of modifications and extensions***

The modifiability and extendability features addressed in **REQ 5** shall be correlated with an appropriate tracing mechanism to keep track of the deviations of the core solution and to simplify the comparison and the evaluation of different working styles across teams, projects, or completely different organizations. Thereby, all changes and additions made in the safety flows (**REQ 1**) or in the safety data formalization (**REQ 2**) must be uniquely identified, appropriately justified, and conveniently documented.

REQ 7 : ***Appropriate framework updates***

In some cases, the supported modifications or extensions (see **REQ 5**) might induce inconsistencies or malfunctions in the underlying safety evaluation environment. To avoid this, the capabilities of the environment must be improved, either by enhancing the functionality of an existing framework/tool or by developing a new one to support the potentially missing feature. In all cases, a centralized environment per organization shall be targeted, where different tool versions, packages, and/or plugins corresponding to the different team or project requirements are gathered and maintained. An appropriate tracing and versioning mechanism (similarly to **REQ 6**) is required for the framework changes and/or extensions.

4.1.3 Automation Support

REQ 8 : ***Automated data extraction***

To overcome the manual character of traditional safety analysis methods, appropriate mechanisms for data capture and handling shall be developed in compliance with **REQ 1** and **REQ 2**. The manual tasks commonly used for entering the relevant data for functional safety evaluation into the analysis tools (e.g., FMEDA Excel spreadsheet, graphical fault tree editor, etc.,) must be substituted (as far as possible) by dedicated tools for automated extraction from input documents and databases. It should be noted that some information may be directly entered by the safety analyst or engineer without any supporting documents. Thereby, he or she relies on experience, rational thinking, and expert judgment. An example for this is the definition of *expected failure effects* for specific failure modes depending on a given system configuration during the FMEDA. Such manual step cannot be fully automated by the tool. It can however be assisted when the analysis is linked to a fault injection and simulation platform, where failure effects can be monitored after inserting the faults which correspond to the considered failure modes into the design model.

REQ 9 : ***Automated tool synthesis***

As mentioned in **REQ 5** and **REQ 7**, the safety analysis flows and/or the formats of the associated data may change over time, so that the accordingly developed frameworks and tools must be updated. To reduce the efforts of such updates, standard tools for data import/export, handling, and visualization shall be automatically synthesized instead of being manually implemented. It should be noted that in some cases, the synthesized tools are not completely adequate, so that further manual optimizations are required. Nevertheless, the effort reduction through the synthesis of the tool basis remains significant.

REQ 10 : ***Semi-automated link between analysis and simulation***

To address the problematic divergence between the safety analysis context on the one hand and the fault injection and simulation context on the other hand, it is required to develop a mechanism for the semi-automated data mapping between different data sets involved in both aspects. The equivalences and/or correspondences of the data artefacts used in safety analyses (FMEDA, FTA, and DFA) to those used in fault injection and simulation shall be investigated. Subsequently, a data mapping and/or transformation tool must be developed. The degree of automation that can be achieved depends on how far the syntactic and semantic discrepancies between both contexts can be mitigated.

REQ 11 : ***Dynamic data updates***

To enable more reuse and to take advantage of previous experiences, dynamic updates and corrections in input documents and databases for safety analysis shall be supported. For example, for the FMEDA, a failure modes database is commonly used. In addition to the automated data extraction from the database addressed in **REQ 8**, extending it by newly considered failure modes for novel technologies or design methodologies must be supported.

REQ 12 : ***Generation of safety evaluation views and documents***

In compliance with **REQ 1** and **REQ 2**, generation capabilities must be investigated for diverse safety evaluation views such as failure propagation graphs, segments of FMEDA table, fault trees, and fault libraries serving as starting point for fault injection and simulation.

4.1.4 Interoperability and Data Exchange

REQ 13 : ***Exchange and reuse of safety evaluation artefacts***

In compliance with the requirements stated in 4.1.1, 4.1.2, and 4.1.3, the exchange of safety evaluation artefacts shall be supported. For this, links between the different frameworks, and tools applied for safety evaluation must be established. Thereby, data exchange and/or transformation are enabled within the same team or between different teams working on the same project. For example, the communication between the analysis team and the fault injection team must be enhanced through a systematic data transfer and/or reporting. Furthermore, the reuse of safety evaluation data must be supported for related projects addressing different derivatives of the same products for example.

REQ 14 : ***Cross-domain migration of dependability data***

To deal with changes in development environments, technical infrastructures, and/or domain-specific standards for safety and more generically for dependability, data migration capabilities must be investigated. A first use case example for this requirement is the translation of concepts, methods, and tools developed for functional safety in the automotive domain into the railway and/or aerospace domains. A second example is the migration of functional safety data into a new platform developed according to the same approach for security evaluation purposes. It should be noted that this requirement is only partly addressed in the scope of this thesis.

4.1.5 Enhanced Usability

REQ 15 : ***Convenient Graphical User Interface(s)***

For the frameworks and tools developed in accordance with the requirements in 4.1.1 and 4.1.2, convenient and customizable Graphical User Interfaces (GUIs) shall be created. The potential features of these GUIs include (i) *data visualization and handling*: the safety evaluation data can be visualized and modified or extended by the user through the GUI across the safety lifecycle stages, (ii) *facilitated tool usage*: tools and scripts can be graphically invoked through the GUI, (iii) *dynamic data responsiveness*: changing the value of a specific data element leads to the update of all dependent data elements, and (iv) *plausibility checks*: if the user input violates data consistency, an error message shall be issued. It should be noted that this requirement is only partly addressed in the scope of this thesis.

4.2 Solution Fundamentals

This section gives an overview of the solution concepts used in this thesis to achieve the requirements defined in Section 4.1 and to overcome the challenges described in Section 1.4. As already mentioned, the major objectives of the thesis are the formalization and the automation of functional safety analysis as well as its linking to fault injection and simulation. To realize these objectives, a comprehensive approach is developed relying on *Model Driven Development (MDD)* [104, 105] whose basics are outlined in Section 4.2.1. Two related aspects are also particularly relevant for the approach development and implementation, namely (i) *metamodeling and code generation* (see Subsection 4.2.2) and (ii) *metasynthesis* (see Subsection 4.2.3). Furthermore, the technicalities of *data transformation and mapping*, which are applied in this thesis, are briefly presented in Subsection 4.2.4, specially in relation to MDD.

4.2.1 Model-Driven Development

The Model Driven Development (MDD) concept originates from the software engineering domain. It emerged as an alternative organization for the software production cycle and has been increasingly applied in the last two decades to tackle the long lasting complexity, standardization, and efficiency challenges in the software development field. By managing different abstraction layers and separating between multiple system aspects, MDD marks significantly the today's state of the art software implementation flows, helps designers to face competitiveness and time-to-market challenges, and promotes the accelerated production of larger amounts of operational code with improved consistency and expressiveness [5].

From a conceptual point of view, MDD consists in the elevation of the abstraction level at which software is developed by using models and model technologies [104]. Instead of considering all possible details, which depend on the design configuration and the surrounding environment, software developers focus on the functionality and the general architecture of the intended applications and build appropriate models depicting them. Indeed, at least considerable parts of these applications are not implemented per hand anymore, but systematically and automatically derived from the corresponding models. For this purpose, several computer-based technologies are applied to enable the transformation of models into running implementations [8, 106].

Representing the fundamental element in MDD, a *model* is defined as a *"coherent set of formal elements describing something [...] built for some purpose that is amenable to a particular form of analysis"* [105]. In other words, a model is an abstract description of a certain system aspect, which is both human-understandable and machine-readable. Consequently it can be either further on refined by the developers or provided as input to a dedicated software tool. Thereby, a systematic analysis of the model results into an additional representation at a lower level of abstraction or produces the targeted implementation [5].

There are several frameworks implementing the MDD concept. However, the most popular and frequently used MDD framework is the Model Driven Architecture (MDA) [6]. Defined by the Object Management Group (OMG) (an international industrial organization concerned with software interoperability in many domains such as telecommunications and manufacturing [105]), MDA relies particularly on the Unified Modeling Language (UML) language to create system models at different abstraction levels and to develop appropriate tools for automation and generation support [105, 7].

MDA divides the software system into a platform independent specification (core business logic) and a platform dependent implementation technology (configuration characteristics of development environment such as hardware and operating system) [7, 6, 107]. Hence, two types of models are differentiated in the MDA context:

- *Platform Independent Model (PIM)*: It is an abstract description of system functionality and behavior. All details related to the implementation technology and development platform are discarded in the PIM. That is why, it can be used on many platforms related to the same domain. The PIM developer is concerned only with the general problem. Thus, he or she analyzes the appropriate approaches to solve it without taking hardware characteristics, programming language semantics, and other details into account [104, 107, 6].

- *Platform Specific Model (PSM)*: Combining the system specification (PIM) and the characteristics of the underlying hardware and software platform, the PSM is the representation of the detailed system implementation containing all relevant information about the concrete behavior in a certain environment [104, 107, 6].

Hence, to apply the Model Driven Architecture, a PIM is first created. It specifies the functionality of the target system and remains the same throughout all possible technological changes. Then, one or more PSMs are derived from the PIM in compliance with the supported platform(s), the technology characteristics, and the implementation preferences of the software developer [5]. Finally, different *system views* depicting the lowest level of abstraction and having the highest level of detail (e.g. system code, test bench, verification files, documentation, etc.,) are generated from the PSM [104, 108]. In Figure 4.1, an MDA overview is illustrated along with the relationship between the PIM and the PSM.

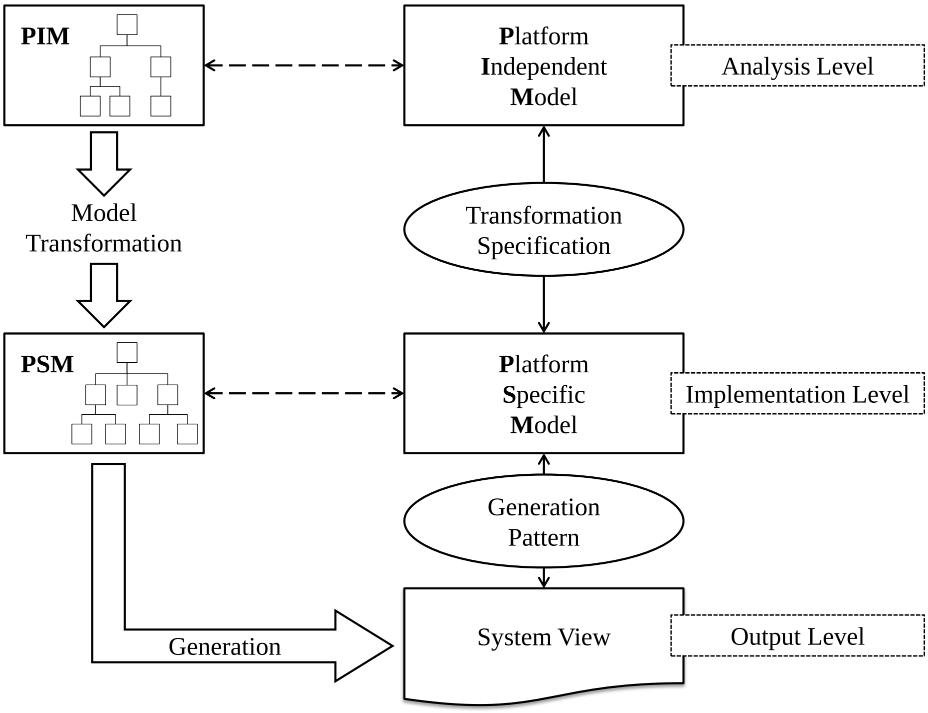


Figure 4.1: Model-Driven Architecture Overview [5, 6, 7]

As already mentioned earlier, MDD has progressively become an established methodology within the software development community because of its several benefits listed in [5]. It also started to gain popularity within other domains, such as hardware engineering and functional safety evaluation [109, 110].

- *Productivity*: Through abstraction, MDD offers a significant reduction of the technical details that must be manually manipulated by the developers. They can henceforth focus on the overall logic, functionality, and architecture of the targeted applications instead of considering minor implementation details. Subsequently, the time-to-market is considerably reduced, especially in the case of multi-platform deployment [107].
- *Interoperability and Reusability*: The PIM/PSM separation makes it possible to

export multiple domain-specific solutions on different platforms. Thereby, the reuse factor is improved and drastic time and effort savings are achieved [8, 107].

- *Automation Level*: By automatically generating system views from PSMs, consistency, quality, and expressiveness levels are improved. In fact, syntactic errors are minimized, a well-defined coding style is systematically followed, and extra *finalization items* such as comments can be conveniently created [6, 107].

4.2.2 Metamodeling and Code Generation

Beyond models, which are the basis of Model Driven Development (4.2.1), *metamodels* offer an additional abstraction depicting model properties, describing the relationships between their elements, and defining the constraints they must comply with [111, 112]. *Meta* is a Greek prefix which means "after" or "beyond". It appears in the terms *metadata* and *metamodel* denoting "data about data" and "model for model" [113, 114]. For example, when a given set of *objects* is considered to be a model, then the corresponding metamodel is the group of *classes* whose *instances* are the model objects. Similarly, programming language grammars may be considered as metamodels and the respectively created programs as models [111].

Metamodeling is commonly used in software development to describe so-called *Domain Specific Languages DSLs* and define their syntax and semantics. The level of abstraction of DSLs makes them valid for different development environments and platforms associated with the same domain, subsequently leading to more interoperability and reuse [5]. In other words, metamodeling allows a translation of the developer's focus from the solution domain to the problem domain. It also enhances the design tasks ranging from the early specification to the final verification through concise meta-information and well-structured representation formats [115, 116].

A concrete reflection of the metamodeling concept in association with MDD is given within the Model Driven Architecture defined by OMG and already introduced earlier. In fact, MDD relies on a standard metamodeling infrastructure including four modeling levels called *metalayers*, which are hierarchically organized and connected by "instance-of" relationships. This hierarchy of metalayers [8, 117] is illustrated in Figure 4.2 and commented below.

- *M0 Level*: This level addresses the concrete user data describing the observed system (e.g., data objects to be handled by software applications).

- *M1 Level*: A this level, a model of the M0 user data is built. It contains a structural and/or behavioral description of the system. M1 models are constructed using domain-specific concepts and are frequently stored and/or visualized as UML object diagrams.

- *M2 Level*: Here, models of the M1 models are constructed and commonly denoted as metamodels. Such metamodels represent the domain-specific modeling language expressing the structure and the semantics of the appropriately defined meta-data. In many cases, UML class diagrams are used to create and/or display metamodels.
- *M3 Level*: At this level, a model of the M2 metamodel is built. It consists in a *self-describing* (*metacircular*) meta-metamodel which is capable of specifying DSLs. This level is also referred to as Meta Object Facility (MOF) [105].

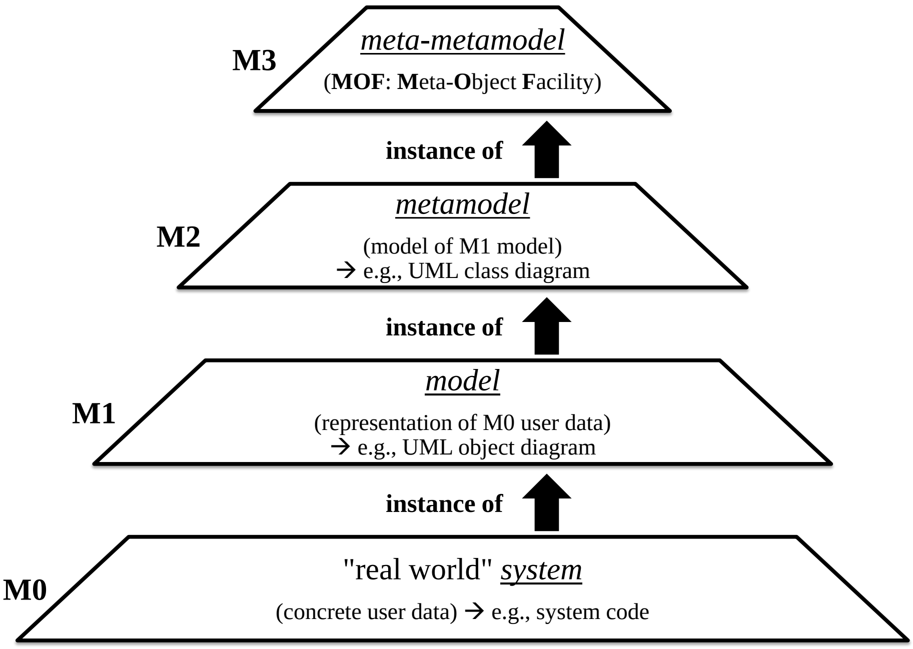

M3
meta-metamodel
(MOF: Meta-Object Facility)
instance of
M2
metamodel
(model of M1 model)
→ e.g., UML class diagram
instance of
M1
model
(representation of M0 user data)
→ e.g., UML object diagram
instance of
M0
"real world" system
(concrete user data) → e.g., system code

Figure 4.2: Object Management Group Metamodeling Layers [5, 8, 9]

In the last decade, in addition to its frequent usage in software development, metamodeling has also been increasingly applied in hardware design, particularly to enable reuse of Intellectual Property (IP) in System-on-Chip (SoC) design [5]. In this context, the IP-XACT standard [118] has been introduced by the SPIRIT consortium. It provides a generic metamodel for the description of IP-modules and their unified specification through a well-structured meta-data pattern. With IP-XACT, the automation level of IP creation, configuration, and integration is enhanced. Moreover, productivity improvement and time-to-market reduction are reached thanks to the exchange and reuse factors promoted by vendor-neutral IP descriptions in compliance with IP-XACT [118, 119].

Another important solution concept for this thesis is *code generation*, which is inherently related to MDD and metamodeling. In fact, the automated derivation of complex, detailed, and verbose source code from the correspondingly constructed simple, abstract, and concise models is a key objective in this work.

Today, the omnipresence of code generation approaches in software and hardware projects is obvious. It is motivated by the rising concerns about (i) code size, (ii) memory requirements, and (iii) error monitoring and handling across the development stages [5]. Therefore, several generation techniques are applied to consume less resources and simplify the analysis, verification, and optimization tasks for the developers. However, the benefits of code generation can only be realized when the corresponding code generators are carefully implemented, taking into account the required properties of the intended code with respect to the architecture, the functionality, and the behavior. Certainly, fulfilling such requirements necessitates investing significant efforts in the generator writing task. Nevertheless, these efforts will be compensated through the boost of performance, effectiveness, and efficiency resulting from the application of generated code instead of manually-written code [10].

Among the most relevant code generation techniques, three are mentioned below [5, 10]. A more detailed explanation of these and other techniques can be found in [5].

- *Template-Based Code Generation*: Templates are special files which define the layout of the intended output. They are either applied to filtered parts of textual specifications (e.g., XML files) or to metamodel instances which have been correspondingly filled with the specification data. The concept of template-based code generation using metamodels is illustrated in Figure 4.3.
- *API-Based Code Generation*: Using an *Application Programming Interface (API)*, which relies on the syntax of the programming language of the expected code, the abstract structure from which the code shall be derived is systematically traversed and appropriately transformed by the code-generating programs, which apply the functionalities and features provided by the API.
- *Inline Code Generation*: By compiling or interpreting manually written programs, which have been appropriately processed by dedicated pre-compilers, the expected code is produced.

In the context of this thesis, two of the code generation techniques mentioned above are applied, namely the template-based and the API-based techniques.

4.2.3 Metasynthesis

In Subsections 4.2.1 and 4.2.2, the critical challenges of automation and flexibility in software and hardware design have been extensively addressed, as well as the commonly used methodologies to tackle them, namely Model Driven Development,

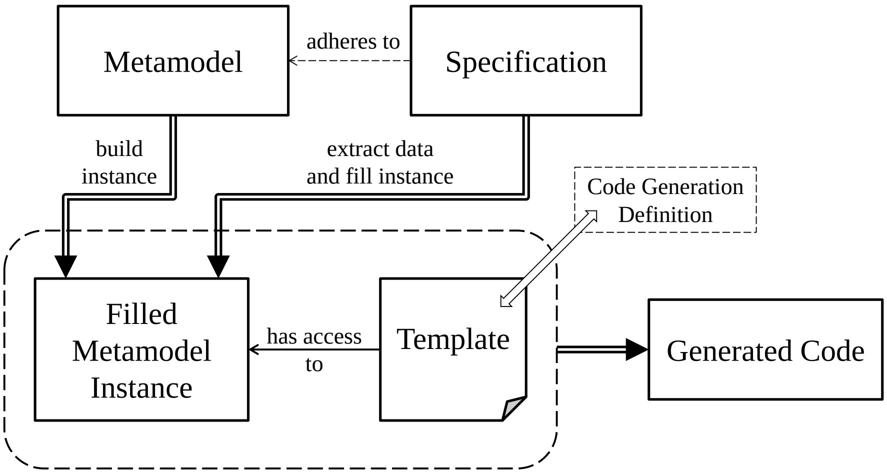

Metamodel
adheres to
Specification
build instance
extract data and fill instance
Code Generation Definition
Filled Metamodel Instance
has access to
Template
Generated Code

Figure 4.3: Template-Based Code Generation using Metamodeling [5, 10]

metamodeling, and code generation. These methodologies rely on building abstract descriptions (models and/or metamodels) and applying generation techniques to subsequently derive concrete system implementations, views, and applications from them. However, these capabilities are mainly ensured by software infrastructures (containing modeling platforms, programming and editing frameworks, synthesis tools, etc.,), whose implementation (mostly manual) is still a complex and time-consuming procedure. In [120], a new concept called *metasynthesis* is introduced to address the unsupported fields of automation by the Electronic Design Automation (EDA) industry, especially in the automotive SoC design domain. In the following, the general concepts and terms of metasynthesis are first presented. Then, a quick overview of metasynthesis applications is given.

- **General Concepts of Metasynthesis**:
  Beyond the *system synthesis* concept, which is becoming an intrinsic part of ESL design and verification, metasynthesis is a flexible technique aiming at the synthesis of the system synthesis tools themselves. Once automatically synthesized, such tools transform an abstract description (e.g., requirements list, tabular specification, graphical block diagram) into a concrete implementation (e.g., C program, RTL code, or netlist). The naming of the methodology reflects (ii) the idea of having *a synthesis tool "beyond" another synthesis tool* and also (ii) the underlying metamodeling technology [120]. It should be noted that the two-step synthesis approach is correlated with additional efforts resulting (i) from the development of the metasynthesis infrastructure (reusable platform implemented only once and then appropriately maintained) and (ii) the creation of the system synthesis tools (multiple tools depending on the use cases).

However, the high automation level of the second step and the input properties with respect to simplicity and compactness lead to an overall reduction of the design time [120].

- **Metasynthesis Applications**:
  Metasynthesis is a widely applied approach at Infineon. It has already about 100 use cases and it contributes significantly to the productivity improvement in automotive SoC design. The effort reduction reaches 70% due to the removal of repetitive error-prone manual tasks mainly related to data entry and there are reports that the synthesized tools are able to generate up to 80% of the overall RTL code of certain chips [120]. Beyond automotive products, metasynthesis addresses other products at Infineon which are related to energy efficiency and to security for example. With respect to ISO 26262's safety lifecycle presented in Subsection 2.2.1, metasynthesis applications covers the majority of the stages, e.g., the requirements specification, HW design (virtual prototyping using TLM and RTL design), SW design (more precisely firmware), verification, and test [120]. That is why, it is a central solution concept in this thesis, especially with respect to the requirements stated in Subsection 4.1.3.

4.2.4 Data Transformation and Mapping

With respect to requirements **REQ 10**, **REQ 13**, and **REQ 14**, data exchange, transformation, and mapping between multiple tools, platforms, and domains related to functional safety evaluation shall be supported in the context of this thesis. In the following, the model and data transformation challenge in MDD, which is the fundamental solution concept for this work, is briefly studied. Then, a brief general overview of data mapping techniques is given.

- **Model and Data Transformation in MDD**:
  In [106], the major challenges related to model transformation in MDD are presented. In general, model transformation is a relationship between a *source model* and a *target model*. The transformation mechanism takes the elements of the source model as input, manipulates them appropriately, and subsequently produces the elements of the target model. There are different types of transformations, such as:
  - *model refinement* (more details are added to the source model elements),
  - *model composition* (multiple source models are integrated together into one single target model),
  - *model abstraction* (certain details of the source model are discarded),
  - *model decomposition* (one single source model is segmented in many target

models), and

  - *model translation* (the source code is translated into another language to produce the target model) [106].

  Among the known approaches for data transformation in model-driven development, OMG's Query/View/Transformation (QVT) standard addresses the description of relations at different levels of abstraction either in a declarative or in an imperative manner [106].

  Many challenges are associated with model transformation [106] such as:

  - The consistency maintenance across different views: plausibility checks and reviews are required.
  - The testability of the transformation with respect to its correctness and/or efficiency: test cases based on target codes are needed.
  - The integration of model-to-code transformations with manually-written code or with legacy code: "glue" code recommended to implement the missing interfaces.

- **Data Mapping**:
  Data mapping is an important aspect in data management and computing. It consists in creating links between two different data models. These links are the basis for further data handling activities, such as:

  - *data transformation* addressed above,
  - *data mediation* which is a special type of data transformation where a third *mediating* data model is used to establish the connection between the source and the target,
  - *data lineage analysis* which deals with the provenance of data and the visualization of its flow of movement from its origin to its destination, and
  - *data gathering and consolidation* with the objective of building a comprehensive database without unnecessary redundancies for example.

  There are many approaches to perform data mapping. The most relevant are:

  - *Hand-coded, graphical and/or manual mapping*: In this approach, a procedural code is manually written to bring together the elements of the source model on the one hand and the elements of the target model on the other hand. Alternatively, a graphical interface is used to create the mapping links.
  - *Data-driven mapping*: Here, advanced heuristic and statistical techniques are used to detect similarity patterns between the considered data sets.

- *Semantic mapping*: In this approach, semantic equivalences with respect to meta-data information are applied through dedicated data mapping tools to establish the intended links.

In this thesis, both *hand-coded, graphical and/or manual mapping* and *semantic mapping* are applied.

5 Metamodeling-Based Formalization of Functional Safety Analysis

In this chapter, the formalization strategies developed in the context of this work to address functional safety analysis are presented.

First, a general introduction is given in Section 5.1, highlighting the main objectives of the intended formalization in relation to the overall requirements explained in Chapter 4 and displaying the basic techniques enabling such formalization.

Then, Sections 5.2 and 5.3 address the two central formalization approaches implemented during this thesis with respect to their theoretical basics and to their technical implementation.

5.1 Introduction

In this section, the main objectives of the formalization are briefly recapitulated (Subsection 5.1.1) as well as the technical enablers which are applied to achieve it (Subsection 5.1.2). Afterwards, the organization of the chapter remainder is outlined (Subsection 5.1.3).

5.1.1 Main Objectives

As already addressed in Section 1.4 of the introduction chapter, functional safety evaluation has become a central topic in the industry over the last decades. Due to the significant importance and the considerable challenges it represents, it also has been extensively addressed by the research community to develop appropriate theories, approaches, and tools, particularly for functional safety analysis. Among the frequently studied topics, (i) fault and failure modeling, (ii) failure propagation analysis, and (iii) model-based support for safety analysis techniques have already been introduced in Sections 3.1 and 3.2 of the related work chapter. Nevertheless, many issues still persist, such as the informality of the available descriptions for modeling strategies and analysis techniques. The subjectivity of such descriptions and their lack of structure and standardization make them difficult to understand and to apply. Furthermore, they limit the opportunities for data exchange and/or reuse.

Hence, the formalization aspect is addressed in this thesis to tackle such inconveniences, reduce the tedious and error-prone characteristics of functional safety analysis, and create a solid basis for the later development of systematic tools to assist safety engineers and analysts throughout the safety lifecycle stages, particularly in automotive applications in compliance with the ISO 26262 standard.

Concretely, the objectives to be achieved through formalization are intrinsically correlated with the overall requirements defined in Chapter 4, especially those of Subsections 4.1.1, 4.1.2, and 4.1.4. These objectives are listed below along with references to the respective requirements.

- Depict the tasks of safety analysis methods in structured flows to simplify their reflection through formal descriptions (**REQ 1**).
- Ensure that these flows are easily modifiable and extendable in accordance with team, application-domain, and/or project specificities (**REQ 5**).
- Provide a formal description of safety analysis artefacts and interactions between them which is both understandable by safety engineers and supportable as input for automated framework synthesis and tool generation (**REQ 2**).
- Ensure the clarity, the portability, and the flexibility (with respect to changes and/or additions) of the representation formats used to depict the formal description related to functional safety (**REQ 3** and **REQ 5**).
- Keep track of all changes and/or extensions performed during the formalization procedure and enable a comparison mechanism between different versions of the formalized descriptions (**REQ 6**).
- Anticipate later exchange and/or reuse of functional safety data through dedicated features in the formalized descriptions of the different analysis techniques, such as references and mapping patterns (**REQ 13** and **REQ 14**).

5.1.2 Technical Enablers

The achievement of the objectives listed in Subsection 5.1.1 is enabled through three basic solution concepts that have already been presented and detailed in Section 4.2. In the context of Model Driven Development (MDD) (see Subsection 4.2.1), models are used to abstract certain system aspects and to describe them in a formal, concise, and reusable way. In this thesis, the focus is on functional safety as the system aspect to be covered through this modeling concept. Thereby, data models are used to substitute large and cumbersome documents addressing system safety at different levels of abstraction and on different stages of the design and manufacturing cycle. Building such models has been already addressed in the literature and has also become a concern in the industry because of the increasing relevance and complexity of safety-related data. However, the structure, properties, and constraints of these models have been often neglected, although they represent the basis for a consistent and efficient application and deployment across organizations, domains, and projects. Therefore, MDD is correlated with two further technical enablers in this thesis, namely metamodeling and code generation (see Subsection 4.2.2). Abstracting models by providing well-defined and well-structured syntactic and semantic descriptions for them is the primary function of metamodels (more details are given in Subsection 4.2.2). Moreover, code generation is applied on top of metamodels and in compliance with different techniques mainly to support the construction of associated metamodel instances (i.e., data models) and derive appropriate views out of them (the details are also given in Subsection 4.2.2). In order to emphasize the formalization strategy using MDD and metamodeling, the two basic terms *model* and *metamodel* are formally defined below [109, 57].

Def. 1 *A metamodel is a tuple* $\mathscr{M} = (\mathbb{M}, \mathscr{C}, \mathscr{R})$*:*

- $\mathbb{M}$ *is the self-describing meta-metamodel which is capable of specifying all metamodels, including itself.*
- $\mathscr{C}$ *is a set of classes. Each class* $c \in \mathscr{C}$ *determines syntactical properties of later instances such as types, multiplicities, and potential attribute initializations.*
- $\mathscr{R}$ *is a set of relationships between classes including associations, references, and inheritances.*

Def. 2 *A model is a tuple* $\mathcal{M} = (\mathscr{M}, \mathcal{O}, \mathcal{R})$*:*

- $\mathscr{M}$ *is the metamodel formalizing the structure of* $\mathcal{M}$*.*
- $\mathcal{O}$ *is a set of valid class instances referred to as objects. Each object* $o \in \mathcal{O}$ *is an instance of a class* $c \in \mathscr{M}.\mathscr{C}$*. It is characterized by a unique identifier, a set of attributes, and a set of related objects.*
- $\mathcal{R}$ *is a set of object links such as associations, references, and inheritances (instances of* $r \in \mathscr{M}.\mathscr{R}$*).*

5.1.3 General Organization

In the remainder of this chapter, the developed formalization approaches for functional safety analysis in the context of this thesis are presented. First, *Metamodeling-based Failure Propagation Analysis (MetaFPA)* is addressed in Section 5.2 mainly through (ii) a methodology overview, (ii) an explanation of the new enhancements of the already publicized Failure Logic Modeling theory (see Subsection 3.1.2), and (iii) the detailed description of the created metamodel for failure propagation modeling. After that, the metamodeling-based formalization of classical safety analysis techniques is described in Section 5.3. For the three considered techniques FMEDA (2.2.7.5), FTA (2.2.7.6), and DFA (2.2.7.7), the respective flows and documentation formats are depicted. Then, the correspondingly formalized descriptions through metamodels are presented.

5.2 MetaFPA: Metamodeling-Based Failure Propagation Analysis

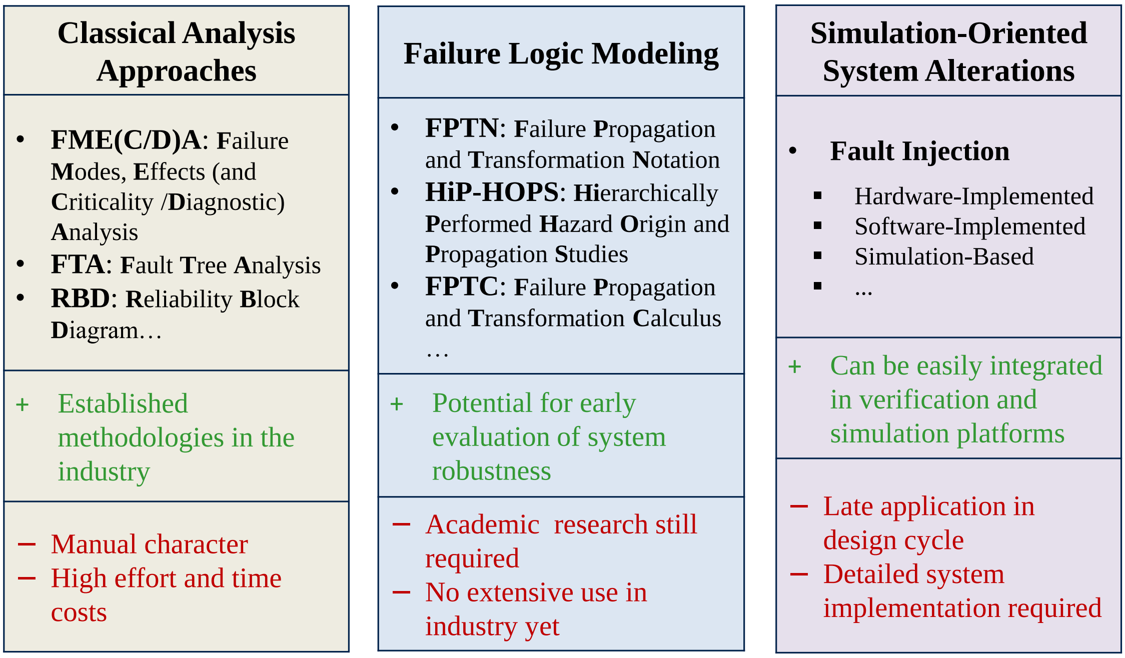
In Chapter 2, the significance of system robustness against faults, errors, and failures has been emphasized. Multiple robustness aspects are addressed throughout the system design and manufacturing cycle, but the focus of this work is functional safety, i.e., the absence of catastrophic consequences on the users and the environment.

Functional safety is particularly important in safety-critical domains where human lives can be harmed in failure cases. Therefore, it gets evaluated on multiple stages of the product lifecycle and at different abstraction levels. Functional safety evaluation starts already during the concept phase. At that early stage, analytical and probabilistic methodologies are applied to perform Hazard Analysis and Risk Assessment (HARA) either qualitatively or quantitatively. The latest step of functional safety evaluation is the physical prototype testing where the system hardware and software get certified with respect to the safety requirements. In between, simulation-oriented system alterations are carried out through fault injection campaigns.

Several new model-based safety assessment approaches have been proposed during the last two decades to reduce the gap between the classical safety analysis approaches on the one hand and the simulation-oriented system alterations on the other hand (see Figure 5.1). These approaches try to address the limitations of the usual techniques on both sides. They are referred to in the literature as Failure Logic Modeling (FLM) techniques [64, 66].

The basis of FLM methods is the compositional description used to evaluate system safety at an early stage. For this, the failures affecting the system components either internally or on the interfaces are specified and the cause-to-effect relationships are defined.

Although these methodologies show potential for early evaluation of system robustness, they still require further academic research and need examination of their productive use in the industry. Indeed, this is one of the motivations of this work.



Classical Analysis Approaches	Failure Logic Modeling	Simulation-Oriented System Alterations
• **FME(C/D)A**: Failure **M**odes, **E**ffects (and **C**riticality /**D**iagnostic) **A**nalysis • **FTA**: **F**ault **T**ree **A**nalysis • **RBD**: **R**eliability **B**lock **D**iagram…	• **FPTN**: Failure **P**ropagation and **T**ransformation **N**otation • **HiP-HOPS**: **Hi**erarchically **P**erformed **H**azard **O**rigin and **P**ropagation **S**tudies • **FPTC**: Failure **P**ropagation and **T**ransformation **C**alculus …	• **Fault Injection** ▪ Hardware-Implemented ▪ Software-Implemented ▪ Simulation-Based ▪ ...
+ Established methodologies in the industry	+ Potential for early evaluation of system robustness	+ Can be easily integrated in verification and simulation platforms
– Manual character – High effort and time costs	– Academic research still required – No extensive use in industry yet	– Late application in design cycle – Detailed system implementation required

Figure 5.1: Failure Logic Modeling as a Bridge between Classical Safety Analysis and Fault Injection and Simulation

In this section, FLM techniques are addressed from three different perspectives: (i) formalized definitions and descriptions of failure logic modeling aspects (**REQ 2**), (ii) enhanced compatibility with conventional simulation-oriented system modeling guidelines (**REQ 4** and **REQ 10**), and (iii) enhanced flexibility, reusability, automation, and maintenance capabilities (**REQ 5** and **REQ 7**). The objective is to use them for high-level failure propagation analysis in industrial applications.

For this purpose, the MetaFPA approach is introduced [11], offering a comprehensive platform for failure propagation analysis and simulation. The remainder of this section is organized as follows. First, an overview of the MetaFPA methodology is given in Subsection 5.2.1. Then, the extensions provided by MetaFPA in comparison to the Failure Logic Modeling (FLM) theory are explained in Subsection 5.2.2. Afterwards, Subsection 5.2.3 presents the metamodel developed in the MetaFPA context to formalize failure logic and propagation modeling.

5.2.1 Methodology Overview

As already mentioned, FLM methodologies represent the academic starting point to reduce the gap between the analytical safety analysis and the simulation-oriented system alteration consisting in the fault injection. The challenge in this work is to build upon the concepts offered by the FLM theory to develop a formalized technique for failure propagation analysis which is more compatible with the simulation world and to create a corresponding platform with extensive and flexible capabilities.

Subsection 3.1.2 gives an overview of the theoretical basics of the most relevant FLM approaches such as FPTN (3.1.2.1), FPTC (3.1.2.1), and HiP-HOPS (3.1.2.3). It shows the compositional description of the system structure and the interdependencies between the contained components in failure cases. In fact, FLM methods model the system as a set of interacting components and specify their relationships in terms of potential deviations from the initial *design intent* [66, 11]. In addition to that, FLM techniques describe the *failure flow* across the defined structure by defining the logical relationships between input deviations, internal malfunctions, and output deviations.

The FLM basics mentioned above remain valid for MetaFPA (see Figure 5.2). Indeed, the system is modeled as a set of blocks in an abstract description provided commonly as a block diagram by concept engineers. The potential threats which might affect the system blocks are then specified. Possible deviations on the input and the output interfaces of the blocks are considered as well as the malfunctions which may occur internally inside the block.

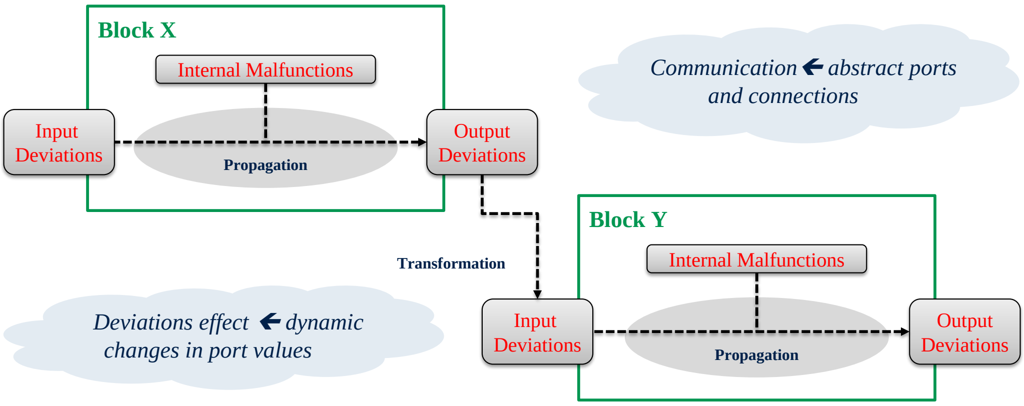


Figure 5.2: Basic Idea of the MetaFPA Approach [11]

In MetaFPA, the failure logic is taken into account. The relationships between the different threats inside the blocks and also between the different blocks are covered. The mapping between input deviations and internal malfunctions on the one hand and output deviations of a specific block on the other hand is referred to as *propagation*. Furthermore, *transformation* is the term used to denote the connection between output deviations of one specific block and input deviations of the next blocks in the system architecture. This information, which is related to the behavior of the system blocks in failure cases, can theoretically be provided by the user of the MetaFPA platform. However, practically, it is derived from already existing safety data which is available for the different system components, e.g., component FMEDA tables, local fault trees, etc. The essential benefit of reusing the available knowledge about the failure behavior of the system blocks is the evaluation of safety integrity level of the complete system in a different way from the classical analysis approach [11].

The main difference is actually the new simulation aspect provided by MetaFPA. This aspect is realized through the usage of ports and connections to dynamically capture the communication between the blocks. In comparison to the traditional FLM techniques, where the failure behaviors of the different blocks are statically defined upfront, MetaFPA enables a dynamic computation of the block state including the internal malfunctions and the values of the respective ports. This dynamic computation is based on the propagation and the transformation patterns. In fact, deviation effects are modeled through changes in the port values. Furthermore, the connections are activated to capture how the failures spread across the system structure [11].

Based on the general concepts and ideas described above, a comprehensive methodology is developed to support the intended model-based and simulation-oriented failure propagation analysis at higher levels of abstraction and on early concept stages of the safety lifecycle. In compliance with ISO 26262, MetaFPA is applicable during the concept phase (Part 3) to assist the specification of the safety integrity levels and the functional safety requirements. The outcomes of the failure propagation analysis at this level affect the safety concept and enable a more efficient exploration of the intended safety mechanisms with respect to their adequacy and effectiveness in covering the potential threats which may affect the system. An overview of the MetaFPA methodology is given in Figure 5.3.

The initial step in the development of the MetaFPA methodology is the construction of the Metamodel for Failure Propagation Analysis which formalizes the abstract system modeling on the one hand and the failure propagation modeling on the other hand. Confirming to this metamodel, the MetaFPA platform is created. For this, the Metasynthesis concept (see Subsection 4.2.3), which is based on synthesizing tools [120], is used. Within the synthesized MetaFPA platform, generated tools such as standard readers, parsers, and writers for common data formats like XML and XLS are used to build, visualize, and manipulate data models, which are actually instances of the metamodel. In addition to these generated tools, handwritten extensions, plugins, and further generators are developed, always confirming to the terms and to the structure of the metamodel. As previously mentioned, flexibility is one of the key requirements in the context of this thesis (**REQ 5**). The fulfillment of this requirement in the MetaFPA context is illustrated in Figure 5.3 by the customization facility offered at the *User Input Interface* to change, configure, or extend the metamodel depending on specific needs or requirements either for the system description or for the failure behavior specification. The user can also implement targeted add-ons for the

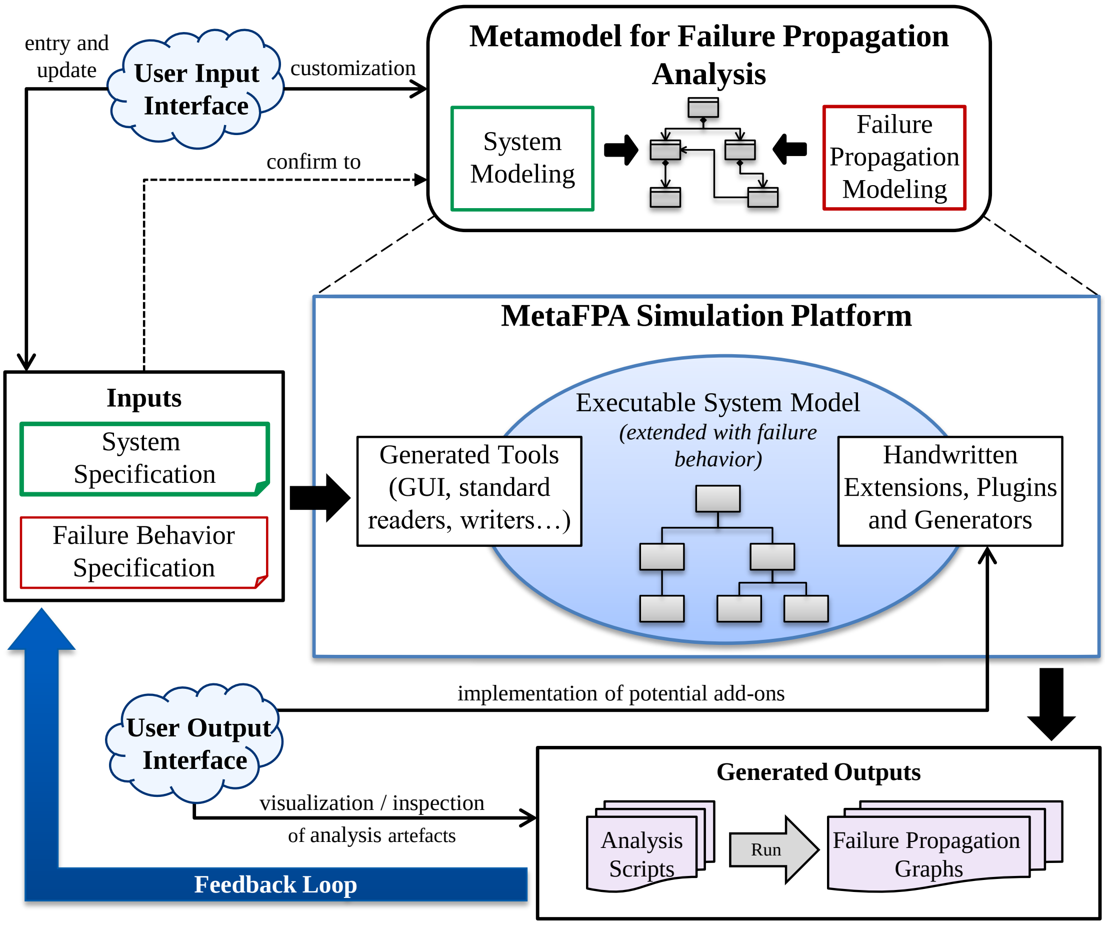

entry and update
User Input Interface
customization
confirm to
Metamodel for Failure Propagation Analysis
System Modeling
Failure Propagation Modeling
MetaFPA Simulation Platform
Inputs
System Specification
Failure Behavior Specification
Executable System Model
(extended with failure behavior)
Generated Tools (GUI, standard readers, writers…)
Handwritten Extensions, Plugins and Generators
implementation of potential add-ons
User Output Interface
visualization / inspection
of analysis artefacts
Generated Outputs
Analysis Scripts
Run
Failure Propagation Graphs
Feedback Loop

Figure 5.3: Overview of the MetaFPA Methodology [11]

MetaFPA platform at the *User Output Interface*, such as special readers or writers if a new format of input or output data is used. Further details about the MetaFPA analysis and simulation platform and its use-case dependent customizations are given in Section 6.2.

Once the MetaFPA platform is adapted to the special requirements of the use case, the application starts with the entry of the user inputs consisting of the system specification confirming to the system modeling part of the metamodel and the failure behavior specification related to the failure propagation modeling rules. *Entry and update* in Figure 5.3 do not necessarily depict manual tasks. They also refer to the automated extraction of information delivered by concept engineers and safety analysts in structured formats. The model filled with the extracted data from these inputs is the basis on which the failure propagation analysis is performed. Python scripts are generated, covering a big number of faulty configurations of the system.

Running these scripts produces the analysis results which are displayed in the form of tree graphs generated in the XML based GraphML format. These graphs document the runs leading to critical system failures. An example of a generated graph using MetaFPA is given in Figure 5.4. More details about the structure of the graph and the use case behind it are given later in Subsection 6.2.3.

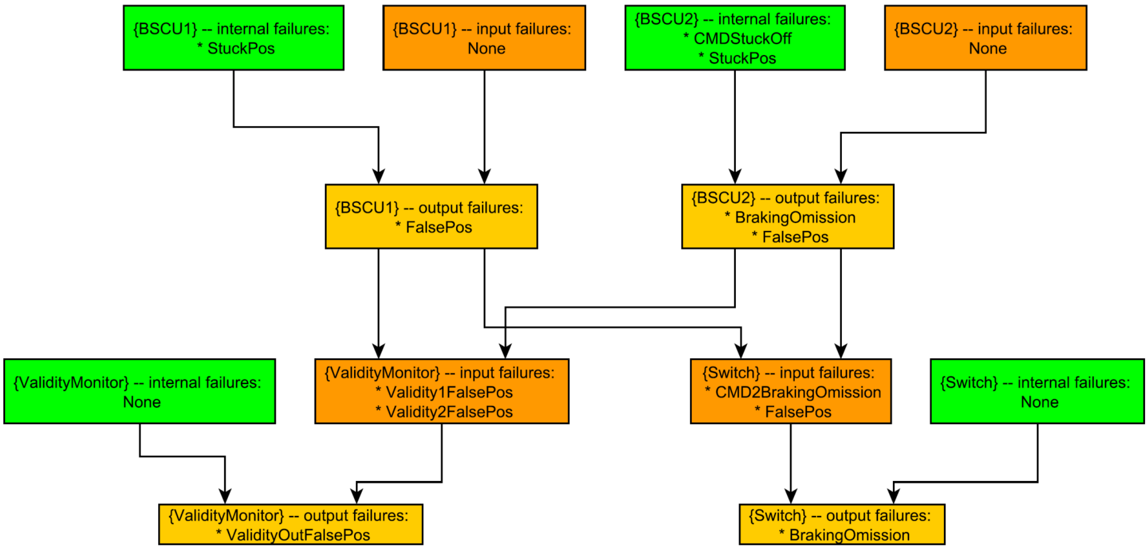

{BSCU1} -- internal failures:
* StuckPos
{BSCU1} -- input failures:
None
{BSCU2} -- internal failures:
* CMDStuckOff
* StuckPos
{BSCU2} -- input failures:
None
{BSCU1} -- output failures:
* FalsePos
{BSCU2} -- output failures:
* BrakingOmission
* FalsePos
{ValidityMonitor} -- internal failures:
None
{ValidityMonitor} -- input failures:
* Validity1FalsePos
* Validity2FalsePos
{Switch} -- input failures:
* CMD2BrakingOmission
* FalsePos
{Switch} -- internal failures:
None
{ValidityMonitor} -- output failures:
* ValidityOutFalsePos
{Switch} -- output failures:
* BrakingOmission

Figure 5.4: Example of Generated Failure Propagation Graph using MetaFPA

The analysis results can be observed and inspected by the user. With the knowledge gained from this inspection, it becomes possible to define actions to improve the system quality with respect to its functional safety. These actions are actually recommendations to revise the high-level system specification by adding new safety mechanisms for example. A semi-automation capability in the context of this feedback loop is investigated. Thereby, the enabling of systematic extraction of candidates for proposed mitigation techniques is targeted. The essence of the feedback remains however a subjective task requiring human intelligence and hence an unavoidable contribution of the user. However, the offered semi-automation by MetaFPA offloads the user from simple, repetitive, and error-prone tasks.

5.2.2 Failure Logic Modeling Extensions

In this part of the thesis, the primary challenge is to overcome the limitations of the failure logic modeling theory. MetaFPA builds upon the traditional FLM techniques in order to overcome the limitations given in Figure 5.1 consisting in the need for further academic research on the one hand and for industrial evaluation and qualification on the other hand. Moreover, MetaFPA addresses different technical weaknesses that have been witnessed in existing FLM tools and platforms.

In fact, MetaFPA provides alternatives and/or enhancements in comparison to the already existing FLM approaches with respect to the following three aspects: (i) *dynamic failure description*, (ii) *flexibility*, and (iii) *automation level*. These three aspects are detailed below.

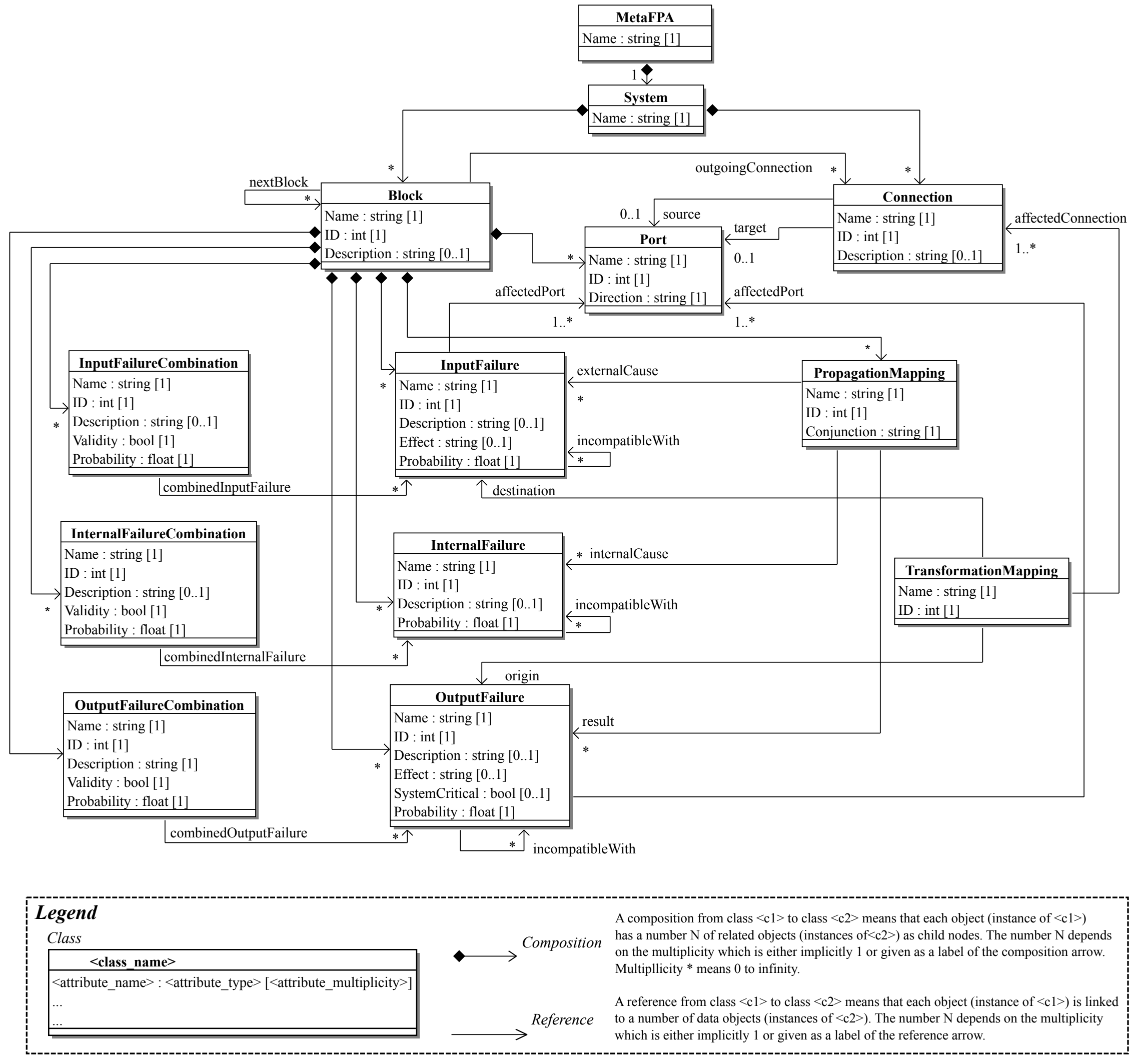
- **Dynamic failure description**: In known FLM approaches, failure logic is described statically. In fact, the specification of the component failures and most importantly the mapping between the different components is predefined and manually entered. As an alternative to the static description, MetaFPA captures the failure behavior of the system in a dynamic way which is more compatible with the conventional simulation-oriented modeling styles. This is enabled by the abstract ports and connections described in Subsection 5.2.1.

- **Automation level**: FLM has a limited automation level. Until now, there was no platform which offers a partial synthesis of the tools used to implement the failure logic modeling concepts or provides a mechanism to systematically create analysis artefacts (such as programs or scripts) to assist the safety engineer or analyst. The tool support described in the literature requires a considerable contribution of the user not only in terms of modeling but also in terms of writing and/or programming the necessary analysis routines. In contrast to that, MetaFPA offers a high level of automation due to the model-driven support, the underlying metamodeling and code generation techniques, the application of the metasynthesis concept to automatically generate extensive parts of the MetaFPA platform, etc. Moreover, MetaFPA assists the safety engineers not only in building the underlying model for the failure propagation analysis, but also in performing the analysis itself and evaluating its results (generation of analysis scripts and feedback loop already mentioned in Subsection 5.2.1).

- **Flexibility**: Reconfigurability, interoperability, reuse, and other flexibility aspects are major issues in the FLM context. The customization capabilities of the accessible FLM tools are very limited which makes them almost unusable for other use cases than those they were originally implemented for. Several MetaFPA features, such as the use-case dependent customization of the metamodel, the extendability of the platform by user add-ons, and the reuse of analysis artefacts across projects and teams, help to overcome the flexibility concerns encountered in the FLM context and offers instead a manifold and powerful tool-set for failure propagation analysis and simulation which offers multiple facilities and which can be flexibly rearranged.

5.2.3 Metamodel for Failure Propagation Analysis

As already mentioned, the technical enablers for the MetaFPA methodology are model-driven development, metamodeling, and code generation. The MetaFPA metamodel for failure propagation analysis is the basis of the methodology. It represents the core artefact of the approach and the starting point for all modeling, data handling, and evaluation tasks. Illustrated in a simplified form through a UML class diagram in Figure 5.5, the metamodel contains a formalization of the compositional system structure used for the analysis and covers the failure logic modeling aspects addressed in Subsection 5.2.1.



A composition from class <c1> to class <c2> means that each object (instance of <c1>) has a number N of related objects (instances of<c2>) as child nodes. The number N depends on the multiplicity which is either implicitly 1 or given as a label of the composition arrow. Multipllicity * means 0 to infinity.

A reference from class <c1> to class <c2> means that each object (instance of <c1>) is linked to a number of data objects (instances of <c2>). The number N depends on the multiplicity which is either implicitly 1 or given as a label of the reference arrow.

Figure 5.5: Simplified Metamodel for Failure Propagation Analysis

In the following, the aspects which are related to the system modeling and which are covered by the metamodel are recapitulated in 5.2.3.1. Then, an overview of the failure modeling aspects is given in 5.2.3.2. After that, the remaining syntactic and semantic details of the metamodel are described in 5.2.3.3.

5.2.3.1 System Modeling Aspects

In accordance with most FLM approaches and taking the extensions and enhancements presented in Subsection 5.2.2 into account, MetaFPA's metamodel addresses the following system modeling aspects:

- **System compositional structure**: Each **System** contains a set of **Blocks**. These blocks are associated with a highly abstracted architecture of the considered system which evolves during the concept phase.
- **Block interfaces**: Each block has a set of **Ports**, which can have different directions (IN, OUT, INOUT). The values of these ports are dynamically updated during the analysis. This is an important prerequisite for supporting the dynamical flow aspect and facilitating the failure capturing mechanisms throughout the system structure.
- **Communication between blocks**: The interactions between the different blocks are ensured through **Connections**. Each block has a number (zero or more) *outgoing connections*, and each connection is optionally associated (through references) to a *source* port and a *target* port.

5.2.3.2 Failure Modeling Aspects

Similarly to almost all FLM approaches, MetaFPA's failure propagation modeling is based on the following:

- **Block failures**: The behavior of the blocks in failure cases is covered in the metamodel through the classes **InputFailure** and **OutputFailure** which depict respectively the single deviations that may occur on the inputs or the outputs of each block. In addition to that, the possible failures that can internally affect the block itself are denoted as **InternalFailures**.
- **Failure combinations**: Combing failures is an aspect which has been neglected by many FLM approaches on the pretext that the probability of multiple concurrent failures is extremely low. However, a thorough and complete functional safety assessment, as required by currently relevant safety standards (e.g., [1] and [2]), does not allow such disregard. Therefore, MetaFPA offers the possibility to analyze the impact of failure combinations (formalized

by **InputFailureCombination** and **InternalFailureCombination** in Figure 5.5) which may lead to single output deviations or also to combinations (**OutputFailureCombination**). Assuming that a block has n input failures, the number of the corresponding input failure combinations will be 2^n (power set). However, not all combinations are *valid* as there may be some contradictory failures in the model specified by the user such as omission and commission failures. Theoretically, the user can manually specify the valid failure combinations, but MetaFPA offers a feature to facilitate this complex and error-prone task. Once the user has specified the so-called *incompatibility* references between failures, an extension in the MetaFPA platform goes over all 2^n possible combinations, discards all irrelevances, and keeps only the valid combinations.

- **Failure propagation and transformation**: The failure propagation logic is described through the *PropagationMapping* class which captures the failure behavior of each block by linking input failures (*externalCauses* as they are caused by other blocks or by the environment), internal failures (*internalCauses* as they occur within the block itself), and consequent output failures (*results*). MetaFPA enables the user to create instances of the **PropagationMapping** classes through manual data input. Though, due to the significant complexity of this task, especially when all failure combinations must be exhaustively taken into consideration, an alternative generation method is supported in MetaFPA. Based on parsing the occurrence conditions of the block output failures, given by the user as DNFs[1] of input and internal failures, the appropriate **PropagationMapping** instances are automatically generated and appended to the constructed failure propagation model.

5.2.3.3 Further Metamodel Details

Among the additional aspects covered by MetaFPA's metamodel, and which cannot be immediately categorized as system modeling aspects or as failure logic modeling aspects, the following two are worth mentioning:

- **Link between system modeling and failure logic modeling**: The concrete impact of the failures on the system is modeled in a dynamic simulation-oriented fashion. For this, the *effect* attribute in the classes **InputFailure** and **OutputFailure** is used in correlation with the *affectedPort* references (from both classes to the **Port** class). The activation of the system model's connections induces the update of the ports status. Through an inspection of the ports, occurring input failures are detected making the **TransformationMapping** class obsolete. However, this class ensures the basic FLM feature of defining manually the primary relationships between output failures (*origins*) and input

[1] **DNF**: Disjunctive Normal Form

failures (*destinations*).

- **Criticality**: For the blocks of the system structure, certain deviations from the originally intended functionality are qualified as *critical.* Such categorization can be derived for example from the safety goals or from the top-level functional safety requirements. To enable this through the metamodel, the attribute *SystemCritical* is added to the **OutputFailure** class.
- **Statistical aspect**: Qualifying a system as functionally safe requires the computation of multiple metrics (failure rate, diagnostic coverage, etc.,). When critical values of these metrics are yielded by the analysis, re-design is prescribed. In MetaFPA, system functional safety, with respect to the very early and abstract model covered by the approach, is correlated with the number of analysis runs leading to a *system failure.* Through the probability attributes in the failure classes, the statistical distribution of the input and internal threats can be given by the user for so-called *entry* blocks (blocks which get their inputs from the environment and not as results of previous blocks in the system structure). Probabilities of output failures can then be computed and propagated to input failures of subsequent blocks. A *system failure* takes place when at least one *system critical* output failure occurs at one or more final blocks of the system structure with a probability greater than a minimum value specified by the user (e.g., 10^{-5}). Every analysis run leading to a system failure is documented through the generation of a graph (currently in the XML-based GraphML format) illustrating the propagation path (see Figure 5.4).

5.3 Metamodels for Functional Safety Analysis

In Chapter 2, the importance of safety evaluation in system design process has been extensively addressed. Associated activities are performed throughout the design and manufacturing process at different abstraction levels, including (i) conceptual risk assessment and safety analysis, consisting mostly in probabilistic, (ii) simulation-oriented system alterations are carried out through fault injection campaigns, and (iii) physical prototype testing where the system hardware and software are certified with respect to the safety requirements.

In the automotive context, the evaluation guidelines are addressed by the ISO 26262 standard for functional safety of road vehicles [2]. As previously presented in Section 2.2, an extensive analysis and assessment procedure is required to ensure the targeted safety integrity level of the system (see Figure 5.6). It starts by predicting potential risks, whose causes and effects are subsequently identified. To mitigate the failure effects, appropriate countermeasures are deployed. And finally, evaluation metrics are computed to provide evidence about the system safety integrity level.

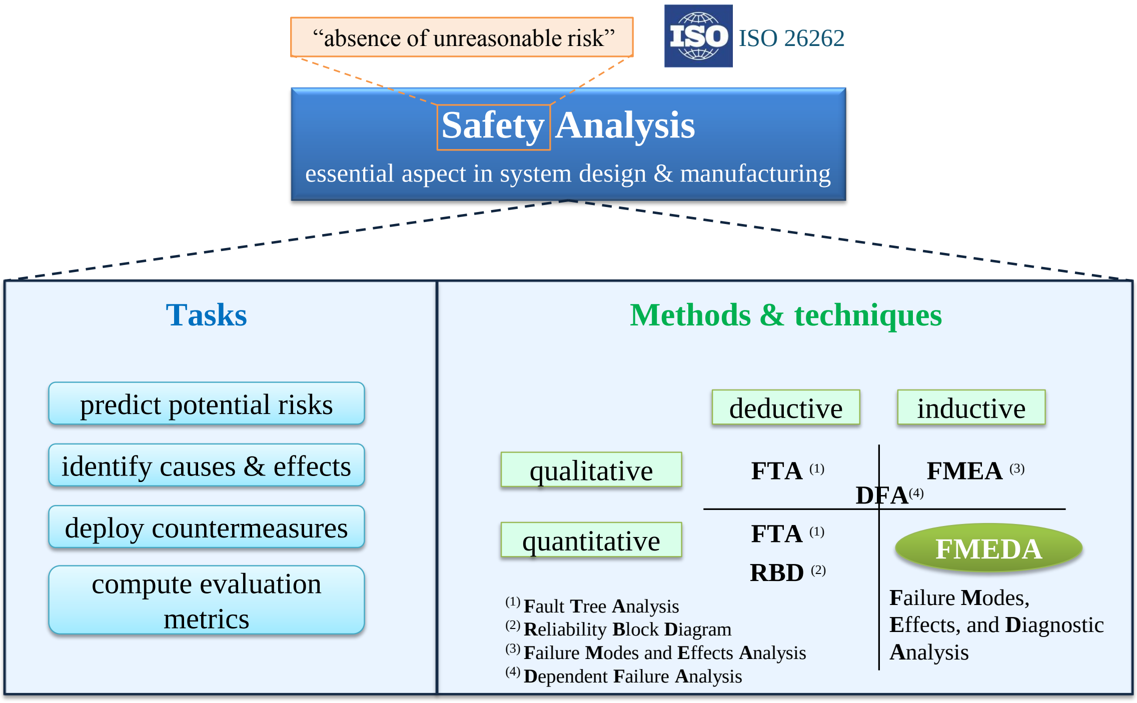

"absence of unreasonable risk"
ISO 26262
Safety Analysis
essential aspect in system design & manufacturing
Tasks
predict potential risks
identify causes & effects
deploy countermeasures
compute evaluation metrics
Methods & techniques
deductive
inductive
qualitative
quantitative
FTA (1)
FMEA (3)
DFA(4)
FTA (1)
RBD (2)
FMEDA
Failure Modes, Effects, and Diagnostic Analysis
(1) Fault Tree Analysis
(2) Reliability Block Diagram
(3) Failure Modes and Effects Analysis
(4) Dependent Failure Analysis

Figure 5.6: Overview of Functional Safety Analysis in the Automotive Context

Safety analysis tasks are performed through different methods and techniques which are classified in the standard according to multiple criteria. There is a distinction between deductive and inductive approaches. A deductive analysis is performed top-down starting from problematic situations on system-level, and looking for the corresponding causes in the components. Inductive approaches start bottom-up from local component malfunctions moving towards the corresponding effects on the complete system. Safety analysis can also be either qualitative or quantitative (more details in 2.2.7.3). Many analysis techniques are addressed by the ISO 26262 standard, for example Fault Tree Analysis (FTA) (2.2.7.6), Reliability Block Diagram (RBD)s (2.2.7.8), Dependent Failure Analysis (DFA) (2.2.7.7), Failure Modes and Effects Analysis (FMEA) and its quantitative variant, Failure Modes, Effects, and Diagnostic Analysis (FMEDA) (2.2.7.5).

In this thesis, FMEDA, FTA, and DFA are considered. The interest in these specific procedures is justified by (i) the quantitative safety assessment they enable, offering the evidence and the figures needed by the customers, (ii) their compliance with the ISO 26262 standard, and (iii) their established status in the industry.

Despite these characteristics making FMEDA, FTA, and DFA the state of practice analysis techniques for the automotive industry and for its suppliers, especially system and semiconductor manufacturers, their application is still correlated with many problems. Traditionally, the analysis is done manually and it includes complicated tasks leading to high effort and time costs. The created documents such as the FMEDA spreadsheets and the fault trees are very large and complex for industrial applications (thousands of lines in the FMEDA spreadsheets for example), which makes them very hard to explore and to maintain.

These challenges have already been subjects of research in the literature. However, most of the proposals suggest new data formats for the analysis. This may be possible for small experimentation examples, but it is difficult to apply in industrial contexts, where a considerable amount of safety analysis data is already available and shall be reused. Moreover, the provided toolsets are usually limited to data management and they do not establish any link to simulation contexts.

In this thesis, the challenges mentioned above are addressed by opting for a formalization of available data formats and templates. Thereby, the reuse of already existing analysis artefacts is enabled. This formalization is also the prerequisite for an automated support based on model-driven engineering. It is also used later to bridge the gap between the analytical data and the fault simulation on executable models through model-to-model transformations and corresponding data mapping.

The remainder of this section is organized as follows. In Subsection 5.3.1, the metamodel-based FMEDA formalization is described after a brief overview of the traditional FMEDA flow and documentation style. Similarly, the formalization of FTA and DFA through metamodels is presented in Subsections 5.3.2 and 5.3.3 respectively.

5.3.1 Metamodel-Based FMEDA Formalization

In Subsection 2.2.7.5, the inductive safety analysis approach referred to as Failure Modes and Effects Analysis (FMEA) is introduced. FMEA has been applied for many successive decades since the 1960s in different domains such as the chemical process and the aerospace industries. Beyond the basic FMEA capabilities consisting in defining potential component failures and deriving the expected impact at system level, different additional aspects are taken into consideration in the several methodology variants, e.g., criticality, failure rates, and on-line diagnostic mechanisms (more details in 2.2.7.5). FMEDA is an FMEA extension developed in the 1990s within the company Exida [39, 40]. In this work, the traditional FMEDA procedure is considered from three perspectives: (i) a formalized description, (ii) an automated support, and (iii) a consistent link with fault injection and simulation environments.

5.3.1.1 Traditional FMEDA Flow

As already mentioned in 2.2.7.5, the term *FMEDA* is not mentioned in the ISO 26262 standard, although it is the commonly used denomination for the quantitative FMEA, which is particularly conducted within system and semiconductor companies. The precise activities where FMEDA is applied in compliance to ISO 26262 are located in Part 5 (Product development at hardware level). In the context of the hardware *safety analyses* (5-7.4.3), a rather generic FMEDA is conducted. In practice, safety engineering teams refer to this iteration of the FMEDA as a *Concept FMEDA*, as it does not take all design details into account and relies on estimated values for the quantitative derivation of the safety metrics. Once the final design is available, the activities of the ISO 26262 clauses **5-8**: (*evaluation of the hardware architectural metrics*) and **5-9**: (*evaluation of safety goal violations due to random hardware failures*) are performed. The outcome is the so-called *detailed* or *final FMEDA*.

In other words, in the context of ISO 26262 related safety analysis, the detail level of the FMEDA is variable. It depends on the system representation and on the respective abstraction layer (conceptual design, detailed development, etc.). Furthermore, for large systems addressed in industrial contexts, FMEDA is performed recursively. Thus, for the different system components, a separate analysis is carried out. Complex system components may be again decomposed in subcomponents, for which further FMEDAs are performed. The obtained results are then inspected and gathered to conduct an overall analysis for the complete system. That is why, there is a differentiation between the *Component FMEDA* and the *System FMEDA*.

The FMEDA process is represented through a flowchart in Figure 5.7. It should be noted that the flowchart addresses the overall System FMEDA. However, a similar procedure is followed to perform the analysis at component or subcomponent level.

Step 2 in Figure 5.7 shows that safety-relevant components of the system (i.e., those related to the safety requirements) should be identified and, in most cases, classified according to the considered safety standard (e.g., Table D.1 in ISO 26262-5). As FMEDA is performed by teams not by individuals, the appropriate members should be identified (step 3) and the specific role of every team member should be determined: who is responsible for the analysis and maintenance of which components, who reviews the outcome, etc., (step 4).

As long as the considered components are functionally independent, their respective analyses may be done in parallel by different FMEDA team members (starting at step 5). Step 6 provides an understanding of the functions normally fulfilled by the component (information about functionality generally provided by design engineers) which is required to answer the question “How may the component fail?”.

Identifying the component failure modes (step 7) is a crucial part of the FMEDA

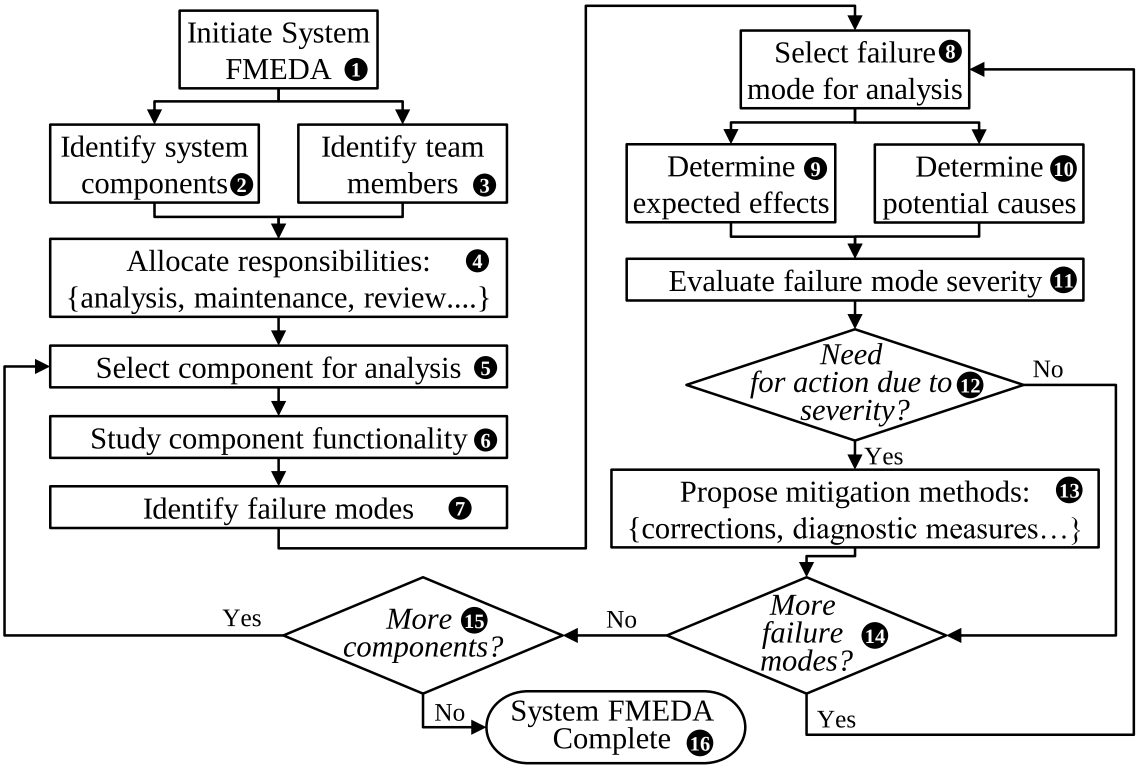

Initiate System FMEDA ❶
Identify system components ❷
Identify team members ❸
Allocate responsibilities: ❹ {analysis, maintenance, review....}
Select component for analysis ❺
Study component functionality ❻
Identify failure modes ❼
Select failure mode for analysis ❽
Determine expected effects ❾
Determine potential causes ❿
Evaluate failure mode severity ⓫
Need for action due to severity? ⓬
No
Yes
Propose mitigation methods: ⓭ {corrections, diagnostic measures…}
More failure modes? ⓮
Yes
No
More components? ⓯
Yes
No
System FMEDA Complete ⓰

Figure 5.7: Flowchart of Basic FMEDA Steps [3]

flow where reliable resources are required. In fact, component failure rates, lists of associated failure modes, and corresponding probabilistic distributions are commonly extracted from internal catalogues, old project reports, or recognized databases (SN 29500 [121], IEC 61709 [122], Exida Reliability Handbook [123]).

In steps 8 to 11, each failure mode should be analyzed according to the component specificities and to the safety requirements. First, the impact of the component failure mode on the system behavior should be determined by the safety engineer. A failure mode can have multiple effects depending on many parameters such as location or timing.

Then, all possible root causes of the failure mode are listed. It should be noted that the root cause listing is not always performed in FMEDAs.

Finally, the failure modes are ranked according to their impact on the safety. The so-called *severity* levels are safety standard-specific, and in ISO 26262, they range between *safe* and *residual* (uncovered by safety mechanisms). The need for action is determined by the severity ranking. For example, for all *safe* failure modes, no recommendations are required. For the rest, measures should be undertaken (step 12).

Metric	Unit	Comments
$\boldsymbol{\lambda}$	[FIT]	Total failure rate: sum of given component failure rates: $\lambda = \sum_{i=1}^{N} \lambda_{C_i}$.
$\boldsymbol{\lambda_{\mathbf{S}}}$	[FIT]	Failure rate of safe faults: for safety-related elements, safe faults do not violate the safety goal, neither directly nor in combination with other independent faults.
$\boldsymbol{\lambda_{\mathbf{SPF}}}$	[FIT]	Failure rate of single-point faults: single-point faults are not covered by safety mechanisms and lead directly to a violation of the safety goal.
$\boldsymbol{\lambda_{\mathbf{DPF}}}$	[FIT]	Failure rate of dual-point faults: a dual-point fault leads only in combination with another independent fault to a violation of the safety goal. A DPF can be perceived, detected or latent(1): $\lambda_{\mathrm{DPF}} = \lambda_{\mathrm{DPF,D}} + \lambda_{\mathrm{DPF,L}}$.
$\boldsymbol{\lambda_{\mathbf{MPF}}}$	[FIT]	Failure rate of multiple-point faults: multiple-point failures with order ≥ 3 are commonly considered as safe faults, so that only MPFs with order 2 (i.e., dual-point faults) are relevant for the calculation of the LFM. This assumption can however be omitted by the technical safety concept.
$\boldsymbol{\lambda_{\mathbf{RF}}}$	[FIT]	Failure rate of residual faults: when a failure mode is addressed by a safety mechanism and the related coverage factor is α, then the remaining uncovered $(100-\alpha)\%$ of the faults related to that failure mode are residual faults.
SPFM	[%]	Single-Point Fault Metric: this metric reflects the robustness against SPFs and RFs. This robustness is evaluated with respect to the coverage offered by safety mechanisms and to the properties of the design itself enabling a primarily high amount of safe faults. Assuming that all components of the considered design are safety-critical, equation **C.5**(2) defining SPFM in ISO 26262-5 can be simply written as follows: $\mathrm{SPFM} = 1 - \frac{\lambda_{\mathrm{SPF}} + \lambda_{\mathrm{RF}}}{\lambda}$.
LFM	[%]	Latent Fault Metric: this metric reflects the robustness against latent faults. This robustness is evaluated with respect to the coverage offered by safety mechanisms, to the recognition of faults by the driver before the safety goal violation, and to the properties of the design itself enabling a primarily high amount of safe faults. Assuming that all components of the considered design are safety-critical and that all MPFs with order ≥ 3 are considered safe, equation **C.6**(3) defining LFM in ISO 26262-5 can be simply written as follows: $\mathrm{LFM} = 1 - \frac{\lambda_{\mathrm{DPF,L}}}{\lambda - \lambda_{\mathrm{SPF}} - \lambda_{\mathrm{RF}}}$.
MTTF	[yrs]	Mean Time To Failure: the mean time till the first failure under specified experimental conditions. MTTF can be calculated as: $\mathrm{MTTF} = \frac{1}{\lambda}$. If only failures with dangerous consequences are considered, the Mean Time To Dangerous Failure is determined as: $\mathrm{MTTF_D} = \frac{1}{\lambda_{\mathrm{SPF}} + \lambda_{\mathrm{RF}}}$.

(1) A *latent fault* is a "multiple-point fault whose presence is not detected by a safety mechanism nor perceived by the driver within the multiple-point fault detection interval" [2].

(2) Single-Point Fault Metric **SPFM** $= 1 - \frac{\sum_{SR,HW} (\lambda_{\mathrm{SPF}} + \lambda_{\mathrm{RF}})}{\sum_{SR,HW} \lambda}$ [Equation **C.5** in ISO 26262-5]

where $\sum_{SR,HW} \lambda_{\mathrm{x}}$ is the sum of λ_{x} of the considered safety-critical HW elements for the metric calculation [2].

(3) Latent Fault Metric **LFM** $= 1 - \frac{\sum_{SR,HW} (\lambda_{\mathrm{MPF,latent}})}{\sum_{SR,HW} (\lambda - \lambda_{\mathrm{SPF}} - \lambda_{\mathrm{RF}})}$ [Equation **C.6** in ISO 26262-5]

where $\sum_{SR,HW} \lambda_{\mathrm{x}}$ is the sum of λ_{x} of the considered safety-critical HW elements for the metric calculation [2].

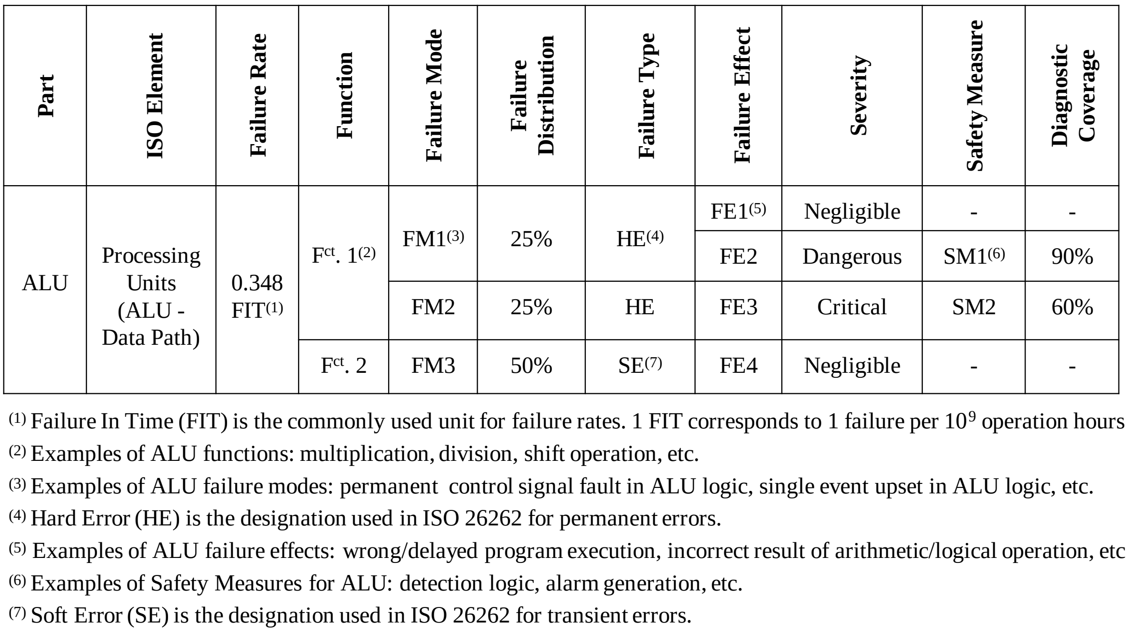
Table 5.1: Typical ISO 26262 Metrics Derived as FMEDA Results [2]

The appropriate mitigation techniques (step 13) can be derived from the considered safety standard or at least aligned with it. For example, Tables D.2 - D.12 in ISO 26262-5 give lists over safety measures for different element types. The typically achievable diagnostic coverage by the mechanisms is also qualified in the standard (low, medium, high). However, within the scope of the FMEDA, approximate (or if available, more accurate) percentage values should be considered to enable the further calculation of the safety metrics required by the ISO 26262 standard.

The progressive character of the FMEDA flow can be observed in the flowchart steps 14 and 15 in Figure 5.7. As long as not all failure modes of the currently considered component are already analyzed, steps 8 to 13 should be repeated. And as long as not all components of the system are already analyzed, steps 5 to 14 should be repeated. Once the FMEDA is completed, the standard-specific numerical results such as the total system failure rate and the specific coverage metrics (e.g., Single-Point Fault Metric (SPFM) and Latent Fault Metric (LFM) in ISO 26262) should be derived. An overview of the ISO 26262 metrics, which are commonly calculated as FMEDA results, is given in Table 5.1.

5.3.1.2 Traditional FMEDA Documentation

FMEDA documentation is done in tabular form, as seen in this simplified extract of an FMEDA spreadsheet (Table 5.2).



Part	ISO Element	Failure Rate	Function	Failure Mode	Failure Distribution	Failure Type	Failure Effect	Severity	Safety Measure	Diagnostic Coverage
ALU	Processing Units (ALU - Data Path)	0.348 FIT(1)	F^{ct}. 1(2)	FM1(3)	25%	HE(4)	FE1(5)	Negligible	-	-
							FE2	Dangerous	SM1(6)	90%
				FM2	25%	HE	FE3	Critical	SM2	60%
			F^{ct}. 2	FM3	50%	SE(7)	FE4	Negligible	-	-

(1) Failure In Time (FIT) is the commonly used unit for failure rates. 1 FIT corresponds to 1 failure per 10^9 operation hours.

(2) Examples of ALU functions: multiplication, division, shift operation, etc.

(3) Examples of ALU failure modes: permanent control signal fault in ALU logic, single event upset in ALU logic, etc.

(4) Hard Error (HE) is the designation used in ISO 26262 for permanent errors.

(5) Examples of ALU failure effects: wrong/delayed program execution, incorrect result of arithmetic/logical operation, etc.

(6) Examples of Safety Measures for ALU: detection logic, alarm generation, etc.

(7) Soft Error (SE) is the designation used in ISO 26262 for transient errors.

Table 5.2: Example of FMEDA Table [3]

For each safety-relevant system part, an element type according to the ISO standard and a failure rate are allocated. The multiple functions of the part can be affected by different failure modes which are then characterized by their failure distributions (basically the percentage of their contribution to the failure rate of the whole part), their failure types (hard errors vs. soft errors), and their failure effects. These are classified according to their severity. Furthermore, appropriate safety measures, which ensure the mitigation of the identified failure effects, are documented along with their diagnostic coverage values in the FMEDA table.

5.3.1.3 FMEDA Metamodel

A simplified version of the FMEDA metamodel is represented as a UML class diagram in Figure 5.8.

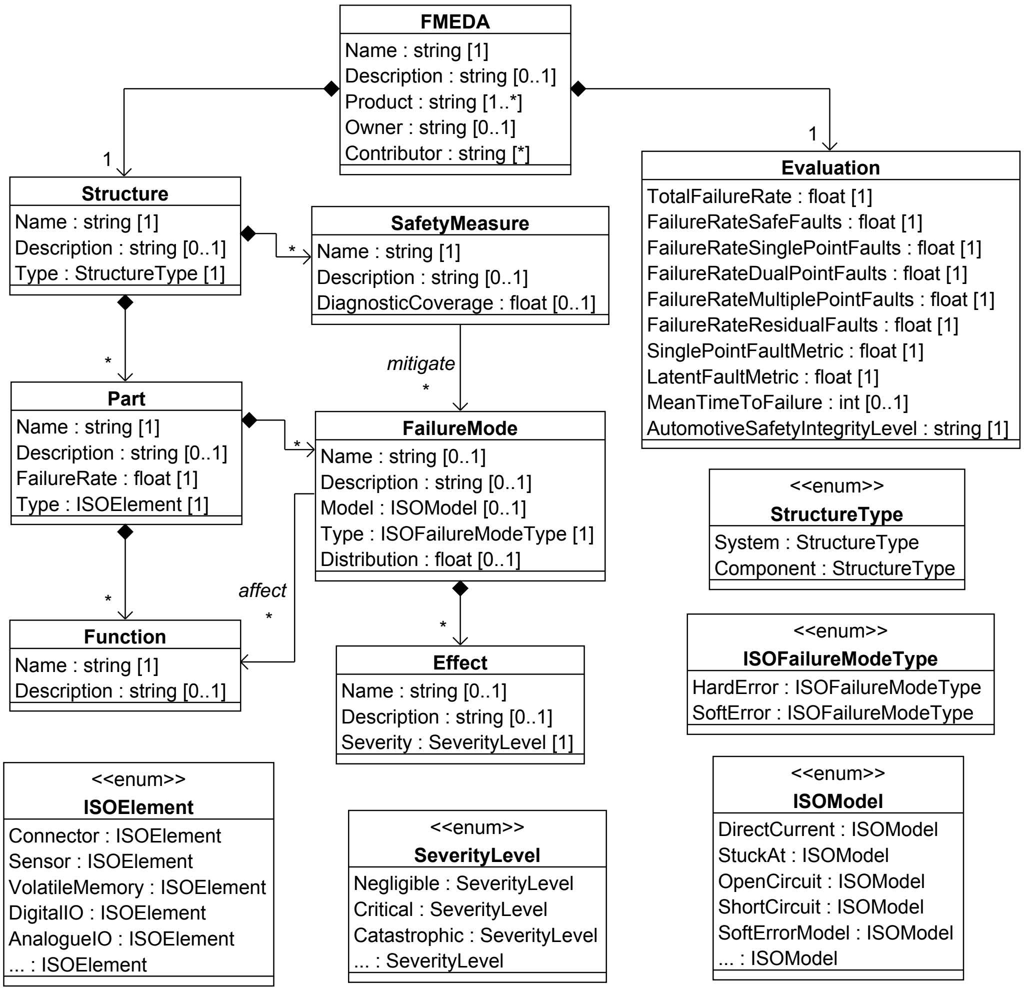
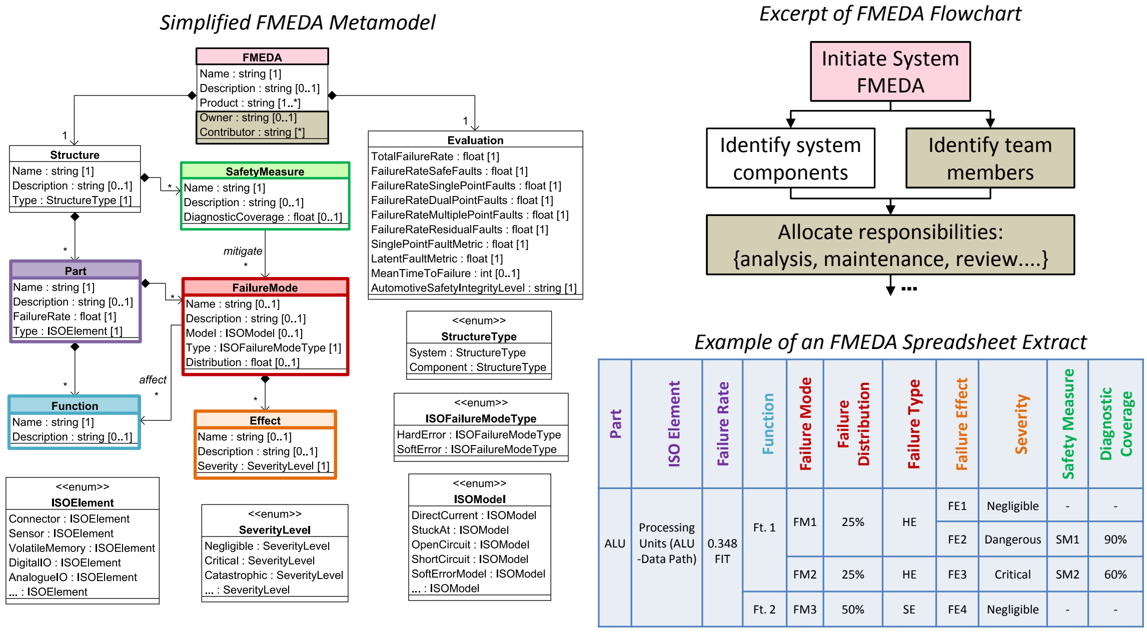

FMEDA
Name : string [1]
Description : string [0..1]
Product : string [1..*]
Owner : string [0..1]
Contributor : string [*]
1
1
Structure
Name : string [1]
Description : string [0..1]
Type : StructureType [1]
*
SafetyMeasure
Name : string [1]
Description : string [0..1]
DiagnosticCoverage : float [0..1]
Evaluation
TotalFailureRate : float [1]
FailureRateSafeFaults : float [1]
FailureRateSinglePointFaults : float [1]
FailureRateDualPointFaults : float [1]
FailureRateMultiplePointFaults : float [1]
FailureRateResidualFaults : float [1]
SinglePointFaultMetric : float [1]
LatentFaultMetric : float [1]
MeanTimeToFailure : int [0..1]
AutomotiveSafetyIntegrityLevel : string [1]
*
mitigate
*
Part
Name : string [1]
Description : string [0..1]
FailureRate : float [1]
Type : ISOElement [1]
*
FailureMode
Name : string [0..1]
Description : string [0..1]
Model : ISOModel [0..1]
Type : ISOFailureModeType [1]
Distribution : float [0..1]
<<enum>>
StructureType
System : StructureType
Component : StructureType
affect
*
*
*
Function
Name : string [1]
Description : string [0..1]
Effect
Name : string [0..1]
Description : string [0..1]
Severity : SeverityLevel [1]
<<enum>>
ISOFailureModeType
HardError : ISOFailureModeType
SoftError : ISOFailureModeType
<<enum>>
ISOElement
Connector : ISOElement
Sensor : ISOElement
VolatileMemory : ISOElement
DigitalIO : ISOElement
AnalogueIO : ISOElement
... : ISOElement
<<enum>>
SeverityLevel
Negligible : SeverityLevel
Critical : SeverityLevel
Catastrophic : SeverityLevel
... : SeverityLevel
<<enum>>
ISOModel
DirectCurrent : ISOModel
StuckAt : ISOModel
OpenCircuit : ISOModel
ShortCircuit : ISOModel
SoftErrorModel : ISOModel
... : ISOModel

Figure 5.8: Simplified FMEDA Metamodel for ISO 26262 Safety Analysis [3]

Two basic sources are taken into account for its construction, namely the steps of the traditional FMEDA flow (Figure 5.7) and the spreadsheet structure used by safety engineers as exemplified in Table 5.2. Team members and the allocation of their responsibilities are captured by the root class in the FMEDA metamodel. **Parts** of the **Structure** being analyzed (*system* or *component*) are characterized through their *types*, their *failure rates*, and their **Functions**. **Failure Modes** affecting **Functions** are covered along with their *failure types* and *distributions*. Failure **Effects** and *severity levels* are also taken into account and **Safety Measures** used to mitigate one or more failure modes are addressed along with their *diagnostic coverage*.

Figure 5.9 illustrates some examples of the correspondences between the constructed FMEDA metamodel on the one hand and the traditional FMEDA flow and documentation on the other hand. Through color coding, the equivalent and/or associated classes and attributes of the metamodel, tasks in the flowchart excerpt, and columns in the FMEDA table are appropriately highlighted.


Simplified FMEDA Metamodel
FMEDA
Name : string [1]
Description : string [0..1]
Product : string [1..*]
Owner : string [0..1]
Contributor : string [*]
1
1
Structure
Name : string [1]
Description : string [0..1]
Type : StructureType [1]
SafetyMeasure
Name : string [1]
Description : string [0..1]
DiagnosticCoverage : float [0..1]
Evaluation
TotalFailureRate : float [1]
FailureRateSafeFaults : float [1]
FailureRateSinglePointFaults : float [1]
FailureRateDualPointFaults : float [1]
FailureRateMultiplePointFaults : float [1]
FailureRateResidualFaults : float [1]
SinglePointFaultMetric : float [1]
LatentFaultMetric : float [1]
MeanTimeToFailure : int [0..1]
AutomotiveSafetyIntegrityLevel : string [1]
*
*
mitigate
*
Part
Name : string [1]
Description : string [0..1]
FailureRate : float [1]
Type : ISOElement [1]
*
FailureMode
Name : string [0..1]
Description : string [0..1]
Model : ISOModel [0..1]
Type : ISOFailureModeType [1]
Distribution : float [0..1]
<<enum>>
StructureType
System : StructureType
Component : StructureType
*
affect
*
*
Function
Name : string [1]
Description : string [0..1]
Effect
Name : string [0..1]
Description : string [0..1]
Severity : SeverityLevel [1]
<<enum>>
ISOFailureModeType
HardError : ISOFailureModeType
SoftError : ISOFailureModeType
<<enum>>
ISOElement
Connector : ISOElement
Sensor : ISOElement
VolatileMemory : ISOElement
DigitalIO : ISOElement
AnalogueIO : ISOElement
... : ISOElement
<<enum>>
SeverityLevel
Negligible : SeverityLevel
Critical : SeverityLevel
Catastrophic : SeverityLevel
... : SeverityLevel
<<enum>>
ISOModel
DirectCurrent : ISOModel
StuckAt : ISOModel
OpenCircuit : ISOModel
ShortCircuit : ISOModel
SoftErrorModel : ISOModel
... : ISOModel
Excerpt of FMEDA Flowchart
Initiate System FMEDA
Identify system components
Identify team members
Allocate responsibilities: {analysis, maintenance, review....}
...
Example of an FMEDA Spreadsheet Extract

Part	ISO Element	Failure Rate	Function	Failure Mode	Failure Distribution	Failure Type	Failure Effect	Severity	Safety Measure	Diagnostic Coverage
ALU	Processing Units (ALU -Data Path)	0.348 FIT	Ft. 1	FM1	25%	HE	FE1	Negligible	-	-
							FE2	Dangerous	SM1	90%
				FM2	25%	HE	FE3	Critical	SM2	60%
			Ft. 2	FM3	50%	SE	FE4	Negligible	-	-

Figure 5.9: Correspondence between the FMEDA Metamodel and the Traditional FMEDA Flow and Documentation

5.3.2 Metamodel-Based FTA Formalization

In Subsection 2.2.7.6, the deductive safety analysis approach referred to as Fault Tree Analysis (FTA) is introduced. FTA has been originally developed in the 1960s and applied in the aerospace domain. Indeed, the details of the FTA process are given in aircraft and aviation related standards such as **SAE ARP4761**: *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment* [13] and **IEC 61025**: *Fault Tree Analysis (FTA)* [124].

Qualitative FTA consists in determining undesirable events leading to system failures and subsequently decomposing them with respect to their potential causes in a recursive way until reaching the leaf nodes, namely the so-called elementary events at component level. A quantitative aspect is also addressed in the FTA context. Thereby, the occurrence probabilities and/or frequencies for events as well as for their contributing factors are taken into account [124] so that overall failure rates and safety metrics at system level are accordingly derived. In this thesis, the traditional FTA procedure is considered from two perspectives: (i) a formalized description and (ii) an automated fault tree synthesis through a link to failure logic modeling (see Subsection 3.1.2 and Section 5.2).

5.3.2.1 Traditional FTA Flow and Documentation

As already mentioned in Subsection 2.2.7.6, FTA is a top-down failure analysis method using Boolean logic to depict the top-level system failures and the different levels of hazardous events causing them. All combinations and propagations of these events are denoted through the acyclic graphic representation called *fault tree*.

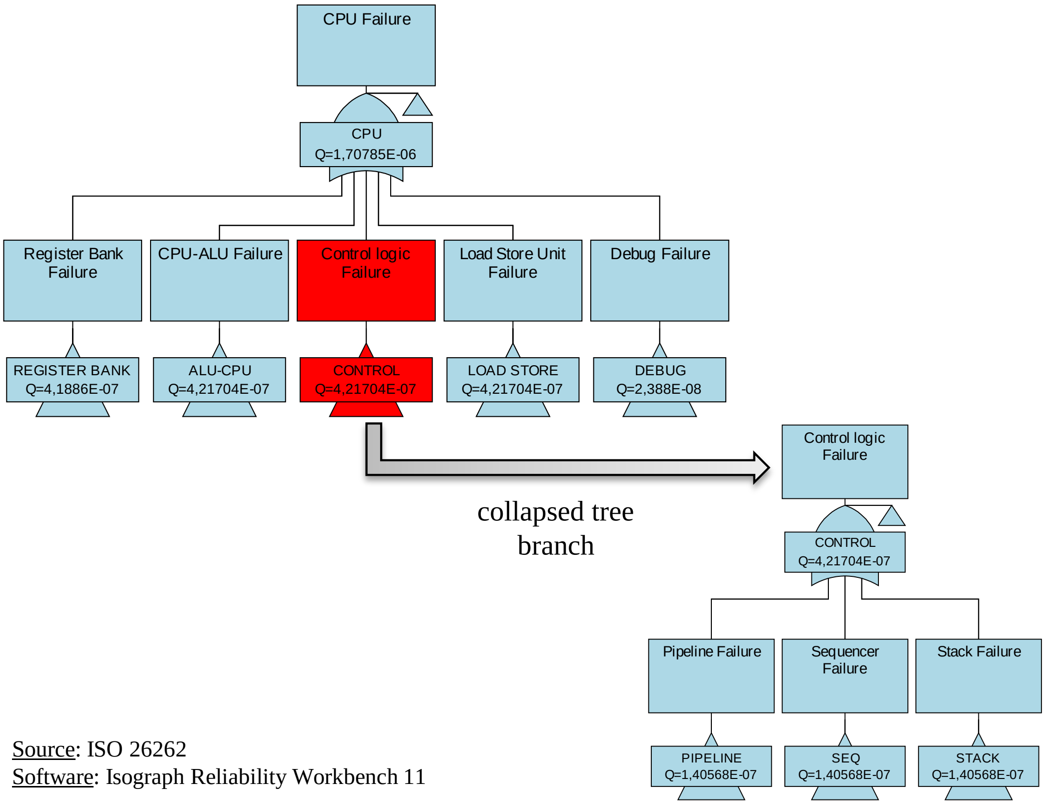
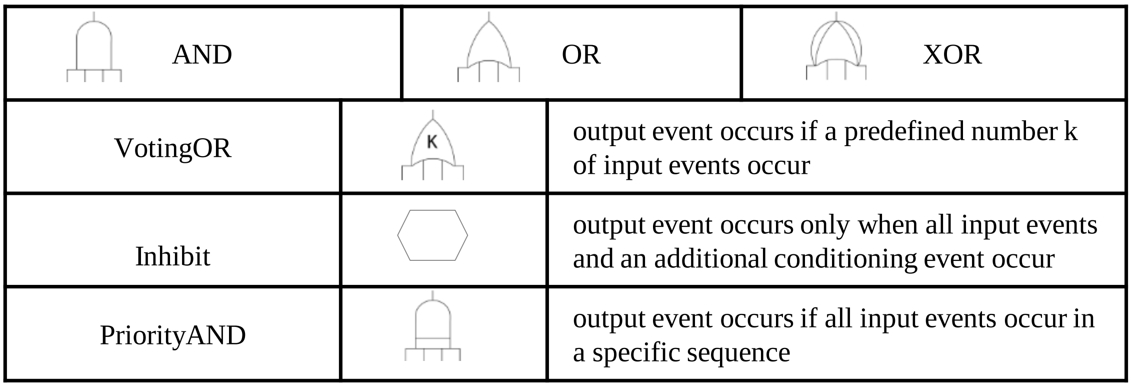


Figure 5.10: Example of a Fault Tree Extract [2]

In Figure 5.10, an exemplary fault tree extract is shown. It is derived from the ISO 26262 standard and addresses the analysis of a microcontroller, particularly the CPU failures which can be classified into a register bank failure, an ALU failure, a control logic failure, etc. Most fault tree editors, such as Isograph Reliability Workbench^{TM} used here to create the example, offer the possibility to collapse tree branches to make the exploration easier. In the example of Figure 5.10, the control logic failure of the CPU is further on detailed into pipeline failure, sequencer failure, and stack failure.

The symbols used in the fault trees are called *logic gates* and are very similar in form and significance to the symbols used in electronic circuit design (see Table 5.3). In addition to the commonly known *AND*, *OR*, and *XOR* gates, FTA introduces several special gates [125] such as:

- *VotingOR* gate: Output event occurs if a predefined number of input events occur.
- *Inhibit* gate: Output event occurs only when all input events and an additional conditioning event occur.
- *PriorityAND* gate: Output event occurs if all input events occur in a specific sequence.



AND	OR	XOR
VotingOR	K	output event occurs if a predefined number k of input events occur
Inhibit		output event occurs only when all input events and an additional conditioning event occur
PriorityAND		output event occurs if all input events occur in a specific sequence

Table 5.3: Overview of Logic Gates in FTA

FTA is considered as an *event-based* and *status-driven* analysis. On the one hand, each input of a specific logic gate in the tree represents a *hazardous event* corresponding to an undesired status of a certain part of system and potentially other factors taken into account for the analysis, such as external events related to the system environment. On the other hand, the output of a logic gate depicts either the resulting status of the affected system parts or a failure condition of the whole system.

The basic FTA flow is illustrated in Figure 5.11. Once the analysis is initiated for a given system, the first step is to collect all required information to perform the

FTA. While gathering the available system data, the FTA analyst shall pay attention to the correctness, completeness, and currentness criteria. These are actually the prerequisites to perform a consistent analysis whose outcomes are useful for the assessment of the safety integrity level and of the fulfillment of the safety goals and requirements [13].

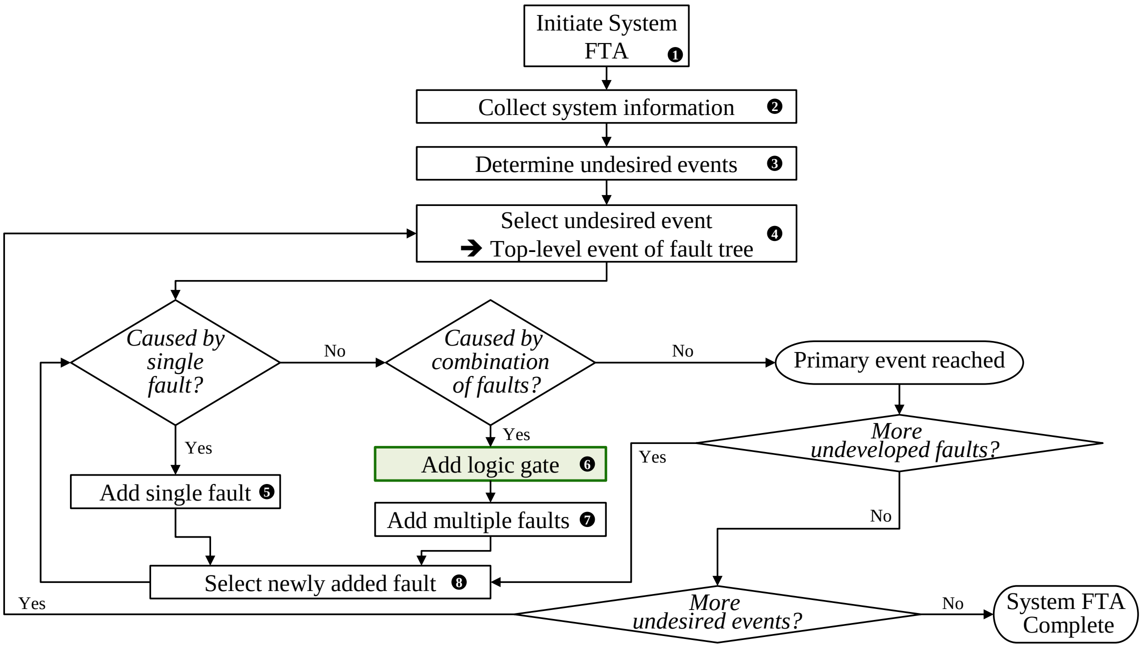


Figure 5.11: Flowchart of Basic FTA Steps

The needed system information is commonly obtained from the documents created during the first stages of the design cycle such as the concept phase, the requirements definition phase, and the high-level architectural design phase. In the context of automotive system design in compliance with the ISO 26262 standard, several work products such as the (i) specifications of safety goals, (ii) functional and technical safety concepts, (iii) hardware design specification, etc., are used as sources to derive the dynamically evolving system data during the subsequent development stages which is required to perform the FTA.

The second step in the FTA flow consists in determining the *undesired events* at system level. Such undesired events have a direct harmful impact on the system user(s) and/or environment. In the ISO 26262 context, safety goal violations represent undesired events at system level. For example, if a safety goal is *"unintended steering shall be avoided"*, then the corresponding safety goal violation (*"unintended steering cannot be avoided"* or simply *"unintended steering"*) is an undesired event at system level (assuming that the considered system is an EPS (Electric Power Steering) system). For each of the undesired system-level events, a fault tree is constructed (Step 4). The uppermost node (i.e., root node) of the tree is referred to as the *top-level event* or simpler as the *top-event*.

Starting from the top-level event, the fault tree structure gets progressively constructed through a top-down *backward stepping* process. The first task in this process is to determine all potential causes of the top-level event.

In the case that there is only one potential cause for the top-level event, a single *intermediate event* is added. It should be noted that intermediate events are frequently referred to in the literature as *faults*. Therefore, they are called so in the flowchart (Figure 5.11). It is also possible, and rather more probable, that the occurrence of the top-level necessitates that a combination of intermediate events take place (e.g., random hardware failure in a certain component AND dysfunction or coverage inability of the dedicated safety measure). In that case, a logic gate shall be added as well as the multiple intermediate events which are logically connected through it.

Afterwards, each newly added element in the fault tree is treated similarly to the top-level event, so that the tree structure gets iteratively extended. In certain cases, no further analysis is required for a certain event, i.e., neither a single intermediate event nor a logic gate gets attached to it, which makes it a *primary event*. In other words, such events are the leaf nodes of the fault tree. When all paths across the tree from the top-level event to the primary events are completed, the analysis process is reiterated from Step 4 for the next system-level undesired event. The overall FTA of the system is completed when the construction of all fault trees corresponding to all undesired events is done.

5.3.2.2 FTA Metamodel

In accordance to the FTA process flow and to the structure of the fault tree presented in the previous subsection 5.3.2.1, an FTA metamodel is constructed. It is represented in a simplified form as UML class diagram in Figure 5.12.

The FTA metamodel addresses three key aspects in the FTA methodology:

1. ***Overall structure***: The top-down analysis starts at the system failures and investigates all potential causes in a recursive way. All data related to the FTA are structured through a tree format containing a number of *events* which are interconnected using *logical gates*. For the multiple undesired events at system level (multiple safety goal violations in the automotive context), multiple fault trees are respectively constructed. That is why the root node class of the metamodel named **FTA** is related to the **Tree** class through a composition relationship characterized by the multiplicity [1..*]. For the sake of modularization and to enable and separate analysis of different system parts by different team members, the fault tree can be

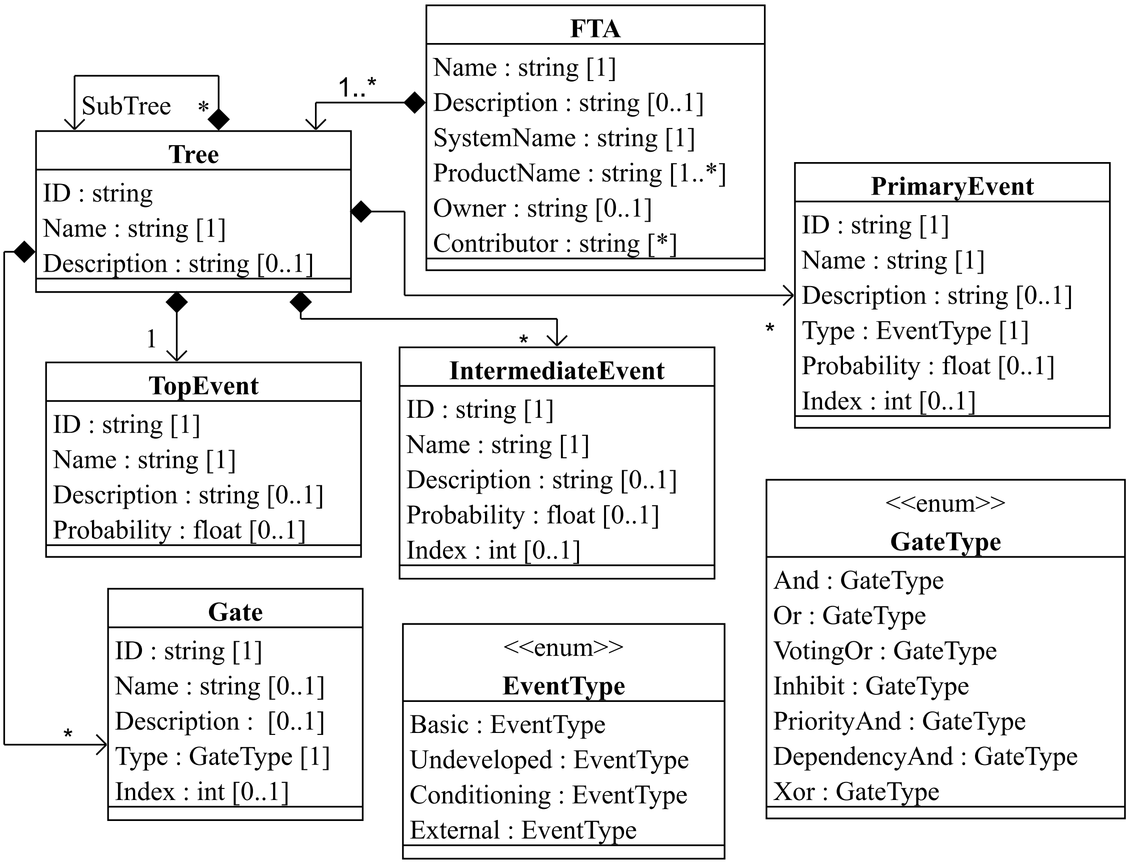


Figure 5.12: Simplified FTA Metamodel [12]

optionally decomposed in further sub-trees with the same structure. Therefore the metamodel class **Tree** has a self-composition named *SubTree*. Each tree contains necessarily one and exactly one **top event** representing the starting point of the analysis. The details of all potential causes leading to the top-event are further on developed in the different fault tree branches.

2. ***Events***: In the metamodel, three types of events are differentiated and reflected through respectively defined classes: (i) the already mentioned **TopEvent**, (ii) **IntermediateEvent**, and (iii) **PrimaryEvent**. Each event in the fault tree is identified by its *ID*, its *Name* and an optional *Description*. In addition to these generic attributes, primary events are also characterized by a *Type*. The four items of the enumeration class **EventType** in the metamodel depict the four different types that can be assigned to primary events [125]:

   a) *Basic events*: These events require no further development in the fault tree.

   b) *Undeveloped events*: These events are not further on developed in the fault tree because of unavailable information for example.

c) *Conditioning events*: These events represent specific conditions or restrictions that may apply to the logic gates.

d) *External events*: These are events which are normally expected to occur.

The *Probability* attribute of the event classes is used when quantitative FTA is required.

3. ***Failure logic***: The construction of the fault tree is based on defining the potential initiators for the top-event and for each intermediate event in compliance with the Boolean logic rules. This is covered in the FTA metamodel by the **Gate** class which is characterized by a mandatory *ID*, an optional *Name*, an optional *Description*, and a mandatory *Type* to be selected from the list of items in the enumeration class **GateType** which correspond to the defined logic gates in Table 5.3. As already explained Subsection 5.3.2.1, the top-event can be initiated either by one single intermediate (or primary) event or by a combination of intermediate (and/or primary) events. Similarly, each intermediate event can be initiated either by one single event (intermediate or primary) or by a combination of further events (intermediate or primary). This is covered in the metamodel through the cross-references between the classes **TopEvent**, **Gate**, **IntermediateEvent**, and **PrimaryEvent** as shown in Figure 5.12.

5.3.3 Metamodel-Based DFA Formalization

In Subsection 2.2.7.7, the Dependent Failure Analysis (DFA) approach recommended by the ISO 26262 standard to evaluate *independence* and *freedom of interference* is introduced.

Beyond being one of the extensively addressed safety analysis techniques in the ISO 26262 standard, DFA is the state-of-practice technique in several domains and particularly in the semiconductor industry, when it comes to assessing dependent failures (characterized by the fact that the probability of simultaneous or successive occurrence of two given dependent failures cannot be simply expressed as the product of their respective occurrence probabilities). Indeed, DFA must be conducted in the following cases:

- Hardware or software parts containing safety-critical diagnostic functions.
- ASIL decomposition (see Subsection 2.2.3).
- Software partitioning, i.e. software allocation to shared hardware resources.
- Temporary execution of software tasks on shared hardware.

- Shared resources in hardware (e.g., shared memory).

In this work, the traditional procedure is considered to create a formalized metamodel covering the analysis aspects and artefacts in the DFA context and subsequently enable an associated model-driven support and establish potential synergies with FMEDA and FTA as well as possible links to the fault simulation context.

5.3.3.1 Traditional DFA Flow and Documentation

The basic DFA steps are derived from the ISO 26262 standard and illustrated in a flowchart (see Figure 5.13). Once the system DFA is initiated, general system information covered by the design description for example shall be collected. The safety requirements related to the independence within hardware and/or software parts and to the freedom of interference must be extracted from the complete list of safety requirements in order to perform the DFA properly.

The first major step in the DFA process is to identify so-called *vulnerable* system parts which may be affected by dependent failures. Redundant elements and shared resources are the most common candidates to dependence-related vulnerability. The second step is to determine and classify the dependencies by which the identified vulnerable parts are interrelated. *Interfacing mechanism*, *shared supply*, and *shared clock* are examples of the considered dependency classes in the ISO 26262 context.

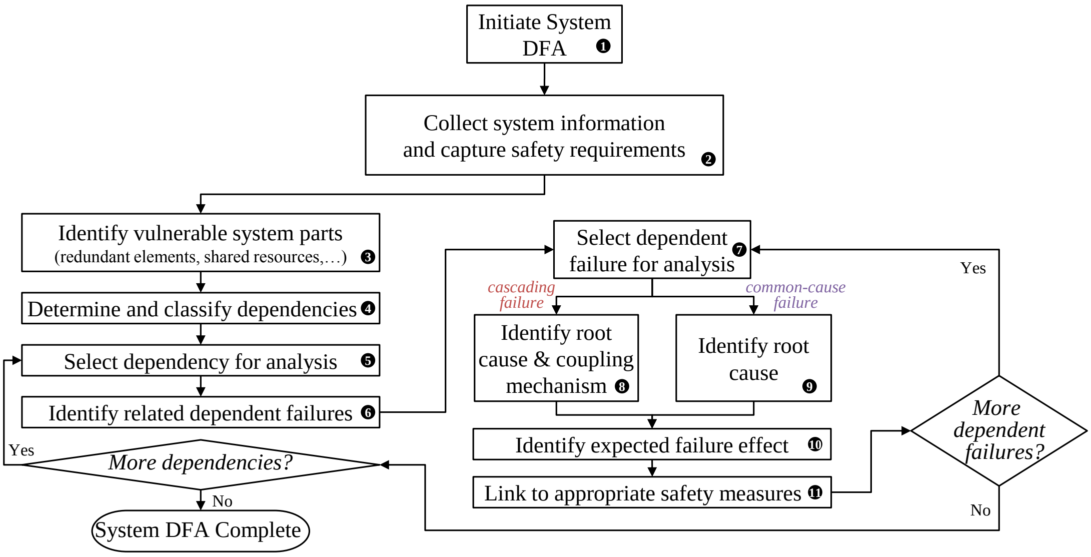

Initiate System DFA ❶
Collect system information and capture safety requirements ❷
Identify vulnerable system parts (redundant elements, shared resources,...) ❸
Determine and classify dependencies ❹
Select dependency for analysis ❺
Identify related dependent failures ❻
Yes
More dependencies?
No
System DFA Complete
Select dependent failure for analysis ❼
cascading failure
common-cause failure
Identify root cause & coupling mechanism ❽
Identify root cause ❾
Identify expected failure effect ❿
Link to appropriate safety measures ⓫
More dependent failures?
Yes
No

Figure 5.13: Flowchart of DFA Steps

In the remaining steps of the process flow shown in Figure 5.13, the iterative character of the DFA is illustrated. In fact, for each identified dependency, all related potential dependent failures must be designated. Afterwards, the defined dependent failures are progressively analyzed. The differentiation between *cascading failures* and *common cause failures* affects the subsequent flow. In fact, common cause failures result from a single specific event or root cause, while cascading failures are failures of a device element causing at least one other element to fail because of *coupling mechanisms* in the architecture (see Figure 5.14).

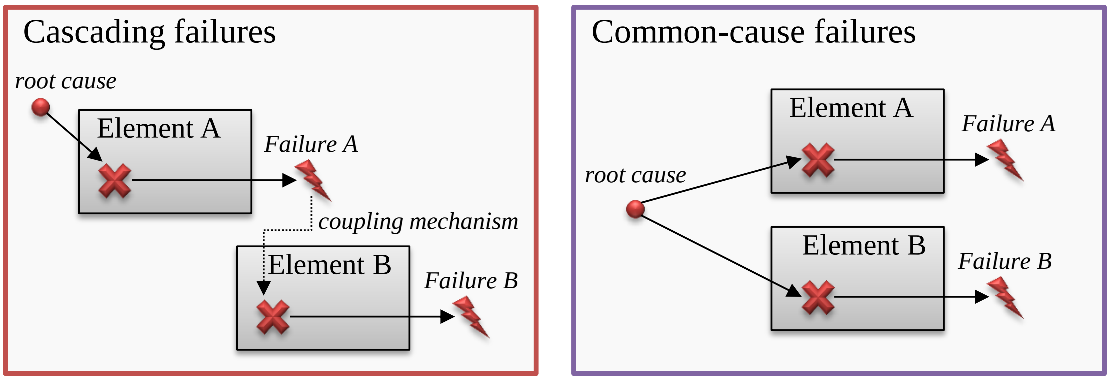


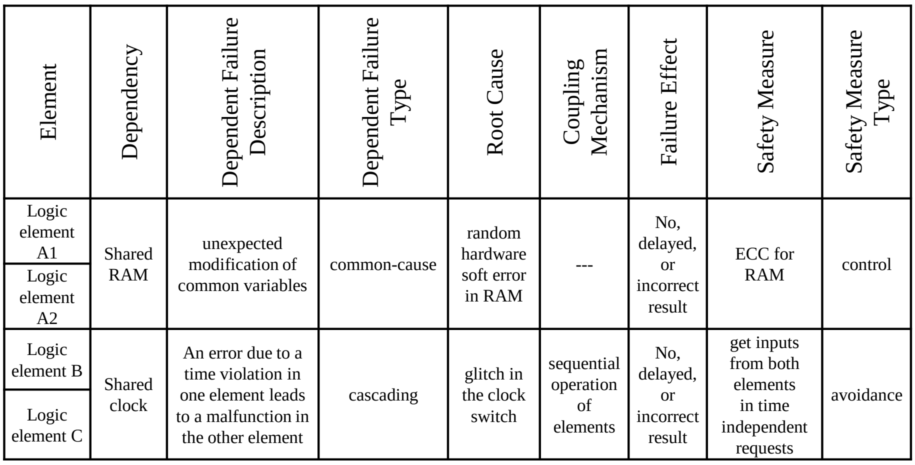
Figure 5.14: Differences between Cascading Failures and Common-Cause Failures [2]

For cascading failures, potential root causes and the related coupling mechanisms must be defined. For common cause failures, only the potential root causes (failure sources) behind them need to be identified. Physical and logical dependent failure sources are considered. Independently from the type of the dependent failure, the expected failure effect must be defined. Finally, the DFA analyst shall list the safety measures and the further action items (e.g., design changes) that must be undertaken to prevent dependent failures and/or to control their effects. These measures are qualitatively evaluated in the scope of the DFA.

In the industrial DFA practice, the documentation of analysis data and results is done in tabular form. An example of a DFA documentation is given in Table 5.4.

5.3.3.2 DFA Metamodel

In accordance to the DFA process flow and to the structure of its tabular documentation presented in the previous subsection 5.3.3.1, a DFA metamodel is constructed. It is represented in a simplified form as UML class diagram in Figure 5.15. The metamodel covers the DFA artefacts through the different classes such as **RedundantElement**, **DependentFailure**, **FailureEffect**, **RootCause**, **CouplingMechanism**, and **SafetyMeasure**. The *composition* and *reference* association in the metamodel reflect the relationships between the analysis artefacts as previously



Element	Dependency	Dependent Failure Description	Dependent Failure Type	Root Cause	Coupling Mechanism	Failure Effect	Safety Measure	Safety Measure Type
Logic element A1	Shared RAM	unexpected modification of common variables	common-cause	random hardware soft error in RAM	---	No, delayed, or incorrect result	ECC for RAM	control
Logic element A2								
Logic element B	Shared clock	An error due to a time violation in one element leads to a malfunction in the other element	cascading	glitch in the clock switch	sequential operation of elements	No, delayed, or incorrect result	get inputs from both elements in time independent requests	avoidance
Logic element C								

Table 5.4: Example of a DFA Table

explained. For example, the **Structure** considered during the analysis, which might be either a complete *System* or just a *Component*, has as child nodes a set of **VulnerableElement**, **Dependency**, **DependentFailure** and **SafetyMeasure** instances, and each **Dependency** class instance has a number of references to the related **VulnerableElement** class instances.

The formalization offered by the metamodel in Figure 5.15 provides a clear structure of the traditional DFA process and of the underlying documentation. Beyond this formalization, the metamodel represents a foundation for a model-driven support of the analysis. Through tool synthesis, a software framework, called *DFA MetaLib* is created in compliance to the metamodel (more details about MetaLibs and the underlying implementation environment are given in Subsection 6.1.2).

Within the DFA MetaLib, capturing the system structure (elements and dependencies) and the related safety data becomes easier through the application of dedicated parsers for structured files describing the system architecture and extracting relevant failure modes and safety mechanisms from the available catalogues. DFA analysts can also be assisted in performing their traditional manual tasks through a generated Graphical User Interface (GUI) based on the DFA metamodel. For example, the identification of the dependent failures, their association with the appropriate safety measures, and the classification of the dependencies can be considerably simplified.

Furthermore, the metamodel-based DFA formalization is a first step towards linking the dependent failure analysis to the two other safety analyses addressed in this work: FMEDA 5.3.1 and FTA 5.3.2. The respective data models are connected together, and through model-to-model transfer or transformation, already captured data in one analysis may be reused in another analysis. An example for this is to import the

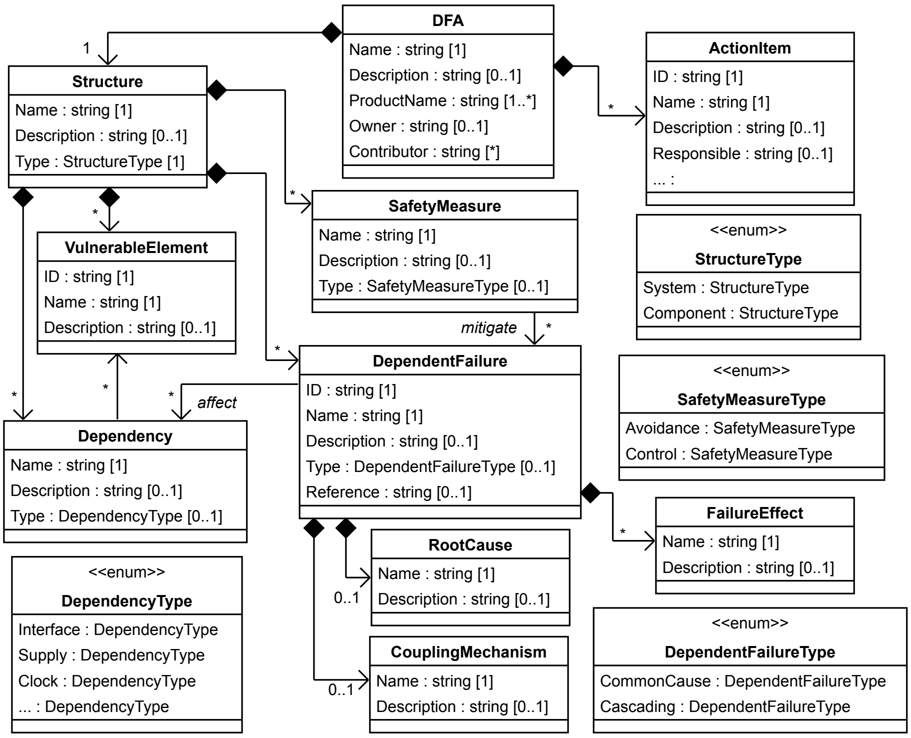

DFA
Name : string [1]
Description : string [0..1]
ProductName : string [1..*]
Owner : string [0..1]
Contributor : string [*]
ActionItem
ID : string [1]
Name : string [1]
Description : string [0..1]
Responsible : string [0..1]
... :
Structure
Name : string [1]
Description : string [0..1]
Type : StructureType [1]
1
*
SafetyMeasure
Name : string [1]
Description : string [0..1]
Type : SafetyMeasureType [0..1]
<<enum>>
StructureType
System : StructureType
Component : StructureType
VulnerableElement
ID : string [1]
Name : string [1]
Description : string [0..1]
mitigate
DependentFailure
ID : string [1]
Name : string [1]
Description : string [0..1]
Type : DependentFailureType [0..1]
Reference : string [0..1]
affect
<<enum>>
SafetyMeasureType
Avoidance : SafetyMeasureType
Control : SafetyMeasureType
Dependency
Name : string [1]
Description : string [0..1]
Type : DependencyType [0..1]
FailureEffect
Name : string [1]
Description : string [0..1]
RootCause
Name : string [1]
Description : string [0..1]
0..1
<<enum>>
DependencyType
Interface : DependencyType
Supply : DependencyType
Clock : DependencyType
... : DependencyType
CouplingMechanism
Name : string [1]
Description : string [0..1]
0..1
<<enum>>
DependentFailureType
CommonCause : DependentFailureType
Cascading : DependentFailureType

Figure 5.15: Simplified DFA Metamodel

failure modes which have been identified during an FMEDA for a given system into the DFA platform for the same system.

Finally, compositional failure models created using MetaFPA 5.2 can be used to assist the identification of dependencies between failures in DFA. In fact, the failure propagation graphs are an indication to the analyst on how certain malfunctions are coupled within the system structure. A semi-automated extraction of such dependency information from MetaFPA data-models and/or propagation graphs is possible. However, it has not been investigated in the scope of this thesis.

6 Model-Driven Support of Functional Safety Evaluation

This chapter addresses the model-driven platforms developed during this thesis to apply the formalization strategies presented in Chapter 5 and enable automated support of functional safety evaluation.

6.1 Introduction

In this section, the main objectives of model-driven functional safety evaluation are briefly recapitulated in Subsection 6.1.1. Then, the underlying environment for the implementation of the model-driven support is presented in Subsection 6.1.2. Afterwards, the organization of the chapter remainder is outlined in Subsection 6.1.3.

6.1.1 Main Objectives

Because of its constantly rising importance in the industry – particularly in the automotive sector, functional safety evaluation is extensively addressed by the research community with respect to its efficiency and to the link between its two major aspects, namely analysis and simulation. In Chapter 3 (*Related Work*), Section 3.2 gives a compilation of several methodologies for safety analysis automation found in the literature and Section 3.3 provides an overview of the known approaches to connect safety analysis methods to fault injection and simulation techniques. It is however also shown that many limitations are still to be overcome in the context of the automated support of functional safety analysis (e.g., the missing quantitative aspect and the incompatibility with the industrial practices of certain model-based FMEA automation approaches – see Subsection 3.2.2) and also in the context of the seamless linking between analysis and simulation (e.g., the limited flexibility and usability of most commercial FMEDA tools – see Subsection 3.3.2).

Thus, model-driven support of functional safety evaluation is addressed in this thesis to tackle the inconveniences detected in the existing methodologies. Through developing appropriate software platforms and tools, safety engineering tasks are simplified throughout the ISO 26262 safety lifecycle and the associated automation, interoperability, and reuse levels are enhanced.

The objectives to be fulfilled through model-driven support are inherently related to the overall requirements defined in Chapter 4, especially those of Subsections 4.1.3, 4.1.4, and 4.1.5. These objectives are listed below along with references to the respective requirements.

- Develop model-based software applications in compliance with the metamodels for failure propagation modeling and functional safety analysis as presented in Chapter 5. The defined process flows and data structures shall be reflected and/or supported by the developed tools (**REQ 4**). When the underlying flows and/or metamodels are changed and/or extended due to project, team, or organization specificities, the associated frameworks and tools shall be accordingly updated. This framework update shall be performed automatically as far as possible and a consistent report shall be generated to trace back the undertaken changes (**REQ 7**).
- Ensure automated data extraction from the relevant inputs for the functional safety evaluation through dedicated tools (**REQ 8**).
- Reduce the implementation efforts of the functional safety evaluation platforms by generating the rather generic parts which do not necessitate an advanced customization instead of developing them manually (**REQ 9**).
- Establish a semi-automated link between safety analysis and fault injection/simulation. It is required to perform an investigation of data equivalences and/or correspondences between the supported safety analyses (FMEDA, FTA, and DFA) on the one hand and fault injection and simulation on the other hand. Based on the outcomes of this investigation, tools for data mapping and/or transformation shall be developed (**REQ 10**).
- Support dynamic data changes within the inputs and the underlying data models and/or databases (**REQ 11**).
- Enable the systematic generation of safety evaluation views out of the safety data models handled within the created platforms, such as failure propagation graphs, fault trees, and fault libraries to be injected (**REQ 12**).
- Support the exchange and the reuse of safety evaluation artefacts across the developed platforms and anticipate later migration to different standards or domains though dedicated tools (**REQ 13** and **REQ 14**).
- Enhance the usability of the developed software platforms and tools by providing user-friendly graphical interfaces for data input, handling, and visualization (**REQ 15**).

6.1.2 Underlying Environment

The achievement of the objectives listed in Subsection 6.1.1 relies in the context of this thesis on the application of the solution concepts presented in Section 4.2. Thereby, the guidelines of model-driven development (4.2.1) are followed, the template-based and API-based code generation techniques are applied in correlation with metamodeling (4.2.2), and the hand-coded and semantic data mapping approaches are utilized (4.2.4). Furthermore, the metasynthesis concept denoting the synthesis of synthesis tools is considered (4.2.3).

Concretely, these concepts are embodied in *Metagen*, Infineon's in-house metamodeling and code generation environment. Along with few commercial tools used traditionally in the industrial safety evaluation context such as Excel (for FMEDA spreadsheets) and Isograph's Reliability Workbench™ (for creation, edition, and visualization of fault trees), the Metagen environment is the central platform on which the implementation of all thesis concepts is performed.

Metagen is a software development platform which has been in use at Infineon for more than 5 years. It offers highly-automated capabilities for data modeling and code generation using Application Programming Interfaces (APIs) and text templates [5]. Metagen can be defined as a "design building box for a metamodeling infrastructure". It enables the automated derivation of expected target code instead of manually typing it. The basic three-step process flow of Metagen is described as follows [126]:

1. Create a *metamodel* capturing the structure of the targeted data model.
2. Implement a specification *reader* to fill the data model (instance of the metamodel).
3. Write a corresponding code generator referred to as *template*.

Among the basic features of Metagen, a metamodel-compliant API gets automatically generated. It contains a set of object classes and functions enabling to fill, manipulate, and access the correspondingly created data model. Besides, Metagen provides tools for generating GUIs (Graphical User Interfaces) as well as generic data parsers and writers to simplify the construction and the handling of data models. A plug-in mechanism is supported within Metagen to glue all the different pieces together. Furthermore, it is possible in Metagen to combine different data models and establish mapping, transition, and transformation relationships between them through a linking mechanism relying on internal and external references [126].

The key advantage of Metagen is the tool-based automation it provides as well as the considerable flexibility allowed by its dedicated extension mechanism. Beyond the systematic, highly consistent, and efficient production of countless types of model views (RTL and/or C code, verification test-benches, documentation files, data graphs, etc.,) the solutions promoted by Metagen are easily expendable to cover new domains of design, verification, and testing. This is enabled by creating so-called *MetaLibs* correspondingly for those new domains. In the context of this thesis, the ability of Metagen to contribute to the ISO 26262 safety lifecycle by assisting related safety engineering tasks is demonstrated

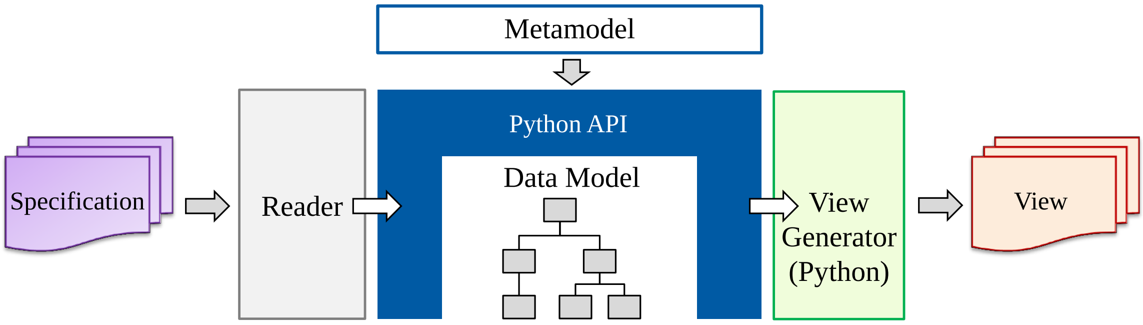


Figure 6.1: Basic Metagen Framework [5]

Figure 6.1 illustrates the basic Metagen setting and shows the relations between the different levels of abstraction for a given system. The metamodel is defined at the highest abstraction level as the structural organization of the model. In Metagen, the metamodel is stored either as a UML class diagram or as an equivalent textual file written for example in the XMI (XML Metadata Interchange) format. In compliance with the metamodel, Metagen generates the corresponding API in Python, using a precoded generator. The data model is located at one abstraction level below. It is constructed by the API using the information extracted from the specification through the reader. At the lowest abstraction level, the output system view, e.g., VHDL code, SystemC code, documentation file etc., is generated through a view generator which is granted data model access by the API [127, 5].

In Metagen, the view generator is an object-oriented program which is produced from a hand-written template through a template engine, instead of being typed manually. The template engine used per default in Metagen is *Mako*, a library developed in Python. The Mako language is considered an *embedded* Python language and is characterized by a compact template syntax and a rich template rendering API [5, 128]. Mako supports the three basic features of template rendering [129, 130, 131, 5]:

1. *Value substitution*: replacement of placeholders with corresponding data by evaluating the contained expressions, pulling the appropriate piece of information from the model and writing it into the output view.

2. *Template logic*: logic and control-flow statements such as *If conditionals*, *For loops*, and *recursive macros*.

3. *Embedded code*: using unrestricted Python code to raise template capabilities.

Figure 6.2 illustrates the role of the Mako template engine in the generation process within Metagen. It should be noted that the view generator is not implemented manually, but generated from a template through the Mako template engine. In fact, writing the corresponding template is much easier for the user, because of the higher abstraction level and the simpler syntax.

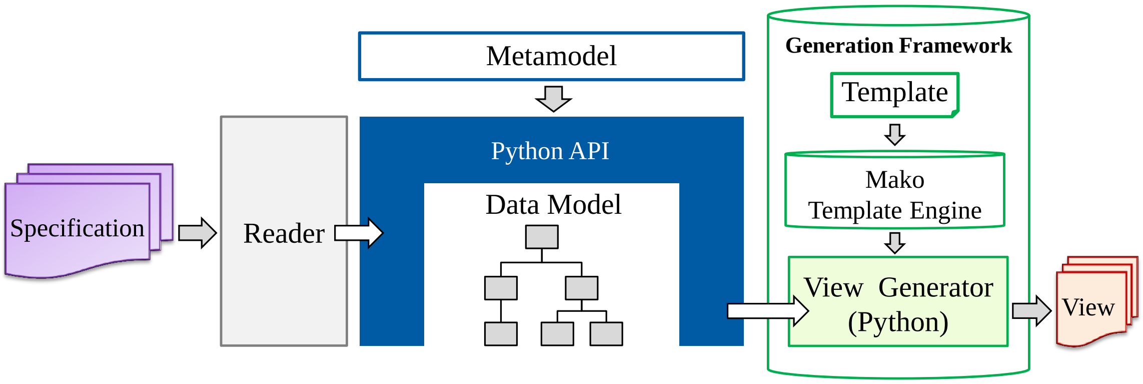


Figure 6.2: Role of the Mako Template Engine in the Metagen Framework [5]

6.1.3 General Organization

In the remainder of this chapter, the model-driven platforms and tools developed in the context of this thesis are presented.

First, Section 6.2 describes the MetaFPA framework which is developed to realize the enhanced failure propagation analysis introduced in Section 5.2. The section contains mainly (ii) a general overview of the MetaFPA framework, (ii) a brief presentation of the associated GUI, and (iii) an explanation of the functionality of the generation tool for failure propagation graphs.

In Section 6.3, the so-called safety analysis *MetaLibs* created in accordance to the FMEDA (5.3.1) and FTA (5.3.2) metamodels are presented. Beyond the general setting of each MetaLib, special tools for automation support and/or safety documents generation are addressed.

Finally, Section 6.4 introduces *SaVer*, a safety verification framework developed to establish the targeted link between safety analysis on the one hand and fault injection and simulation on the other hand.

6.2 MetaFPA Framework for Failure Propagation Analysis

In this section, the implementation of the MetaFPA framework for failure propagation modeling and analysis in accordance with the theoretical foundation presented in Section 5.2 is addressed. First, in Subsection 6.2.1, the general structure and setting of the MetaFPA framework is presented. Then, a quick overview on the MetaFPA user interface is provided in Subsection 6.2.2. After that, the tool-based generation of failure propagation graphs within MetaFPA is explained in Subsection 6.2.3. Finally, the interactions of MetaFPA with other model-driven safety frameworks developed in the context of this thesis, particularly for FMEDA and FTA support, are outlined in Subsection 6.2.4.

6.2.1 General Setting

As already explained in Subsection 5.2.1, MetaFPA is a model-based and simulation-oriented approach for failure propagation analysis, which connects concepts from the failure logic modeling theory with ideas and techniques used in fault injection to simulate system alterations. The purpose of MetaFPA is to explore the failure behavior of a given system at a high abstraction level during the early concept stages of the ISO 26262 safety lifecycle. The outcomes of the MetaFPA analysis are used to refine the safety concept, especially with respect to the adequacy and effectiveness properties of the intended safety mechanisms.

In the following, the basic artefacts of the MetaFPA methodology (see Figure 5.3) are listed along with their reflection on the MetaFPA framework which is implemented within the Metagen environment (see Subsection 6.1.2) and illustrated in Figure 6.3:

- **Metamodel for Failure Propagation Analysis**: The metamodel (see Figure 5.5 in Subsection 5.2.3) is the foundation of the MetaFPA framework. It complies with the basics of the Failure Logic Modeling theory (see Subsection 3.1.2) and incorporates the enhancements related to dynamic failure descriptions, automated analysis, and flexible deployment (more details in Subsection 5.2.2). The metamodel is created within the Metagen environment as a UML class diagram. Based on it, the initial setting of the MetaFPA simulation platform is synthesized. **A1** in Figure 6.3 corresponds to the metamodel for failure propagation analysis.
- **MetaFPA Simulation Platform**: The platform for simulation-oriented failure propagation analysis is depicted in Figure 6.3 as **A2**. It contains the application programing interface (Python API – **A3**) generated by Metagen in

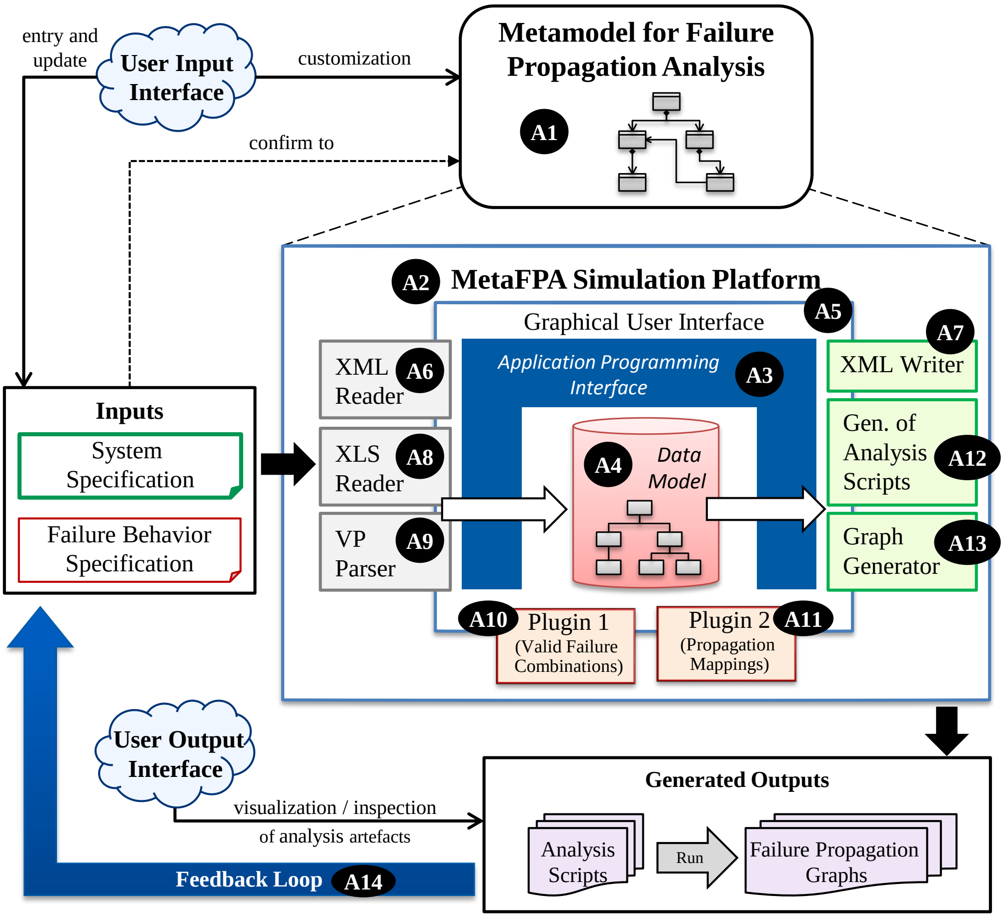

entry and update
User Input Interface
customization
confirm to
Metamodel for Failure Propagation Analysis
A1
A2 MetaFPA Simulation Platform
Graphical User Interface A5
Application Programming Interface A3
XML Reader A6
XLS Reader A8
VP Parser A9
A4 Data Model
XML Writer A7
Gen. of Analysis Scripts A12
Graph Generator A13
A10 Plugin 1 (Valid Failure Combinations)
Plugin 2 A11 (Propagation Mappings)
Inputs
System Specification
Failure Behavior Specification
User Output Interface
visualization / inspection of analysis artefacts
Feedback Loop A14
Generated Outputs
Analysis Scripts
Run
Failure Propagation Graphs

Figure 6.3: Overview of the MetaFPA Framework for Failure Propagation Analysis

accordance to the metamodel **A1**. On this platform, the corresponding data model is constructed, handled, and visualized through several tools which are either systematically generated or hand-written.

- **Executable System Model (extended with failure behavior)**: MetaFPA's data model (depicted in Figure 6.3 as **A4**) is a filled instance of the metamodel **A1**. It is an executable model of the system being analyzed where the nominal description is extended with the abstract failure behavior as defined in the metamodel **A1** (more details in Subsection 5.2.3).

- **Generated Tools**:

  ∗ *GUI*: Metagen generates a basic Qt GUI based on the metamodel. The GUI (artefact **A5** in Figure 6.3) corresponds to the input and output user interfaces previously mentioned in 5.2.1 and illustrated in Figure 5.3.

More details about the MetaFPA GUI are addressed in Subsection 6.2.2.

  - *Standard readers*: The default format for data storage in Metagen is XML. Therefore, a standard XML reader (artefact **A6** in Figure 6.3) gets systematically generated.
  - *Standard writers*: Similarly, a standard XML writer (artefact **A7** in Figure 6.3) is systematically generated.

- **Handwritten Extensions, Plugins, and Generators**:
  - *Input extensions for capturing the system architecture*: In addition to the manual data input supported through the GUI, it is possible in the MetaFPA framework to automatically extract information about the early system architecture at concept level either from a tabular specification provided in Excel through the XLS reader (**A8** in Figure 6.3) or from a virtual prototype (VP) developed in SystemC for example through the VP parser (**A9**).
  - *Plugin for determination of valid failure combinations*: In contrast to other failure logic modeling techniques, MetaFPA considers combinations of multiple failures occurring either inside specific system blocks or on their respective interfaces. There are however contradictory failures which cannot occur simultaneously (e.g., omission and commission failures). Such incompatibilities are specified as references in the metamodel **A1**. Once these references are instantiated in the data model **A4** to specify failure incompatibilities, the valid failure combinations are determined systematically by the plugin depicted as **A10** in Figure 6.3.
  - *Plugin for derivation of failure propagation mappings*: In MetaFPA, there is the possibility of defining the failure logic of a certain block either manually through the GUI or to derive it automatically using Boolean expressions. In fact the failure logic which links the input deviations and internal malfunctions on the one hand with the output deviations on other hand can be captured through Boolean expressions denoting the occurrence condition of output deviations depending on the input deviations and the internal malfunctions of the considered block. The plugin depicted as **A11** in Figure 6.3 takes these occurrence conditions as input and subsequently generates the corresponding failure propagation mappings in the data model **A4**.
  - *Generator for failure propagation analysis scripts*: Once the MetaFPA data model **A4** is fully constructed (system structure and failure behavior), the template-based script generator **A12** is used to generate special Python programs whose execution runs correspond to the multiple failure

configurations of the system. The impact of these failure configurations on the overall system behavior is monitored while executing the scripts. In other words, the purpose of this script-based failure propagation analysis is to detect those configurations leading to a critical system failure by monitoring the system outputs throughout the runs.

    - *Generator for failure propagation analysis graphs*: The outcomes of the script-based analysis are used for the generation of so-called *failure propagation graphs.* The corresponding generation tool, depicted in Figure 6.3 as **A13**, is addressed in details in Subsection 6.2.3.

- **Feedback Loop**: The MetaFPA framework provides assistance to concept and safety engineers at an early design stage. Through the feedback loop embodied in the MetaFPA framework (artefact **A14** in Figure 6.3), early system exploration with respect to safety goals is supported. The results of the failure propagation analysis represent valuable indications for potential deficiencies in the system architecture, particularly in the safety measures it contains. Through an investigation and/or a walk-through of the generated failure propagation graphs, the safety engineer identifies problematic locations in the system architecture, where a hardening is required. In such vulnerable locations, the system design shall be changed. Thereby, the back-tracing from the failure propagation to the system specification through the MetaFPA data model is particularly convenient.

6.2.2 MetaFPA GUI

Based on the metamodel for failure propagation analysis, a Qt graphical user interface is generated. It should be noted that Qt is a C++ library originally published in 1995. It has been since then most frequently used for the development of software applications with graphical user interfaces.

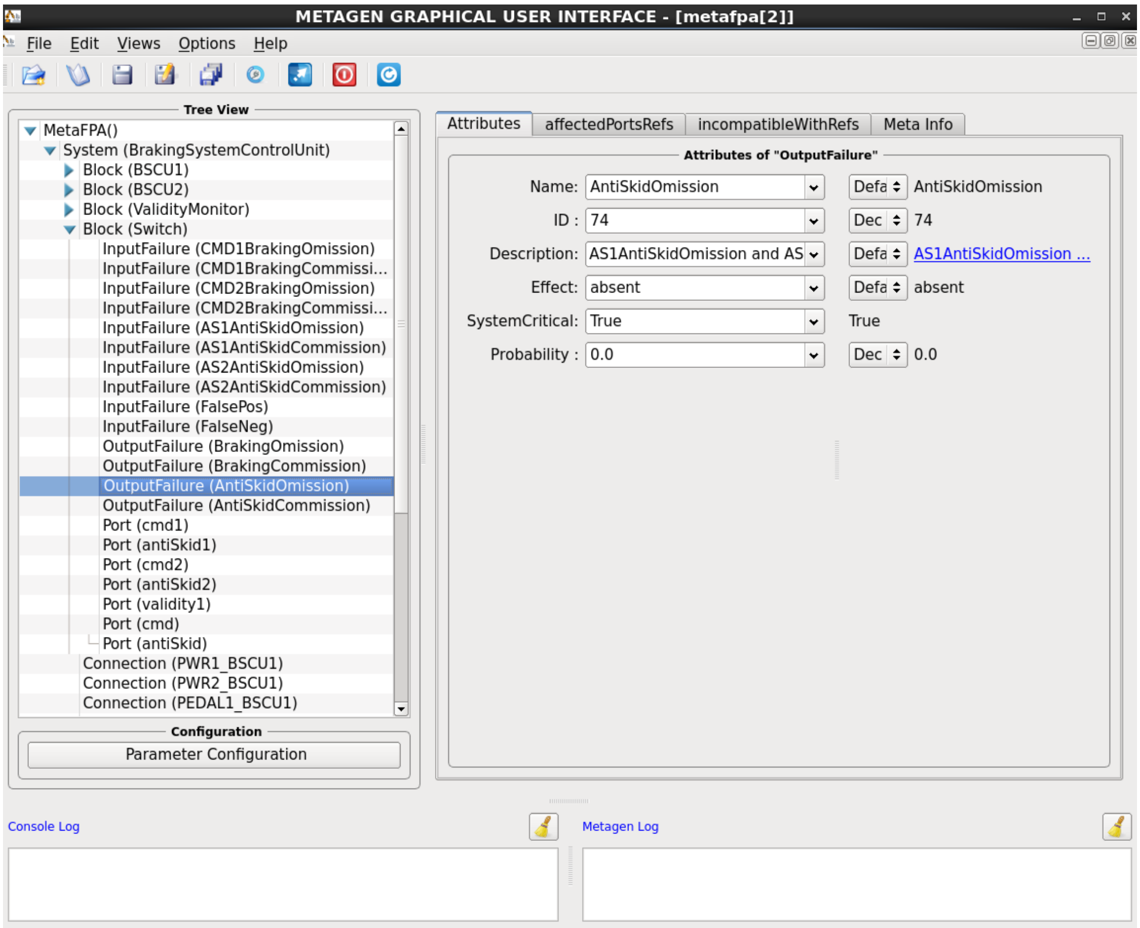
In the context of this thesis, the MetaFPA GUI is applied for:

1. **Data entry**: The GUI enables the specification of the abstract system structure (blocks, ports, and connections) and of the basic failure behavior (input/internal/output failures).

2. **Data visualization**: Through the GUI, already existing data models can be explored in a user-friendly way.

3. **Data model editing and handling**: Manual extensions and/or changes can be performed on existing data models, such as specifying incompatible failures as well as defining failure propagation mappings.

4. **Tool invocation**: Data readers and writers, extensions, and plugins created within the MetaFPA framework can be invoked from the GUI to (i) extract data and accordingly fill an empty metamodel instance, (ii) to process an existing data model and systematically extend or change it, or (iii) to generate a special view out of a data model.



MetaFPA()
System (BrakingSystemControlUnit)
Block (BSCU1)
Block (BSCU2)
Block (ValidityMonitor)
Block (Switch)
InputFailure (CMD1BrakingOmission)
InputFailure (CMD1BrakingCommissi...
InputFailure (CMD2BrakingOmission)
InputFailure (CMD2BrakingCommissi...
InputFailure (AS1AntiSkidOmission)
InputFailure (AS1AntiSkidCommission)
InputFailure (AS2AntiSkidOmission)
InputFailure (AS2AntiSkidCommission)
InputFailure (FalsePos)
InputFailure (FalseNeg)
OutputFailure (BrakingOmission)
OutputFailure (BrakingCommission)
OutputFailure (AntiSkidOmission)
OutputFailure (AntiSkidCommission)
Port (cmd1)
Port (antiSkid1)
Port (cmd2)
Port (antiSkid2)
Port (validity1)
Port (cmd)
Port (antiSkid)
Connection (PWR1_BSCU1)
Connection (PWR2_BSCU1)
Connection (PEDAL1_BSCU1)

Attributes of "OutputFailure"

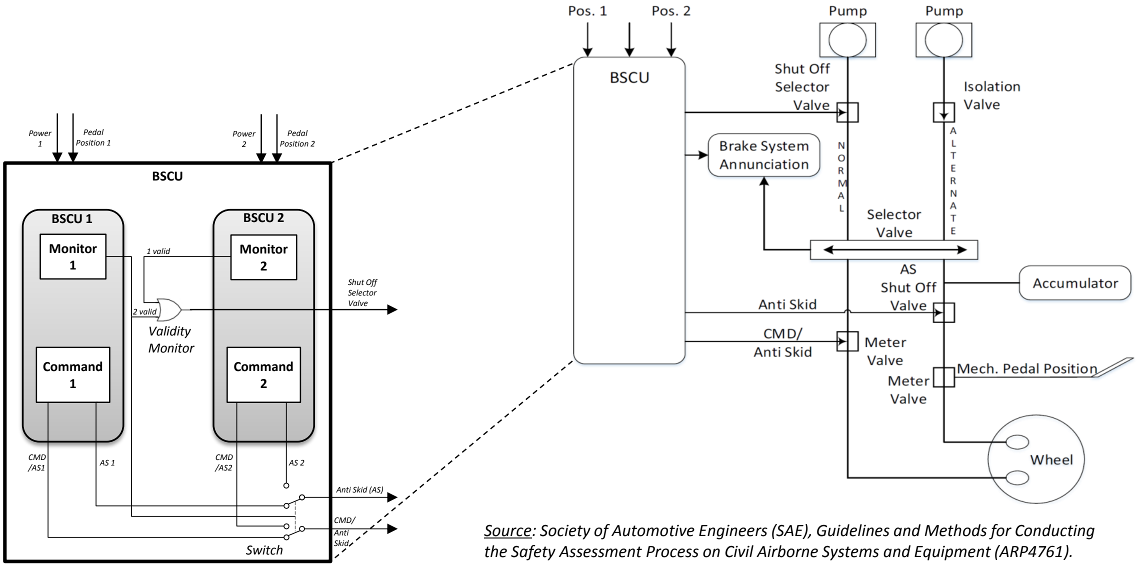
Name:	AntiSkidOmission	Defa	AntiSkidOmission
ID :	74	Dec	74
Description:	AS1AntiSkidOmission and AS	Defa	AS1AntiSkidOmission ...
Effect:	absent	Defa	absent
SystemCritical:	True		True
Probability :	0.0	Dec	0.0

Figure 6.4: Snapshot of the Graphical User Interface of the MetaFPA Framework

The utility of the MetaFPA GUI (see snapshot in Figure 6.4) is illustrated through a simple use case addressing the *Wheel Braking System (WBS)* described in [13]. This hydraulic braking system is applied in the avionics domain to "provide primary stopping force" by applying hydraulic pressure to the brake assemblies of the main landing gear wheels [13, 132].

The WBS contains a physical part and a control part. The physical part which consists of hydraulic pumps, circuits, and brakes is controlled either mechanically by the pilot through the pedals' positions or electrically through the control part denoted as *Braking System Control Unit (BSCU)* [13, 132].

There are different operation modes of the WBS. As illustrated in Figure 6.5, the normal mode of the WBS relies on the output of the BSCU to control the meter valves, more specifically the combined brake and anti skid command. It should be noted that the brake command is derived from the pilot pedal position while the anti skid command is derived from the sensor inputs indicating mainly the wheel and the ground speeds. In the alternate mode of the WBS, the meter valves are controlled by the associated pedal positions. The switching between the two modes is ensured by the selector valve and is triggered by pressure fluctuations [13, 132].



Source: Society of Automotive Engineers (SAE), Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment (ARP4761).

Figure 6.5: Overall Architecture of the WBS and of the Contained BSCU [13]

The focus of the failure modeling in this use case is the BSCU which is the only digital component of the complete wheel braking system. Therefore, its structure is illustrated in Figure 6.5 (on the left). The BSCU consists of two redundant *Monitor* and *Command* units. Each pair of them forms a subsystem (*BSCU1* and *BSCU2*) and is independently powered by the power supplies illustrated in Figure 6.5. In addition to the power sources, the BSCU receives two redundant “Pedal Position” signals from the cockpit. The primary output of the BSCU is the combined control signal *CMD/Anti Skid* which controls the meter valves at the higher level of the WBS in normal mode. Furthermore, the BSCU produces the pure Anti Skid (AS) signal which is needed for the alternate mode of the WBS and the Shut Off Selector Valve signal which is necessary to prevent hazardous hydraulic flow if the outputs of the control unit are considered untrustworthy. The BSCU outputs are considered untrustworthy if an internal malfunction is detected or if a discrepancy between the pedal inputs is diagnosed. That is why, the Shut Off Selector Valve signal is produced by the so-called *Validity Monitor* within the BSCU [133, 66, 132].

Using the generated GUI, a MetaFPA structure model for the BSCU is created according to Figure 6.6.

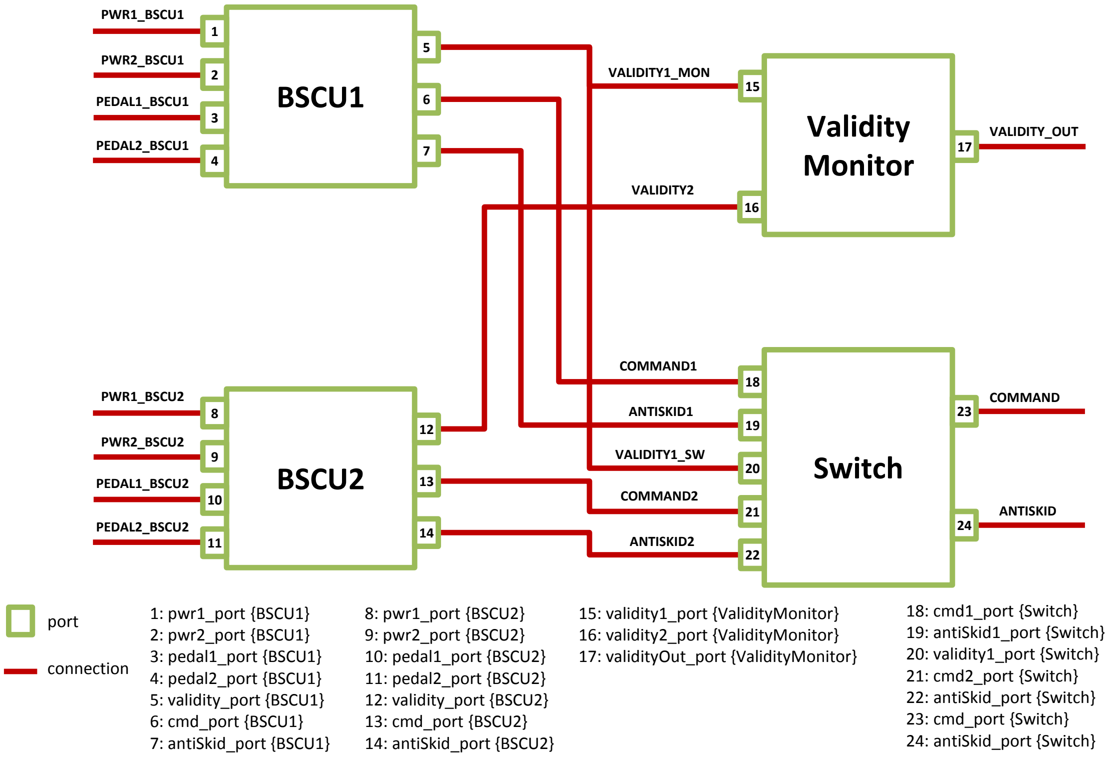


Figure 6.6: MetaFPA Structure Model of the Braking System Control Unit (BSCU)

In this use case, four types of failures are considered:

- **Omission**: In this use case, omission refers to the absence of an expected entry and to low value failures (entry is present but its value is below the valid range).
- **Commission**: In this use case, commission refers to the occurrence of an inadvertent or unexpected entry and to high value failures (expected entry is present but its value is beyond the valid range).
- **False Negative**: A test or diagnostic is false negative if it results in a so-called *rejection failure*. In reality, the test/diagnostic results are problematic and must be rejected, however the rejection does not happen and the results are accepted.
- **False Positive**: A test or diagnostic is false positive if it results in a so-called *incorrect rejection*. In reality, the test/diagnostic results are correct and must

be accepted, however they get rejected. This situation is equivalent to a false alarm.

In Figure 6.7, the considered input deviations, internal malfunctions, and output deviations for *BSCU1* are listed along with the ports they potentially affect in the structure model.

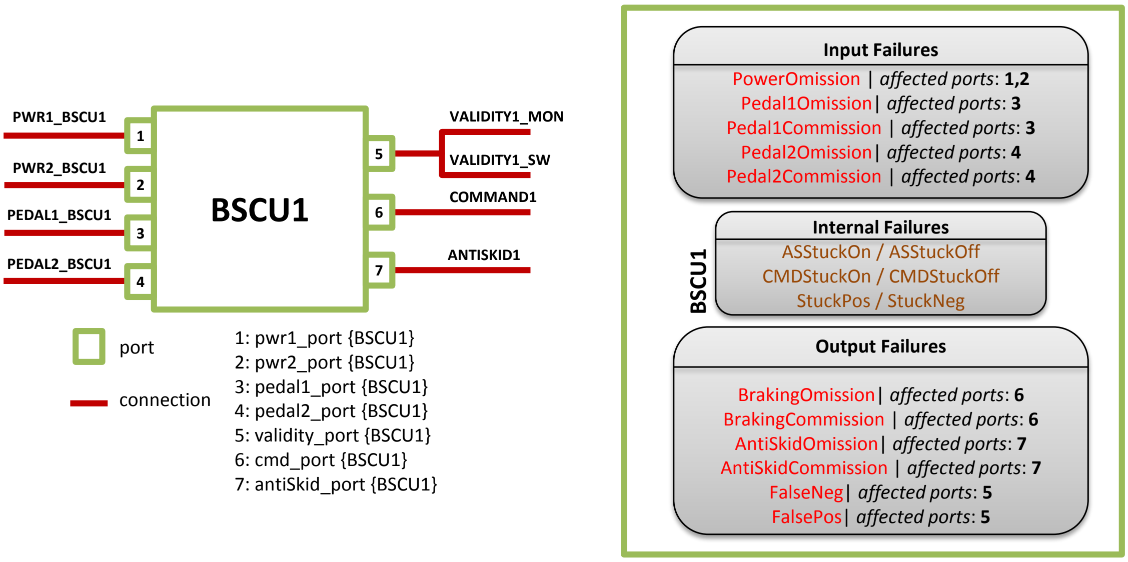


Figure 6.7: Examples of Braking System Control Unit Failures

The usage of the MetaFPA GUI to perform such failure logic modeling ranges from specifying the system composition (contained blocks, their respective ports, connections between them) to defining the failure behavior of the different blocks (input/output failures and their respective affected ports, internal malfunctions, etc.,). Furthermore, the GUI can be used to display probabilities of system output failures which are calculated during the failure propagation analysis runs. A snapshot of the GUI is given in Figure 6.4. It shows the structure of the tree view and how the attributes of each object of the data model are displayed.

6.2.3 Failure Propagation Graphs Generator

Among the handwritten tools applied within the MetaFPA framework, the failure propagation graphs generator (artefact **A13** in Figure 6.3) transforms the outcomes of the script-based failure propagation analysis into graphical representations depicting the propagation paths leading to critical system failures.

The generation of the propagation graphs relies on the numerous analysis runs performed through the generated scripts. In fact, for every possible combination of input deviations and internal malfunctions of the starting (entry) blocks, the resulting output deviations are determined based on the propagation mappings and consequently deployed on the output ports. Afterwards, the outgoing connections are activated to propagate the result on the target ports. Then, the rest of the architecture is recursively analyzed (detection of propagated input deviations, deployment of internal malfunctions, identification of output deviations, propagation through connections, etc.,). When an analysis run results in a critical system failure already specified by the user in the MetaFPA data model, a GraphML diagram is generated to visualize the propagation flow which leaded to that system failure. In Subsection 6.2.3.1, the output format of the generation, which is GraphML, is briefly described and illustrated through an example. The details of the failure propagation analysis and the generation algorithm are addressed in Subsection 6.2.3.2.

6.2.3.1 Output Format

In MetaFPA, the failure propagation graphs are generated in the XML-based format GraphML, which is a comprehensive and user-friendly file format to create and store graphs [134]. Consisting of a language core describing structural graph properties and an extension mechanism to flexibly add application-specific data, GraphML supports the creation of (i) directed, undirected, and mixed graphs, (ii) hypergraphs, and (iii) hierarchical graphs [134]. The motivation to use GraphML is that instead of having a custom syntax like other graph file formats, it relies on XML. Subsequently it is easily applicable for generating, archiving, and/or processing graphs [134]. Figure 6.8 depicts an example of a generated graph for the Braking System Control Unit (BSCU) introduced in Subsection 6.2.2 and illustrated in Figure 6.5.

The critical system failure of the BSCU which is addressed by the graph in Figure 6.8 is the combination of the output deviation “ValidityOutFalsePos” of the ValidityMonitor and the output deviation “BrakingOmission” of the Switch (see Figure 6.6). In other words, the validity signal indicates erroneously that the COMMAND1 output of the BSCU1 block is invalid and cannot be trusted because of an internal failure called “StuckPos” in BSCU1. Therefore, the COMMAND2 signal of the BSCU2 block will be used to produce the COMMAND output of the switch. But the COMMAND2 signal itself is erroneous (“CMD2BrakingOmission”) because of an internal failure in the BSCU2 block. That is why, the braking command expected at the output of the BSCU is not correctly generated and the overall wheel braking system is affected: a lack of braking occurs.

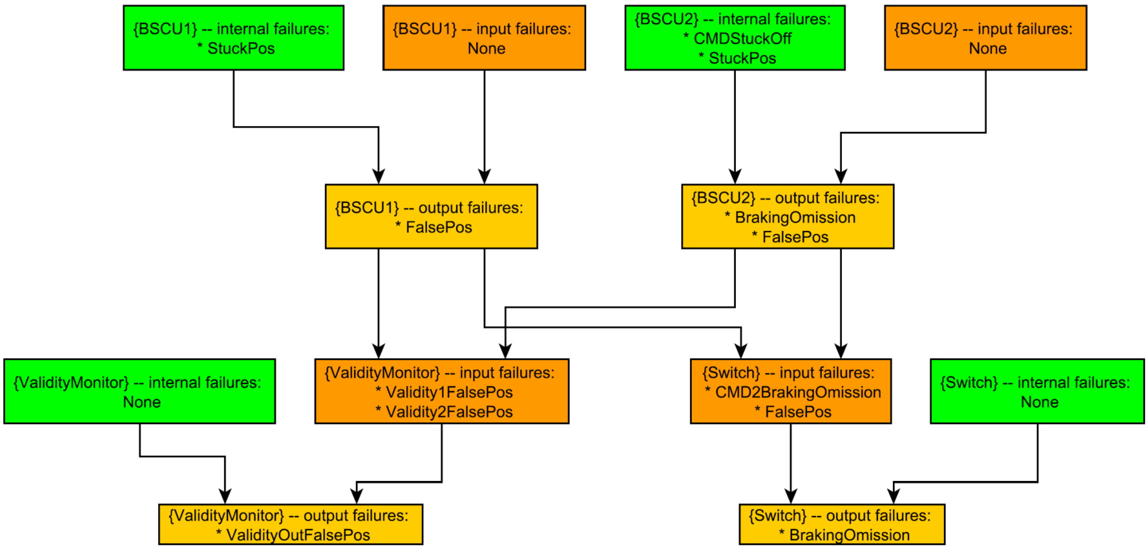

{BSCU1} -- internal failures:
* StuckPos
{BSCU1} -- input failures:
None
{BSCU2} -- internal failures:
* CMDStuckOff
* StuckPos
{BSCU2} -- input failures:
None
{BSCU1} -- output failures:
* FalsePos
{BSCU2} -- output failures:
* BrakingOmission
* FalsePos
{ValidityMonitor} -- internal failures:
None
{ValidityMonitor} -- input failures:
* Validity1FalsePos
* Validity2FalsePos
{Switch} -- input failures:
* CMD2BrakingOmission
* FalsePos
{Switch} -- internal failures:
None
{ValidityMonitor} -- output failures:
* ValidityOutFalsePos
{Switch} -- output failures:
* BrakingOmission

Figure 6.8: Example of Generated Failure Propagation Graph (BSCU Use Case)

6.2.3.2 Generation Algorithm

In the MetaFPA context, a system S is specified by a set of blocks $\mathcal{B}$ and a set of connections $\mathcal{C}$: $S = (\mathcal{B}, \mathcal{C})$ with $\mathcal{B} = \{B_i\}$; $i = 1...N_{\mathcal{B}}$ and $\mathcal{C} = \{C_i\}$; $i = 1...N_{\mathcal{C}}$ ($N_{\mathcal{B}}$ and $N_{\mathcal{C}}$ are respectively the numbers of blocks and connections in S). Each block B_i is characterized by a set of input ports $\mathcal{P}_i^I = \{p_{ik}^I\}$, a set of output ports $\mathcal{P}_i^O = \{p_{ik}^O\}$, a set of input deviations $\mathcal{D}_i^I = \{d_{ik}^I\}$, a set of output deviations $\mathcal{D}_i^O = \{d_{ik}^O\}$, and a set of internal malfunctions $\mathcal{M}_i = \{m_{ik}\}$. Thus, a block is denoted as follows: $B_i = (\mathcal{P}_i^I, \mathcal{P}_i^O, \mathcal{D}_i^I, \mathcal{D}_i^O, \mathcal{M}_i)$. A propagation mapping function $\mathbb{P}_i : \mathcal{P}(\mathcal{D}_i^I) \times \mathcal{P}(\mathcal{M}_i) \to \mathcal{P}(\mathcal{D}_i^O)$ defines the relations between the *power-sets* of input deviations, internal malfunctions, and output deviations. Furthermore, the failure propagation analysis is based on the dynamic evaluation of each block's state at a given simulation iteration x. This block state denoted as $\mathbb{S}^{(x)}(B_i)$ is given by the port values and the occurring internal malfunctions at simulation iteration x. The detected port values allow the determination of currently occurring input and/or output deviations.

Performing a failure propagation analysis in the MetaFPA framework requires a number of preprocessing steps to be executed and a multitude of conditions to be met. First, so-called *entry* (first) and *exit* (final) blocks are either defined by the user or automatically detected. On the one hand, an *entry* block is a block with at least one *incoming* connection with no *source* reference. On the other hand, an *exit* block has at least one *outgoing* connection with no *target* reference. Then, the *back-couplings* in the system are detected and classified according to the involved blocks and to the

Algorithm 1 Failure Propagation Simulation Flow

```
procedure SIMULATE($B_i$, $x_i^{former}$)
    $READY \leftarrow TRUE$
    for $p_{ik}^I \in \mathcal{P}_i^I$ do
        $READY \leftarrow READY$ AND $\Theta(p_{ik}^I)$
    end for
    if $READY = TRUE$ then
        $x \leftarrow x_i^{former} + 1$
        $\delta_i^{I(x)} \leftarrow$ set of occurring input deviations
        $\mu_i^{(x)} \leftarrow$ set of occurring internal malfunctions
        $\delta_i^{O(x)} \leftarrow \mathbb{P}_i(\delta_i^{I(x)},\ \mu_i^{(x)})$
        $\mathbb{S}^{(x)}(B_i) \leftarrow (\delta_i^{I(x)},\ \mu_i^{(x)},\ \delta_i^{O(x)})$
        if $x > 1$ AND $\mathbb{S}^{(x)}(B_i) = \mathbb{S}^{(x-1)}(B_i)$ then        ▷ fix-point reached
            return
        end if
        Update output ports set of $B_i$ ($\mathcal{P}_i^O$)
        Activate outgoing connections set of $B_i$
        for each next block $B_j$ do
            SIMULATE($B_j$, $x_j^{former}$)
        end for
        $x_i^{former} \leftarrow x$
    else
        return
    end if
end procedure
```

connection path loops.

Back-couplings in the system composition are addressed in the MetaFPA context through a fix-point approach, where the simulation iterations for a specific block are terminated when its state gets stabilized, i.e., when the set of occurring deviations remains unchanged. Infinite loops are avoided through a forced termination with error logging after a certain number of iterations.

Furthermore, the criticality characteristics of system failures to be detected and documented through the analysis are captured. In general, these characteristics consist in the output deviations of the exit blocks which are considered as critical for the system safety. The maximum tolerable probability of occurrence is also specified; if it is reached or exceeded, then the corresponding system configuration consisting of the different block states has to be documented.

The concrete simulation flow on the MetaFPA platform is started by the deployment of a valid combination of input deviations for every entry block. The particularity of entry blocks is precisely the possibility to force input deviations by manipulating input port values. For all other blocks, the input port values are obtained through the propagation across the connections. A block is considered *"READY"* to be simulated if all its input ports have specific values which are either forced to them (for entry blocks) or obtained through previous simulation iterations. This information is stored for each input port within the considered block through the Boolean variable $\Theta(p^I_{ik})$ which is set to True once the port is assigned a specific value. The rest of the simulation steps are the same for all blocks. These steps are summarized in Algorithm 1.

6.2.4 Synergies with FMEDA and FTA

The MetaFPA framework is an extended and enhanced adaptation of the Failure Logic Modeling (FLM) theory which is traditionally applied to reduce the gap between safety analysis and fault injection. In the context of this thesis, MetaFPA is used as a model-based and simulation-oriented platform for failure propagation analysis. It offers a wide range of capabilities to evaluate the system safety at a high level of abstraction and early in the design cycle and to generate graphs depicting the propagation paths leading to critical system failures. Furthermore, the utility of MetaFPA to offer improvements within the traditional safety analysis context is investigated. Indeed, MetaFPA is applied to achieve a combination of inductive and deductive failure analysis approaches, more precisely FMEDA and FTA [12].

Within the comprehensive safety evaluation environment represented in Figure 6.9, synergies between inductive and deductive failure analysis are realized by associating corresponding tools and working flows. MetaFPA, which is used in this environment as a bridge between top-down and bottom-up safety analyses, simplifies the data exchange between FMEDA Excel spreadsheets on the one hand and fault trees created and/or edited within Isograph's Reliability Workbench™ (RWB) on the other hand. Dedicated tools are created to support data exchange and automation as follows [12]:

- Capture input/output deviations by inspecting and classifying fault tree events.
- Generate FMEDA table segments out of data models created within MetaFPA.
- Extract information about internal failure behavior as specified in component FMEDA sheets and fill it into MetaFPA's models.
- Synthesize fault trees out of those models to be visualized and potentially refined and maintained in RWB (more details in Subsection 6.3.2.3).

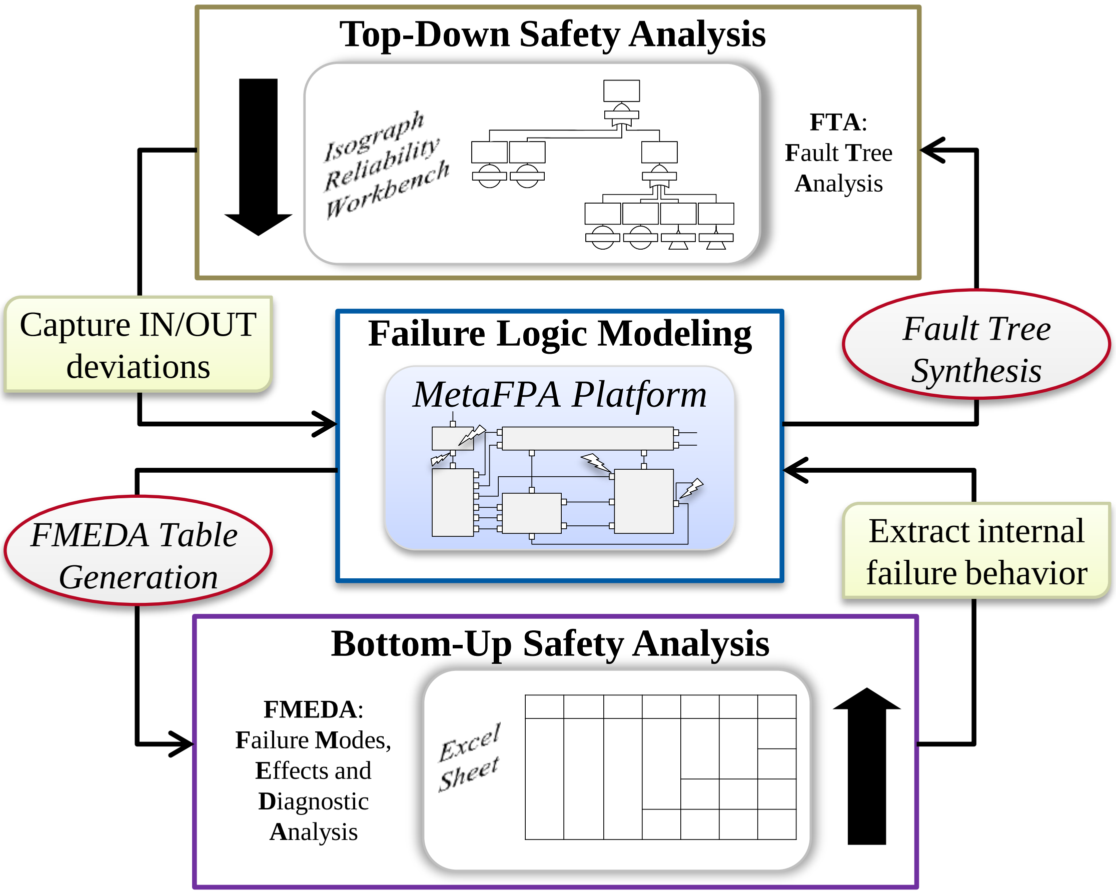

Top-Down Safety Analysis
Isograph Reliability Workbench
FTA: Fault Tree Analysis
Capture IN/OUT deviations
Failure Logic Modeling
MetaFPA Platform
Fault Tree Synthesis
FMEDA Table Generation
Extract internal failure behavior
Bottom-Up Safety Analysis
FMEDA: Failure Modes, Effects and Diagnostic Analysis
Excel Sheet

Figure 6.9: Linking Top-Down & Bottom-Up Safety Analysis through MetaFPA [12]

6.3 Safety Analysis MetaLibs

Section 5.3 introduced formalization approaches for the three major analysis techniques in the context of automotive functional safety in compliance with the ISO 26262 standard, namely FMEDA (Subsection 5.3.1), FTA (Subsection 5.3.2), and DFA (Subsection 5.3.3). By structuring available data formats and templates in formal and concise metamodels, reuse of already existing analysis artefacts and automated support based on model-driven engineering are enabled. In accordance with the developed metamodels for FMEDA, FTA, and DFA, dedicated frameworks, referred to as *MetaLibs*, are created within the Metagen environment (see Subsection 6.1.2) by applying the metasynthesis concept (see Subsection 4.2.3). A MetaLib is basically a package of tools and utilities which are generated or manually implemented around a constructed metamodel (e.g., UML class diagram). For instance, it includes an application programming interface (API), an optional graphical user interface (GUI), a set of data readers and parsers, and an extension/plug-in mechanism to allow more functionalities. In this section, the MetaLibs developed respectively for FMEDA and FTA are described with respect to their general setting, to the specific tools they contain or are linked to, and to their potential relations with other frameworks developed in the context of this thesis such as the MetaFPA platform for failure propagation analysis (see Section 6.2).

6.3.1 FMEDA MetaLib

To overcome the challenges encountered by safety engineers in the traditional FMEDA procedure, a model-based and simulation-assisted FMEDA approach is developed and implemented. The resulting framework is called *FMEDA MetaLib*. In the following, the general setting of the FMEDA MetaLib is presented in Subsection 6.3.1.1. After that, the failure modes database used in the context of the FMEDA automation is described in Subsection 6.3.1.2.

6.3.1.1 General Setting

The overview of the model-based and simulation-assisted FMEDA approach [3] is illustrated in Figure 6.10.

For large systems addressed in industrial contexts, FMEDA is performed recursively. Indeed, for the different system components, a separate analysis is carried out using the flow already presented in Subsection 5.3.1.1. Complex system components may be again decomposed in subcomponents, for which further Component-FMEDAs (abbreviated as C-FMEDAs) are performed. The obtained results are then manually inspected and gathered to conduct an overall analysis for the complete system (System-FMEDA abbreviated as S-FMEDA). The model-driven alternative proposed in this thesis relies on the FMEDA metamodel (Subsection 5.3.1). Through parsing and data extraction, confirming data models are built to substitute huge amounts of tabular data and to simplify exploration, maintenance, and reuse. Here, the term data model refers to a set of associated objects which can be visualized as an object diagram and which can be handled using a common object-oriented language, such as Python or C++. Most importantly, these data models enable a link to the simulation environment and provide a connection between analytical data and executable system models (e.g., SystemC, SystemVerilog, and VHDL).

Another key point of the model-based FMEDA methodology is to derive an S-FMEDA model by automatically assembling a set of related C-FMEDA models. The prerequisites of this automated assembly consist in (i) a system architecture provided

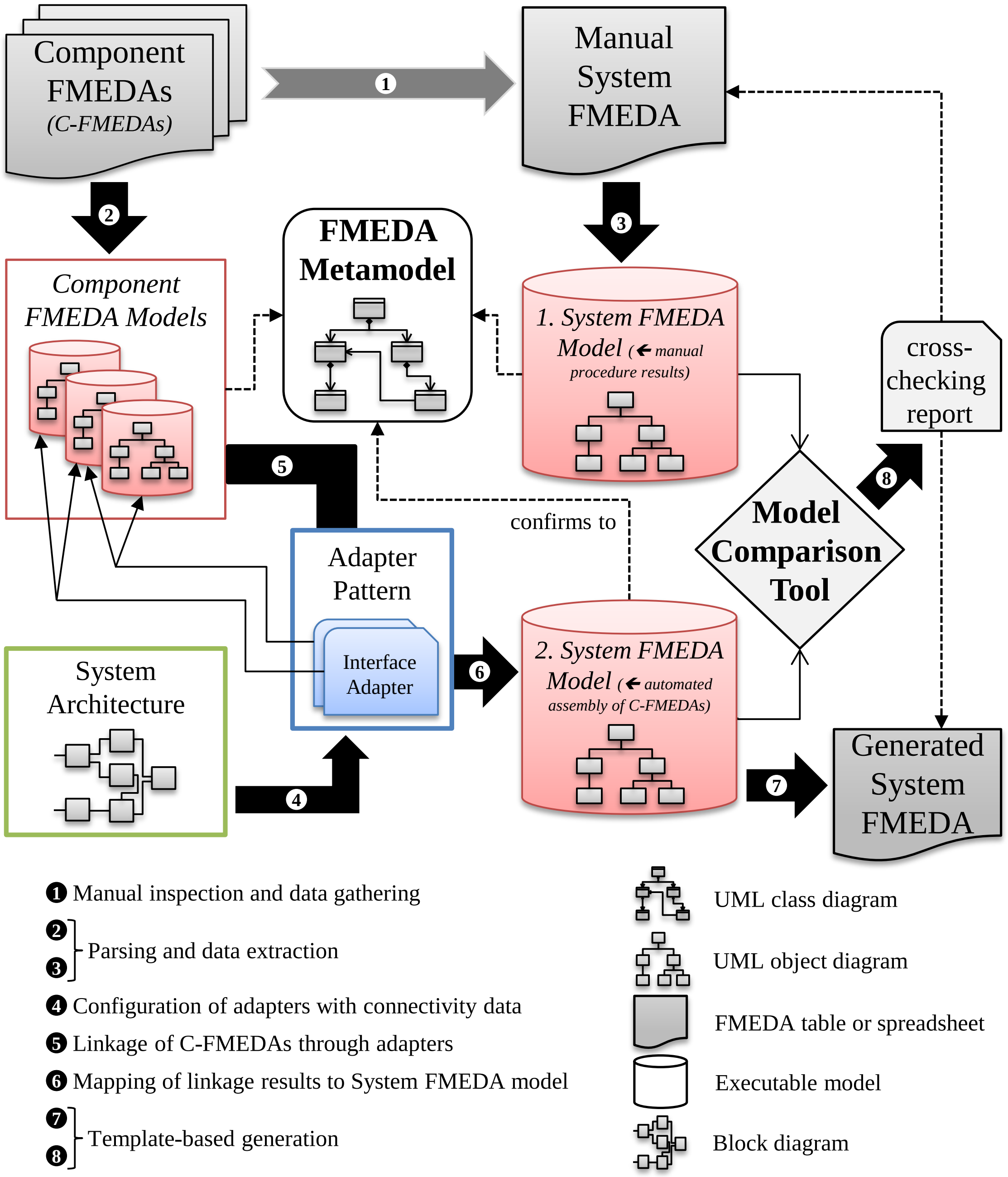

Component FMEDAs (C-FMEDAs)
1
Manual System FMEDA
2
3
Component FMEDA Models
FMEDA Metamodel
1. System FMEDA Model (← manual procedure results)
cross-checking report
5
confirms to
Model Comparison Tool
8
Adapter Pattern
Interface Adapter
6
2. System FMEDA Model (← automated assembly of C-FMEDAs)
System Architecture
4
7
Generated System FMEDA
1 Manual inspection and data gathering
2
3 Parsing and data extraction
4 Configuration of adapters with connectivity data
5 Linkage of C-FMEDAs through adapters
6 Mapping of linkage results to System FMEDA model
7
8 Template-based generation
UML class diagram
UML object diagram
FMEDA table or spreadsheet
Executable model
Block diagram

Figure 6.10: Approach Overview of Model-Based FMEDA [3]

as an architectural block diagram and (ii) a set of *interface adapters* organized in an *adapter pattern* to depict interrelations between components in failure cases. The interface adapter is derived based on the connectivity information between the different blocks as depicted in the system architecture. Linking the C-FMEDA models through the interface adapters leads eventually to the S-FMEDA model corresponding to the safety evaluation of the complete system [3]. As the obtained S-FMEDA model confirms syntactically to the FMEDA Metamodel, a generator can be used to transform it into an S-FMEDA table.

The proposed FMEDA methodology has also a simulation-assisted aspect. In fact, a simulation platform supporting nominal and failure-modified system models, such as the MetaFPA simulation platform (see Subsection 6.2), is considered. By applying model-to-model transformation algorithms on Component-FMEDA data models, compositional failure specifications are created. Nominal system models are accordingly extended or annotated with these compositional failure descriptions. The outcome of the subsequently performed failure propagation simulation is mapped into an S-FMEDA data model which corresponds also to the safety evaluation of the complete system and which can be transformed into an FMEDA table [3].

Furthermore, the methodology supports a verification capability for the manually performed analysis. By comparing the two metamodel instances: (i) the one derived from the results of the manual analysis procedure and (ii) the one generated through automated assembly as described above, a cross-checking report highlighting detected inconsistencies and providing corrective recommendations is produced.

The structure of the model-based framework for FMEDA support and automation (i.e., the FMEDA MetaLib) is illustrated in Figure 6.11. As already mentioned above, the MetaLib is based on the FMEDA metamodel and consists of an Application Programming Interface and a set of tools including data readers and parsers, writers, and other plugins used for the construction, visualization, and modification of FMEDA models.

On the input side of the framework, a failure modes database (see Subsection 6.3.1.2) and a set of block diagrams capturing the hierarchical structure of components or systems are considered. When the target of the analysis is a large system, manually created tables for FMEDA components or subcomponents are also taken into account. In some cases, the purpose of applying the FMEDA MetaLib is to verify the consistency of an FMEDA which has already been conducted using the traditional manual procedure. In this case, the FMEDA table documenting the outcomes of the manual analysis is a further input [3].

To achieve the automated FMEDA assembly, the extracted data from the inputs mentioned above is used to fill Component-FMEDA data models. A set of scripts are then applied to process those C-FMEDA models and correspondingly derive the overall System-FMEDA model.

On the output side, the table generator produces the final model view which is precisely the conventional FMEDA table documenting the overall system analysis.

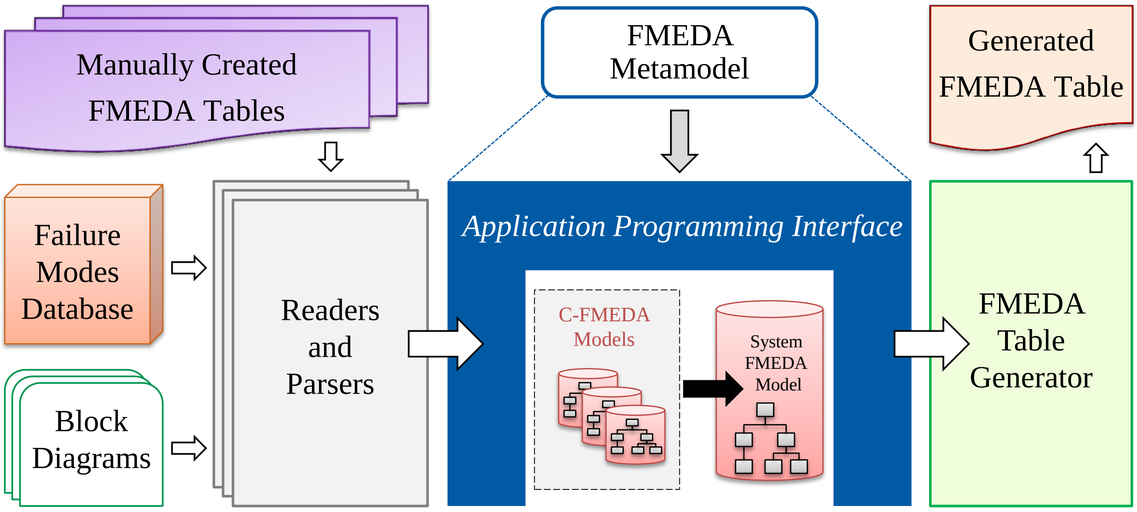


Figure 6.11: Framework for Model-Based FMEDA Automation [3]

6.3.1.2 Failure Modes Database

An essential element of the model-based and simulation-assisted FMEDA framework is the failure modes database containing the information derived from different reliability and safety sources, mainly the internal failure catalogue and the field return data of older projects. It also takes into account the tables D.1 to D.14 in Annex D of Part 5 of the ISO 26262 standard, where common fault models, safety mechanisms, and typical diagnostic coverage values are listed. The structure of the failure modes database is covered by the metamodel illustrated in Figure 6.12.

The failure modes database also includes the base failure rates for the components which are derived for example from the Siemens Norm SN 29500 [121] or from the Exida Reliability Handbook [123].

In compliance with the metamodel illustrated in Figure 6.12 and using the available information sources, a correspondingly filled data model is constructed and used as starting point for the data extraction within the FMEDA MetaLib when it comes to (i) common fault models and/or failure modes for specific component types, (ii) potential candidates of appropriate safety mechanisms for failure mitigation, and (iii) base failure rate values for specific component types, configuration, and implementation technologies. To simplify management, visualization, and data queries within the failure modes database, an SQL (Structured Query Language) interface is correspondingly generated. In Figure 6.13, extracts of the SQL database and their relationships to the classes of the metamodel are illustrated.

In the failure modes database, three directories, corresponding to the three men-

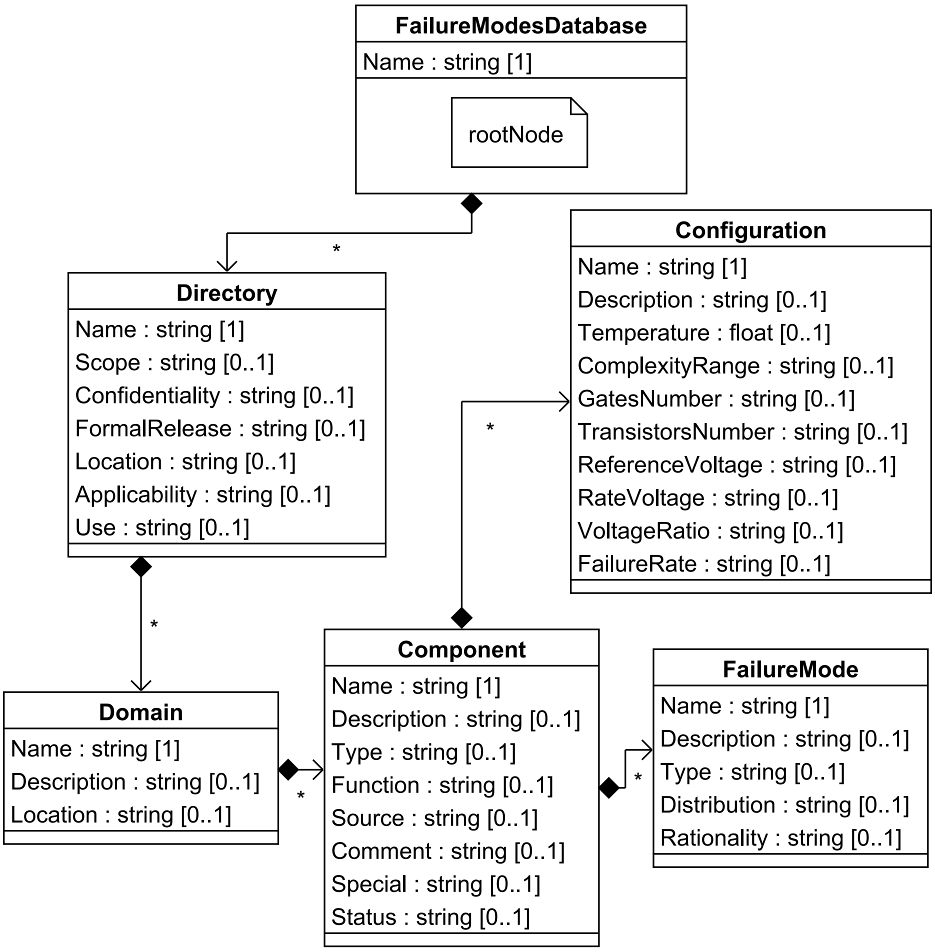
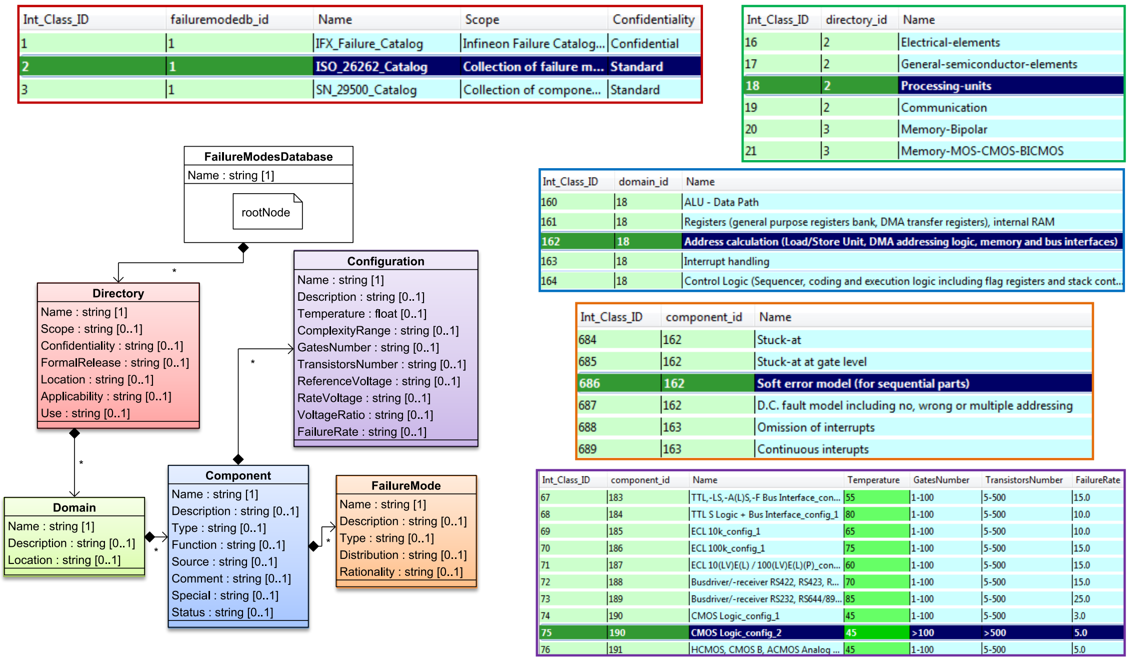

FailureModesDatabase
Name : string [1]
rootNode
Configuration
Name : string [1]
Description : string [0..1]
Temperature : float [0..1]
ComplexityRange : string [0..1]
GatesNumber : string [0..1]
TransistorsNumber : string [0..1]
ReferenceVoltage : string [0..1]
RateVoltage : string [0..1]
VoltageRatio : string [0..1]
FailureRate : string [0..1]
Directory
Name : string [1]
Scope : string [0..1]
Confidentiality : string [0..1]
FormalRelease : string [0..1]
Location : string [0..1]
Applicability : string [0..1]
Use : string [0..1]
*
*
*
*
*
Domain
Name : string [1]
Description : string [0..1]
Location : string [0..1]
Component
Name : string [1]
Description : string [0..1]
Type : string [0..1]
Function : string [0..1]
Source : string [0..1]
Comment : string [0..1]
Special : string [0..1]
Status : string [0..1]
FailureMode
Name : string [1]
Description : string [0..1]
Type : string [0..1]
Distribution : string [0..1]
Rationality : string [0..1]

Figure 6.12: Metamodel-Based Structure of Failure Modes Database

tioned sources (i.e., internal failure catalogue at Infineon, tables of ISO 26262 - Part 5 (Annex D), and the Siemens Norm SN 29500) are included. In each directory, multiple domains are taken into account, for example, the processing units and communication domains in the ISO 26262 standard. A set of components, or more precisely component types, are related to every domain and each component is then subject to a group of failure modes, characterized for example by the distribution and the rationale behind it. The component failure rates, which are mainly derived from the Siemens Norm, depend on the technology and operation parameters. This is covered in the database metamodel through the **Configuration** class. For instance, for a common CMOS logic component, two configurations are given depending on the complexity and showing two different base failure rates (see Figure 6.13).



Int_Class_ID	failuremodedb_id	Name	Scope	Confidentiality
1	1	IFX_Failure_Catalog	Infineon Failure Catalog...	Confidential
2	1	ISO_26262_Catalog	Collection of failure m...	Standard
3	1	SN_29500_Catalog	Collection of compone...	Standard

Int_Class_ID	directory_id	Name
16	2	Electrical-elements
17	2	General-semiconductor-elements
18	2	Processing-units
19	2	Communication
20	3	Memory-Bipolar
21	3	Memory-MOS-CMOS-BICMOS

Int_Class_ID	domain_id	Name
160	18	ALU - Data Path
161	18	Registers (general purpose registers bank, DMA transfer registers), internal RAM
162	18	Address calculation (Load/Store Unit, DMA addressing logic, memory and bus interfaces)
163	18	Interrupt handling
164	18	Control Logic (Sequencer, coding and execution logic including flag registers and stack cont...

Int_Class_ID	component_id	Name
684	162	Stuck-at
685	162	Stuck-at at gate level
686	162	Soft error model (for sequential parts)
687	162	D.C. fault model including no, wrong or multiple addressing
688	163	Omission of interrupts
689	163	Continuous interupts

Int_Class_ID	component_id	Name	Temperature	GatesNumber	TransistorsNumber	FailureRate
67	183	TTL,-LS,-A(L)S,-F Bus Interface_con...	55	1-100	5-500	15.0
68	184	TTL S Logic + Bus Interface_config_1	80	1-100	5-500	10.0
69	185	ECL 10k_config_1	65	1-100	5-500	10.0
70	186	ECL 100k_config_1	75	1-100	5-500	15.0
71	187	ECL 10(LV)E(L) / 100(LV)E(L)(P)_con...	60	1-100	5-500	15.0
72	188	Busdriver/-receiver RS422, RS423, R...	70	1-100	5-500	15.0
73	189	Busdriver/-receiver RS232, RS644/89...	85	1-100	5-500	25.0
74	190	CMOS Logic_config_1	45	1-100	5-500	3.0
75	190	CMOS Logic_config_2	45	>100	>500	5.0
76	191	HCMOS, CMOS B, ACMOS Analog ...	45	1-100	5-500	5.0

FailureModesDatabase
Name : string [1]
rootNode
*
Directory
Name : string [1]
Scope : string [0..1]
Confidentiality : string [0..1]
FormalRelease : string [0..1]
Location : string [0..1]
Applicability : string [0..1]
Use : string [0..1]
Configuration
Name : string [1]
Description : string [0..1]
Temperature : float [0..1]
ComplexityRange : string [0..1]
GatesNumber : string [0..1]
TransistorsNumber : string [0..1]
ReferenceVoltage : string [0..1]
RateVoltage : string [0..1]
VoltageRatio : string [0..1]
FailureRate : string [0..1]
*
*
Domain
Name : string [1]
Description : string [0..1]
Location : string [0..1]
*
Component
Name : string [1]
Description : string [0..1]
Type : string [0..1]
Function : string [0..1]
Source : string [0..1]
Comment : string [0..1]
Special : string [0..1]
Status : string [0..1]
*
FailureMode
Name : string [1]
Description : string [0..1]
Type : string [0..1]
Distribution : string [0..1]
Rationality : string [0..1]

Figure 6.13: SQL Management of Failure Modes Database

6.3.2 FTA MetaLib

In comparison to FMEDA, FTA is considered as a less cumbersome analysis method, especially because of (i) its top-down exploration of the system failure behavior, (ii) its reliance on Boolean logic to capture the interconnections between failures, and (iii) its graphical documentation format (trees). It should however be noted that the results of the FTA are in certain cases insufficient with respect to quantitative safety assessment. Even though FTA can be conducted in a quantitative way to provide numerical results of the overall system failure rate for example, there is no computation support within common FTA tools of the advanced safety metrics prescribed by the ISO 26262 such as the SPFM (Single Point Fault Metric) and LFM (Latent Fault Metrics) like in the FMEDA (see Subsection 5.3.1.1).

FTA is therefore considered to be more manageable than FMEDA. Nevertheless, the significant volume and complexity of fault trees in industrial applications addressing highly complex systems still represent a challenge for safety engineers as well as for quality analysts and managers. That is why, the formalization of FTA has been addressed in the context of this thesis (see Subsection 5.3.2) as well. In accordance to that formalization, a model-based framework is developed to address data handling, visualization, and generation issues within FTA. This framework, namely the FTA MetaLib, is described in Subsection 6.3.2.1 with respect to its general setting. In this thesis, the transition from compositional failure propagation models developed within the MetaFPA framework (see Subsection 6.2.1) to FTA is addressed with the objective of simplifying the creation of fault trees. After presenting the basics of this transition in Subsection 6.3.2.2, the associated fault tree synthesis aspect relying on algorithmic model-to-model transformation is explained in Subsection 6.3.2.3.

6.3.2.1 General Setting

The overview of the FTA MetaLib within its primary usage context in this thesis, namely its association with MetaFPA for the purpose of systematically generating fault trees [12], is illustrated in Figure 6.14.

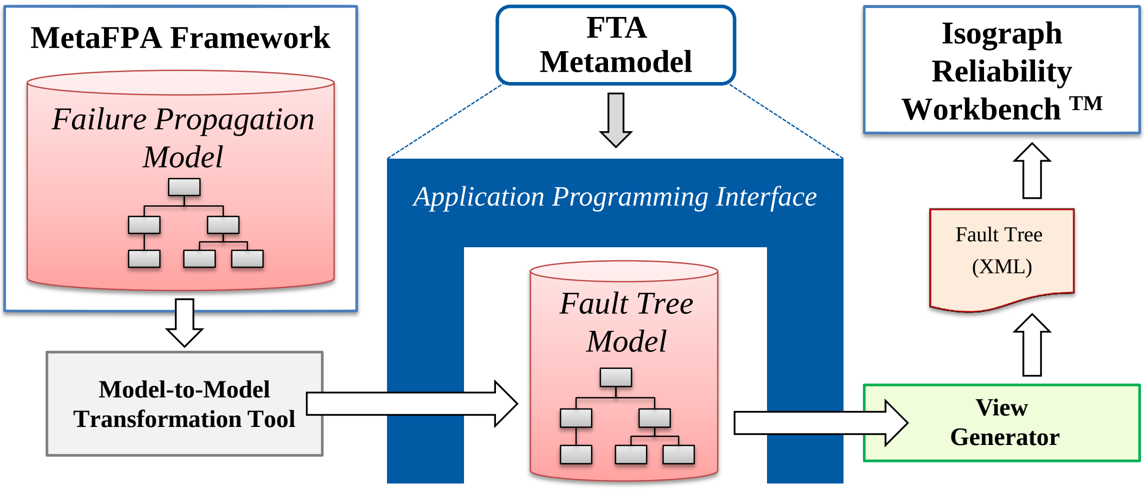

MetaFPA Framework
Failure Propagation Model
FTA Metamodel
Isograph Reliability Workbench ™
Application Programming Interface
Fault Tree (XML)
Model-to-Model Transformation Tool
Fault Tree Model
View Generator

Figure 6.14: Overview of the FTA MetaLib and its Link to the MetaFPA Framework

The development of the FTA MetaLib is based on the FTA metamodel presented in Subsection 5.3.2 which covers the four fundamental features of the fault tree: (i) the modularization aspect denoting that a fault tree can be decomposed in further subtrees, (ii) the failure logic covered through dedicated gate classes (traditional gates as well as FTA specific gates such as VotingOR and PriorityAND), (iii) classified events (top, intermediate, and primary events), and (iv) the quantitative aspect addressed by the probability attributes in the event classes. In compliance to the FTA metamodel, the metasynthesis concept (see Subsection 4.2.3) is applied in the Metagen environment (see Subsection 6.1.2), so that major parts of the MetaLib are automatically generated, such as the Application Programming Interface – API (see Figure 6.14) and a basic Graphical User Interface – GUI (not shown in Figure 6.14). Within the FTA MetaLib, structured and concise substitutes of graphical fault trees are created, visualized, and handled. These substitutes are the fault tree models which are either constructed by user input through the GUI or systematically derived from MetaFPA's failure propagation models through a dedicated model-to-model transformation tool also shown in Figure 6.14. On the output side, a view generator produces XML files depicting fault trees in a format which is compatible with the commercial tool Isograh Reliability Workbench™ (RWB) [135]. The generated XML file is then imported into RWB so that further inspection, refinement, and edition steps can be manually performed by the safety engineering team.

6.3.2.2 Transition from MetaFPA to Fault Trees

In this subsection, the focus is on the model-to-model transformation tool which is shown in Figure 6.14 and which implements the transition between failure propagation models created in MetaFPA and the fault tree models. From a conceptual perspective, the transition from MetaFPA to FTA is illustrated in Figure 6.15 in a simplified way.

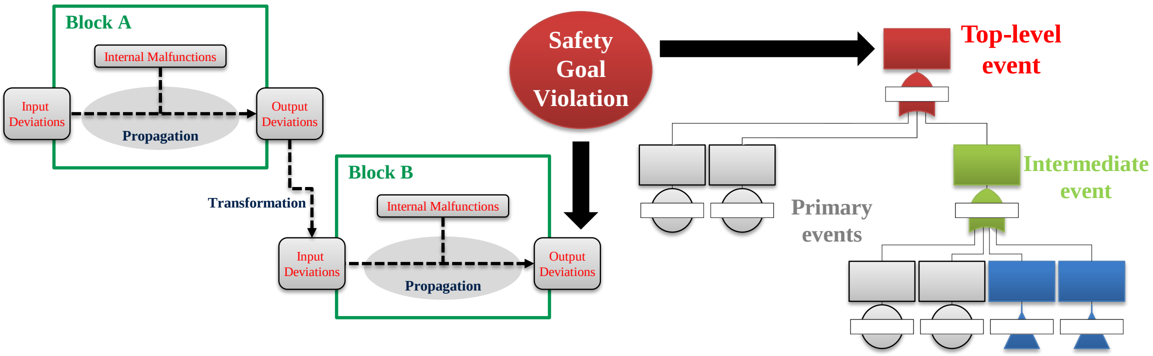


Figure 6.15: Transition from MetaFPA to FTA – Basic Concepts

The starting point is a fully constructed failure propagation model in MetaFPA (see Section 6.2). The target is a fully constructed fault tree model in compliance to the FTA metamodel (see Subsection 5.3.2). As already mentioned, the overall fault tree is in practice a set of sub-trees due to the modularization feature. Each sub-tree is characterized by a unique top-level event which is equivalent in our context to a so-called *safety goal violation*. Safety goals (see Subsection 2.2.4) are actually the top-level safety requirements of the system according to the ISO 26262 standard. They are specified in the safety concept (see Subsection 2.2.5.1). Hence, they represent a prerequisite to perform the safety analysis [12].

In relation to MetaFPA, safety goal violations are mapped to system output deviations, i.e., output failures occurring at the final blocks of the system architecture located at its interface with the surrounding environment (other systems, users, etc.). After this mapping, the fault sub-tree logic is constructed progressively by traversing the failure propagation model created in MetaFPA and by adding intermediate and/or primary events in correlation to input deviations and/or internal malfunctions in the respective blocks [12].

In the simple example depicted in Figure 6.15, internal malfunctions occurring in block B are reflected by the primary events which are appended to the top-level event. Assuming that there is an input deviation of block B, which is actually a transformed output deviation of block A, an intermediate event is added to the fault tree. Consequently, the primary events corresponding to the internal malfunctions of block A are accordingly appended. The collapsed branches (blue triangular symbols) are used in the figure as a generalization of the possible alternatives of how the sub-tree structure may be extended. Indeed, potential further tree events are correlated with the input deviations of block A and the construction depends on whether there are other blocks upfront in the system architecture. It should be noted that for the sake of simplification in Figure 6.15, only OR gates are considered. However, the actual synthesis algorithm takes the exact occurrence conditions of the deviations into account and properly generates the according logic gates in the fault tree.

In the next subsection 6.3.2.3, the algorithm of the fault tree synthesis is detailed.

6.3.2.3 Fault Tree Synthesis

As already stated in Subsection 6.2.3.2, a system S_χ is specified in MetaFPA by a set of blocks and a set of connections: $S_\chi = (\mathcal{B}, \mathcal{C})$. In addition to the sets of input and output ports characterizing each block B_i, the sets $\mathcal{D}_i^I$ of input deviations, $\mathcal{D}_i^O$ of output deviations, and $\mathcal{M}_i$ of internal malfunctions are particularly important for the description of the failure propagation and subsequently for its transformation into the fault tree structure.

Another prerequisite for the transition from MetaFPA to the fault tree analysis is the consideration of safety goal violations. For this, $\Sigma = \{\sigma_l\}$ depicts the set of such violations and is a further characteristic of the system (l is the iterating index ranging between one and L, which is the total number of safety goal violations derived from the safety concept).

The fault tree is denoted as Γ and contains a set of gates γ and a set of events ε. Thus, the system definition is correspondingly extended to $S = (\mathcal{B}, \mathcal{C}, \Sigma, \Gamma)$.

As already mentioned above, the fault tree Γ is a group of several sub-trees. Each of them corresponds to a safety goal violation σ_l in Σ.

Each safety goal violation σ_l is caused by a combination of output deviations across the different blocks located at the system output interface (so-called *exit* blocks in MetaFPA). The set of all these output deviations is denoted as $\mathcal{D}_X^O$. The set of combinations which are taken into account as potential causes of safety goal violations

Algorithm 2 Synthesis Flow of a Fault Sub-Tree

```
function SynthesizeSubtree($S$, $\Gamma$, $\sigma_l$)
    $\Delta_{\sigma_l} \leftarrow \emptyset$
    for each block $B_i \in \mathcal{B}$ do
        for each output deviation $d_{ik}^O \in \mathcal{D}_i^O$ do
            if $d_{ik}^O$ causes $\sigma_l$ then
                $\Delta_{\sigma_l} \leftarrow \Delta_{\sigma_l} \cup \{d_{ik}^O\}$
            end if
        end for
    end for
    if $\Delta_{\sigma_l} \neq \emptyset$ then
        if $|\Delta_{\sigma_l}| > 1$ then
            add new OR gate $\gamma_{\sigma_l}$ to fault tree $\Gamma$
            append $\gamma_{\sigma_l}$ to $\Gamma$'s top-event
        else
            $\gamma_{\sigma_l} \leftarrow$ $\Gamma$'s top event
        end if
        for each $\delta \in \Delta_{\sigma_l}$ do
            add new OR gate $\gamma_\delta$ to $\Gamma$
            append $\gamma_\delta$ to $\gamma_{\sigma_l}$
            for each internal malfunction leading to $\delta$ do
                add new primary event $\varepsilon$ to $\Gamma$
                append $\varepsilon$ to $\gamma_\delta$
            end for
            for each input deviation $\vartheta$ leading to $\delta$ do
                CreateBranch($S$, $\Gamma$, $\vartheta$, $\gamma_\delta$)
            end for
        end for
    end if
    return $\Gamma$
end function
```

is the subset of non-empty combinations within the power-set $\mathcal{P}(\mathcal{D}_X^O)$.

Each output deviation d_{ik}^O for a given block B_i is induced by a valid non-empty combination of input deviations and internal malfunctions of the considered block B_i. For this, the *propagation modeling function* denoted for B_i as $\mathbb{P}_i : \mathcal{P}(\mathcal{D}_i^I) \times \mathcal{P}(\mathcal{M}_i) \rightarrow \mathcal{P}(\mathcal{D}_i^O)$ is taken into account. Through this function, all failure combinations are considered, except those removed from the sets of permissible inputs and outputs by *invalid combination* cross-references captured in MetaFPA's metamodel 5.3.2.2.

For a given safety goal violation σ_l, the synthesis of the corresponding fault sub-tree starts by detecting relevant combinations of output deviations within $\mathcal{P}(\mathcal{D}_X^O)$ and grouping them in a set Δ_{σ_l}. The detection relies on the comparison of specified port values for safety goal violations on the one hand and for output deviations on the other hand [11, 12].

The root node of the sub-tree which is related to σ_l is an *OR* disjunction γ_{σ_l} of the corresponding combinations of output deviations which have been identified in Δ_{σ_l}. Intermediate events are added as inputs of γ_{σ_l} for single deviations causing the safety goal violation (singletons in Δ_{σ_l}). For the remaining combinations in Δ_{σ_l}, *AND* conjunctions are appended to γ_{σ_l}.

The reflection of each output deviation δ that has been added in the sub-tree is either an *OR* or an *AND* gate depending on the propagation modeling function of the corresponding block. The appended inputs are either (i) primary events depicting single internal malfunctions which might cause δ or (ii) further logic segments which are iteratively built through a recursive CreateBranch function. In this function, output deviations of other blocks which are logically connected to a certain input deviation ϑ are identified. Their occurrence logic, which is accordingly derived from the MetaFPA failure propagation model, is reflected into the fault tree.

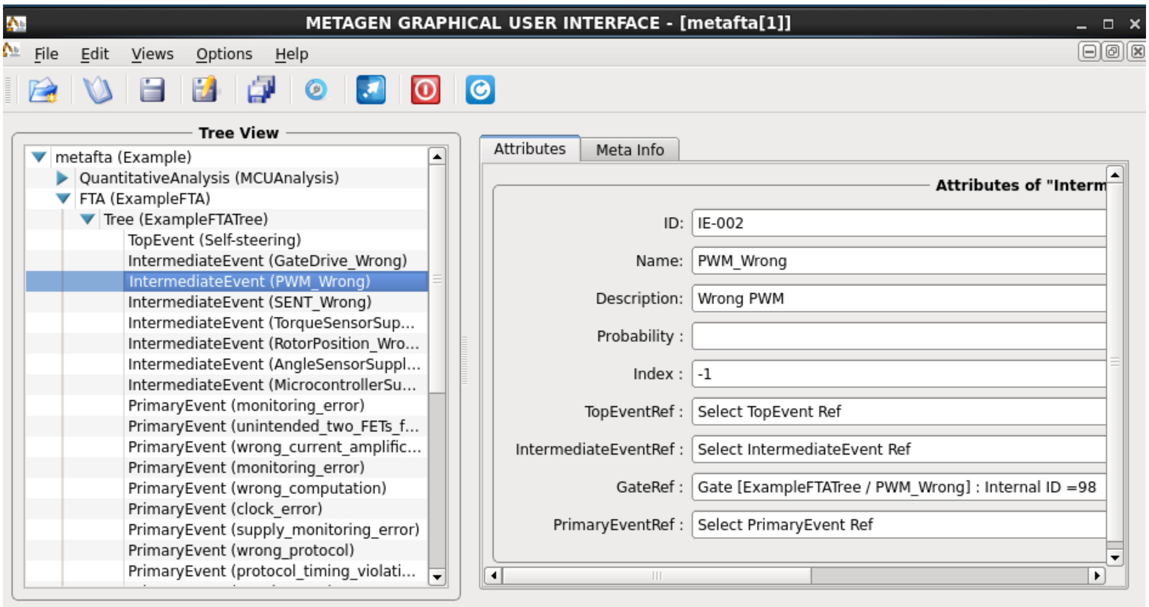
Algorithm 2 shows a simplified version of the general flow for sub-tree synthesis to make it clearer and more understandable. In this simplified version, the following two assumptions are made:

- *Assumption A*: Each safety goal violation σ_l is caused by single output deviations of components at the system output interface.
- *Assumption B*: Each output deviation d_{ik}^O for a given block B_i is induced either by exactly one input deviation or exactly one internal malfunction.

Assumptions A and *B* are irrelevant for the full algorithm which is actually implemented. In fact, safety goal violations which are caused by combinations of output deviations are considered along with those caused by single output deviations. Furthermore, the synthesis algorithm takes into account all possible combinations causing a given output deviation according to the propagation modeling function.

All functions for the fault tree synthesis are implemented in Python and are gathered in the model-to-model transformation tool illustrated in Figure 6.14. The synthesized fault tree model can be visualized and further on refined and/or extended within the FTA MetaLib (see Figure 6.14) through the GUI which is illustrated by the snapshot in Figure 6.16.

Applying the view generator on the fault tree model creates an XML file (see extract example in Figure 6.17) which can be later on imported in the Reliability Workbench™ (RWB) tool thanks to its compliance to the XML schema supported by RWB. There, the safety engineers obtain the traditional graphical tree they are used to, such that

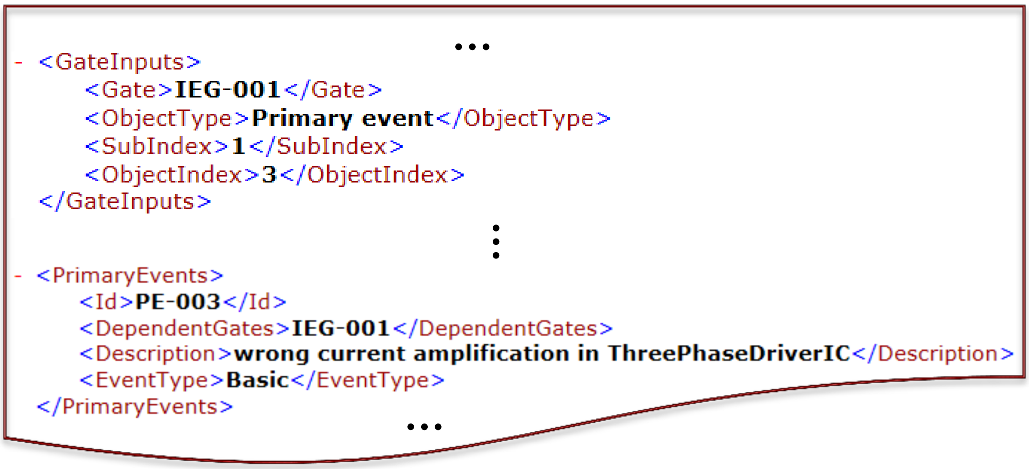


- metafta (Example)
  - QuantitativeAnalysis (MCUAnalysis)
  - FTA (ExampleFTA)
    - Tree (ExampleFTATree)
      - TopEvent (Self-steering)
      - IntermediateEvent (GateDrive_Wrong)
      - IntermediateEvent (PWM_Wrong)
      - IntermediateEvent (SENT_Wrong)
      - IntermediateEvent (TorqueSensorSup...
      - IntermediateEvent (RotorPosition_Wro...
      - IntermediateEvent (AngleSensorSuppl...
      - IntermediateEvent (MicrocontrollerSu...
      - PrimaryEvent (monitoring_error)
      - PrimaryEvent (unintended_two_FETs_f...
      - PrimaryEvent (wrong_current_amplific...
      - PrimaryEvent (monitoring_error)
      - PrimaryEvent (wrong_computation)
      - PrimaryEvent (clock_error)
      - PrimaryEvent (supply_monitoring_error)
      - PrimaryEvent (wrong_protocol)
      - PrimaryEvent (protocol_timing_violati...

Attributes of "Interm

ID: IE-002
Name: PWM_Wrong
Description: Wrong PWM
Probability :
Index : -1
TopEventRef : Select TopEvent Ref
IntermediateEventRef : Select IntermediateEvent Ref
GateRef : Gate [ExampleFTATree / PWM_Wrong] : Internal ID =98
PrimaryEventRef : Select PrimaryEvent Ref

Figure 6.16: Snapshot of the Graphical User Interface in the FTA MetaLib



```
                              ...
- <GateInputs>
      <Gate>IEG-001</Gate>
      <ObjectType>Primary event</ObjectType>
      <SubIndex>1</SubIndex>
      <ObjectIndex>3</ObjectIndex>
  </GateInputs>
                              ⋮

- <PrimaryEvents>
      <Id>PE-003</Id>
      <DependentGates>IEG-001</DependentGates>
      <Description>wrong current amplification in ThreePhaseDriverIC</Description>
      <EventType>Basic</EventType>
  </PrimaryEvents>
                           ...
```

Figure 6.17: Simplified Extract of Generated XML File in Compliance with the XML Schema of Isograph's Reliability Workbench

they can review the generation outcome and potentially refine or extend it. The major benefit of applying the FTA MetaLib is that the safety engineering team does not start from scratch when it comes to FTA. It is no longer necessary to construct the whole graphical fault tree manually because most of it is systematically generated. The prerequisite however is to provide the according system architecture along with its abstracted failure behavior in accordance to the MetaFPA methodology.

6.4 SaVer: Safety Verification Framework

SaVer is a heterogeneous safety verification framework developed at Infineon in the context of the EffektiV project [136] to establish a link between safety analysis on the one hand and fault injection and simulation on the other hand [137, 14, 138]. As already mentioned in 6.1.1, one of the major objectives of the model-driven support of functional safety evaluation is to use data equivalences and/or correspondences between the supported safety analyses (FMEDA, FTA, and DFA) and the supported fault injection and simulation techniques, e.g., simulation-based system alteration at virtual prototype level, to correspondingly enable data mapping and/or transformation. The importance of linking analysis and simulation in the functional safety evaluation context is summarized in two major aspects. First, by conveniently using safety analysis outcomes as guidance for fault injection and simulation, the efficiency is enhanced by reducing the fault space and the related injection scenarios. Second, the results of the fault injection campaigns and the associated simulation runs provide accurate information about the safety integrity level of the system, e.g., percentage of safe faults and diagnostic coverage values, which can be used to verify the consistency of safety analysis outcomes.

The development of the SaVer framework starts by investigating correspondences and/or equivalences between analysis and simulation with respect to the involved data artefacts. For this, the elements used to perform safety evaluation from the two perspectives are compiled, examined, and compared. On the one hand, during safety analysis, system parts, failure modes that may occur and the failure effects they may lead to, potential undesired events, and safety measures are among the most relevant data elements. On the other hand, fault injection and simulation consider sensitivity zones, related injection points, observation points, diagnostic points, etc.

The investigation shows that independently from the evaluation perspective and from the underlying methodology, data elements can be allocated to on the following three *Safety Evaluation Basic Element Groups*:

- **Targets**: System elements or functions that must be protected to ensure the safe operation of the system.
- **Threats**: Malfunctions, discrepancies, and failures against which the targets must be protected (these are basically all risks that may affect the targets)
- **Counter-measures**: Mechanisms and measures used to protect the targets and mitigate the threats (actions that must be undertaken as response to the threats).

In Figure 6.18, the basic element groups of safety evaluation are illustrated. Exemplary allocations of common elements are also shown in the figure. For instance,

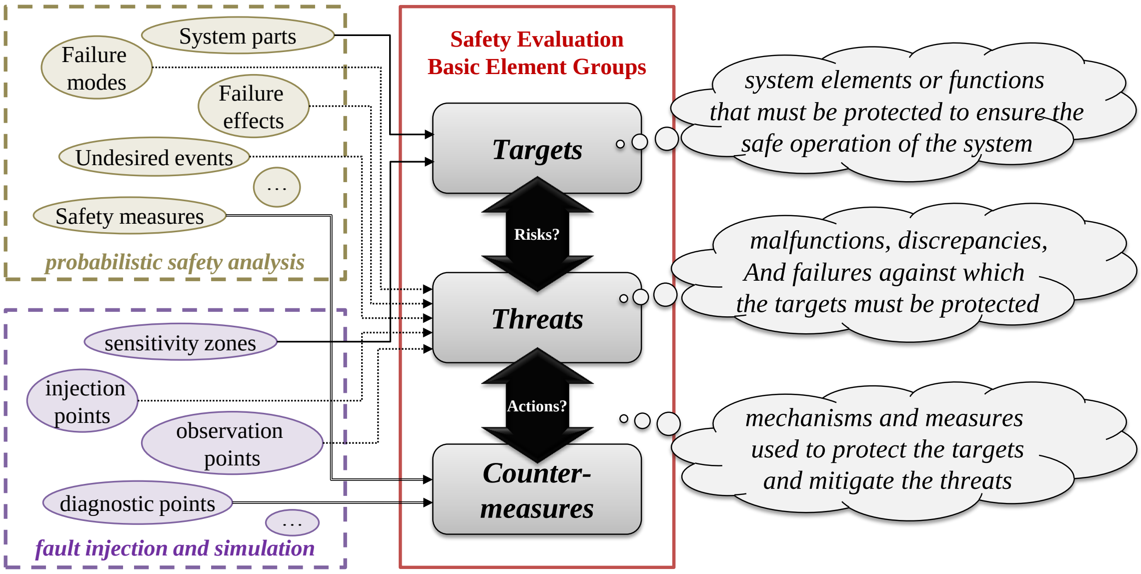

System parts
Failure modes
Failure effects
Undesired events
…
Safety measures
probabilistic safety analysis
sensitivity zones
injection points
observation points
diagnostic points
…
fault injection and simulation
Safety Evaluation Basic Element Groups
Targets
Risks?
Threats
Actions?
Counter-measures
system elements or functions that must be protected to ensure the safe operation of the system
malfunctions, discrepancies, And failures against which the targets must be protected
mechanisms and measures used to protect the targets and mitigate the threats

Figure 6.18: Safety Evaluation Basic Element Groups

system parts and sensitivity zones are potential targets. Furthermore, threats include failure modes, failure effects, and undesired events on the analysis side and they are related to injection points and observation points on the fault injection and simulation side. It should be noted here that terms like “fault”, “error”, “failure” could have been used as elements in the fault injection domain and correspondingly classified as threats. However, it is more convenient to consider the concrete elements in the executable system where faults, errors, and failures are implicitly embodied: for example (i) a signal where a fault might be inserted is an *injection point* and (ii) an output port where the resulting error (or failure in the case of a system output) can be monitored is an *observation point*. Finally, safety measures are counter-measures on the analysis level. Corresponding *diagnostic points*, such as signals for the detection of value discrepancies or extra blocks for error correction, are the respective reflection of counter-measures in the fault injection and simulation context.

6.4.1 General Setting

The general overview of the SaVer framework is illustrated in Figure 6.19 [14]. The key added value of the SaVer framework is the established link between the safety analysis platform and the fault simulation platform. In fact, the FMEDA and FTA MetaLibs described in Subsection 6.3 are integrated along with the DFA MetaLib (see Subsection 5.3.3.2) into the comprehensive SaVer framework. Thereby, *data models* are built as instances of the corresponding *metamodels* to substitute the traditionally used analysis formats (FMEDA spreadsheets, FTA trees, and DFA tables). Such data models enable the transition between the analysis context on the one hand and the simulation context on the other hand.

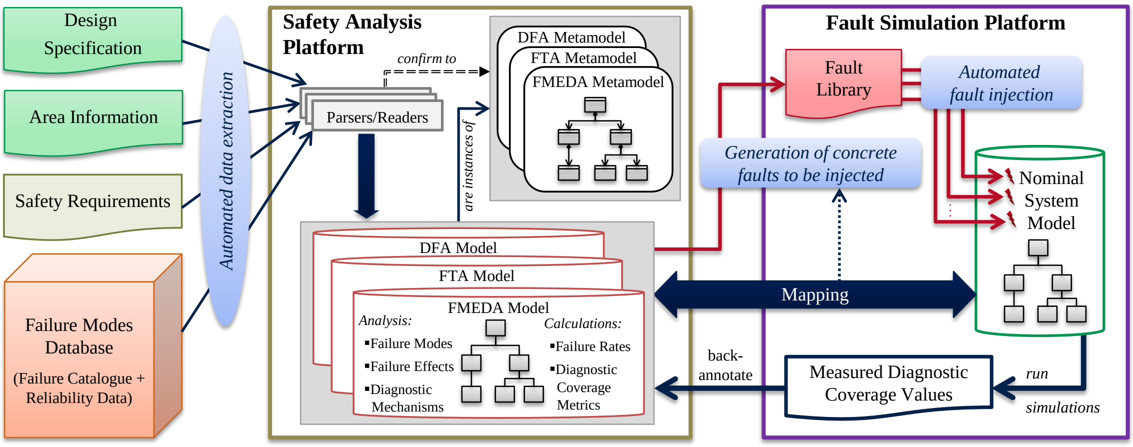


Figure 6.19: Overview of the Safety Verification Framework SaVer [14]

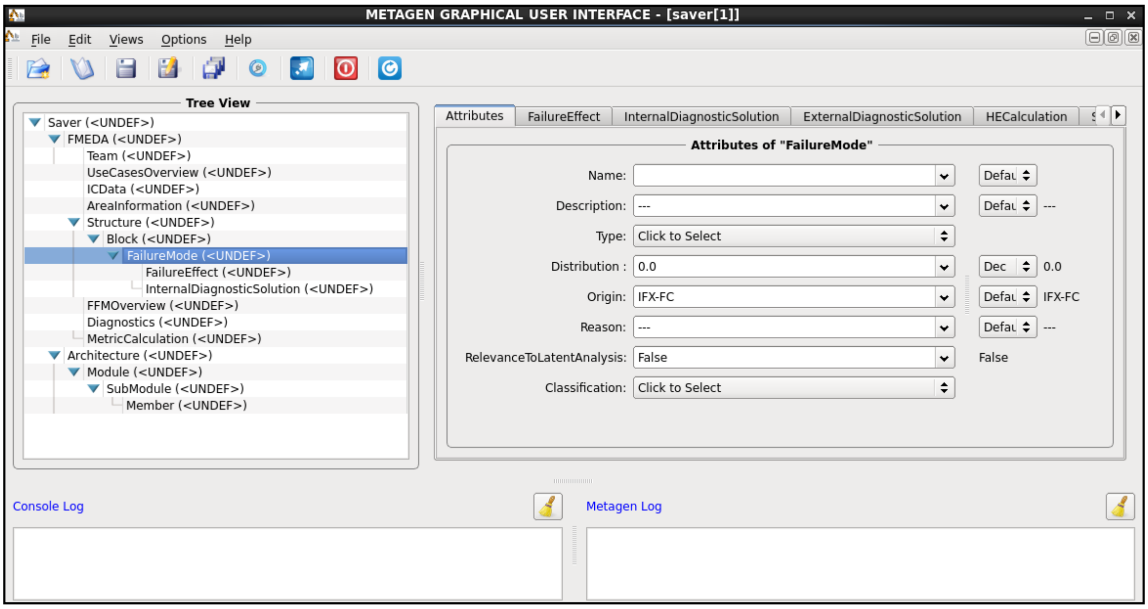
It should be noted that the SaVer framework is developed as an alternative to the traditional environment used in the semiconductor industry to perform functional safety evaluation. Therefore, the conventional inputs of the safety analysis remain unchanged in comparison to the traditional flow. These inputs consist mainly in the design specification, area information, application-dependent safety requirements, and failure modes database combining the information of the failure catalogue and other reliability data derived from different standards, norms, and handbooks. The main difference to the traditional flow is that the traditionally applied manual data entry is replaced by automated data extraction using the parsers and readers contained in the respective safety analysis MetaLibs (see Section 6.3).

The resulting safety analysis data models, which are subsequently constructed in syntactical and semantical accordance with the FMEDA, FTA, and DFA metamodels, cover the different analysis artefacts, such as the failure modes and effects, the dependent failures, the hazardous events, the failure rates, and diagnostic coverage metrics. Furthermore, they ensure the link to the simulation context. Indeed, model-to-model mapping mechanisms are applied to associate specific elements of the safety analysis data models to the corresponding system elements contained in the nominal model to be simulated. In Subsection 6.4.3, the mapping procedure is detailed. The resulting outcome enables the generation of a *fault library*, which is basically a list of all the faults to be inserted into the system model [14].

Another central capability of the SaVer framework is the comparison between fault simulation results and safety analysis outcomes. This comparison, illustrated in Figure 6.19 through the feedback loop labeled *"back-annotate"*, allows a consistency check of the expert judgment on which the analysis is based. For instance, a special plugin implements the back-annotation of the measured diagnostic coverage values throughout the simulation runs into the FMEDA data model. If a mismatch is detected, then the assumptions made during the analysis shall be reviewed and potentially corrected. Appropriate refinements in the system architecture and/or in the associated safety concept must be undertaken when the target safety integrity level is not reached according to the simulation measurements [14].

In summary, the SaVer framework combines the safety analysis MetaLibs developed in the context of this thesis to support model-driven support of FMEDA, FTA, and DFA (Section 6.3 and Subsection 5.3.3.2) with the fault simulation platform developed in the context of the EffektiV project [139, 140, 141, 142]. It should be noted that the FMEDA, FTA, and DFA MetaLibs can also be used standalone as already shown in Section 6.3 and in Subsection 5.3.3.2). However, their integration into the SaVer framework makes them more beneficial in the overall safety evaluation flow with respect to the (i) seamless data mapping of safety data models to executable system models, (ii) the increased efficiency of fault injection and simulation through the analysis-directed generation of the fault library, and (iii) the consistency checking of the analysis results through the feedback loop from the simulation.

The SaVer framework is implemented in Metagen (see Subsection 6.1.2) and contains subsequently a Qt GUI supporting basic data entry and visualization features (see Figure 6.20).


METAGEN GRAPHICAL USER INTERFACE - [saver[1]]
File
Edit
Views
Options
Help
Tree View
Attributes
FailureEffect
InternalDiagnosticSolution
ExternalDiagnosticSolution
HECalculation
Console Log
Metagen Log

Saver (<UNDEF>)
FMEDA (<UNDEF>)
Team (<UNDEF>)
UseCasesOverview (<UNDEF>)
ICData (<UNDEF>)
AreaInformation (<UNDEF>)
Structure (<UNDEF>)
Block (<UNDEF>)
FailureMode (<UNDEF>)
FailureEffect (<UNDEF>)
InternalDiagnosticSolution (<UNDEF>)
FFMOverview (<UNDEF>)
Diagnostics (<UNDEF>)
MetricCalculation (<UNDEF>)
Architecture (<UNDEF>)
Module (<UNDEF>)
SubModule (<UNDEF>)
Member (<UNDEF>)

Attributes of "FailureMode"

Name:		Defau	
Description:	---	Defau	---
Type:	Click to Select		
Distribution :	0.0	Dec	0.0
Origin:	IFX-FC	Defau	IFX-FC
Reason:	---	Defau	---
RelevanceToLatentAnalysis:	False		False
Classification:	Click to Select		

Figure 6.20: Basic Qt Graphical User Interface for the SaVer Framework

Nevertheless, the application of the SaVer framework in a productive context necessitates an enhanced user interface. Therefore, an extensive graphical user interface is developed within the Eclipse Modeling Framework (EMF) to offer a user-friendly cockpit for heterogeneous safety evaluation processes combining analyses and simulations. The technical details and features of the SaVer GUI are given in the next Subsection 6.4.2.

6.4.2 SaVer GUI

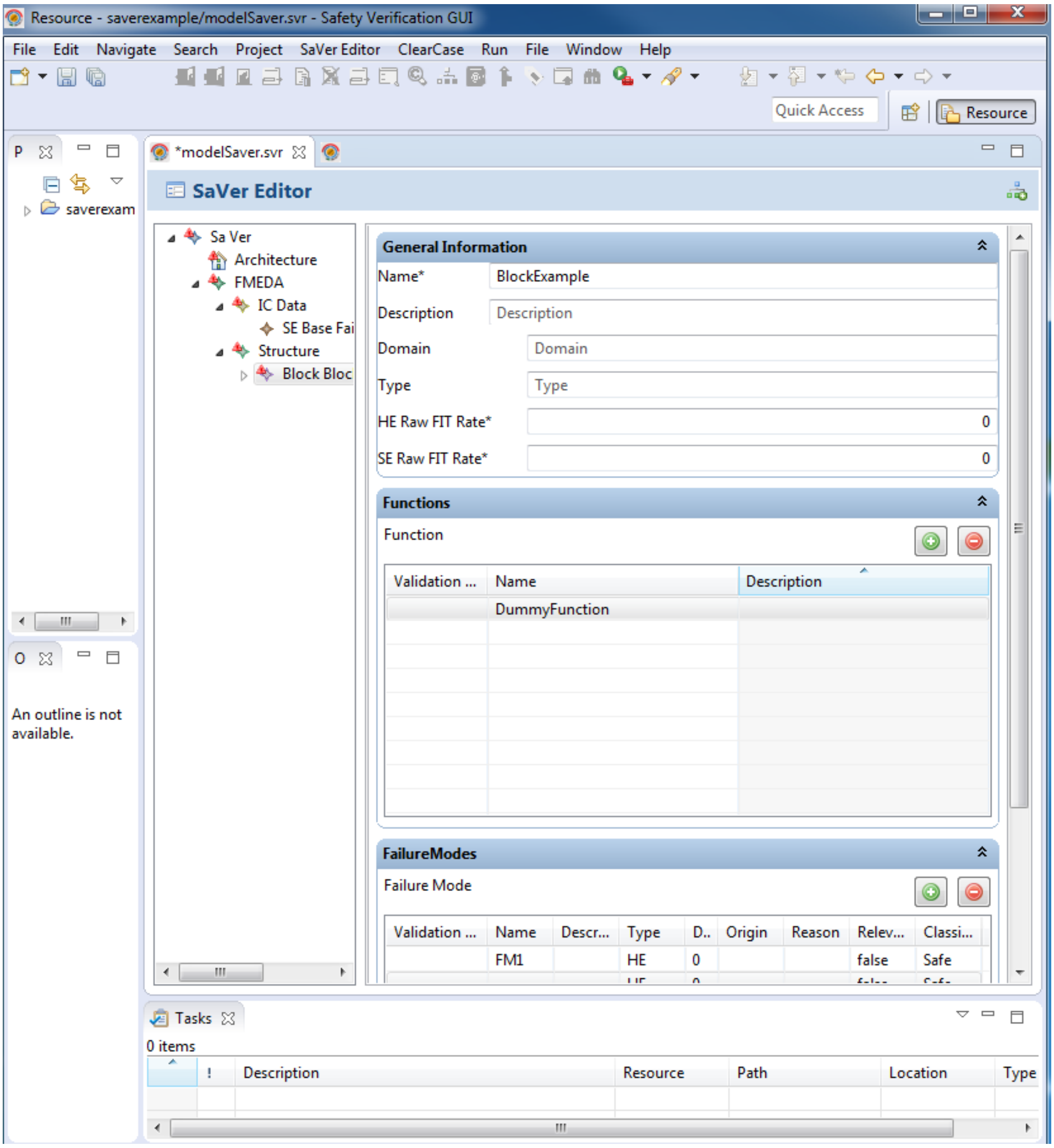
Eclipse is the most widely used Java IDE (Integrated Development Environment) in computer programming. The basic Eclipse workspace is enhanced through an advanced plug-in system which can be extended to customize the development environment and subsequently enable the creation of diversified applications.

Among the popular Eclipse development platforms, EMF is a modeling framework with code generation facilities. Complying with the model-driven development concepts introduced in Section 4.2.1, EMF enables developers to build tools and applications with reduced efforts by considering structured data models [143].

In this work, EMF and particularly EMF Forms (an Eclipse facility to develop form/layout-based user interfaces [144]), are applied to create a graphical user interface for the safety verification framework SaVer. In Figure 6.21, a snapshot of the Eclipse SaVer GUI is shown.

The most important features supported by the Eclipse SaVer GUI are the following:

- *Undo/redo, copy/paste, and search features*: The user is able to modify and explore data within the SaVer framework in a more sophisticated way than with the basic Qt GUI generated by Metagen.
- *Dynamic data responsiveness*: Changing a value leads to the update of all dependent elements in the data model. In other words, the user interface is able to invoke plug-ins and tools running on the background to immediately perform real-time computations when required and accordingly update the data model.
- *Plausibility checks*: If the user input violates data consistency, an error message is issued.
- *Customized formatting*: Using EMF forms, the layout of the GUI is refined and enhanced based on the requests of the safety engineering team.
- *Import and export buttons*: The user interface features dedicated buttons to invoke the responsible tools for data import and/or export to other components of the safety evaluation environment such as Excel Spreadsheets for FMEDA and Isograph's Reliability Workbench™ for FTA.



Name* BlockExample

Description Description

Domain Domain

Type Type

HE Raw FIT Rate* 0

SE Raw FIT Rate* 0

Function

Validation ...	Name	Description
	DummyFunction	

Failure Mode

Validation ...	Name	Descr...	Type	D..	Origin	Reason	Relev...	Classi...
	FM1		HE	0			false	Safe

	!	Description	Resource	Path	Location	Type

Figure 6.21: Snapshot of the Eclipse Graphical User Interface for the SaVer Framework

6.4.3 Mapping between Safety Analysis and Fault Injection

The integration of the FMEDA, FTA, and DFA MetaLibs in the comprehensive SaVer framework is the first step in the transition from safety analysis to fault injection. The accordingly constructed safety analysis data models as a substitution for huge amounts of tabularly and/or graphically organized data do not only simplify exploration, maintenance, and reuse. They are also the starting point of the second step of the transition, namely the mapping towards fault injection and simulation [14].

On the one hand, the safety analysis data models considered in the SaVer framework consist in sets of associated objects which are handled using a common object-oriented programming language such as Python or C++. On the other hand, the nominal system models, which are used as targets for fault injection, are executable models that are generally developed in SystemC, SystemVerilog, VHDL, etc. They can subsequently be considered also as sets of associated objects. Therefore, the transition from the analysis context to the simulation context is established through the application of model-driven techniques and systematic matching algorithms to identify appropriate mapping points [14]. The details of the mapping flow are given in the next subsection 6.4.3.1.

6.4.3.1 Mapping Flow

The data mapping procedure which links safety analysis to fault injection and simulation in the context of the SaVer framework is based on the following four major tasks [14]:

1. ***Capture system model***: Fault injection and simulation are performed on nominal system models with variable levels of complexity and abstraction. Indeed, the system architecture can be considered either at the concept or at the implementation level. TLM (Transaction Level Modeling) and RTL (Register Transfer Level) are among the considered abstraction layers for fault injection and simulation in the EffektiV project [136]. SystemVerilog, VHDL, and SystemC are the most relevant modeling languages in the semiconductor industry, so that they are taken into account for data mapping in the SaVer framework. In the context of Infineon's activities within the EffektiV project, SystemC is used as the default modeling language for virtual prototypes. The motivation for applying SystemC is that it has been extensively addressed in the literature for fault injection, error simulation, and dependability evaluation purposes (e.g., [145, 146, 141, 147]).

   Despite the significant diversity in abstraction layers and modeling languages, the basic system structure remains the same. In fact, the overall system structure can be depicted in a simplified way as a set of interconnected elements forming an architectural hierarchy, independently from all syntactical and semantic differences caused by detail levels and modeling language specificities. Relying on these perceptions, an abstract metamodel is created to depict the general system architecture. The simplified version shown in Figure 6.22, focuses on virtual prototyping using SystemC. Therefore, the architecture is decomposed in a set of

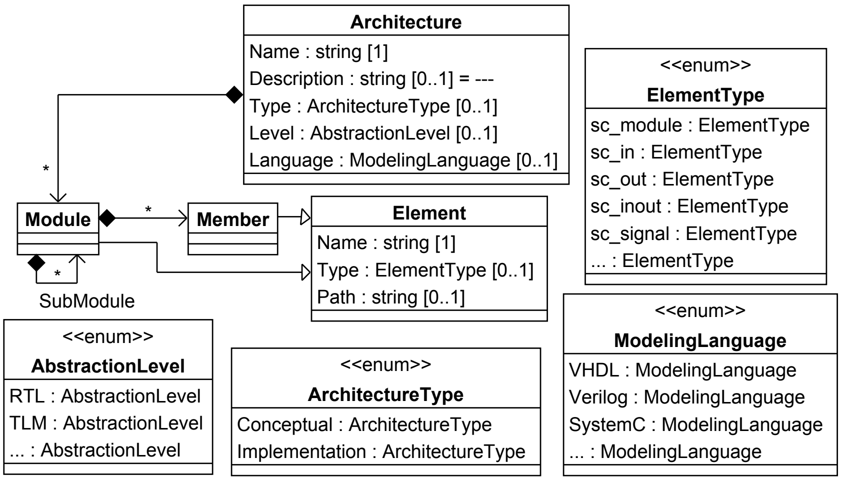


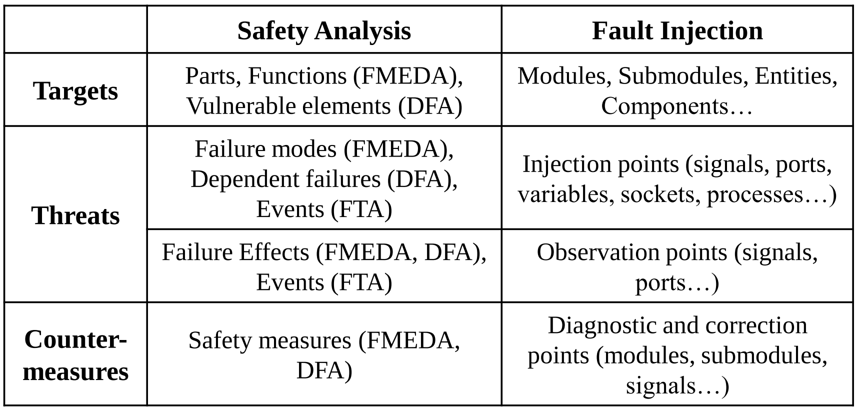
Figure 6.22: Simplified Metamodel of System Architecture

Modules that can be further on decomposed in **SubModules**. Each (sub)module has a number of **Members** (e.g., input and output ports, signals). Capturing the system model is performed by parsers which read the system code and extract the required information to build a data model which syntactically confirms to the architecture metamodel in Figure 6.22.

2. ***Map analysis artefacts to the system model***: During this step, all artefacts identified in the context of the safety analysis procedures (e.g., safety-relevant system parts, failure modes and effects, undesired events, safety measures) are mapped to the appropriate elements of the executable system model. The association between objects of FMEDA/FTA/DFA data models on the one hand and objects of system architecture models created as instances of the metamodel in Figure 6.22) on the other hand is ensured through dedicated *references.* Table 6.1 gives an overview of the correspondences between safety analysis artefacts and system model elements in accordance with the *Safety Evaluation Basic Element Groups* introduced in Subsection 6.4.1 and illustrated in Figure 6.18.

   So-called *matching algorithms* are used to technically enable the data mapping. The application of these algorithms leads to the identification of all potential *mapping candidates*, i.e., system model elements that may be assigned to a given safety analysis artefact. The matching algorithms are addressed in more details in Subsection 6.4.3.2.

3. ***Configure fault injection***: After the accomplishment of the mapping between analysis artefacts and system model elements, additional details required for the fault injection are captured. The information that can be derived from the safety analysis for the purpose of the fault injection consists only in (i) locations for



	Safety Analysis	Fault Injection
Targets	Parts, Functions (FMEDA), Vulnerable elements (DFA)	Modules, Submodules, Entities, Components…
Threats	Failure modes (FMEDA), Dependent failures (DFA), Events (FTA)	Injection points (signals, ports, variables, sockets, processes…)
	Failure Effects (FMEDA, DFA), Events (FTA)	Observation points (signals, ports…)
Counter-measures	Safety measures (FMEDA, DFA)	Diagnostic and correction points (modules, submodules, signals…)

Table 6.1: Mapping Between Safety Analysis Artefacts and Relevant System Elements for Fault Injection and Simulation

fault insertion, (ii) locations for fault-effect observation, and (iii) locations for safety-measure monitoring during simulation. Nevertheless, inserting faults into the executable model in a simulation-based way necessitates more input. Therefore, this configuration step allows to specify exact values to be forced into the system model, precise bit positions in signals to be modified, and accurate start and stop times of the injection. The configuration can be given by the user through the SaVer GUI or derived from an operational profile of the system, i.e., from a quantitative characterization of the way a system is used [148].

4. ***Generate fault library***: The transition from safety analysis to fault injection and simulation is finalized by the generation of the fault library. In this last step, the concrete faults to be inserted into the system are compiled into a single file. Template-based generation techniques are applied on top of the safety analysis data model which has been accordingly extended with all information gathered during the mapping and configuration steps. Within the fault simulation platform included in the SaVer framework, a post-processing tool takes the generated textual file as inputs and subsequently invokes the responsible tools which initiate the automated fault injection.

6.4.3.2 Matching Algorithms

In the context of the SaVer framework, matching algorithms are applied to systematically identify relevant mapping candidates. The candidates in question are elements of the considered executable system models for fault injection. Their equivalence or at least correspondence to specific elements of the safety analysis data models make them particularly suitable to perform fault injection and simulation.

Two alternatives are considered in SaVer to ensure data matching (see Figure 6.23):

- **ID-based matching**: When the consistency between the structure used for safety analysis and the one used for system modeling and simulation is guaranteed, then ID-based matching can be used for the identification of the mapping candidates. The consistency between safety analysis models and corresponding system models is given, only when IDs of safety artefacts are available prior to the system modeling. In this case, the IDs which are assigned to system elements shall be derived from the IDs of the associated safety artefacts. For example, when the designer gets the information from the safety concept that an ECC (error code correction) safety mechanism with ID "`SM_ECC_Part_a`" shall be implemented in "`Part_a`", he or she shall assign an ID containing "`SM_ECC_Part_a`" to the related diagnostic signals in the system implementation. Such full consistency based on IDs would be the ideal scenario for the data mapping. However, in practice, lack of communication between different development groups, delays in deliveries, and different identification styles for objects and artefacts make such ID-based matching capabilities rather limited. That is why, a more flexible matching approach is used.

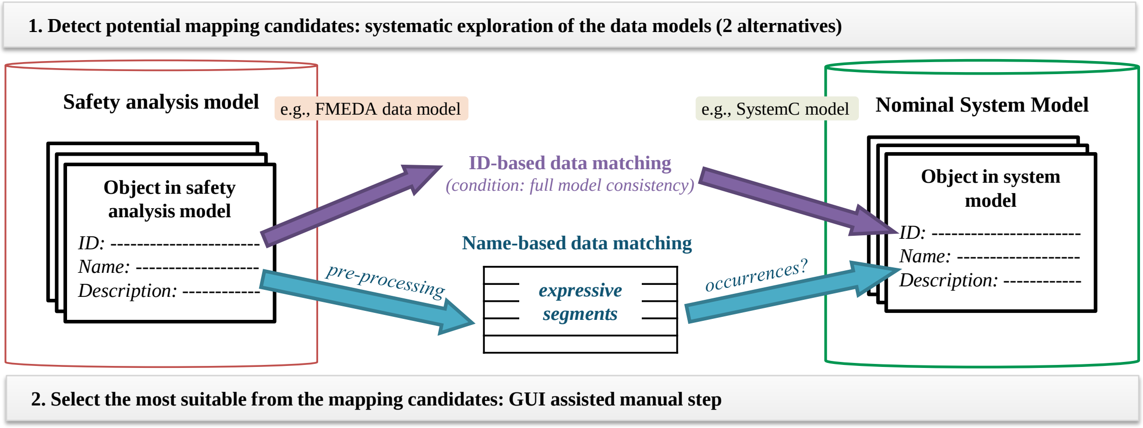

1. Detect potential mapping candidates: systematic exploration of the data models (2 alternatives)
Safety analysis model
e.g., FMEDA data model
e.g., SystemC model
Nominal System Model
Object in safety analysis model
ID: ----------------------
Name: ------------------
Description: -----------
ID-based data matching
(condition: full model consistency)
Name-based data matching
pre-processing
expressive segments
occurrences?
Object in system model
ID: ----------------------
Name: ------------------
Description: -----------
2. Select the most suitable from the mapping candidates: GUI assisted manual step

Figure 6.23: General Mapping Approach

- **Name-based matching**: If no ID consistency is ensured, specific attributes of the objects contained in safety analysis data models (e.g., *Name*, *Description*) are preprocessed to identify so-called *"expressive segments"* before launching the matching algorithm. Expressive segments are all parts of the considered string attributes which are not articles, prepositions, or commonly used words in the naming of failure modes such as *wrong*, *corrupt*, *incorrect*, etc. This list of expressive segments represents then the basis for data matching. In fact, the list items are queried in the considered system model. More precisely, if one or more expressive segments are found in the name or description attribute of a specific object in the system model, then it gets marked as a potential candidate for the mapping.

It should be noted that the mapping procedure is semi-automated. First, the matching algorithms may return multiple mapping candidates. In that case, the user must select the most appropriate candidate for the fault injection. Second, the matching algorithms may terminate without detecting any potential candidate. In such a scenario, the user is asked to manually define a relevant mapping point. The name-based data matching approach described above and as illustrated in Figure 6.23, which is applied to systematically identify relevant mapping candidates, is summarized in Algorithm 3.

Algorithm 3 Data Matching Flow for Safety Evaluation Purposes

```
function FindMappingCandidates($\Psi$, $\Omega$)
    $\Upsilon$ is an empty dictionary
    for each relevant object $\theta_\Psi \in \Psi$ do
        add $\theta_\Psi$ as a key of the dictionary $\Upsilon$
        assign an initially empty list $\upsilon_{\theta_\Psi}$ as the associated value with $\theta_\Psi$ in $\Upsilon$
        $\Phi_{\theta_\Psi} \leftarrow$ list of expressive segments in $\theta_\Psi$'s Name and Description attributes
        for each relevant object $\theta_\Omega \in \Omega$ do
            for each expressive segment $\kappa$ in $\Phi_{\theta_\Psi}$ do
                if $\kappa$ occurs in $\theta_\Omega$'s Name or Description attributes then
                    $\upsilon_{\theta_\Psi} \leftarrow \upsilon_{\theta_\Psi} \cup \{\theta_\Omega\}$
                end if
            end for
        end for
    end for
    return $\Upsilon$
end function
```

The FindMappingCandidates function, which is presented in Algorithm 3, has two inputs, namely the safety analysis model Ψ and the system model Ω. Both inputs are sets of associated objects characterized each by a set of attributes. Data objects of the safety analysis model which are relevant for the data matching algorithm are referred to as θ_Ψ. These include failure modes, failure effects, safety mechanisms, etc. Corresponding data objects in the system model, which are considered as potential mapping candidates (e.g., modules, signals, ports, etc.,), are denoted as θ_Ω. The output of the function is a data structure, which is a Python dictionary Υ in the actual implementation. This dictionary links all relevant objects of the safety model to the lists of respectively found objects of the system model as potential mapping candidates. The user can then explore the data matching results to select the most suitable candidate in each case.

7 Application

This chapter presents the case studies that have been conducted in the context of this thesis. Through these case studies, the developed approaches are evaluated with respect to their usability and practicability in the industrial safety lifecycle. Indeed, the studies demonstrate the application of (i) the formalization methodologies adopted for functional safety analysis techniques, (ii) the associated frameworks for model-based automation support, and (iii) the systematic linking between analysis and simulation for safety evaluation purposes.

7.1 Overview

Throughout this thesis, functional safety evaluation has been addressed from three major perspectives. First, the traditional safety analysis techniques, mainly FMEDA, FTA, and DFA, have been extensively studied and correspondingly formalized using the metamodeling concept. Second, based on the created safety analysis metamodels, corresponding model-driven frameworks and platforms have been developed to enhance automation, efficiency, reuse in the context of safety analysis. Finally, a seamless linking has been established between safety analysis on the one hand and fault injection and simulation on the other hand through dedicated model-to-model mapping tools and data matching algorithms.

The overall requirements which are satisfied by the developed methodologies and tools throughout this thesis have been listed in Section 4.1. They are related to five main aspects: (i) structure and formalism with respect to the descriptions of safety analysis techniques (Subsection 4.1.1), (ii) flexibility and extendability of such descriptions (Subsection 4.1.2), (iii) automation support through accordingly developed frameworks and tools (Subsection 4.1.3), (iv) interoperability and data exchange (Subsection 4.1.4), and (v) enhanced usability of the complete flow through convenient user interfaces (Subsection 4.1.5).

To achieve the defined objectives of the thesis, multiple solution concepts have been applied, mainly (i) model-driven development (Subsection 4.2.1) and its related techniques metamodeling and code generation (Subsection 4.2.2), (ii) the metasynthesis methodology addressing synthesis of generation tools (Subsection 4.2.3), and

(iii) data transformation and mapping methods (Subsection 4.2.4).

Semiconductor and system manufacturing, particularly for automotive products, is the primary domain of application for the developed methodologies and tools in the context of this thesis. Therefore, the IEC 61508 and most importantly the ISO 26262 standard (especially Parts 4, 5, and 6) have been taken into account from the conception to the application of all solutions in this thesis. The guidelines of safety evaluation and the several steps of the safety lifecycle have been considered to allow an easy integration of the developed solutions in the comprehensive safety evaluation environment used within the industry.

To demonstrate the results of the thesis, two application examples are given in this chapter. They are organized as follows.

First, in Section 7.2, the fault tree synthesis described in Subsection 6.3.2.3 is illustrated through the fault tree generation of an Electric Power Steering (EPS) subsystem. The fault tree synthesis application is based on the failure propagation models created in MetaFPA (see Sections 5.2 and 6.2) and the data model transformation concept described in Subsection 6.3.2.2 to establish the transition between MetaFPA and the FTA MetaLib (see Subsection 6.3.2.1).

Then, in Section 7.3, the major capability of the SaVer framework (see Section 6.4) consisting in linking safety analysis outcomes with fault injection campaigns and fault simulation results is exemplified through the mapping of an FMEDA conducted for a CPU model to the fault injection and simulation performed on the corresponding SystemC virtual prototype.

7.2 Fault Tree Synthesis at System Level

The MetaFPA (Metamodeling-based Failure Propagation Analysis) methodology (see Section 5.2) has been developed in the context of this thesis to enhance the Failure Logic Modeling (FLM) theory (see Subsection 3.1.2) with a dynamic failure description and to make it more compatible with conventional simulation concepts. The correspondingly developed MetaFPA framework (Subsection 6.2) has several features with respect to the visualization and handling of failure propagation models and offers high levels of automation and reuse in comparison to other FLM platforms previously created in the context of related academic works.

In addition to the MetaFPA methodology, traditional safety analysis techniques have been addressed from a formalization perspective to subsequently enable model-driven automation support. Among those techniques, Fault Tree Analysis (FTA), which is an ISO 26262-recommended deductive (top-down) analysis method, relies on backward stepping from top-events to primary events and Boolean logic gates to describe the failure behavior and document it in a tree structure (more details in Subsection 5.3.2.1). The FTA flow and the underlying structuring fundamentals for FTA data are formalized in the FTA metamodel (see Subsection 5.3.2.2) and supported by the associated FTA *MetaLib* (see Subsection 6.3.2) created in Infineon's Metagen environment (see Subsection 6.1.2).

One of the most relevant features implemented in this thesis with respect to the ISO 26262 safety evaluation flow is the transition between the MetaFPA framework and the FTA MetaLib. The associated model-to-model transformation tool implements the fault tree synthesis algorithm (Subsection 6.3.2.3) whose application is illustrated in this Section through the following case study addressing an EPS subsystem.

7.2.1 Case Study: Electric Power Steering

Similarly to braking, steering is an ASIL-D automotive application. Related failures may lead to out-of-control vehicles and potentially cause critical accidents. Hence, satisfying safety requirements of steering systems is a major task for automotive manufacturers, as well as for their suppliers. Therefore, an Electric Power Steering subsystem composed of Infineon components is addressed as an application example for the fault tree synthesis. After a general introduction of EPS systems in Subsection 7.2.1.1 and a brief literature review of EPS failures in Subsection 7.2.1.2, the structure of the EPS subsystem which is concretely used for the case study is presented in Subsection 7.2.1.3.

7.2.1.1 Introduction: Electric Power Steering (EPS) Systems

There are two major reasons behind the establishment of power steering in the automotive industry: (i) the increasing front axle loads and (ii) the raising need for higher agility and flexibility of the steering properties [149]. Hence, the trend towards direct transmission steering systems has been progressively confirmed [150, 149].

The basic functionality of power steering systems consists of (i) measuring the steering torque applied by the driver by a dedicated mechanism in the input shaft region of the steering gear or in the steering tube and (ii) accordingly introducing additional forces or moments into the system in order to reduce the steering boost and subsequently achieve a better road contact at high speeds [149].

The traditionally used power steering systems are *hydraulic*. They rely on using oil under pressure to boost the servo [149].

In this case study, a more recent type of power steering systems is addressed, namely EPS (Electric Power Steering) [150] which has become particularly popular in comparison to traditional hydraulic steering systems due to its improved power efficiency. Being only active during the actual steering process, EPS systems consume between 0.3 and 0.5 liter less fuel per 100 km which makes them more economical and environmentally-friendly [151]. In addition to this improved fuel efficiency estimated to reach up to 3%, EPS offers a compact design due to less space requirements and leads subsequently to reduced mounting costs [17].

Currently used EPS systems are designed to capture the driver's steering requests and accordingly provide steering assistance by generating a part of the steering force through an electric motor whose torque is transmitted to the steering column or gear. This force superposition mechanism reduces the steering efforts of the driver, offers higher steering quality, and subsequently enhances the functional safety and has a positive effect on the overall driving experience and comfort [151, 17]. EPS has become an almost standard feature in modern cars, as it is the steering technology on which several advanced driver assistance systems such as (i) lane assist/keeping, (ii) side-wind compensation, and (ii) parking assistance systems are based [17].

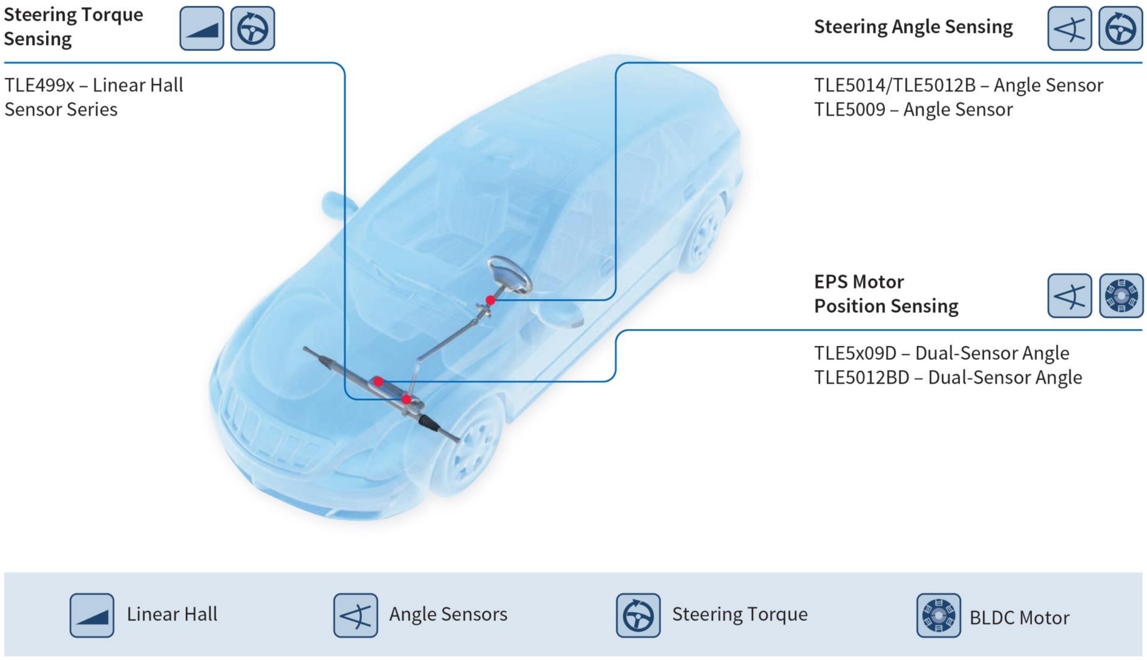

Steering Torque Sensing
TLE499x – Linear Hall Sensor Series
Steering Angle Sensing
TLE5014/TLE5012B – Angle Sensor
TLE5009 – Angle Sensor
EPS Motor Position Sensing
TLE5x09D – Dual-Sensor Angle
TLE5012BD – Dual-Sensor Angle
Linear Hall
Angle Sensors
Steering Torque
BLDC Motor

Figure 7.1: Application of Infineon Magnetic Position Sensors in EPS Systems [15]

In Figure 7.1, the basic functionality of EPS systems is illustrated in a simplified way. In addition to the electric motor which is used for the generation of a part of the steering torque, the EPS system relies on multiple position sensors measuring (i) the input steering torque applied by the driver, (ii) the position of the EPS motor moving the steering rack, and (iii) the absolute position of the steering wheel [15].

7.2.1.2 EPS Failure Analysis: Literature Review

During the last decades, several mechanical actuators traditionally used in the automotive industry have been progressively replaced by electric drives to increase the efficiency and enhance the dynamic performance [152]. Nowadays, it has become common to have electrically assisted braking and steering in mass-production vehicles. It is obvious that failures of such key functionalities have a direct impact on passenger safety. In fact, an electrical incident affecting a traction motor and more generically any fault in an electric drive used for propulsion in the vehicle may lead to an uncontrolled output torque and subsequently have an adverse effect on the vehicle stability. Therefore, it is considered as a critical threat for the overall functional safety [153, 152, 16].

In [153], a general investigation of failures occurring in traction systems within modern vehicles, where electric motors produce the complete or at least a part of the propulsion power, shows that the major cause for motor drive failures consists in inverter faults. Actually, such malfunctions may lead to an asymmetrical distribution of the traction force. This asymmetry is highly hazardous as it may lead to a high *yaw-torque* making the driver unable to control the vehicle [153]. Another severe failure scenario originating from electric drive malfunctions is the so-called *wheel-locking* issue consisting either in (i) the lockup of the rear tires causing the loss of directional stability or (ii) the lockup of the front tires leading to steering instability [153].

In [153], the authors propose the following categorization of motor drive faults:

- **Mechanical malfunctions** such as phase disconnections and magnet breakdowns [153].
- **Electrical malfunctions** including mainly short and open circuits occurring in the 3-phase inverter bridges and which are commonly caused either by (i) over-temperature, (ii) over-voltage, or (iii) over-current [153].
- **Control malfunctions** such as sensor defects and software deficiencies [153]

In [16], the focus of the investigation is on the potential failure modes of EPS systems with Permanent Magnet Synchronous Motors (PMSM). Beyond the analysis of local faults occurring in the different components contained in EPS systems such as in [154, 155, 156], the study documented in [16] addresses in particular the system-level failure behaviors of the EPS.

In [16], the failure modes investigation is based on a simplified block diagram of the EPS system, including:

- a sensing mechanism to get the steering input and subsequently forward the position current and the torque

- an electric controller
- an inverter stage
- a Permanent Magnet Synchronous Motor (PMSM)

The failure causes identified using this basic structure of the EPS system are classified as follows [16]:

- **Malfunctions of the motor position sensor**: Using an encoder with a certain number of pulses per revolution (PPR), the motor position and speed are commonly estimated through a pulse counting mechanism implemented by a motor position sensor in the EPS system [153]. Among the possible failure modes of such a sensor, the delivered number of counted pulses can be wrong. The resulting motor position and speed values are subsequently incorrect and may affect the consistency of the assistance torque which gets later on generated by the EPS system. In [153], the authors mention so-called *missing teeth* and *additional teeth* to depict the deviations from the normal behavior. In correlation with the missed pulses or the erroneously counted pulses, the respectively estimated position gets decreased or increased.

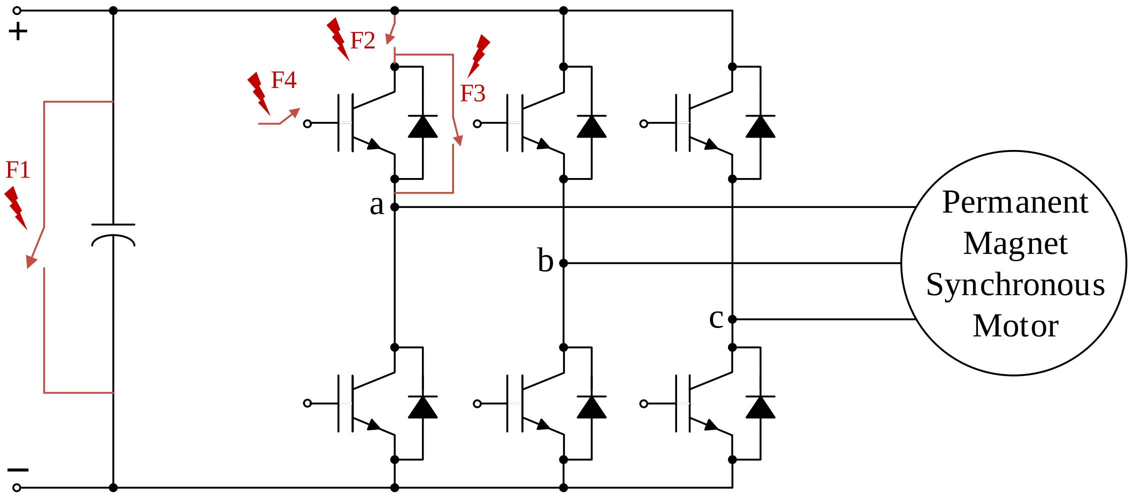


Figure 7.2: Potential Faults of a 3-Phase Voltage Source Inverter Used in an EPS System [16]

- **Malfunctions in the inverter stage**: Based on an easy representation (see Figure 7.2) of the Voltage Source Inverter (VSI) contained in the EPS system, [153] identifies the following four faults which are also annotated on Figure 7.2:
  - *Short circuit at the DC link capacitor (F1)*

  - *Open-switch (F2)*
  - *Short-switch (F3)*
  - *Open-gate (F4)*

- **Malfunctions of the PMSM**: In [153], the addressed faulty configuration of the PMSM is referred to as *turn-to-turn short* affecting the *a-phase winding.* A turn-to-turn fault is commonly induced by mechanical contact which is either the result of mechanical forces in the transformer or of a severe insulation deterioration caused by excessive overloading [157]. To put it simply, the turn-to-turn short fault of the PMSM leads to a faulty winding with a smaller number of turns, and subsequently to severe deviations from the normal behavior of the PMSM with a direct impact on the applied steering torque and consequently on the stability of the vehicle.

7.2.1.3 Considered EPS Subsystem Structure

The *system* definition in the ISO 26262 terminology implies that it must contain at least one sensor, one controller, and one actuator.

Therefore, the structure considered in this case study and which is depicted in Figure 7.3 is an EPS *subsystem* as it does not contain any actuator. In fact, the electric motor is not an Infineon product and that is why it is not considered in the qualification process of the EPS subsystem regarding its compliance with the ISO 26262 standard. The EPS solution at Infineon relies on diverse sensor types and on low-loss MOSFETs. It can be adapted via software such that it can be flexibly deployed in different car models and used in diverse driving modes.

From a structural perspective, the EPS subsystem addressed in this case study includes [17, 12]:

- a torque sensor (Linear Hall Sensor, e.g., TLE4498) to capture the driver steering torque and the front-wheels aligning torque,
- an angle sensor (a *Giant Magnetoresistance* GMR-based sensor, e.g., TLE5012B) to sense the rotor position,
- an MCU: Microcontroller Unit (e.g., Infineon's AURIX™) to process the information acquired from the sensors such as Single Edge Nibble Transmission (SENT) signals and to generate Power Width Modulation (PWM) signals for motor operation,
- a supply device to feed the MCU and the sensors (e.g, the safety system power supply TLF35584 containing a safety watchdog),

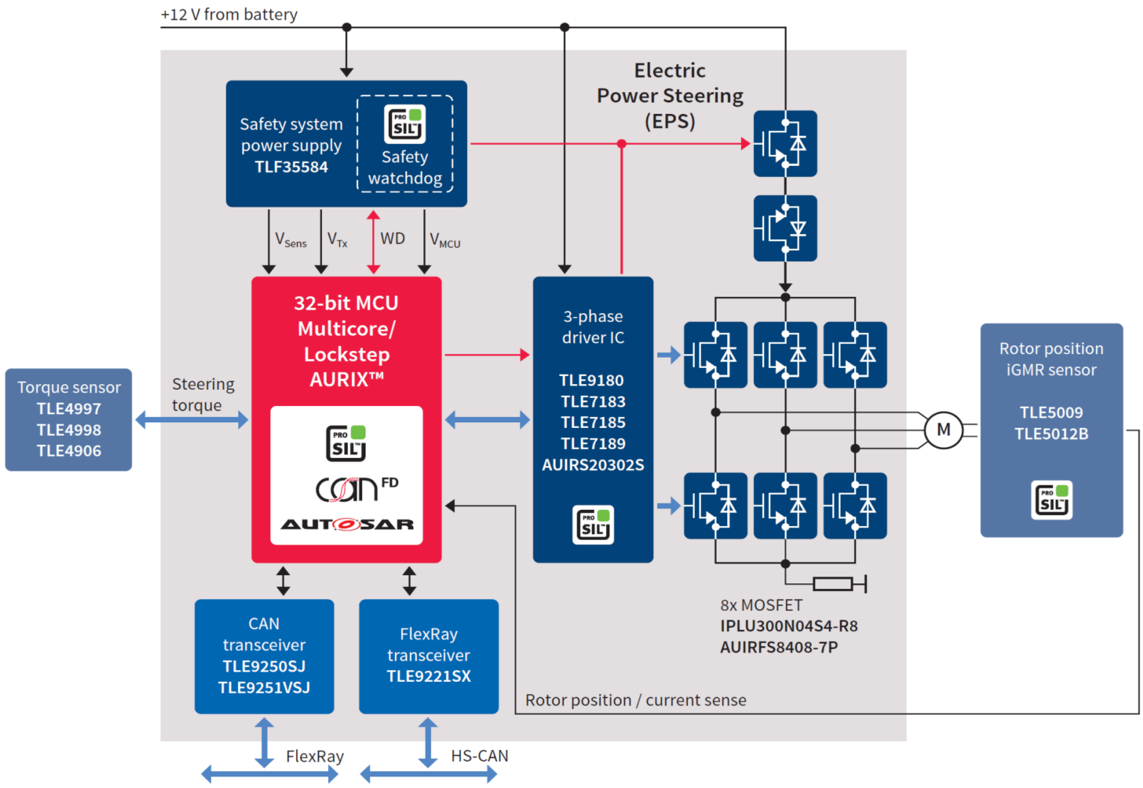


Figure 7.3: Electric Power Steering (EPS) Subsystem Diagram [17]

- a 3-phase Driver IC providing a gate-drive to level-shift PWM signals, and
- an inverter power stage featuring H-bridge arranged low-loss MOSFETs to provide phase signals to operate the motor.

7.2.2 Application Flow

The EPS subsystem structure described above is used as an example to apply the developed methodology of algorithmically transforming failure propagation models created in MetaFPA into fault trees. In this Subsection, the application flow is illustrated. It consists of the following steps:

- building the compositional model of the EPS subsystem in MetaFPA (Subsection 7.2.2.1),
- creating the failure logic model of EPS in MetaFPA (Subsection 7.2.2.2), and
- correspondingly applying the fault tree synthesis algorithm (Subsection 7.2.2.3).

7.2.2.1 Structure Model of EPS in MetaFPA

The simplified subsystem structure shown in Figure 7.4 is constructed in compliance with the system modeling part of the metamodel for failure propagation analysis (see Subsection 5.2.3). In accordance with the system modeling aspects in MetaFPA (see Subsection 5.2.3.1), the EPS model is created as a composition of blocks (supply device, torque sensor, angle sensor, microcontroller unit (MCU), 3-phase driver IC, and inverter power stage), featuring each a set of ports and interconnected through a number of labeled connections. The simplified compositional structure of the EPS subsystem is derived through abstraction from the block diagram of Infineon's EPS solution (see Figure 7.3).

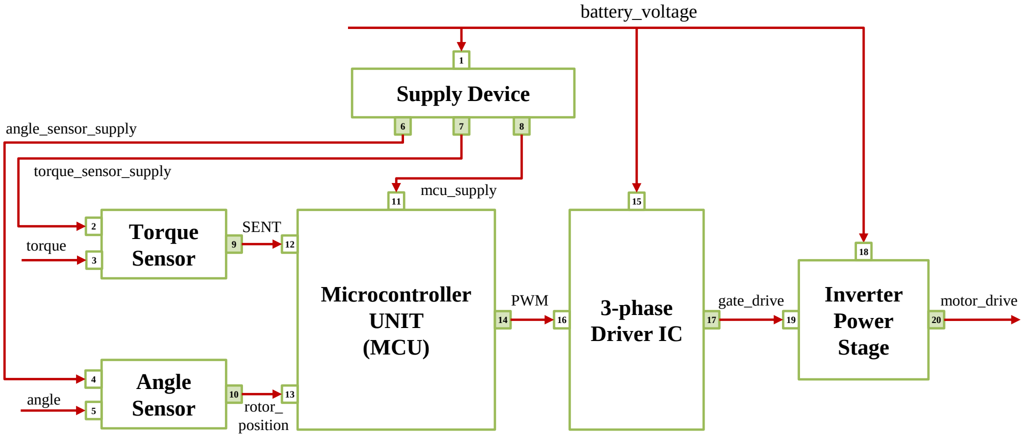


Figure 7.4: Simplified EPS Subsystem Structure

7.2.2.2 Failure Logic Model of EPS in MetaFPA

On top of the compositional structure of the EPS subsystem, the failure behavior is described in MetaFPA in compliance with the failure logic modeling part of the metamodel for failure propagation analysis (see Subsection 5.2.3). In accordance with the failure modeling aspects in MetaFPA (see Subsection 5.2.3.2), the block failures of the EPS system components are specified. The interface deviations occurring either at the input ports or at the output ports of each block, as well as the internal malfunctions taking place inside the blocks are captured in the failure logic model.

In this case study, the two capabilities offered by the MetaFPA framework for specifying block failures are demonstrated. First, manual entry through the MetaFPA GUI (see Subsection 6.2.2) is used for the definition of internal malfunctions. Internal documents, such as the product architecture and the safety concept which were already available for the EPS subsystem at the beginning of the case study, have been used as a source of information to identify all know internal malfunctions and consequently enter them into the MetaFPA data model.

The MetaFPA framework has also the capability of systematically deriving a set of *generic block deviations* based on the ports of the respective blocks. In the EPS example, deviations like “wrong rotor position” on the microcontroller input and “wrong PWM” on its output are examples of such generic interface failures which are automatically generated and added to the failure logic model.

Based on the defined single failures of the EPS blocks and the correspondingly specified *incompatibility references* (see Subsection 5.2.3.2), valid failure combinations are derived and subsequently used for the creation of the **PropagationMapping** and **TransformationMapping** class instances.

7.2.2.3 Applying the Synthesis Algorithm

By applying the model-driven approach described in Subsection 6.3.2.3, an appropriate fault tree is synthesized for the EPS subsystem.

In this case study, two potential safety goal violations of the EPS subsystem are addressed. First, *self-steering* is taken into consideration. It depicts the case where an unintended steering assistance is applied in contradiction with the driver requests. Second, *blocked-steering* is considered. It corresponds to the lockup of the assistance torque at a specific value.

As already described in Subsection 6.3.2.3, each safety goal violation is addressed by a separate fault subtree which gets progressively constructed through Algorithm 2.

The synthesized subtree for the first safety goal violation σ_1 = *self-steering* is represented graphically in Figure 7.5.

σ_1 is the result of a single output deviation, namely δ = *“wrong motor drive”* at the inverter power stage ($|\Delta_{\sigma_1}| = 1$). The upper *OR* gate in Figure 7.5 corresponds to γ_δ (see Algorithm 2) and is appended to the top-event of the complete EPS subsystem fault tree [12].

δ = *“wrong motor drive”* is caused either by one of the internal malfunctions of the inverter power stage or by the input deviation ϑ = *“wrong gate drive”*. For ϑ, the CreateBranch function is used to complete the fault subtree. The triangles labeled IEG-003, IEG-007, and IEG-011 depict collapsed branches [12].

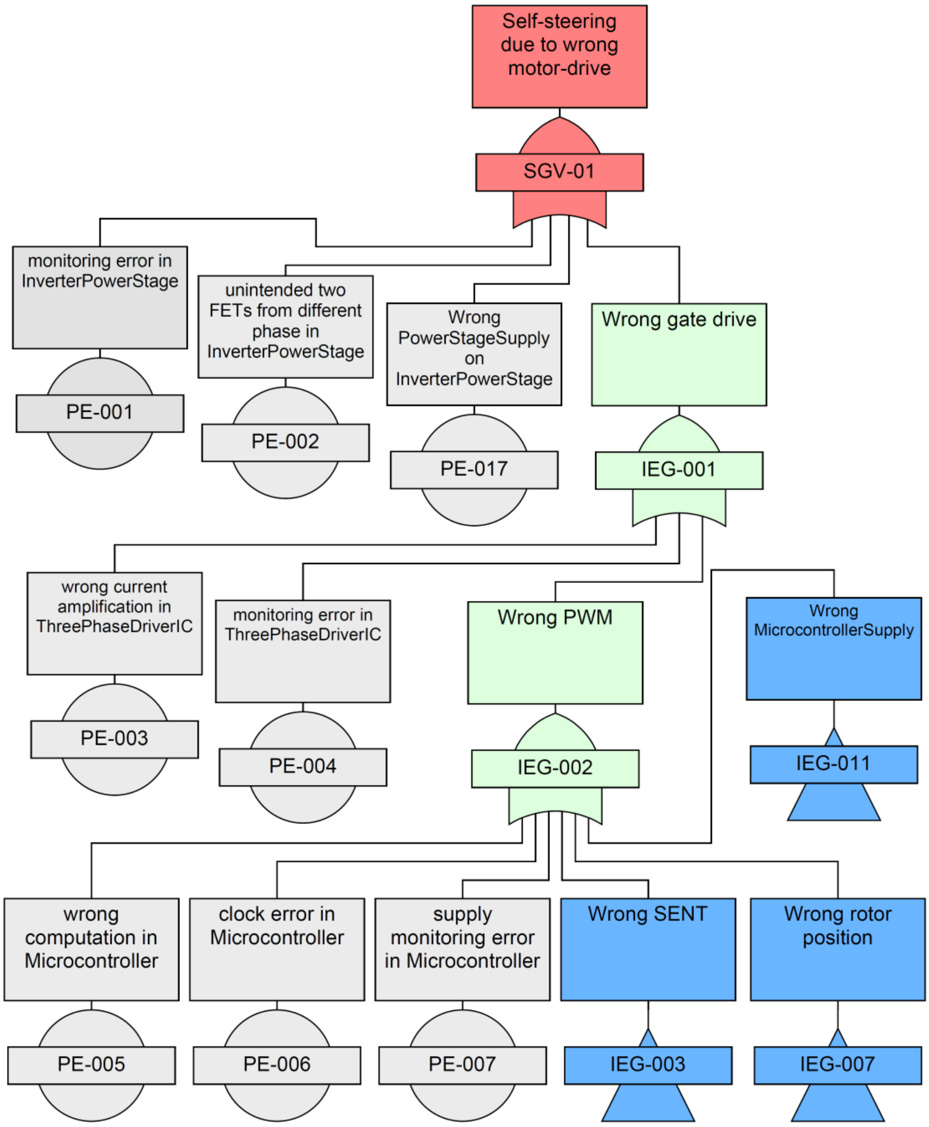

Self-steering due to wrong motor-drive
SGV-01
monitoring error in InverterPowerStage
PE-001
unintended two FETs from different phase in InverterPowerStage
PE-002
Wrong PowerStageSupply on InverterPowerStage
PE-017
Wrong gate drive
IEG-001
wrong current amplification in ThreePhaseDriverIC
PE-003
monitoring error in ThreePhaseDriverIC
PE-004
Wrong PWM
IEG-002
Wrong MicrocontrollerSupply
IEG-011
wrong computation in Microcontroller
PE-005
clock error in Microcontroller
PE-006
supply monitoring error in Microcontroller
PE-007
Wrong SENT
IEG-003
Wrong rotor position
IEG-007

Figure 7.5: Synthesized Fault Subtree for EPS *Self-Steering*

7.3 Linking FMEDA to Fault Injection and Simulation in Virtual Prototypes

The gap between safety analysis on the one hand and fault injection and simulation on the other hand (see Subsection 1.4.1.3) is one of the key challenges that have been addressed in this thesis. From a theoretical point of view, the adopted methodology to overcome this issue consists in three major steps. First, formal metamodels have been created to describe safety analysis artefacts and the relationships between them. Then, the correspondences and/or equivalences between analysis and simulation with respect to the involved data elements have been investigated. Finally, the data mapping patterns between the two contexts have been identified.

Among the tangible embodiments of this methodology, the FMEDA MetaLib (see Subsection 6.3.1) and the *SaVer* Framework (see Section 6.4) have been implemented.

The most important benefits of the link between safety analysis and fault simulation which is achieved by the FMEDA MetaLib and the SaVer Framework are:

- Enhanced efficiency of fault injection and simulation: Through convenient usage of safety analysis outcomes as guidance for fault injection and simulation, the size of the fault space as well as the number of the related injection scenarios are reduced.
- Increased confidence in the safety analysis outcomes: The results of the fault injection campaigns and the associated simulation runs provide accurate information about the safety integrity level of the system (e.g., percentage of safe faults and diagnostic coverage values) which can be used to verify the correctness and consistency of the safety analysis outcomes.

7.3.1 Case Study: Microprocessor with MIPS Architecture

In this thesis, a microprocessor model with MIPS (Microprocessor without Interlocked Pipeline Stages) architecture is used as a test vehicle to evaluate the SaVer framework with respect to its linking feature between FMEDA and fault injection. The considered microprocessor model is called "*Nano*MIPS" because it supports a reduced instruction set in comparison to commercial MIPS processors and because it does not contain a floating point unit.

In this case study, both aspects of safety evaluation are considered: namely analysis and simulation. On the one hand, FMEDA is the adopted technique to perform a quantitative safety analysis of the microprocessor model. On the other hand, a SystemC virtual prototype of NanoMIPS is used for fault injection and simulation.

The work flow of the case study is structured as follows:

1. Construction of the safety analysis model (FMEDA model) using the FMEDA MetaLib (see Subsection 6.3.1).
2. Construction of the nominal system model on which the fault injection and simulation will be performed. This step consists in building an abstract architectural model of the NanoMIPS based on its SystemC implementation as a virtual prototype. This system model contains modules, sub-modules, and members.

3. Mapping of analysis artefacts, i.e., FMEDA data elements such as parts, failure modes, and safety measures to system model elements such as (sub)-modules, and members. This mapping step, which is supported by the SaVer framework, is semi-automated. It relies on detecting potential candidates for mapping by applying the implemented matching algorithms (see subsection 6.4.3).
4. Configuration of fault injection. This configuration step consists for example in defining exact bit positions within signals where faults must be injected as well as the accurate start and potentially stop times of the fault insertions. It is also supported by the SaVer framework, more precisely through the SaVer GUI (see Subsection 6.4.2).
5. Generation of the fault library for the NanoMIPS microprocessor. This generation step, supported by the SaVer framework, is template-based and delivers a text file including all concrete faults to be introduced in the system and corresponding injection details (locations, timing parameters, etc).
6. Fault injection and simulation. In this case study, an in-house SystemC-based fault simulation platform is applied [140, 141]. On this platform, the generated fault library is accordingly processed to initiate the automated fault injection.
7. Back-annotation of measured fault injection and simulation results (mainly diagnostic coverage values of safety mechanisms) into the FMEDA model. This step is automatically supported within the SaVer framework and serves as a consistency check to the estimated values of the analysis.

The remainder of this subsection contains a general description of the MIPS architecture in 7.3.1.1, the details of the NanoMIPS implementation in 7.3.1.2, and a brief literature review of commonly known CPU failure modes in 7.3.1.3.

7.3.1.1 Introduction: MIPS Architecture

MIPS [158] has been first designed in the 1980s at Stanford University as a Reduced Instruction Set Computer (RISC) processor [18]. It is a widely used microprocessor architecture, especially in embedded designs because of its reliability and cost-effectiveness [159]. There are multiple possible implementations of the MIPS architecture, mainly depending on the supported Instruction Set Architecture and the pipeline depth. Nevertheless, MIPS has an important characteristic which remains valid for all those implementations. In fact, locks and stalls do not slow down the execution in cases of inter-dependencies between instructions within the MIPS architecture. By re-ordering the instructions, such dependencies can be resolved. Insertion of empty operations (NOPs) as well as data forwarding are also used as solutions for dependency-caused conflicts [18].

The MIPS architecture consists of five pipeline stages (see simplified diagram in Figure 7.6):

- **Instruction Fetch (IF)**: This stage serves to load the next instruction to be performed from the memory. It includes a program counter (PC) register which holds the actual address pointing to the exact instruction to be read from the memory and to be passed to the next stage. If there is no jump or branch address which overwrites the program counter, then it gets accordingly incremented in order to point to the next instruction in the memory. Therefore, the IF stage contains an adder to increment the PC and a multiplexer to handle branches and jumps.
- **Instruction Decode (ID)**: In the ID stage, the fetched instruction from the memory gets decoded in such a way that it can be executed by the subsequent stages. First of all, the *opcode* segment of the instruction is determined and translated into an internal *ALU vector*, called *AluOp* for instance. Then, the *operands* are fetched from the register file and forwarded to the execution stage. For instructions of type *Immediate*, the immediate value is extracted from the instruction itself (16 lower bits) and sign-extended to 32 bits. Furthermore, the ID stage contains a control logic which generates several control signals such as the write enable signals and the multiplexer signals which facilitate the instruction execution later on.

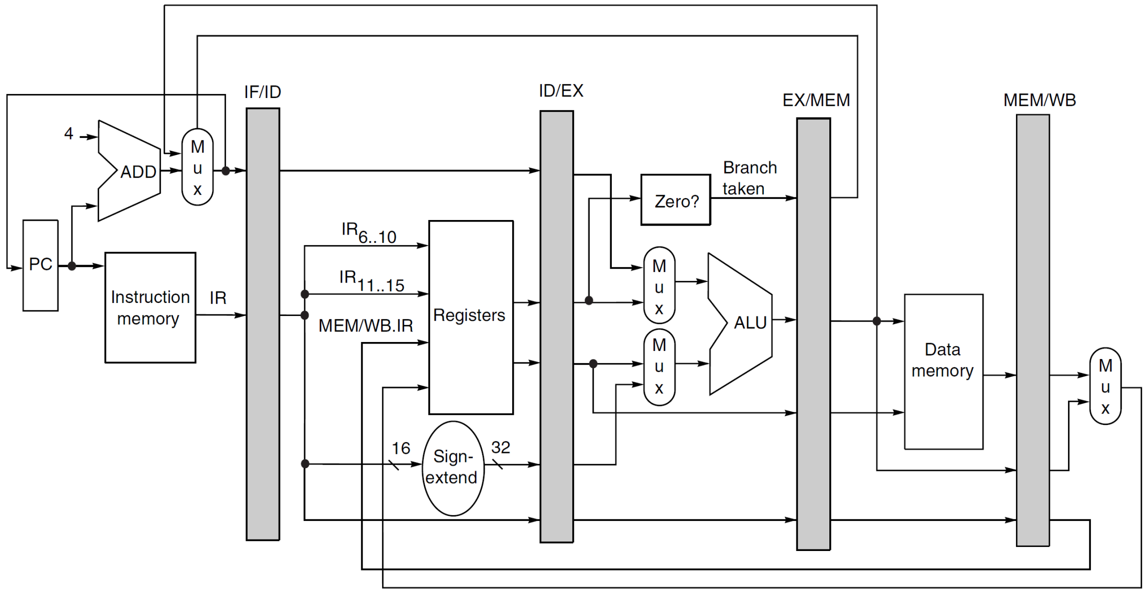


Figure 7.6: Simplified Diagram of Pipelined MIPS Architecture [18]

- **Execution (EX)**: This is the part of the MIPS pipeline where the actual data processing is performed. Therefore, it contains several calculation components (e.g., ALU, adder, shifter, multiplier/divider, etc.). In general, each of these components requires two input operands. One of them is always a register and the other is either a second register or an immediate value (directly derived from the considered instruction). At this stage, jump and branch addresses are calculated. Furthermore, arithmetic operations are performed by the ALU (addition and subtraction, logical operations, and comparisons for conditional branches). In addition to that, shifting operations are performed by the barrel shifter. Multiplication, division, and modulo operations are covered by the multiplier unit. The EX stage contains also a number of multiplexers to correspondingly dispatch the calculation results either to the register file or to the external memory.

- **Memory Access (MEM)**: This is the pipeline stage where the memory is accessed depending on the calculated result in the Execution phase. If a *LOAD* or *STORE* instruction is considered and the calculated result during the execution is consequently a memory address, then the appropriate data is read from or written to the addressed memory location. For other instruction types, the MEM stage forwards the execution result to the next stage, which is the Write Back (WB) stage. This differentiation is handled by a dedicated multiplexer in the MEM stage.
- **Write Back (WB)**: In this final pipeline stage, either the computed result at the Execution phase or the addressed memory content at the Memory Access stage is written into the destination register.

In addition to the pipeline stages mentioned above, the MIPS architecture also includes a set of register banks in between them to shorten the combinational paths, allow higher clock frequencies, and increase the processor throughput. Furthermore, a register file, which does not belong to any specific pipeline stage, enables simultaneous reading of input registers and writing of output registers. In fact, the register file has three 5-bit address inputs and one 32-bit data output.

7.3.1.2 Considered NanoMIPS CPU Model

As already mentioned above, the considered CPU model in this case study, called NanoMIPS, conforms to the MIPS architecture. It has been progressively implemented and refined at Infineon in the context of student internships and theses (e.g., [160] and [19]) and as part of the funded research project EffektiV [136]. The primary implementation guideline of the NanoMIPS has been the education book “Computer Organization and Design” from Patterson and Hennessy [18].

NanoMIPS (see abstracted block diagram in Figure 7.7) supports about 60 instructions including different arithmetic operations such as multiplication and division, shift and logical operations, load and store functions, as well as comparison tests. However, it does not contain a floating point unit as usual in embedded applications.

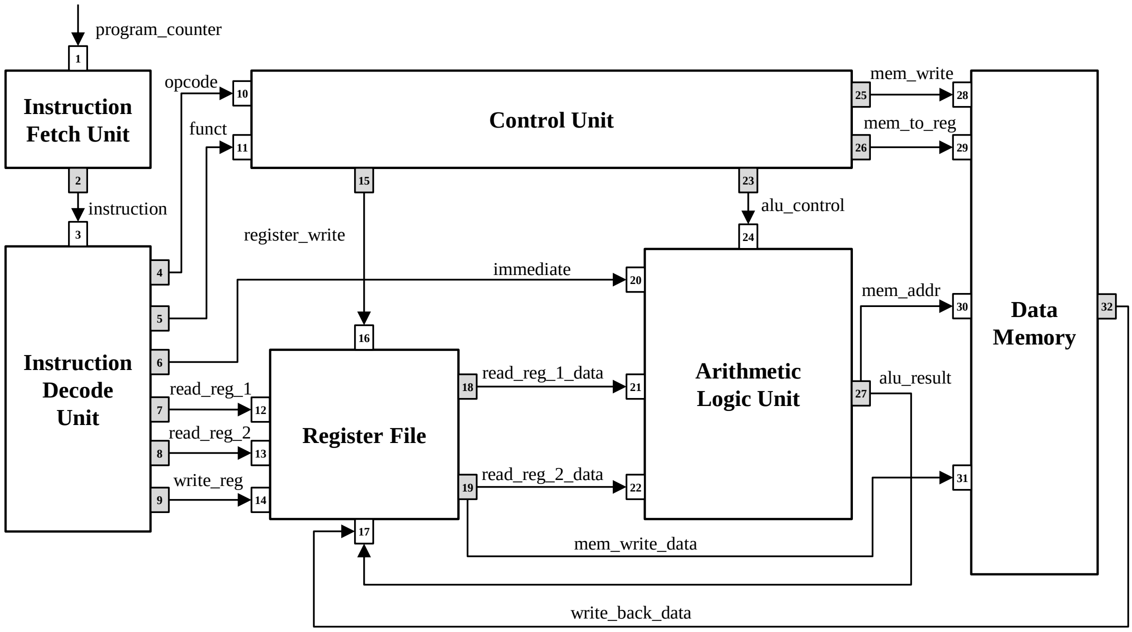


Figure 7.7: Highly Abstracted Block Diagram of *NanoMIPS* Architecture

One of the major concerns that have been addressed while implementing NanoMIPS is power management and efficiency. Therefore, the most power-consuming components (e.g., memory related elements, switching parts, etc) have been identified and appropriate design refinements have been made to reduce power consumption, for example by reducing the number of registers and removing all those which are unnecessary.

The detailed block diagram of the NanoMIPS architecture which has been considered in this case study is illustrated in Figure 7.8. The details of the implementation are available in [19].

7.3.1.3 Microprocessor Failure Modes: Literature Review

Microprocessors are widely used in safety-critical applications (automotive, railways, avionics, health-care, etc.). Therefore, it is frequently addressed in academic research topics such as (i) dependability assessment of computer systems, (ii) safety analysis methodologies and applications, and (iii) fault injection and simulation techniques.

In this subsection, a literature review about microprocessor failure modes is conducted. Several examples are extracted from the literature references and accordingly

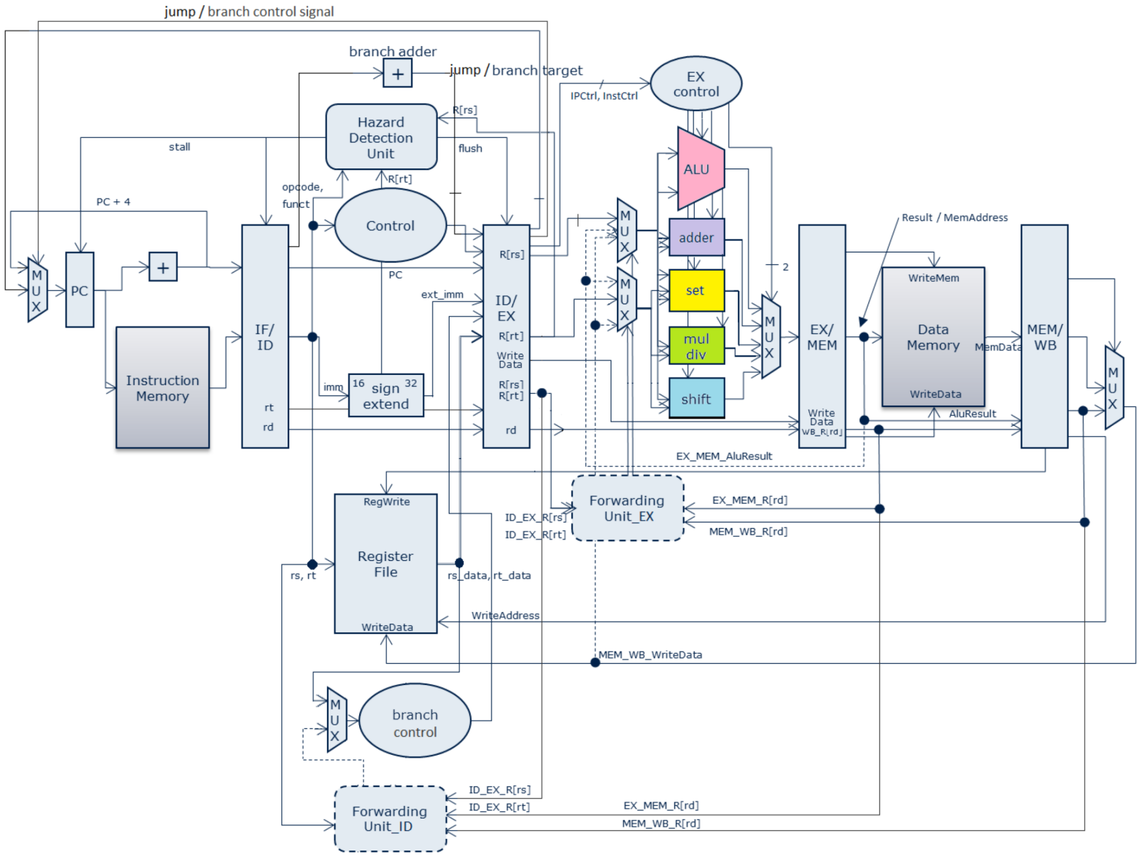


Figure 7.8: Block Diagram of *NanoMIPS* Architecture [19]

reported below. Subsequently a summarizing table gives an overview of the most common failure modes of the different components of a microprocessor (See Table 7.1).

In Chapter 6 of [18], several examples of CPU faults, errors, and failures are given. Their definition and explanation complies to the dependability taxonomy by Laprie [161] which is addressed in Section 1.2. [18] addresses a wide range of fault types including hardware faults, design faults, operation faults and environmental faults. However, in this case study, the focus is on random hardware faults which are the major concern of the FMEDA. Different examples of CPU failure modes are given in [18] as "*exceptions that may occur in the MIPS pipeline*". In the following, some of those exceptions are mentioned for the four first stages of the MIPS pipeline described above [18]:

- **Instruction Fetch (IF)**: memory protection violation, misaligned memory access, page fault[1] on instruction fetch

[1] A **page fault** occurs when a memory page that is not currently mapped by the memory management unit into the virtual address space is accessed

- **Instruction Decode (ID)**: undefined opcode, illegal opcode
- **Execution (EX)**: arithmetic exception (e.g., integer arithmetic overflow or underflow, floating point arithmetic anomaly)
- **Memory access (MEM)**: memory protection violation, misaligned memory access, page fault on data fetch

In [162], the following summary of classified failure mode examples at microprocessor level is given:

- **Erroneous operation for data acquisition**: incorrect value and/or incorrect validity, missing value, missing validity information
- **Erroneous operation for logic processing**: e.g., actuation failure

In [163], following failure modes are considered for different CPU components and/or functions:

- **Register, internal RAM**: Stuck-at, stuck-open, open or high impedance outputs, short circuits between signal lines for data and addresses, dynamic cross-over for memory cells, no addressing, wrong addressing, multiple addressing
- **Program counter, stack pointer**: Stuck-at, stuck-open, open or high impedance outputs, short circuits between signal lines

In [164], the criticality of failure modes affecting memory elements in safety-relevant systems is emphasized, especially when memory failures make the CPU read and/or execute an incorrect instruction. For the CPU itself, [164] presents this list of *functional failure modes* which is adapted from [165]:

- Incorrect result of arithmetic or logical operation (error in ALU)
- Incorrect address leading to incorrect memory contents (error in instruction decoder/pointer)
- Alteration of correct data or address (error in register file)
- Alteration of data and/or instruction to be read from memory or written into it (error in memory data interface)
- Alteration of pointed memory address (error in memory address interface)

Based on the above literature review and on further industry data, a summarizing overview of failure modes occurring in the different components of a microprocessor is given in Table 7.1.

Component	**Failure Mode Examples**
Instruction Fetch Unit	wrong program counter, corrupt program counter, program counter out of sync, etc.
Instruction Decode Unit	no instruction opcode, corrupt or illegal instruction opcode, wrong or undefined instruction opcode, etc.
Control Unit	corrupt or wrong control signals, wrong instruction sequence, wrong processor timing, etc.
Register File	corrupt or wrong data, incorrect write address, etc.
Arithmetic Logic Unit	no program execution, wrong program execution, delayed program execution, incorrect result of arithmetic or logical operation, arithmetic exception, etc.
Data / Instruction Memory	no addressing, wrong addressing, dynamic cross-over for memory cells, etc.

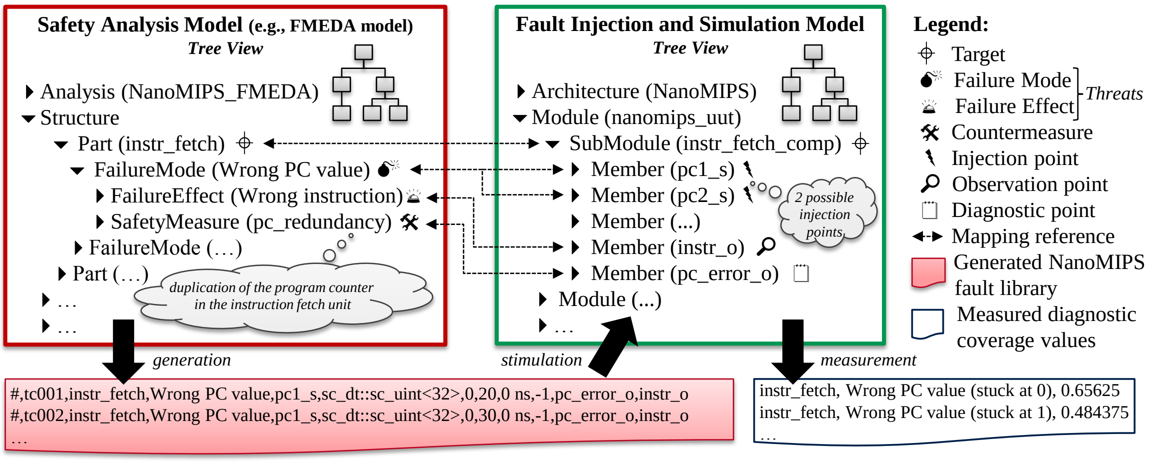
Table 7.1: Failure Mode Examples of Microprocessor Components

7.3.2 Application Steps

In Figure 7.9, the application of the linking approach between FMEDA and fault injection, which has been described above, is illustrated through an example in the context of the NanoMIPS use case.

In this example, the instruction fetch unit of the NanoMIPS is considered. To demonstrate the functionality within the SaVer framework, the effects of faults affecting the program counter (PC) are examined. It should be noted that a diagnostic mechanism addressing the program counter is implemented in NanoMIPS. It is based on the duplication of the PC and the usage of a watchdog timer. In fact, the redundant PC values are compared whenever a new instruction is fetched. If the two values are inconsistent, then further execution is prevented for a predefined period of time. In the case of recovery during that time period, the normal simulation is re-established and the fetched instruction is further on executed. Otherwise, a reset of the NanoMIPS is performed.

In the following subsection, the three remaining application steps of the use case are detailed, namely (i) the mapping of the safety analysis model elements to the


Safety Analysis Model (e.g., FMEDA model)
Tree View
Analysis (NanoMIPS_FMEDA)
Structure
Part (instr_fetch)
FailureMode (Wrong PC value)
FailureEffect (Wrong instruction)
SafetyMeasure (pc_redundancy)
FailureMode (…)
Part (…)
…
…
duplication of the program counter in the instruction fetch unit
Fault Injection and Simulation Model
Tree View
Architecture (NanoMIPS)
Module (nanomips_uut)
SubModule (instr_fetch_comp)
Member (pc1_s)
Member (pc2_s)
2 possible injection points
Member (...)
Member (instr_o)
Member (pc_error_o)
Module (...)
…
Legend:
Target
Failure Mode
Failure Effect
Threats
Countermeasure
Injection point
Observation point
Diagnostic point
Mapping reference
Generated NanoMIPS fault library
Measured diagnostic coverage values
generation
stimulation
measurement

```
#,tc001,instr_fetch,Wrong PC value,pc1_s,sc_dt::sc_uint<32>,0,20,0 ns,-1,pc_error_o,instr_o
#,tc002,instr_fetch,Wrong PC value,pc1_s,sc_dt::sc_uint<32>,0,30,0 ns,-1,pc_error_o,instr_o
…
```

```
instr_fetch, Wrong PC value (stuck at 0), 0.65625
instr_fetch, Wrong PC value (stuck at 1), 0.484375
…
```

Figure 7.9: Application Example: Link Between Safety Analysis and Fault Simulation of *NanoMIPS* CPU Model [14]

nominal system model elements, (ii) the generation of the fault library, and (iii) the back-annotation of the simulation results.

7.3.2.1 Data Mapping from FMEDA to Fault Injection

The prerequisites of the data mapping are the safety analysis model on the one hand and the fault injection and simulation model on the other hand. The left hand side of Figure 7.9 contains a part of the FMEDA model which is constructed using the FMEDA MetaLib based on the highly abstracted NanoMIPS block diagram (see Figure 7.7). Furthermore, the overall composition of the SystemC model of the NanoMIPS which is used for fault injection and simulation is illustrated on the right hand side of Figure 7.9.

According to the previously introduced terminology in Subsection 6.4.3.1 (see Table 6.1), the *targets* which might be affected, the *threats*, and the *counter-measures* represent the data artefacts to be mapped between both contexts (analysis and simulation). In Figure 7.9, multiple examples of those data artefacts are marked and labeled correspondingly to their respective categories (e.g., the Part "instr_fetch" is a target element in the safety analysis model which is reflected by the "instr_fetch_comp" sub-module in the executable system model, the Failure Mode "Wrong PC value" is a threat reflected by faults affecting the PC signals, and "pc_redundancy" is a counter-measure whose concrete reflection in the system model is the observation signal "pc_error_o").

The connections which can be observed between the data model objects from both sides in Figure 7.9 represent the mapping references between both models. Such references are identified through the application of the *matching algorithms* (see Subsection 6.4.3.2). In this use case, the name-based matching algorithm is applied. For instance, the failure mode "Wrong PC value" is mapped to 2 potential injection points: the "pc1_s" and "pc2_s" signals which are members of the "instr_fetch_comp" sub-module.

7.3.2.2 Fault Library Generation

The links which have been established during the mapping step are the first prerequisite for generating the fault library. They are however not sufficient to perform the required fault injection campaigns. Extra information is required to accurately define the concrete bit positions that must be altered during the injection of the considered faults, the simulation time at which the fault must be inserted, and potentially the duration of the fault occurrence (for transient faults only). This extra information is not derived from the safety analysis data. Therefore, it must be manually entered by the user in the appropriate fields of the SaVer GUI.

A small excerpt of the generated fault library for NanoMIPS is illustrated in Figure 7.9). Proper tools and mechanisms are applied within the fault injection and simulation platform to accordingly stimulate the system model and monitor its response to the inserted faults.

7.3.2.3 Back-Annotation

During the fault injection campaign triggered in accordance to the generated fault library, simulation results are progressively gathered, sorted out, and stored into a dedicated logging file. This file contains mainly the measured diagnostic-coverage values. In the considered NanoMIPS example, different ways to inject the "Wrong PC value" failure mode are possible such as a single-bit stuck at 0, a single-bit stuck at 1, a soft error, etc. In this case, the measured diagnostic-coverage value corresponds to the percentage of simulations where the faulty PC value is captured by the safety measure, independently from the injection variant.

The logging file created during the fault simulation is subsequently parsed within the SaVer framework so that its contents are appropriately used to extend the existing FMEDA data model. This back-annotation procedure is intended to make the simulation results a part of the FMEDA data model which can be visualized by the user and used to check the analysis assumptions.

The results of this use case show that the model-based support for safety analysis, which is enabled through the FMEDA metamodel and the correspondingly developed FMEDA MetaLib offers an effort saving reaching up to 60% in comparison to the manual procedure. Furthermore, mapping the FMEDA model to the SystemC model enables the generation of the fault injection, observation, and diagnostic points. The fault list generation offers significant effort and time savings for safety verification engineers. Indeed, about 80% of the manual tasks traditionally required to inspect the safety analysis outcome and derive fault injection lists out of them are fully substituted by systematic generation mechanisms. The remaining 20% consist in the configuration tasks mentioned in Subsection 6.4.3.1, where extra information such as injection time frames are manually entered by the user. In addition to that, the fault list generation allows traceability between the analyzed failure modes and effects on one side and the correspondingly simulated faults and responses on the other side. Moreover, the back-annotation of the simulation results into the FMEDA data enables cross-checking and potential refinements of the analysis assumptions and estimations.

8 Summary and Outlook

Dependability, which has been the primary topic of interest during this thesis, is a major concern in modern system development and manufacturing. In fact, on-top of system functionality, so-called *dependability attributes* such as *reliability*, *safety*, and *security* must be fulfilled.

To address the significance of the dependability topic and overcome the persisting challenges related to it, dependability fundamentals and approaches have been comprehensively studied in this work. Building upon the already existing methodologies to generalize dependability terms and to integrate model-driven development techniques in the dependability context, a metamodel-based approach for cross-domain and multi-level dependability analysis has been developed [57]. Through a generic dependability metamodel and a corresponding customization mechanism, the flexible creation of adapted dependability assessment tools is enabled to better fit the particularities of specific application-domains and/or abstraction-levels [57].

Being one of the major dependability attributes, *safety* has been particularly in the focus of this thesis. In fact, in the context of this work, functional safety of E/E systems has been explored from different perspectives. In particular, functional safety analysis has been addressed, along with its relationship to fault injection and simulation.

To overcome informality and subjectivity issues which are still persisting in the industrial practice of functional safety analysis, a metamodeling-based formalization approach has been developed in this thesis [166, 3, 137, 14, 12, 138]. This formalization approach provides concise, well-structured, and convenient descriptions of (i) safety analysis procedures, (ii) involved data artefacts and relationships between them, and (iii) underlying documentation structures.

Tackling real-life challenges encountered by functional safety engineers and analysts in E/E system manufacturing industries was a central concern throughout this thesis. Several model-driven frameworks have been conceived, developed, and implemented in the context of this work as alternatives to traditional safety analysis work flows which rely on cumbersome, effort-demanding, and time-consuming manual tasks.

First, a metamodeling-based failure propagation analysis framework, called *MetaFPA* has been developed [11]. MetaFPA addresses the limitations of the already existing *Failure Logic Modeling (FLM)* methodologies and provides enhancements with respect to the failure description, the automation level, and the flexibility in the FLM context. In fact, alternatively to the static failure description commonly supported by FLM platforms, MetaFPA enables a dynamic capturing of the failure behavior in a more compatible way with conventional simulation-oriented modeling guidelines. Furthermore, MetaFPA offers a significant level of automation and flexibility due to model-driven support and to the application of metamodeling and code generation techniques. Supporting the failure behavior exploration of a given system at a high level of abstraction, MetaFPA is suitable for an early application during the safety lifecycle. It delivers graphical representations of failure propagation paths by executing simulation-oriented analysis scripts, which are themselves generated from compositional system models including failure scenarios in addition to the nominal system functionality. Hence, MetaFPA outcomes are used particularly to identify critical failure scenarios at an early development stage and consequently improve the safety concept quality with respect to the adequacy and effectiveness of the safety measures which are contained in the design.

MetaFPA has also been applied in the context of this thesis to establish a link between inductive and deductive failure analyses [12]. In fact, dedicated frameworks are created for safety analysis techniques, particularly for Failure Modes, Effects, and Diagnostic Analysis (FMEDA) and Fault Tree Analysis (FTA). These frameworks rely, similarly to MetaFPA, on metamodeling and code generation techniques and are referred to as *Safety Analysis MetaLibs*. By incorporating these MetaLibs with MetaFPA in a comprehensive safety analysis environment, and by applying appropriate model-driven techniques, bi-directional data transfer between fault trees and FMEDA tables through MetaFPA data models is enabled. In addition, convenient tools are developed to support systematic generation of FMEDA table segments and of fault tree branches out of compositional failure models constructed within the MetaFPA platform.

The previously mentioned *Safety Analysis MetaLibs* have been created in Infineon's metamodeling and code generation environment, called Metagen, to support the most relevant functional safety analysis techniques in the industry, namely FMEDA and FTA. These MetaLibs help safety analysts to achieve their activities within the safety lifecycle in reduced time frames and with improved consistency and accuracy levels. Furthermore, the developed MetaLibs address completeness and flexibility issues of already published model-based automation approaches for FMEA and FTA, such as the disregard of the quantitative aspect of the analysis and the usage of special data formats which are not compatible with industrial practices.

The approach behind the first MetaLib, which addresses model-driven and simulation-assisted FMEDA, has been published at the Design Automation Conference in 2015 [3]. It relies on an compliant metamodel with the traditional FMEDA flow and its common documentation structure. The substitution of classical FMEDA tables by object-oriented data models, created as instances of the FMEDA metamodel, enables high automation and reuse levels. Manual data entry is replaced by systematic data extraction, processing, and exploration. Transformations are simplified through utility tools within the model-driven FMEDA framework and the overall quality of the analysis outcome is improved thanks to convenient consistency checks and user-assisting features. When applied in correlation to a simulation-oriented failure analysis platform at a higher abstraction-level, such as MetaFPA, the FMEDA MetaLib supports the hierarchical construction of a *System-FMEDA* based on the assembly of associated *Component-FMEDAs* and a set of *interface adapters* between them. Such adapters are derived from the connectivity information between system components and capture the interactions within the system in failure cases. Furthermore, the FMEDA MetaLib offers a verification capability for manually performed analyses. In fact, through appropriate data readers and parsers, an already exiting System-FMEDA table can be transformed into a data model. This data model is then compared to the respectively generated System-FMEDA data model through automated assembly of Component-FMEDAs. All detected data mismatches are documented in a cross-checking report and represent valuable indications to the safety analyst about potential inconsistencies in the manual analysis.

Applied as part of feasibility studies at first and then within pilot projects, the FMEDA MetaLib showed a significant speed-up of the complete procedure by reducing about 60% of the repetitive manual data entry tasks, simplifying the review process, and offering advanced data handling capabilities.

The second MetaLib addresses the Fault Tree Analysis (FTA) technique. It is based on a data transformation approach from compositional failure propagation models, created with the MetaFPA framework, into fault trees. The FTA MetaLib has been developed to overcome data processing, visualization, and creation issues in the FTA context and to support concept engineers and functional safety analysts at early stages of the safety lifecycle. The underlying data transformation approach, published at the industrial track of the International Conference on Dependable Systems and Networks in 2016 [12], relies on a fault tree synthesis algorithm which enables a systematic and progressive construction of the fault tree logic through traversing the failure propagation model and adding appropriate intermediate and/or primary events in correlation to input deviations and/or internal malfunctions. In practice, the fault tree synthesis algorithm is implemented within the FTA MetaLib by a dedicated model-to-model transformation tool. The MetaLib, which complies to a formalized FTA metamodel, contains also a special view generator which produces XML files depicting fault trees, following the schema of *Isograph Reliability Workbench*™, a widely used commercial tool for fault tree creation.

Furthermore, several perceptions, proof-of-concepts, and early prototypes derived from this work have contributed to the development of a comprehensive safety evaluation framework called *SaVer* [137, 14, 138]. By combining the FMEDA and FTA MetaLibs on the one hand with fault injection and simulation platforms on the other hand, SaVer establishes a seamless link between analysis and simulation. In addition to the publications mentioned above, the basis of the SaVer framework and its relevance for multi-level safety analysis has been addressed in a talk at the Design, Automation & Test in Europe Conference & Exhibition (DATE) in 2016 [167]. The talk emphasized the fact that safety analysis is a multi-domain and multi-skills task which has become an intricate part of the process. Furthermore, it showed the linking approach between safety analysis and fault-effect-analysis, particularly at transaction level. It should be noted that the SaVer framework contains a fault injection and simulation platform relying on virtual prototypes, where gate-level-accurate transaction-level (TL) models with all gate-level matching points are used. Such models are obtained through a systematic abstraction of gate-level netlists and enable a particularly accurate fault-effect-analysis which is simultaneously significantly fast [141, 140, 147, 168, 110, 169].

Due to the considerable benefits it offers with respect to data traceability, exchange, and reuse, SaVer has been further on enhanced and adopted in productive safety-related projects at Infineon.

Beyond the theoretical contributions of this thesis and their respective practical embodiments and applications within industrial projects, several aspects of future work are worth mentioning.

With respect to the automotive safety lifecycle, analysis-based evaluation is performed at concept, system-development, hardware-development, and software-development levels. The developed approaches and tools in the context of this thesis are particularly suitable for concept, system, and hardware levels. Further enhancements and customizations are necessary in order to cover the software level. Dependent Failure Analysis (DFA), which is particularly suited for assessing software failures, has been considered in this thesis and a dedicated metamodel has been constructed to formalize it. This DFA metamodel represents the basis for the DFA MetaLib which is implemented to provide support and automation capabilities in the analysis context. This DFA MetaLib should be experimented in productive contexts, particularly in the firmware development context. Moreover, it can be further on extended to offer more automation capabilities and to support a systematic linking to compositional failure models created using the MetaFPA framework and to fault injection and simulation contexts. In addition, the proof-of-concept created in this thesis about synergies between DFA, FTA, and FMEDA should be evaluated with respect to its practicality and effectiveness in real use-cases.

The link to requirements engineering has been a tangential concern in this work. The increasing importance of safety requirements in the development lifecycle is though a motivation to investigate potential systematic links to the safety evaluation flows and outcomes. The fulfillment of safety requirements is still assessed based on a tedious manual inspection of the analysis or simulation results. To overcome this inconvenience, formalization approaches for safety requirements and according tool-based traceability and assessment methodologies should be investigated.

Finally, in addition to functional safety, the semiconductor industry is increasingly confronted with another challenging dependability aspect, namely *security*. In the era of the Internet-of-Things (IoT), protecting electronic devices from malicious attacks becomes particularly necessary, when those devices contain highly sensitive and confidential data. The perceptions made in this thesis about the disorganization and formality of safety analysis procedures hold true for the assessment of security threats too. Therefore, the formalization and model-driven support approaches which have been developed in the context of this work to overcome persisting issues in the functional safety domain have a considerable potential of being extrapolated to the security area.

Bibliography

[1] International Electrotechnical Commission and others, "Functional safety of electrical/electronic/programmable electronic safety related systems," *IEC 61508*, 2000.

[2] International Organization for Standardization, "ISO 26262, Road vehicles–Functional safety," *International Standard ISO/FDIS*, vol. 26262, 2011.

[3] M. Chaari, W. Ecker, C. Novello, B.-A. Tabacaru, and T. Kruse, "A Model-Based and Simulation-Assisted FMEDA Approach for Safety-Relevant E/E Systems," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, pp. 1–6.

[4] M. Rausand, *Reliability of safety-critical systems: theory and applications.* Second edition, John Wiley & Sons. Online ISBN:9781118776353, 2014.

[5] M. Chaari, "Implementation of a Template Rendering Mechanism based on Model-driven C++ Code Generation," Master's Thesis, Chair of Electronic Design Automation - Technische Universität München, 2013.

[6] J. Miller, J. Mukerji, *et al.*, "MDA Guide Version 1.0.1," *Object Management Group - Document Number: omg/2003-06-01*, vol. 234, p. 51, 2003.

[7] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA explained, the model driven architecture: Practice and promise.* Addison-Wesley Professional, ISBN: 032119442X, 2003.

[8] C. Atkinson and T. Kuhne, "Model-driven development: a metamodeling foundation," *Software, IEEE*, vol. 20, no. 5, pp. 36–41, 2003.

[9] X. Thirioux, B. Combemale, X. Crégut, P.-L. Garoche, *et al.*, "A Framework to formalise the MDE Foundations," *Report for Towers' 07*, pp. 14–30, 2007.

[10] M. Voelter, "A Catalog of Patterns for Program Generation," in *Eighth European Conference on Pattern Languages of Programs (EuroPLoP)*, 2003, pp. 285–320.

[11] M. Chaari, W. Ecker, T. Kruse, and B.-A. Tabacaru, "Automation of failure propagation analysis through metamodeling and code generation."

ITG/GI/GMM-Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen (TuZ), 2015.

[12] M. Chaari, W. Ecker, T. Kruse, C. Novello, and B.-A. Tabacaru, "Transformation of failure propagation models into fault trees for safety evaluation purposes," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Industrial Track*. IEEE, 2016, pp. 226–229.

[13] SAE International, "ARP4761 - Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment," *SAE International, December*, 1996.

[14] M. Chaari, W. Ecker, B.-A. Tabacaru, C. Novello, and T. Kruse, "Linking Model-Based Safety Analysis to Fault Injection and Simulation in Virtual Prototypes." Electronic Design Automation Workshop (edaWorkshop 16), 2016.

[15] (2017, May) Infineon Technologies AG - Sensor Solutions for Automotive, Industrial and Consumer Applications. [Online]. Available: http://www.infineon.com/dgdl/Infineon-Sensor_Solutions_for_Automotive_Industrial_and+Customer_Appl_BR-2015.pdf?fileId=5546d4614937379a01495212845c039f

[16] C. Choi, W. Lee, J. Kim, and S. Kim, "Failure modes investigation and analysis of electric power steering system with PMSM drives," *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, vol. 2, no. 2009-01-0296, pp. 103–108, 2009.

[17] (2017, May) Infineon Technologies AG - Electric power steering (EPS). [Online]. Available: www.infineon.com/cms/en/applications/automotive/safety/eps/

[18] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[19] F. Waheed, "Automated generation of an embedded CPU from ISA with Power Management," Master's Thesis, Institute for Real-Time Computer Systems, Technische Universität München, 2014.

[20] J. Belzer, A. G. Holzman, and A. Kent, *Encyclopedia of Computer Science and Technology: Volume 9-Generative Epistemology of Problem Solving to Laplace and Geometric Transforms*. CRC Press, 1978, vol. 9.

[21] J.-C. Laprie, "Dependability: Basic Concepts and Terminology, volume 5 of Dependable Computing and Fault-Tolerant Systems," *Springer-Verlag*, 1992.

[22] A. Vega, P. Bose, and A. Buyuktosunoglu, *Rugged Embedded Systems: Computing in Harsh Environments*. Morgan Kaufmann, 2016.

[23] D. Heffernan, C. MacNamee, and P. Fogarty, "Runtime verification monitoring for automotive embedded systems using the ISO 26262 functional safety stan-

dard as a guide for the definition of the monitored properties," *Software, IET*, vol. 8, no. 5, pp. 193–203, 2014.

[24] N. Aeronautics and S. Administration. (2016, Feb.) Software Safety Criticality Assessment. [Online]. Available: http://www.hq.nasa.gov/office/codeq/doctree/NS871913C.docx

[25] Cadence. (2016, Feb.) Incisive Functional Safety Simulator. [Online]. Available: http://www.europractice.stfc.ac.uk/vendors/cadence_incisive_functional_safety_simulator_ds.pdf

[26] M. Bozzano and A. Villafiorita, *Design and safety assessment of critical systems.* CRC press, 2010.

[27] N. R. Storey, *Safety critical computer systems.* Addison-Wesley Longman Publishing Co., Inc., 1996.

[28] MTL Instruments Group plc, "An introduction to functional safety and IEC 61508," *Application Note: AN9025-3*, 2002.

[29] A. Fritsch, "Functional Safety and Explosion Protection," *Fundamentals of Functional Safety in Accordance with IEC*, vol. 61508, 2005.

[30] D. J. Smith and K. G. Simpson, *Functional Safety: A Straightforward Guide to Applying IEC 61508 and Related Standards.* Routledge, 2004.

[31] P. Baufreton, J. Derrien, B. Ricque, J. Blanquart, J. Boulanger, H. Delseny, J. Gassino, G. Ladier, E. Ledinot, M. Leeman, *et al.*, "Multi-domain comparison of safety standards," 2011.

[32] C. Ebert, "Implementing functional safety," *IEEE Software*, no. 5, pp. 84–89, 2015.

[33] M. Schmidt, M. Rau, E. Helmig, and B. Bauer, "Functional Safety–Dealing with Independency, Legal Framework Conditions and Liability Issues," *Official Journal of the European Union dated*, vol. 50, no. 200/1, 2009.

[34] B. J. Czerny, J. DAmbrosio, and R. Debouk, "ISO 26262 Functional safety draft international standard for road vehicles: Background, Status, and Overview," *Origins*, vol. 9, no. 1.2004, p. 2003, 2002.

[35] T. A. Kletz, *Hazop and Hazan: Identifying and Assessing Process Industry Hazards.* IChemE, 1999.

[36] B. C. Wei, "A unified approach to failure mode, effects and criticality analysis (FMECA)," in *Reliability and Maintainability Symposium, 1991. Proceedings., Annual.* IEEE, 1991, pp. 260–271.

[37] D. Lawson, “Failure mode, effect and criticality analysis,” in *Electronic Systems Effectiveness and Life Cycle Costing.* Springer, 1983, pp. 55–74.

[38] J. C. Grebe and W. M. Goble, “FMEDA–accurate product failure metrics,” *exida, Sellersville, PA*, vol. 18960, 2007.

[39] W. M. Goble and A. Brombacher, “Using a failure modes, effects and diagnostic analysis (FMEDA) to measure diagnostic coverage in programmable electronic systems,” *Reliability engineering & system safety*, vol. 66, no. 2, pp. 145–148, 1999.

[40] W. M. Goble, *Control systems safety evaluation and reliability.* International Society of Automation, 2010.

[41] H. Czichos, “Scope of technical diagnostics,” in *Handbook of Technical Diagnostics.* Springer, 2013, pp. 3–9.

[42] C. A. Ericson and C. Ll, “Fault tree analysis,” *Hazard analysis techniques for system safety*, pp. 183–221, 2000.

[43] A. Kumar, R. K. Gupta, and A. Kumar, “Fault tree analysis of different systems,” School of mathematics and computer applications. Thapar University, Patiala (Punjab), 2009.

[44] G. K. Palshikar, “Temporal fault trees,” *Information and Software Technology*, vol. 44, no. 3, pp. 137–150, 2002.

[45] J. Bechta Dugan, S. J. Bavuso, and M. A. Boyd, “Dynamic fault-tree models for fault-tolerant computer systems,” *Reliability, IEEE Transactions on*, vol. 41, no. 3, pp. 363–377, 1992.

[46] J. D. Andrews and S. J. Dunnett, “Event-tree analysis using binary decision diagrams,” *Reliability, IEEE Transactions on*, vol. 49, no. 2, pp. 230–238, 2000.

[47] J. V. Bukowski and W. M. Goble, “Using Markov models for safety analysis of programmable electronic systems,” *ISA Transactions*, vol. 34, no. 2, pp. 193–198, 1995.

[48] W. Wang, J. M. Loman, R. G. Arno, P. Vassiliou, E. R. Furlong, and D. Ogden, “Reliability block diagram simulation techniques applied to the IEEE std. 493 standard network,” *Industry Applications, IEEE Transactions on*, vol. 40, no. 3, pp. 887–895, 2004.

[49] S. Distefano and A. Puliafito, “Dependability evaluation with dynamic reliability block diagrams and dynamic fault trees,” *Dependable and Secure Computing, IEEE Transactions on*, vol. 6, no. 1, pp. 4–17, 2009.

[50] S. Bernardi, J. Merseguer, and D. C. Petriu, “Dependability analysis tech-

niques," in *Model-Driven Dependability Assessment of Software Systems.* Springer, 2013, pp. 73–90.

[51] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[52] A. Benso and P. Prinetto, *Fault injection techniques and tools for embedded systems reliability evaluation.* Springer, 2003, vol. 23.

[53] H. Ziade, R. A. Ayoubi, R. Velazco, *et al.*, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.

[54] A. Herkersdorf, M. Engel, M. Glaß, J. Henkel, V. B. Kleeberger, M. Kochte, J. M. Kühn, S. R. Nassif, H. Rauchfuss, W. Rosenstiel, *et al.*, "Cross-layer dependability modeling and abstraction in system on chip," in *Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2013.

[55] A. Herkersdorf, H. Aliee, M. Engel, M. Glaß, C. Gimmler-Dumont, J. Henkel, V. B. Kleeberger, M. A. Kochte, J. M. Kühn, D. Mueller-Gritschneder, *et al.*, "Resilience articulation point (rap): Cross-layer dependability modeling for nanometer system-on-chip resilience," *Microelectronics Reliability*, vol. 54, no. 6, pp. 1066–1074, 2014.

[56] U. Schlichtmann, V. B. Kleeberger, J. A. Abraham, A. Evans, C. Gimmler-Dumont, M. Glaß, A. Herkersdorf, S. R. Nassif, and N. Wehn, "Connecting different worlds: Technology abstraction for reliability-aware design and test," in *Proceedings of the conference on Design, Automation & Test in Europe.* European Design and Automation Association, 2014, p. 252.

[57] M. Chaari, W. Ecker, and B.-A. Tabacaru, "Towards Cross-Domain and Multi-Level Dependability Analysis Through Metamodeling and Code Generation." Electronic Design Automation Workshop (edaWorkshop 16), 2016.

[58] V. Basili, P. Donzelli, and S. Asgari, "A unified model of dependability: Capturing dependability in context," *Software, IEEE*, vol. 21, no. 6, pp. 19–25, 2004.

[59] G. Dobson and P. Sawyer, "Revisiting ontology-based requirements engineering in the age of the semantic web," in *Proceedings of the International Seminar on Dependable Requirements Engineering of Computerised Systems at NPPs*, 2006.

[60] A. Pataricza and F. Györ, "Towards Unified Dependability Modeling and Analysis," in *ARCS Workshops*, 2004, pp. 113–122.

[61] S. Bernardi, J. Merseguer, and D. C. Petriu, "Dependability modeling and

analysis of software systems specified with UML," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 2, 2012.

[62] D. Sojer, *Synthesis of Online Diagnostic Techniques for Embedded Systems.* Verlag Dr. Hut, 2013.

[63] J. L. de la Vara and R. K. Panesar-Walawege, "SafetyMet: A metamodel for safety standards," in *Model-Driven Engineering Languages and Systems.* Springer, 2013, pp. 69–86.

[64] O. Lisagor, J. McDermid, and D. Pumfrey, "Towards a practicable process for automated safety analysis," in *24th International system safety conference.* Citeseer, 2006, pp. 596–607.

[65] L. Grunske and J. Han, "A comparative study into architecture-based safety evaluation methodologies using AADL's Error Annex and failure propagation models," in *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE.* IEEE, 2008, pp. 283–292.

[66] O. Lisagor, "Failure logic modelling: A pragmatic approach," Ph.D. dissertation, University of York, 2010.

[67] P. Fenelon and J. A. McDermid, "An integrated tool set for software safety analysis," *Journal of Systems and Software*, vol. 21, no. 3, pp. 279–290, 1993.

[68] P. Fenelon, J. A. McDermid, M. Nicolson, and D. J. Pumfrey, "Towards integrated safety analysis and design," *ACM SIGAPP Applied Computing Review*, vol. 2, no. 1, pp. 21–32, 1994.

[69] M. Wallace, "Modular architectural representation and analysis of fault propagation and transformation," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 3, pp. 53–71, 2005.

[70] Y. Papadopoulos and J. A. McDermid, "Hierarchically performed hazard origin and propagation studies," in *Computer Safety, Reliability and Security.* Springer, 1999, pp. 139–152.

[71] Y. Papadopoulos and J. McDermid, "Safety-directed system monitoring using safety cases," Ph.D. dissertation, University of York, 2000.

[72] Y. Papadopoulos, J. McDermid, R. Sasse, and G. Heiner, "Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure," *Reliability Engineering & System Safety*, vol. 71, no. 3, pp. 229–247, 2001.

[73] Y. Papadopoulos and M. Maruhn, "Model-based synthesis of fault trees from Matlab-simulink models," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on.* IEEE, 2001, pp. 77–82.

[74] A. Hassan, K. Goseva-Popstojanova, and H. Ammar, "UML based severity analysis methodology," in *Reliability and Maintainability Symposium, 2005. Proceedings. Annual.* IEEE, 2005, pp. 158–164.

[75] S. Lu and W. A. Halang, "A UML profile to model safety-critical embedded real-time control systems," in *Contributions to Ubiquitous Computing.* Springer, 2007, pp. 197–218.

[76] G. Zoughbi, L. Briand, and Y. Labiche, *A UML profile for developing airworthiness-compliant (RTCA DO-178B), safety-critical software.* Springer, 2007.

[77] A. Joshi, S. P. Miller, M. Whalen, and M. P. Heimdahl, "A proposal for model-based safety analysis," in *Digital Avionics Systems Conference, 2005. DASC 2005. The 24th*, vol. 2. IEEE, 2005, pp. 13–pp.

[78] A. Joshi and M. P. Heimdahl, "Model-based safety analysis of Simulink models using SCADE design verifier," in *Computer Safety, Reliability, and Security.* Springer, 2005, pp. 122–135.

[79] A. Joshi and M. P. E. Heimdahl, "Behavioral fault modeling for model-based safety analysis," in *High Assurance Systems Engineering Symposium, 2007. HASE'07. 10th IEEE.* IEEE, 2007, pp. 199–208.

[80] M. Bozzano, A. Villafiorita, P. Bieber, C. Bougnol, M. Bretschneider, A. Cavallo, C. Castel, M. Cifaldi, *et al.*, "ESACS: an integrated methodology for design and safety analysis of complex systems," 2003.

[81] M. Bozzano and A. Villafiorita, "Improving system reliability via model checking: The FSAP/NuSMV-SA safety analysis platform," in *Computer Safety, Reliability, and Security.* Springer, 2003, pp. 49–62.

[82] A. Majdara and T. Wakabayashi, "Component-based modeling of systems for automated fault tree generation," *Reliability Engineering & System Safety*, vol. 94, no. 6, pp. 1076–1086, 2009.

[83] N. H. Ulerich and G. J. Powers, "On-line hazard aversion and fault diagnosis in chemical processes: the digraph+ fault-tree method," *Reliability, IEEE Transactions on*, vol. 37, no. 2, pp. 171–177, 1988.

[84] Y. Wang, T. Teague, H. West, and S. Mannan, "A new algorithm for computer-aided fault tree synthesis," *Journal of Loss Prevention in the Process Industries*, vol. 15, no. 4, pp. 265–277, 2002.

[85] J. D. Andrews and J. Henry, "A computerized fault tree construction methodology," *Proceedings of the Institution of Mechanical Engineers, Part E: Journal of Process Mechanical Engineering*, vol. 211, no. 3, pp. 171–183, 1997.

[86] P. Liggesmeyer and M. Rothfelder, “Improving system reliability with automatic fault tree generation,” in *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on.* IEEE, 1998, pp. 90–99.

[87] A. Rauzy, “Mode automata and their compilation into fault trees,” *Reliability Engineering & System Safety*, vol. 78, no. 1, pp. 1–12, 2002.

[88] Y. Papadopoulos, J. McDermid, A. Mavrides, C. Scheidler, and M. Maruhn, “Model-based semiautomatic safety analysis of programmable systems in automotive applications,” in *Advanced Driver Assistance Systems, 2001. ADAS. International Conference on (IEE Conf. Publ. No. 483).* IET, 2001, pp. 53–57.

[89] P. David, V. Idasiak, F. Kratz, *et al.*, “Towards a better interaction between design and dependability analysis: FMEA derived from UML/SysML models,” in *Proceedings of ESREL 2008 and 17th SRA-EUROPE annual conference*, 2008.

[90] L. Grunske, P. Lindsay, N. Yatapanage, and K. Winter, “An automated failure mode and effect analysis based on high-level design specification with behavior trees,” in *Integrated Formal Methods.* Springer, 2005, pp. 129–149.

[91] J. Elmqvist and S. Nadjm-Tehrani, “Tool support for incremental failure mode and effects analysis of component-based systems,” in *Proceedings of the conference on Design, automation and test in Europe.* ACM, 2008, pp. 921–927.

[92] J. Elmqvist, S. Nadjm-Tehrani, and M. Minea, “Safety interfaces for component-based systems,” in *Computer Safety, Reliability, and Security.* Springer, 2005, pp. 246–260.

[93] M. A. de Miguel, J. F. Briones, J. P. Silva, and A. Alonso, “Integration of safety analysis in model-driven software development,” *Software, IET*, vol. 2, no. 3, pp. 260–280, 2008.

[94] N. Bidokhti, “FMEA is not enough,” in *Reliability and Maintainability Symposium (RAMS), 2009.* IEEE, 2009, pp. 333–337.

[95] (2015, Mar.) YOGITECH S.p.A. Official website. [Online]. Available: http://www.yogitech.com/en

[96] L. Pintard, J.-C. Fabre, M. Leeman, K. Kanoun, and M. Roy, “From safety analyses to experimental validation of automotive embedded systems,” in *Dependable Computing (PRDC), 2014 IEEE 20th Pacific Rim International Symposium on.* IEEE, 2014, pp. 125–134.

[97] L. Pintard, “From safety analysis to experimental validation by fault injection-case of automotive embedded systems,” Ph.D. dissertation, INP Toulouse, 2015.

[98] R. Mariani, P. Fuhrmann, and B. Vittorelli, “Fault-robust microcontrollers for automotive applications,” in *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International.* IEEE, 2006, pp. 6–pp.

[99] R. Mariani, G. Boschi, and F. Colucci, “Using an innovative SoC-level FMEA methodology to design in compliance with IEC61508,” in *Proceedings of the conference on Design, automation and test in Europe.* EDA Consortium, 2007, pp. 492–497.

[100] R. Mariani and G. Boschi, “A systematic approach for failure modes and effects analysis of system-on-chips,” in *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International.* IEEE, 2007, pp. 187–188.

[101] R. Mariani, S. Motto, and M. Chiavacci, “Dependable microcontroller, method for designing a dependable microcontroller and computer program product therefor,” Dec. 30 2008, US Patent 7,472,051.

[102] R. Mariani, “Method for performing failure mode and effects analysis of an integrated circuit and computer program product therefor,” May 3 2011, uS Patent 7,937,679.

[103] K. F. Greb, A. Arora, and R. Mariani, “Tool for automation of functional safety metric calculation and prototyping of functional safety systems,” Sept. 7 2013, US Patent App. 14/020,802.

[104] B. Hailpern and P. Tarr, “Model-driven development: The good, the bad, and the ugly,” *IBM systems journal*, vol. 45, no. 3, pp. 451–461, 2006.

[105] S. J. Mellor, A. N. Clark, and T. Futagami, “Model-driven development,” *IEEE software*, pp. 14–18, 2003.

[106] R. France and B. Rumpe, “Model-driven development of complex software: A research roadmap,” in *2007 Future of Software Engineering.* IEEE Computer Society, 2007, pp. 37–54.

[107] A. MacDonald, D. Russell, and B. Atchison, “Model-driven development within a legacy system: an industry experience report,” in *Software Engineering Conference, 2005. Proceedings. 2005 Australian.* IEEE, 2005, pp. 14–22.

[108] J. E. Fernandes and R. J. Machado, “Model-driven software development for pervasive information systems implementation,” in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the.* IEEE, 2007, pp. 218–222.

[109] W. Ecker, W. Müller, and R. Dömer, “Hardware-dependent software,” in *Hardware-dependent Software.* Springer, 2009, pp. 1–13.

[110] B.-A. Tabacaru, M. Chaari, W. Ecker, T. Kruse, and C. Novello, “Gate-level-

accurate fault-effect analysis at virtual-prototype speed," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2016, pp. 144–156.

[111] S. Shukla, "Metamodeling: What is it good for?" *Design & Test of Computers, IEEE*, vol. 26, no. 3, pp. 96–96, 2009.

[112] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE software*, vol. 20, no. 5, pp. 42–45, 2003.

[113] D. Becker, W. McMullen, and K. Hetherington-Young, "A flexible and generic data quality metamodel," in *ICIQ*, 2007, pp. 50–64.

[114] D. A. Tamburri and P. Lago, "Supporting communication and cooperation in global software development with agile service networks," in *Software Architecture*. Springer, 2011, pp. 236–243.

[115] J. Tolvanen and S. Kelly, "Domain-Specific Modeling Languages for Embedded System Development," in *ESWeek Workshop*, 2012, p. 11.

[116] A. Sangiovanni-Vincentelli, G. Yang, S. K. Shukla, D. A. Mathaikutty, and J. Sztipanovits, "Metamodeling: An emerging representation paradigm for system-level design," *Design & Test of Computers, IEEE*, vol. 26, no. 3, pp. 54–69, 2009.

[117] M. J. Emerson, J. Sztipanovits, and T. Bapty, "A MOF-Based Metamodeling Environment," *J. UCS*, vol. 10, no. 10, pp. 1357–1382, 2004.

[118] SPIRIT, "IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows," *IEEE Std 1685-2009*, pp. C1–360, 2010.

[119] M. Zyś, E. Vaumorin, and I. Sobański, "Straightforward IP Integration with IP-XACT RTL-TLM Switching," in *Design Automation Conference (DAC), USA*, 2008.

[120] W. Ecker, M. Velten, L. Zafari, and A. Goyal, "Metasynthesis for Designing Automotive SoCs," in *Proceedings of The 51st Annual Design Automation Conference on Design Automation Conference*. ACM, 2014, pp. 1–6.

[121] S. SIEMENS AG, "Reliability and quality specifications failure rates of components," 2005.

[122] "Electric components - Reliability - Reference conditions for failure rates and stress models for conversion (IEC 61709)," 2011.

[123] "Electrical & Mechanical Component Reliability Handbook, exida, Sellersville, PA," 2006.

[124] International Electrotechnical Commission and others, "IEC 61025 Fault Tree Analysis," 1990.

[125] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault tree handbook," DTIC Document, Tech. Rep., 1981.

[126] W. Ecker, M. Velten, L. Zafari, and A. Goyal, "Metamodeling and code generation - The Infineon approach," in *ESWeek Workshop MeCoES*, 2012, pp. 1–4.

[127] A. Burger, W. Ecker, O. Bringmann, and W. Rosenstiel, "Meta constraints for consistency checks of embedded system specifications," in *ESWeek Workshop MeCoES*, 2012, pp. 58–66.

[128] Mako. (2013, Aug.) Mako 0.9.0 Documentation. [Online]. Available: http://docs.makotemplates.org/en/latest/

[129] Python. (2013, Aug.) Templating in Python. [Online]. Available: http://wiki.python.org/moin/Templating

[130] M. Van den Brand and A. Serebrenik, *Code generation with templates*. Springer, 2012, vol. 1.

[131] R. Koebler. (2013, Aug.) Template Engines. [Online]. Available: http://www.simple-is-better.org/template/

[132] M. Bozzano, A. Cimatti, A. F. Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta, "Formal design and safety analysis of air6110 wheel brake system," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 518–535.

[133] A. Joshi, M. P. Heimdahl, S. P. Miller, and M. W. Whalen, "Model-based safety analysis," Department of Computer Science and Engineering, University of Minnesota, 2006.

[134] GraphML. (2015, Feb.) http://graphml.graphdrawing.org/. [Online]. Available: http://graphml.graphdrawing.org/

[135] Isograph. (2015, Aug.) . [Online]. Available: http://www.isograph.com/software/reliability-workbench/

[136] edacentrum. (2016, Oct.) Research Project EffektiV. [Online]. Available: https://www.edacentrum.de/effektiv/

[137] M. Chaari, B.-A. Tabacaru, W. Ecker, C. Novello, and T. Kruse, "Bridging the gap between probabilistic safety analysis and fault injection in virtual pro-

totypes," in *1st International Workshop on Resiliency in Embedded Electronic Systems, Amsterdam, The Netherlands*, 2015, pp. 34–35.

[138] M. Chaari, W. Ecker, T. Kruse, C. Novello, and B.-A. Tabacaru, "Efficient exploration of safety-relevant systems through a link between analysis and simulation," in *Design and Verification Conference & Exhibition DVCon Europe*, 2016.

[139] J.-H. Oetjens, N. Bannow, M. Becker, O. Bringmann, A. Burger, M. Chaari, S. Chakraborty, R. Drechsler, W. Ecker, K. Grüttner, *et al.*, "Safety evaluation of automotive electronics using virtual prototypes: State of the art and research challenges," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE.* IEEE, 2014, pp. 1–6.

[140] B.-A. Tabacaru, M. Chaari, W. Ecker, and T. Kruse, "A meta-modeling-based approach for automatic generation of fault-injection processes," *DVCon Europe*, pp. 1–7, 2014.

[141] ——, "Runtime fault-injection tool for executable SystemC models," *DVCon India*, 2014.

[142] S. Reiter, M. Becker, O. Bringmann, A. Burger, M. Chaari, R. Drechsler, W. Ecker, T. Kruse, C. Kuznik, J. Laufenberg, *et al.*, "Fehlereffektsimulation mittels virtueller prototypen," 2015.

[143] Eclipse Modeling Framework. (2015, Feb.) EMF. [Online]. Available: https://eclipse.org/modeling/emf/

[144] Eclipse. (2015, Feb.) EMF Forms. [Online]. Available: http://www.eclipse.org/ecp/emfforms/

[145] C. Bolchini, A. Miele, and D. Sciuto, "Fault models and injection strategies in SystemC specifications," in *Digital System Design Architectures, Methods and Tools, 2008. DSD'08. 11th EUROMICRO Conference on.* IEEE, 2008, pp. 88–95.

[146] G. Beltrame, C. Bolchini, and A. Miele, "Multi-level fault modeling for transaction-level specifications," in *Proceedings of the 19th ACM Great Lakes symposium on VLSI.* ACM, 2009, pp. 87–92.

[147] B.-A. Tabacaru, M. Chaari, W. Ecker, T. Kruse, K. Liu, N. Hatami, C. Novello, H. Post, and A. von Schwerin, "Fault-Injection Techniques for TLM-Based Virtual Prototypes," in *Forum on Specification and Design Languages (FDL) – Work in Progress (WiP)*, 2015, pp. 1–4.

[148] J. D. Musa, "The operational profile in software reliability engineering: an

overview," in *Software Reliability Engineering, 1992. Proceedings., Third International Symposium on.* IEEE, 1992, pp. 140–154.

[149] J. Reimpell, H. Stoll, and J. Betzler, *The automotive chassis: engineering principles.* Elsevier, 2001.

[150] S. Oshita, T. Mouri, and Y. Uemura, "Electric power steering system," May 12 1987, US Patent 4,664,211.

[151] M. Würges, "New electrical power steering systems," *Encyclopedia of Automotive Engineering*, pp. 1–17, 2014.

[152] O. Wallmark, L. Harnefors, and O. Carlson, "Control algorithms for a fault-tolerant pmsm drive," in *31st Annual Conference of IEEE Industrial Electronics Society, 2005. IECON 2005.* IEEE, 2005, pp. 7–pp.

[153] Z. Rahman, M. Ehsani, and K. Butler, "Effect of motor short circuit on EV and HEV traction systems," SAE Technical Paper, Tech. Rep., 2000.

[154] A. El-Antably, X. Luo, and R. Martin, "System simulation of fault conditions in the components of the electric drive system of an electric vehicle or an industrial drive," in *Industrial Electronics, Control, and Instrumentation, 1993. Proceedings of the IECON'93., International Conference on.* IEEE, 1993, pp. 1146–1150.

[155] N. Bianchi, S. Bolognani, and M. Zigliotto, "Analysis of PM synchronous motor drive failures during flux weakening operation," in *Power Electronics Specialists Conference, 1996. PESC'96 Record., 27th Annual IEEE*, vol. 2. IEEE, 1996, pp. 1542–1548.

[156] B. Cui, "Simulation of inverter with switch open faults based on switching function," in *2007 IEEE International Conference on Automation and Logistics.* IEEE, 2007, pp. 2774–2778.

[157] G. D. González, J. G.-A. Fernández, and P. A. Arboleya, "Diagnosis of a turn-to-turn short circuit in power transformers by means of zero sequence current analysis," *Electric Power Systems Research*, vol. 69, no. 2, pp. 321–329, 2004.

[158] (2017) MIPS Technologies Inc., Mountain View, CA, USA. [Online]. Available: http://www.mips.com

[159] D. De Andrés, J.-C. Ruiz, D. Gil, and P. Gil, "Towards dependability benchmarking of hardware cores for embedded systems," in *Workshop on Resilience Assessment and Dependability Benchmarking at International Conference on Dependable Systems and Networks.* Citeseer, pp. 24–27.

[160] B. Backs, "Power Aware Generation of an Embedded CPU," Master's Thesis,

Institute for Real-Time Computer Systems, Technische Universität München, 2014.

[161] A. Avizienis, J.-C. Laprie, B. Randell, *et al.*, *Fundamental concepts of dependability.* University of Newcastle upon Tyne, Computing Science, 2001.

[162] T.-L. Chu, M. Yue, and W. Postma, "A summary of taxonomies of digital system failure modes provided by the digrel task group," Brookhaven National Laboratory (BNL), Tech. Rep., 2012.

[163] T. Malm and M. Kivipuro, *Safety validation of complex components: Validation by analysis.* VTT Technical Research Centre of Finland, 2000.

[164] K. Korsah, M. Muhlheim, and D. Holcomb, "Industry Survey of Digital I&C Failures," Citeseer, 2007.

[165] W. R. Dunn, "Practical design of safety-critical computer systems," Reliability Press, 2002.

[166] M. Chaari, W. Ecker, T. Kruse, and B.-A. Tabacaru, "Automation of failure propagation analysis through metamodeling and code generation." ITG/GI/GMM-Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen (TuZ), 2015.

[167] B.-A. Tabacaru, M. Chaari, W. Ecker, T. Kruse, and C. Novello, "Safety Analysis on Multiple Abstraction Levels." Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016.

[168] ——, "Fault-effect analysis on system-level hardware modeling using virtual prototypes," in *Specification and Design Languages (FDL), 2016 Forum on.* IEEE, 2016, pp. 1–7.

[169] ——, "Speeding up safety verification by fault abstraction and simulation to transaction level," in *Very Large Scale Integration (VLSI-SoC), 2016 IFIP/IEEE International Conference on.* IEEE, 2016, pp. 1–6.

[170] G. G. Wang, "Definition and review of virtual prototyping," *J. Comput. Inf. Sci. Eng.*, vol. 2, no. 3, pp. 232–236, 2002.

9 Glossary

ASIL decomposition

redundant apportioning of safety requirements to different elements which are sufficiently independent, aiming at the reduction of the ASIL of the redundant safety requirements allocated to the independent elements [2] (35, 36, 197)

ASIL tailoring

modification of assigned ASILs to safety requirements through ASIL decomposition [2] (35)

availability

dependability of a system with respect to the readiness for correct service [21] (1, 2, 57)

cascading failure

type of dependent failure depicting a failure of an item element which causes the failure one or more other element of the same item [2] (48, 49, 110, 200)

common cause failure

type of dependent failure depicting a failure of two or more elements of an item which results from a single specific event or root cause [2] (48, 49, 110)

component

non-system level element which can be logically and/or technically separated from other elements and which includes one or more hardware parts or one or more software units [2] (29, 203)

configuration management

discipline of identifying the components of an evolving system to control changes and maintain continuity and traceability throughout the lifecycle [1] (30)

deductive analysis

safety analysis performed top-down starting from problematic situations on system-level and looking for the corresponding causes in the components (9, 45, 48, 49, 95, 102, 178)

dependability

the ability of a computing system to deliver a service that can be justifiably trusted [21] (1–4, 53–57, 149, 170, 171, 177, 181, 197, 203, 204, 207)

dependent failure

dependent failures are failures whose probability of simultaneous or successive occurrence cannot be simply expressed as the product of the independent occurrence probabilities of each of them [2] (44, 48, 49, 108–111, 145, 197)

diagnostic coverage

proportion of the failure rate of a hardware element which is detected or controlled by the implemented safety mechanisms, in other words fraction of dangerous failures detected by automatic diagnostic tests [1, 2] (9, 10, 43, 46, 49, 61, 62, 94, 99, 101, 102, 134, 143, 145, 146, 166, 167)

diagnostic point

location in a hardware part or in a software unit where the response of an already implemented safety measure can be monitored (9, 10)

dual-point fault

single fault which in combination with another independent fault leads to a so-called dual-point failure resulting directly into a safety goal violation [2] (99, 211)

E/E system

system consisting of Electrical and/or (potentially programmable) Electronic elements [2] (1–5, 11, 13, 22, 177, 200, 205, 206)

E/E/PE system

system for control, protection or monitoring based on one or more Electrical / Electronic / Programmable Electronic devices, including power supplies, sensors, communication paths, and actuators [1] (22, 23, 27–29, 33, 200, 205–207)

element

system or part of a system including hardware parts as well as software units and performing one or more safety functions [1, 2] (29, 45, 197–199, 201, 203, 204, 206, 207, 209)

Equipment Under Control

equipment, machinery, apparatus or plant used for manufacturing, process, transportation, medical activities, etc. [1] (22, 205, 212)

error

discrepancy between a computed, observed or measured value or condition and the true, specified, or theoretically correct value or condition [1, 2] (24, 48, 50, 51, 53, 84, 144, 149, 171, 172)

failure

non-compliance of a system's delivered service with its specified service [21], in other words the inability of an given element or system to perform its required function [2] (3, 4, 9, 19–22, 24, 26, 28, 30, 35, 39, 41, 43–51, 53–60, 69, 81, 84–94, 96, 99, 100, 103, 104, 106, 110, 112, 144, 156, 157, 159, 160, 162–164, 171, 172, 177–180, 197, 198, 202, 203, 206–208)

failure catalogue

artefact or document where all known failure modes of a certain technology, of an element or system family, or within an organization are compiled and classified in a structured way (e.g., database) (14, 134, 135, 145)

failure effect

consequence of a the occurrence of a failure mode within an element (8, 62, 68, 94, 100, 101, 143, 144, 153)

failure mode

way in which an element, a system, or an item fails [21, 2] (14, 47, 49, 56, 58, 60, 62, 68, 69, 97–100, 102, 111, 112, 131, 133–135, 143–145, 150, 152, 153, 159, 160, 167, 170–172, 174–176, 199, 200, 204)

failure rate

frequency of element or system failure, expressed in failures per unit of time (FIT) and commonly denoted by the Greek letter λ (9, 14, 20, 43, 46, 47, 60, 94, 96, 98–100, 102, 103, 134–136, 145, 198, 203, 204)

fault

abnormal condition, such as defect in a hardware part (e.g., short circuit, broken wire, etc.), that might lead to an element or item failure [2] (3, 9–11, 24, 37–39, 43, 44, 48, 50, 51, 53, 54, 58, 60, 61, 68, 69, 81, 84, 99, 106, 144, 145, 151, 159–161, 164, 166, 167, 171–176, 198, 200, 203, 204, 206, 208)

fault effect simulation

simulation-based safety evaluation technique applied in correlation with fault injection to monitor the effects of faults on systems (14)

fault injection

deliberate introduction of faults into a target system [52], and execution of controlled experiments where the system's behavior is observed in presence of those faults (vii, 5, 9–16, 32, 50, 51, 53, 61–63, 65, 68–70, 84, 85, 94, 96, 113, 114, 117, 118, 129, 143, 144, 146, 148–151, 153, 155, 156, 165–167, 170, 173–175, 177, 180, 200, 203, 204, 207–209)

fault modeling

representation of failure modes resulting from faults which is generally based on field experience or on reliability handbooks [2] (14)

fault tolerant time interval

time-frame in which the presence of one or more faults in a system can be tolerated before the occurrence of a hazardous event [2] (37, 40)

fault tree

acyclic graphical representation used in the Fault Tree Analysis (FTA) scope including gates and logical connectors to structure successive levels of events [41, 42, 43] (48, 49, 56–59, 68, 69, 87, 96, 103, 104, 106–108, 114, 115, 129, 136–142, 156, 157, 162, 164, 178, 179)

freedom of interference

absence of cascading failures between two or more elements which have the potential to violate a safety requirement [2] (48, 49)

functional safety

part of the overall safety that depends on a system or equipment operating correctly in response to its inputs, more precisely absence of unreasonable risk due to hazards caused by malfunctioning behavior of E/E system [1, 2] (vii, 1, 2, 4, 5, 11, 16, 19–21, 26–30, 32, 33, 38, 41, 47, 49, 54, 60–62, 69, 82, 84, 89, 92, 94, 155, 158, 159, 177, 179, 181, 200)

functional safety assessment

investigation based on evidence to judge the functional safety achieved by one or more safety-related E/E/PE systems and/or other risk reduction measures [1] (26, 32, 41)

functional safety concept

set of functional safety requirements along with their associated information, their allocation to architectural elements, and their necessary interactions to fulfill the safety goals [2] (7, 29, 31, 32, 38–41, 44–46, 87, 105, 118, 138, 139, 146, 152, 163, 178)

functional safety requirement

implementation-independent specification of a safety behavior or a safety measure derived from the analysis of the safety goals and including all relevant safety-related attributes such as the corresponding ASIL [2] (7, 29, 31, 35, 37–42, 44, 46, 87, 94, 201, 207, 209)

functional verification

procedure applied to determine whether a given design conforms to its specification and fulfills all its functional requirements (5, 10, 11)

hard error

permanent change to the physical circuit [1] (100)

hardware part

piece of hardware that cannot be sub-divided [2] (8, 29, 32, 40, 41, 43, 46, 197, 198, 203, 207, 208)

harm

physical injury or damage to human health [2] (7, 19–21, 29, 33–35, 37, 39, 42, 201, 204, 208, 209)

hazard

possible source of harm resulting from a malfunctioning behavior of an item [2] (2, 7, 10, 20, 21, 31, 33, 35–37, 44, 46, 49, 56, 123, 201, 205, 208)

Hazard Analysis and Risk Assessment

method to identify and categorize hazardous events of items and to specify safety goals and ASILs related to the prevention or mitigation of the associated hazards in order to avoid unreasonable risk [2] (29, 31, 33, 35, 36, 42, 44, 84, 205, 213)

hazardous event

hazard combined with an operational situation that might occur during the product (e.g., vehicle) lifetime (e.g., parking, driving, maintenance, etc.) [2] (7, 21, 31, 33–35, 37, 47, 58, 103, 145, 200, 201, 205)

inductive analysis

safety analysis performed bottom-up starting from local component malfunctions and moving towards the corresponding effects on the complete system (9, 45, 47, 49, 95, 96, 178)

item

system or group of systems that implements a function at the vehicle level, to which ISO 26262 is applied [2] (21, 29–33, 35–38, 40–45, 197, 199, 201–206)

latent fault

multiple-point fault whose presence is not detected by a safety mechanism nor perceived by the driver (within a certain time interval) [2] (43, 99, 202)

latent fault metric

metric reflecting a system's robustness against latent faults, which is evaluated with respect to the coverage offered by safety mechanisms, to the recognition of faults by the driver before the safety goal violation, and to the properties of the design itself enabling a primarily high amount of safe faults (99, 100, 206, 213)

maintainability

ability of a system under stated usage conditions, to be retained and/or restored to a state where it can deliver its required service (57)

malfunctioning behavior

failure or unintended behavior of an item [2] (200, 201)

Markov model

stochastic models, which are represented as state diagrams, and used to model changing systems over time under the assumption that the future state only depends on the current state (i.e., the sequence of previous states is not relevant for the determination of the new state) [47] (45, 49)

metamodel

abstraction of a given model depicting its properties, describing the relationships between its elements, and defining the constraints they must comply to (vii, 54, 55, 63, 70, 73–76, 83–85, 87, 88, 90–94, 96, 101, 102, 106–111, 114–122, 130, 131, 133–135, 137, 138, 140, 144, 145, 149, 150, 155, 157, 163, 165, 175, 177–180)

multiple-point fault

fault which, in combination with other independent faults, leads to a multiple-point failure which results in the violation of a safety goal [2] (43, 99, 202, 214)

mutant

altered version of a system component used in the fault injection scope to cause a non-compliant behavior with the system specification [51, 52, 53] (51)

New Symbolic Model Verifier

symbolic model checker which supports state-of-the art verification techniques, such as those based on Binary Decision Diagrams (BDD) or on Boolean Satisfiability (SAT) [81] (214)

observation point

location in a hardware part or in a software unit where the effect of an injected fault can be monitored (9)

operating mode

functional state that can be perceived on a specific item or element (e.g., system active, system off, degraded operation, etc.) [2] (22, 33, 37, 40, 204)

operational

a system is considered operational when it correctly delivers the service which is expected from it (2, 22, 33–35, 49)

preliminary architectural assumption

supporting data in the context of the ISO 26262 which is derived from external sources (e.g., customer requests) and which provides information about the intended system architecture at an early development stage [2] (7, 31, 36, 39–41)

random hardware failure

failure unpredictably occurring during the hardware lifetime and following a probability distribution that can be estimated through hardware failure rates [2] (26, 28, 37, 39, 42–45, 48, 49, 97, 106, 206)

reliability

dependability of a system with respect to the continuity of its service delivery [21] (1, 2, 14, 57, 134, 145, 167, 177, 200)

residual fault

portion of faults related to a failure mode with potential to violate a safety goal which is not covered by the safety mechanism implemented to mitigate that failure mode [2] (43, 98, 99, 208, 215)

residual risk

remaining risk after safety measures are deployed [2] (7, 21, 29, 35, 37)

risk

probability of occurrence of harm combined with the severity of that specific harm [2] (6–8, 19–22, 24, 26, 28, 33–35, 37, 38, 41, 47, 49, 200, 204, 206, 207, 209)

robustness

ability of a system to function correctly even in the presence of invalid inputs and/or stressful environmental conditions and to provide safe behavior at its boundaries [2] (2, 3, 53, 84, 99)

saboteur

additional piece of hardware or software used in the fault injection scope which gets activated only when a fault is being injected in order to alter the correct values of specified signals or to modify certain timing characteristics [51, 52, 53] (51)

safe fault

fault whose occurrence does not significantly increase the probability of violation of a safety goal [2] (43, 98, 99, 143, 166, 202, 208)

safe state

an operating mode of an item without an unreasonable level of risk [1] [2] (1, 37, 39, 40, 49, 205–207)

safeness

proportion of the failure rate of a hardware element which does not contribute to the violation of the safety goals [2] (9)

safety

dependability of a system with respect to the absence of harmful failures with catastrophic consequences on its human user(s) and its surrounding environ-

[1] A safe state must not necessarily implement the intended functionality of the item. Therefore modes like switched off or modes with degraded functionality can be considered as safe states.

ment [21], in other words: freedom of system operation from the occurrence of catastrophic failures [26] (1, 2, 4, 9, 10, 13, 14, 19, 20, 53, 55–60, 66–69, 77, 82–84, 87, 88, 90, 92, 95, 97–99, 101, 111, 177, 179, 200, 206)

safety analysis

branch of safety engineering dealing with analyzing safety threats, defining their potential effects and assessing their severity, identifying appropriate safety measures, and providing qualitative and quantitative evidence about the achievement of safety integrity levels (vii, 5, 8–16, 19, 30, 32, 42–47, 49, 50, 53–55, 58–63, 65–68, 70, 81–85, 94–97, 102, 108, 111, 113, 114, 117, 129, 138, 143–146, 148–153, 155, 156, 165, 166, 170, 174, 175, 177, 178, 180, 181)

safety case

evidence-based argumentation about the completeness and the effective fulfillment of safety requirements for an item which is compiled from the work products of the different safety activities of the safety lifecycle [2] (41)

safety compliance

state of being in accordance with established safety standards and regulations, such as IEC 61508 [1] and ISO 26262 [2] for E/E systems (4, 5, 205)

safety culture

policies and strategies applied within an organization to enable and support developing, producing and operating safety-related systems, such as establishing dedicated rules, offering appropriate trainings, and deploying safety compliant processes and reporting mechanisms [2] (30)

safety evaluation

generic term used to denote both analysis-based activities and simulation-based activities which are commonly applied in the industry to provide evidence about the safety level of a certain product (vii, 4, 5, 8–11, 13, 14, 19–23, 32, 50, 61–63, 65–70, 72, 77, 81, 84, 94, 113–115, 129, 133, 143, 145–147, 155–157, 166, 180, 181, 200)

safety function

function to be implemented by a safety-related E/E/PE system that is intended to achieve or maintain a safe state for the Equipment Under Control (EUC), in respect of a specific hazardous event [1] (22, 24, 28, 29, 40, 198, 206, 207)

safety goal

top-level safety requirement derived as a result of the Hazard Analysis and Risk Assessment (HARA) and which can be related to multiple hazards [2] (7, 28, 29, 31–33, 35–42, 44, 45, 47, 48, 59, 94, 97, 99, 105, 106, 121, 138–141, 164, 198, 201–204, 207, 208)

safety integrity

probability that a safety-related E/E/PE system satisfactorily performs the specified safety functions under all the stated conditions within a stated period of time [1] (24, 206)

safety integrity level

extent of safety integrity corresponding to a provided or targeted level of risk reduction by a specific safety function or safety measure, ranging from the least to the most stringent level from SIL 1 to SIL 4 in IEC 61508 and from ASIL A to ASIL D in ISO 26262 [1, 2] (vii, 8, 9, 11, 13, 22, 24, 26, 28–31, 35, 37, 60, 61, 87, 94, 105, 143, 146, 166, 205–207, 209, 211, 215)

safety lifecycle

sum of stages that an item goes through from concept until decommissioning, i.e., all necessary activities involved in the implementation of the involved safety-related systems, starting at the concept phase of a project and finishing when they are no longer available for use [1, 2] (vii, 4, 5, 8–10, 13, 22, 23, 26, 29–33, 37, 38, 41, 42, 70, 82, 87, 114, 116, 118, 155, 156, 178–180, 205, 207, 209)

safety measure

activity or technical solution to avoid and/or control systematic failures, to detect and/or control random hardware failures, and to mitigate their harmful effects [2] (4, 8–11, 13, 24, 35, 38, 39, 42, 99, 100, 106, 110, 111, 121, 143, 144, 150, 167, 175, 178, 198, 201, 204–207)

safety mechanism

technical reflection of safety measure which is implemented by E/E functions or elements or by a different technology and which aims at maintaining the safe state by detecting faults and/or controlling failures [2] (33, 35, 37, 38, 43, 44, 46, 50, 61, 62, 87, 89, 98, 99, 111, 118, 134, 152, 153, 167, 198, 202, 204, 208)

safety metric

metric used to assess the design effectiveness with respect to safety (e.g., Single-Point Fault Metric (SPFM) and Latent Fault Metric (LFM) [2]) (9, 13, 43, 46, 97, 99, 103)

safety requirement

requirement which is defined with respect to the safety of a system and in correlation with the intended safety integrity level and which can be classified for example as a safety goal, as a functional safety requirement, as a technical safety requirement, as a hardware safety requirement, or as a software safety requirement [2] [2] (4, 5, 8, 10, 13, 16, 22, 26, 32, 35–38, 41–46, 48, 50, 61, 84, 94, 97, 98, 109, 138, 145, 157, 180, 181, 197, 200, 205)

safety validation

safety activity prescribed in the ISO 26262 context and performed through examinations and tests to ensure the sufficiency and the achievement of the defined safety goals [2] (8, 19, 22, 28, 30, 32, 41, 42, 44)

safety verification

branch of safety engineering dealing with verifying the effectiveness of safety measures through simulation-based techniques such as fault injection (19, 32, 62, 117, 143, 147)

safety-critical

a safety-critical system is a system whose malfunction or failure may potentially lead to a serious human injury or to loss of life (1, 3, 20, 58, 84, 99, 108, 170)

safety-related

(i) character of an element or a system whose failures increase the overall risk when they concur with other failures affecting different elements or a systems [1, 30, 26]
(ii) character of a system which implements the required safety functions necessary to achieve or maintain a safe state for the EUC and which is intended to achieve the necessary safety integrity level for those safety functions either on its own or with other safety-related E/E/PE systems [1]
(iii) character of activity performed during the safety lifecycle which is not commonly performed in safety-independent development cycles
(4–8, 13, 15, 20–24, 26–30, 41, 43, 60, 83, 99, 200, 205–207)

safety-relevant

character of an element or a system whose failures increase the overall risk when they concur with other failures affecting different elements or a systems [1, 30, 26] (20, 21, 97, 100, 172)

security

dependability of a system with respect to its ability to prevent unauthorized data access and/or handling [21] (1, 2, 4, 177, 181)

[2] Hardware and software safety requirements are the most detailed and implementation-specific requirements to be fulfilled by hardware parts or software units

sensitivity zone

zone in a hardware part or in a software unit which is vulnerable to the occurrence of certain faults and which is consequently suitable for the according fault injection (9)

service

system behavior as it is perceived by another special system(s) interacting with the considered system: its user(s) [21] (1, 2, 53, 197–199, 202, 203)

severity

estimation of the harm extent which might affect one or more persons in hazard situations [2] (7–9, 13, 21, 31, 33–35, 47, 58, 60, 98, 100, 102, 204, 205)

single-point fault

fault in an element that is not covered by a safety mechanism and that leads directly to the violation of a safety goal [2] (43, 99, 208, 215)

single-point fault metric

metric reflecting a system's robustness against single-point faults and residual faults, which is evaluated with respect to the coverage offered by safety mechanisms and to the properties of the design itself enabling a primarily high amount of safe faults (99, 100, 206, 215)

soft error

erroneous change to data content but no change to the physical circuit itself [1] (100, 175)

software unit

atomic piece of the software architecture which can be tested standalone [2] (8, 29, 32, 40, 41, 43, 197, 198, 203, 207, 208)

system

entity that contains different interacting components (relating at least one sensor, one controller and one actuator in the ISO 26262 context [2]) and that is able to interact with other similar entities [21] (29, 30, 45, 197–200, 202–205, 207–209)

systematic failure

failure which is deterministically correlated with a specific cause and which can only be avoided through design or manufacturing changes, documentation refinements, modifications in the operational procedures, or similar factors [2] (26, 28, 42–45, 49, 206)

technical safety concept

set of technical safety requirements along with their associated information, their allocation to system elements, and their implementation by the system design [2] (7, 29, 32, 38, 40, 41, 99, 105)

technical safety requirement

implementation-specific requirement derived from associated functional safety requirements [2] (7, 29, 30, 32, 35, 37, 38, 40–42, 44, 207, 209)

threat

potential violation of the safety integrity level resulting in immediate harm (13, 56, 58, 144, 205)

unreasonable risk

unacceptable risk in a specific context according as judged by valid societal moral concepts [2] (33, 200, 201)

verification

confirmation by examination and provision of objective evidence that the requirements of a safety lifecycle phase are complete, correctly specified, and effectively fulfilled [1, 2] (30, 32, 35, 41, 44)

verification plan

artefact or document describing how a given element or system is verified, listing all verification activities along with their nature and timing, and including a series of directed test cases used to exercise the device in order to assess the fulfillment of its functional requirements (10, 11)

virtual prototype

an abstracted representation of a physical product which can be displayed, analyzed, and tested with respect to different life-cycle aspects such as design, manufacturing, and service similarly to a real physical model [170] (61, 77, 120, 143, 149, 156, 166)

workload

suitable test case data for fault injection (9)

10 Acronyms

API

Application Programming Interface (75, 115, 116, 118, 131, 137)

ASIL

Automotive Safety Integrity Level (4, 5, 7, 9, 28, 30, 31, 33, 35–38, 42, 48, 197, 201, 206)

BSCU

Braking System Control Unit (123–126)

CPU

Central Processing Unit (156, 167, 169, 171, 172)

DFA

Dependent Failure Analysis (9, 44, 48, 49, 62, 63, 66, 68, 84, 95, 96, 108–112, 114, 130, 143–146, 148, 150, 155, 180)

DFT

Dynamic Fault Tree (48)

DNF

Disjunctive Normal Form (93)

DPF

Dual-Point Fault (99)

DSL

Domain Specific Language (73, 74)

DUT

Device Under Test (11)

EDA

Electronic Design Automation (11, 76)

EMF

Eclipse Modeling Framework (147)

EPS

Electric Power Steering (156–164)

ESACS

Enhanced Safety Assessment for Complex Systems (58)

ESL

Electronic System Level (76)

ETA

Event Tree Analysis (43, 45, 46, 49)

EUC

Equipment Under Control (22, 205, 207)

FIT

Failure In Time (99, 199)

FLM

Failure Logic Modeling (55, 56, 58, 84–87, 89, 90, 92, 93, 129, 156, 178)

FMEA

Failure Modes and Effects Analysis (5, 9, 42–47, 49, 56–63, 95–97, 113, 213)

FMECA

Failure Modes, Effects, and Criticality Analysis (47, 60, 61)

FMEDA

Failure Modes, Effects, and Diagnostic Analysis (vii, viii, 14, 37, 42, 43, 46–48, 63, 66, 68, 69, 84, 87, 95–102, 109, 111–115, 117, 118, 129–134, 136, 143–148, 150, 155, 156, 166, 167, 171, 173–176, 178–180)

FPGA

Field-Programmable Gate Array (51)

FPTC

Failure Propagation and Transformation Calculus (56, 57, 86)

FPTN

Failure Propagation and Transformation Notation (56, 57, 86)

FTA

Fault Tree Analysis (vii, viii, 5, 9, 42–46, 48, 58, 61–63, 66, 68, 84, 95, 96, 102–106, 108, 109, 111, 114, 117, 118, 129–131, 136–138, 141–148, 150, 155–157, 178–180, 200)

GUI

Graphical User Interface (69, 70, 115, 117, 119–122, 124, 125, 131, 137, 138, 141, 146, 147, 151, 163, 167, 175)

HARA

Hazard Analysis and Risk Assessment (7, 29, 31, 33, 35, 36, 42, 44, 84, 205)

HAZOP

HAZard and OPerability study (44–47)

HCID

Hot-Carrier Induced Degradation (3)

HiP-HOPS

Hierarchically Performed Hazard Origin and Propagation Studies (56, 57, 59, 86)

IF-FMEA

Interface Focused-FMEA (57)

IP

Intellectual Property (74)

ISAAC

Improvement of Safety Activities for Aeronautical Complex systems (58)

LFM

Latent Fault Metric (99, 100, 206)

MBSA

Model-Based Safety Analysis (58)

MCU

Microcontroller Unit (161, 163)

MDA

Model Driven Architecture (71–73)

MDD

Model Driven Development (70–73, 75, 77, 82, 83)

MetaFPA

Metamodeling-based Failure Propagation Analysis (83, 85–94, 112, 117–122, 124–129, 131, 133, 137–142, 156, 157, 162–164, 177–180)

MIPS

Microprocessor without Interlocked Pipeline Stages (166–171, 173–175)

MOF

Meta Object Facility (74)

MPF

Multiple-Point Fault (99)

NBTI

Negative-Bias Temperature Instability (3)

NuSMV

New Symbolic Model Verifier (58)

OMG

Object Management Group (71, 73, 78)

PIM

Platform Independent Model (71, 72)

PMSM

Permanent Magnet Synchronous Motor (159–161)

PSM

Platform Specific Model (71–73)

PWM

Power Width Modulation (161, 162, 164)

QVT

Query/View/Transformation (78)

RBD

Reliability Block Diagram (9, 43, 45, 46, 49, 50, 95)

RF

Residual Fault (99)

RTL

Register Transfer Level (51, 76, 77, 115, 149)

RWB

Reliability Work Bench (129, 138, 141)

SENT

Single Edge Nibble Transmission (161)

SIL

Safety Integrity Level (24, 28, 61, 206)

SoC

System-on-Chip (53, 62, 74, 76, 77)

SPF

Single-Point Fault (99)

SPFM

Single-Point Fault Metric (99, 100, 206)

SQL

Structured Query Language (134)

TFA

Tabular Failure Annotation (57)

TFT

Temporal Fault Tree (48)

TLM

Transaction Level Modeling (77)

UMD

Unified Model of Dependability (54)

UML

Unified Modeling Language (54, 57, 58, 60, 71, 73, 74, 91, 101, 106, 110, 116, 118, 130)

VHDL

VHSIC Hardware Description Language (51, 116, 131, 149)

VHSIC

Very High Speed Integrated Circuit (216)

WBS

Wheel Braking System (122, 123)

XMI

XML Metadata Interchange (116)

XML

Extensible Markup Language (75, 87, 89, 94, 116, 120, 126, 138, 141, 179, 216)