Proposal of a model-driven approach for software safety - Application to the software architecture of connected and autonomous vehicles

Yandika Sirgabsou

Université Fédérale
Toulouse Midi-Pyrénées

THÈSE

En vue de l'obtention du
DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Institut National des Sciences Appliquées de Toulouse

Présentée et soutenue par

Yandika SIRGABSOU

Le 10 février 2023

Proposition dune approche dirigée par les modèles pour la sûreté de fonctionnement logicielle - Application à l'architecture logicielle des véhicules connectés et autonomes

Ecole doctorale : **SYSTEMES**

Spécialité : **Informatique et Systèmes Embarqués**

Unité de recherche :
LAAS - Laboratoire d'Analyse et d'Architecture des Systèmes

Thèse dirigée par
Claude BARON et Philippe ESTEBAN

Jury

M. Nicolas DACLIN, Rapporteur
M. Frédéric KRATZ, Rapporteur
Mme Rabea AMEUR-BOULIFA, Examinatrice
Mme Claude BARON, Directrice de thèse
M. Philippe ESTEBAN, Co-directeur de thèse
M. Jean-Michel BRUEL, Président

En mémoire mon père, Raphaël,

A ma mère, Myriam,

A ma femme Adeline et à mes filles, Kalsinbé et Tamiba,

A mes frères et sœurs, Olivier, David, Geneviève, Ernestine, Parfaite et Moise,

A ma grande famille, mes amis et toutes les personnes qui ont cru en moi, soutenu et encouragé.

Acknowledgments

Over the years, I had the opportunity to be surrounded by amazing people that contributed in various ways to the completion of this thesis. Without being exhaustive, I would like to take the opportunity to thank them all and mention a few names here.

First, I would like to express my deep gratitude to Mrs. Claude BARON, Professor at INSA of Toulouse, head of the ISI team at LAAS-CRNS and Mr. Philippe ESTEBAN, Lecturer at the University of Toulouse 3-Paul Sabatier. I am very grateful for the honor they gave me to conduct this thesis under their supervision. Their advice, encouragements, patience, and kindness are what allowed me to carry on against all challenges. I am especially grateful to Mrs. Claude BARON for the opportunity she gave me and for believing in me.

I would also like to thank Mr. Laurent PAHUN, head of the Platform Software Architecture team at Renault Software factory for welcoming me in his team and awarding me with the opportunity to carry this thesis project under his caring supervision. I thank him for his unveiling support and endless kindness and for raising me up in times of challenges.

My gratitude also goes to Mr. Hamid DEMMOU, retired Lecturer at University of Toulouse 3-Paul Sabatier and to Mr. Rob VINGERHOEDS, Professor at ISAE-SUPAERO for accepting to be part of my follow-up committee. I thank them for their availability and sound advice.

I would also like to thank all my jury members: Mr. Nicolas DACLIN, Lecturer at IMT Mines d'Alès, Mr. Frédéric KRATZ, Professor at INSA Centre Val-de-Loire, Mr. Jean-Michel BRUEL, Professor at IRIT/IUT Blagnac, Mrs. Rabea AMEUR-BOULIFA, Lecturer at Telecom Paris for giving me the honor of being part of my thesis jury.

Finally, my deepest gratitude goes to my late father for inspiring me to carry on in this journey, my mother, brothers and sisters for their moral support and unconditional love, my wife and children who shared with me the joys and struggles of this journey, my extended family and all my friends that encouraged me.

Summary

This research work is motivated by the need to cope with the increasing complexity of software architectures, in particular in the context of automobile, and to overcome the limitations of current industrial practices in terms of security analysis. Despite the development of MBSE (Model Based Systems Engineering), these practices are still characterized by the reliance on manual traditional safety analysis techniques such as Fault Tree Analysis (FTA) or Failure Modes and Effect Analysis (FMEA). Although still useful, these techniques fall short when faced with complexity with the possibility of resulting in subjective, inefficient, poor quality and error-prone analyses. Hence, to improve the state of the current practice in the automotive context, our proposal is to apply the Model Based Safety Analysis (MBSA) approach that is a relevant Model Driven Engineering approach applied to safety. However, the review of the state of the art of MBSA approaches suggests that most are systems oriented and lack clear methodological support. In addition, some of them, especially those relying on a dedicated model, require deep understanding (in terms of modeling paradigm) and can be challenging to implement in the case of complex systems. Another issue with current industrial practices is that safety analysis at software level suffers of poor integration with the software development process, which can result in inconsistent analyses. To address these issues, the essence of our contribution is to provide a methodology that adapts the concepts, principles, and methods of MBSA for the purpose of improving the practice of software safety analysis, taking into consideration the current state of practices (in the existing software development process).

The contribution of the PhD is a guided methodology to perform software safety analysis using a model-based approach. It has been validated for embedded automotive software architectures. The methodology proposes to first define the safety analysis context then to build the dysfunctional architecture, and finally to use this architecture to conduct safety analyses based on a dedicated model approach. One of the main difficulties was the lack of adequate input data to conduct these analyses due to the use of document-centric and not model-centric artifacts in the software engineering process. To overcome this issue, the methodology proposes to definite the context of the safety analysis by identifying the component to model depending on whether they are safety related, their expected normal behavior, and the safety measures they implement. To support the application of the methodology, we compared, selected and experimented a commercial off-the-shelf safety analysis tool and a language. The methodology was validated through two case studies from industrial projects.

Extensions of the methodology have also been proposed, aiming to address some remaining challenges related to complexity brough by the limitations of a dedicated model approach. The first proposal consists of using software fault patterns based on ISO 26262 software fault templates to ease the construction of the dysfunctional model. Through this proposal, prototypes of common software fault patterns are developed and reused to build the dysfunctional model. The second proposal is a tooling proposal to partially automated and ease the construction of software component's fault behavior and propagation through functional to dysfunctional logic translation. It aims to ensure a better consistency of software safety analyses with the software development process constantly with ISO 26262 recommendations.

Résumé

Dans le contexte du développement de logiciels automobiles, le problème général qui a motivé ce travail était la complexité croissante des architectures logicielles et les limites des pratiques actuelles en termes d'analyses de sécurité. Malgré le développement du MBSE (Model Based Systems Engineering), ces pratiques sont toujours caractérisées par le recours à des techniques manuelles traditionnelles d'analyse de la sécurité telles que l'analyse par arbre de défaillance (FTA) ou l'analyse des modes de défaillance et de leurs effets (AMDE). Bien qu'elles soient toujours utiles, ces techniques sont insuffisantes face à la complexité avec la possibilité d'aboutir à des analyses subjectives, inefficaces, de mauvaise qualité et sujettes aux erreurs. Par conséquent, pour améliorer l'état de la pratique actuelle dans le contexte automobile, notre proposition est d'appliquer l'approche d'analyse de la sécurité basée sur un modèle (MBSA) qui est une approche d'ingénierie pilotée par modèle pertinente appliquée à la sécurité. Toutefois, l'examen de l'état actuel de la technique des approches actuelles des MBSA suggère que la plupart de ces approches sont axées sur les systèmes et manquent d'un soutien méthodologique clair. En outre, certaines des approches MBSA (en particulier celles qui reposent sur un modèle dédié) nécessitent une compréhension approfondie (en termes de paradigme de modélisation) et peuvent être difficiles à mettre en œuvre dans le cas de systèmes complexes (limites de la modélisation manuelle). De même, dans les pratiques actuelles, l'analyse de la sécurité au niveau logiciel souffre d'une mauvaise intégration avec le processus de développement logiciel, ce qui peut entraîner des analyses de sécurité incohérentes. Pour résoudre ces problèmes, l'essence de notre contribution est de fournir une méthodologie qui adapte les concepts, les principes et les méthodes de MBSA dans le but d'améliorer la pratique de l'analyse de la sécurité des logiciels, en tenant compte de l'état actuel des pratiques (dans le processus de développement de logiciels existant).

Notre première contribution consiste en une méthodologie couvrant toutes les étapes nécessaires pour effectuer une analyse de la sécurité sur les architectures logicielles automobiles en utilisant l'approche basée sur des modèles tout en répondant aux défis présentés par le manque d'intrants inadéquats apportés par l'utilisation d'artefacts centrés sur les documents dans certaines parties du processus de génie logiciel. Grâce à cette contribution, nous proposons une méthodologie étape par étape pour définir le contexte d'analyse de la sûreté, construire l'architecture dysfonctionnelle et l'utiliser pour des analyses de sûreté s'appuyant sur une approche modèle dédiée. Pour surmonter ce problème, la méthodologie propose de définir le contexte de l'analyse de sûreté en identifiant les composants à modéliser selon qu'ils sont liés à la sécurité, leur comportement normal attendu et les mesures de sécurité qu'ils mettent en œuvre. Pour soutenir l'application de la méthodologie, nous avons comparé, sélectionné et expérimenté un outil logiciel d'analyse de sécurité du commerce et un langage. La méthodologie a été validée par deux études de cas issues de projets industriels.

Des extensions de la méthodologie ont également été proposées. La première, également méthodologique, vise à relever certains défis liés à la complexité due aux limites d'une approche basée sur un modèle dédiée. Elle consiste à utiliser des modèles de pannes logicielles basés sur des exemples de pannes logicielles définie par la norme ISO 26262 pour faciliter la construction du modèle dysfonctionnel. Grâce à cette proposition, des prototypes de modèles de défaillance logicielle courants sont développés et réutilisés pour construire le modèle dysfonctionnel. La deuxième extension, encore en cours de développement, est une proposition d'outillage visant à automatiser partiellement et à faciliter la construction du comportement et de la propagation des défauts des composants logiciels par traduction logique fonctionnelle à dysfonctionnelle. Elle vise à assurer une meilleure cohérence des analyses de sécurité logicielle avec le processus de développement logiciel en permanence avec les recommandations ISO 26262.

Table of content

Table of contents

List of acronyms

AADL	Architecture Analysis and Design Language
AD	Autonomous /Automated Driving
ADAS	Advanced Driver Assistance Systems
CPA	Critical Path Analysis
EAST-ADL	Electronics Architecture and Software Technology -Architecture Description Language
DFA	Dependent Failure Analysis
EMV2	Error Model Version 2
FEM	Failure Effect Modeling
FFI	Freedom From Interference
FMEA	Failure Modes and Effects Analysis
FMECA	Failure Modes Effects and Criticality Analysis
FPL	Failure Propagation Logic
FPTN	Failure Propagation and Transformation Notation
FTA	Fault Tree Analysis
FTT	Failure Truth Table
HARA	Hazard and Risk Assessment Analysis
HiP-HOPS	Hierarchically Performed Hazard Origin and Propagation Studies
MBSA	Model Based Safety Analysis/Assessment
MBSE	Model-based Systems Engineering
MBSW	Model Based Software Engineering
MDE	Model Driven Engineering
PHA	Preliminary Hazard Assessment
SAML	Safety Analysis Modeling Language
SFP	Software Fault Pattern
STAMP	System-Theoretic Accident Model and Processes
STPA	System-Theoretic Process Analysis
SysML	Systems Modeling Language
UML	Unified Modeling Language

List of figures

List of tables

Glossary

This subsection defines the meanings of some expressions employed all-through the document.

Fault Tree Analysis (FTA)

FTA is a deductive (top-down) safety analysis technique used to determine the combination of causes called basic events that may lead to a known feared event called top event.

Failure Propagation logic

Failure propagation is a form of abstract (abnormal) dataflow representing the way in which failure modes within a system interact.

Hazard

A hazard is a system state or set of conditions that, together with a particular set of worst-case environmental conditions, will lead to a loss.

Minimal Cut

A minimal cut set is any set of conditions necessary and sufficient to cause the loss event described at the top of the tree.

Safety

The expectation that a system does not, under defined conditions, lead to a state in which human life, health, property, or the environment is endangered." [ISO/IEC 15026:1998].

MBSA (Model Based Safety Analysis)

A technique which models system content and behavior in order to provide safety analysis results. MBSA employs an analytical model called a Failure Propagation Model (FPM) [ARP4761A].

Model Driven Engineering

A software development technology that combines domain-specific modeling languages with transformation engines and generators to enable 1) the formalized specification of the application requirement, structure and behavior using models within particular domains, 2) the analysis of certain aspects of the models and 3) the synthesis of various types of artifacts

Model-Based Systems Engineering (MBSE)

The formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases.

Chapter 1. Introduction

Abstract: *In our society undergoing profound technological and societal changes, we are witnessing the development around automated driving nowadays. In particular, autonomous vehicles are characterized by a growing complexity in their embedded software architectures implying new safety assurance challenges. Consequently, there is a growing need for more efficient and rigorous analysis techniques to ensure safety, prove its evidence, comply with regulations and achieve societal acceptance. In this context, this first chapter introduces the general context and states the motivations for this thesis. It also outlines the main research challenges that make it difficult to fulfill the needs identified in the context. Then, the chapter states the thesis conceptual proposal which is to explore the use of a model-based approach. The general problem is broken down from general research questions into specific research questions to formalize the problem statement. The chapter lays out the research method we adopted to address the different questions, outlines what our contributions are, explains how we validated them, and finally shows how the proposal allows answering the research problem. Beyond a simple introduction, it constitutes an extended abstract of the thesis.*

1.1. Thesis context

This work results from a collaboration between Renault Software Labs (RSL) and the French Laboratory of Analysis and Architecture of Systems (Laboratoire d'Analyse et d'Architecture des Systèmes, LAAS). It benefited from the support of the French National Technological Research Association (Association Nationale de la Recherche Technologique, ANRT) through the granting of a CIFRE (Conventions Industrielles de Formation par la Recherche) funding. CIFRE is a French government funding system that promotes the development of public-private research partnership by placing PhD students in employment conditions.

LAAS is a research unit of the French National Center for Scientific Research (CNRS) located in Toulouse. The work was conducted in the ISI team (Systems Engineering and Integration) of the laboratory. ISI research focuses on the design of complex systems and the improvement of life-cycle processes, including requirement management, modeling, model integration, verification and validation, simulation or virtual prototyping. The work of this PhD is complementary to the work of another thesis, carried out within the framework of the S2C (System & Safety Continuity) project of the IRT Saint Exupéry, which addresses the digital continuity between systems engineering and safety assessment models in order to propose a model dedicated to the diagnostic of satellites while in operation. The research collaboration between RSL and LAAS-ISI was initiated during a previous thesis by Yann ARGOTTI [1]. Our work is part of this ongoing collaboration.

Renault Software Labs (RSL) is a subsidiary of Renault Group born from the acquisition by the group of the French R&D branch of the Intel corporation back in 2017. It is a software R&D center in charge of developing embedded software for Renault's connected and autonomous cars. The creation of this software division was part of Renault's vision and desire to promote embedded software, whose importance has become growing amid the advent of the automated, connected and electric cars [2].

Following the creation of RSL, a new entity, the Renault Software Factory, was created in 2020, bringing together all of the Renault Group's software skills and expertise. Renault Software Factory is at the heart of Renault Group's technological transformation and the challenges of the mobility of the future. Located on two sites, Sophia-Antipolis and Toulouse, the entity brings together more than 600 engineers and, in addition to the Paris Technocentre, is at the heart of two ecosystems focused on the intelligent car. It is in this context that our thesis took place. The thesis focuses on the analysis of software safety through model-based approach applied to software architecture of connected and autonomous cars.

Thesis title

"Proposal of a model-driven approach for software safety - Application to the software architecture of connected and autonomous vehicles."

Moreover, our thesis is complementary to another ongoing thesis [3] at Renault Software Factory. With the latter our thesis shares the common goal of improving the current software engineering practices through the use of models and formal methods.

1.2. Scientific context

The recent decades have seen a tremendous rise in the development of autonomous and intelligent systems across various embedded systems industries such as autopilots in aeronautics, highly complex software operated space missions, or self-driving vehicles in automotive. One common characteristic shared across these domains is the increasing use of complex embedded software to ensure various functionalities including autonomy and critical (safety related) functions. Consistently with this trend, the automotive industry seems on its way to a radical transformation as vehicles are becoming more electric, connected and autonomous. With the advent of autonomous vehicles, the automotive industry is increasingly relying on embedded software to enable various features. As a result, new concepts such as software defined vehicles are emerging, implying serious changes in the very concept of what a vehicle is. In this context, from being a neglectable part of the vehicle at the lowest level in the past, today's embedded software is at the centerstage in the automotive industry with various critical and safety related features being partially or fully assumed through software. But if software is at the center of this revolution, there are challenges that remain among which are mastering complexity and ensuring safety, defined as the ability of the system to prevent failures that could lead to injuries and damages [1].

The first component of this challenge, that is technological, is related to the technical means for ensuring the safe behavior of the embedded critical software in systems architectures that are becoming increasingly complex. The extensive use of embedded software in autonomous vehicles has profound implications for safety that must be rigorously ensured. With an average of one hundred million lines of code (compared to about ten million in a Boeing B787) distributed among nearly one hundred embedded computers, the complexity of automotive computer architectures is such that traditional design and

analysis practices (based on document production and analysis) have reached their limits. Indeed, it can be stated without a shadow of a doubt that due to their complex nature, ensuring safety of complex software intensive systems such as autonomous vehicles are extremely challenging using the existing method. Additionally, the reliance on software for safety critical function has also introduced new considerations for the overall system safety [4]. For instance, it is admitted throughout the critical systems industry that safety is a system property and that embedded software on its own cannot cause harm [5] [6]. However, with software increasingly assuming such an active and crucial role in vehicle safety functions, an increasing number of accidents have seen their root causes associated with software failures across numerous software intensive critical systems. In the automotive context, the growing use of software-enabled features has now led to the development of autonomous driving systems that aim to assist or replace the driver, through various autonomy levels ranging from assisted driving to fully autonomous driving [7]. While this technological change is introducing interesting features for the modern vehicle, it also has profound ramifications for safety as well as well-established beliefs from the society and trusted engineering practices from the industry. Indeed, the use of embedded software enables new and various innovative features in autonomous vehicles. However, their failure resulting from improper execution, development errors, or unforeseen unsafe scenarios can lead to serious human harm and property damages. This can imply economic cost (related to recalls, litigations) for automakers as well as challenges in fulfilling required regulatory compliances that are preconditions to vehicles commercialization.

In addition to the described technological challenge, there is a societal component to this challenge. This second aspect, still related to safety, is the trustworthiness of software operated vehicles that can affect the acceptance of automated vehicles. Today, it can be stated that the trustworthiness of autonomous and highly computerized vehicles to safely transport humans remains very fragile in the light of recent road incidents involving autonomous. Indeed, with our society undergoing perpetual technological revolutions, we are often faced with new technological concepts that promise us better lives, more comfort, and safety. With time, these innovations often live up to the goods they promise. But the fact remains that their acceptance by society can be reluctant, or not always unanimous especially if they have the potential of affecting the safety of the users or radically changing our habits. Therefore, to achieve automated driving trustworthiness, it is crucial to ensure and guarantee safety as a precondition for the acceptance of autonomous vehicles by our society. The ability to provide safety guarantees will be key to the acceptance of autonomous vehicles by society.

1.3. Motivation

In the critical systems industries such as military, aviation, space or railways, the proof and guarantee of safety is an important requirement supported by standards. Such standards include IEC 61508 [8] for electrical/electronic/programmable electronic safety-related systems, MIL-STD882E [9] for military systems and SAE ARP 4754A [10] as well as SAE ARP 4761 [11] for aviation. This notion of standard led safety assessment is also well rooted in the automotive engineering practices where safety assessment, tests and validation activities are performed as part of a formal development process. The automotive development process is governed by the ISO 26262 standard [12], which focuses on functional safety for road vehicles. This standard requires that safety analyses be performed, proven and guaranteed. More recently, the ISO/PAS 21448 “Safety of the Intended Functionality” (SOTIF) standard was created by the

automotive industry to addresses the issues for driver assistance functions that could fail to operate properly even in the absence of systems equipment failures.

However, the current complexity of embedded systems in vehicles means that guaranteeing this safety is extremely difficult. In this context, traditional document-based safety analysis techniques such as Fault Tree Analysis (FTA) [13] or Failure Modes Effect Analysis (FMEA) [14] are used to assess the safety of the system, hardware, and embedded software. Although still useful, these techniques fall short when faced with complexity with the possibility of resulting in subjective, inefficient, poor quality and error-prone analyses. However, autonomous vehicles' embedded software is highly complex. They often consist of over a hundred million lines of codes distributed across various communicating computing units. This complexity makes their safety evaluation through the traditional techniques extremely challenging. In addition, it has become increasingly difficult to keep safety analyses up to date with the evolution of engineering artifacts, which can be rapid in the case of agile development. Moreover, the development of vehicles currently requires highly collaborative work between different teams coming from different disciplines such as Systems Engineering, Software Engineering or Safety Engineering comprising specialists from different business sectors with different intentions. In the context of increasing systems architecture complexity, such collaboration remains difficult especially when they rely on traditional document centric artifacts to convey design ideas. Therefore, there is a growing need from an organization standpoint for more efficient methods that favor collaboration.

Faced with this increasing use of software in vehicles, the difficulty of evaluating and guaranteeing driving safety, and the increased need to exchange consolidated, simulable and verifiable engineering data within and between teams, the current need is to define and adopt new practices. To meet these needs, most organizations lean on Model-Driven Engineering (MDE) solutions: a set of practices based on the principle of the conceptual domain model, which aims (among other things) to automate the production of systems and software [15] [16] [17]. The design process can then be seen as an ordered set of model transformations that lead to usable artifacts, which encourages reuse and early verification in particular. It consists in weaving different models together (including functional and organic models) or dealing with different extra-functional requirements such as safety, reliability, or performance. However, the heterogeneity of these different models makes their cohabitation difficult. In addition, despite the development of MBSE (Model Based Systems Engineering), the current practices are still characterized by the reliance on the manual document centric FMEA or FTA safety analysis techniques.

The general problem that motivated this work was the growing complexity of software architectures in automotive and the limitations of the current practices in terms of safety analyses.

General problem

ISO 26262 software safety analysis current practice rely on manual traditional safety analysis techniques resulting in subjective, inefficient, poor quality, error prone analyses.

This induced our general research question, ‘How to improve the state of the current practice in order to guarantee rigor and consistency of analyses and prove compliance to authorities?’

General research question

How can the current safety analysis practices be improved to guarantee more rigorous, consistent analysis and show proof of compliance?

Our high-level answer to this question is to adopt a model driven approach to support the safety analyses, as Model-Driven Engineering (MDE). MDE also called Model Driven Development (MDD) [18] is a software development technology that combines domain-specific modeling languages with transformation engines and generators to enable 1) the formalized specification of the application requirement, structure and behavior using models within particular domains, 2) the analysis of certain aspects of the models and 3) the synthesis of various types of artifacts, such as source code, deployment descriptions, or alternative models [19].

General answer to the general research question

Apply Model-Driven Engineering

MDE promotes the systematic use of models as primary artifacts during a software engineering process [20]. MDE practices proved to increase efficiency and effectiveness in software development, as demonstrated by various quantitative and qualitative studies [16]. These benefits have also been promoted through the application of MBSE, defined as the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities throughout the system engineering process. Although both methods promote the use of models as primary artifacts, MDE relies more on domain specific and formal semantics that enables automated analysis, transformation, and alternative artifacts generation.

In the domain of software or systems safety assessment, applying MDE principle means relying on a safety Domain Specific Language (DSL) to enable the modeling, analysis, and generation of alternative safety models. This practice is found in Model-Based Safety Analysis (MBSA), defined as a technique which models system content and behavior in a failure-oriented analytical model in order to provide safety analysis results. Although the less strong term ‘model-based’ is used, many MBSA approaches rely on modeling languages that are formal enough to enable formal model checking, automated safety analysis

and the generation of classical models making. These capabilities make these MBSA approaches to be compatible with the MDE approach definition.

1.4. Research method

To conduct our research and make the several proposals, we used a research method based on literature review (according to the typology proposed in [21]). The choice for a literature review-based method is mainly motivated by the nature of the problem to solve. Indeed, the nature of the problem we aim to address is related to the adoption a new methodology. Therefore, it can be classified as a methodological problem. Like the majority of software engineering issues, the solution of software engineering methodological problems often consists in new methodologies, algorithms or pieces of software. One way to identify gaps and issues within the current methodology is to conduct a literature review.

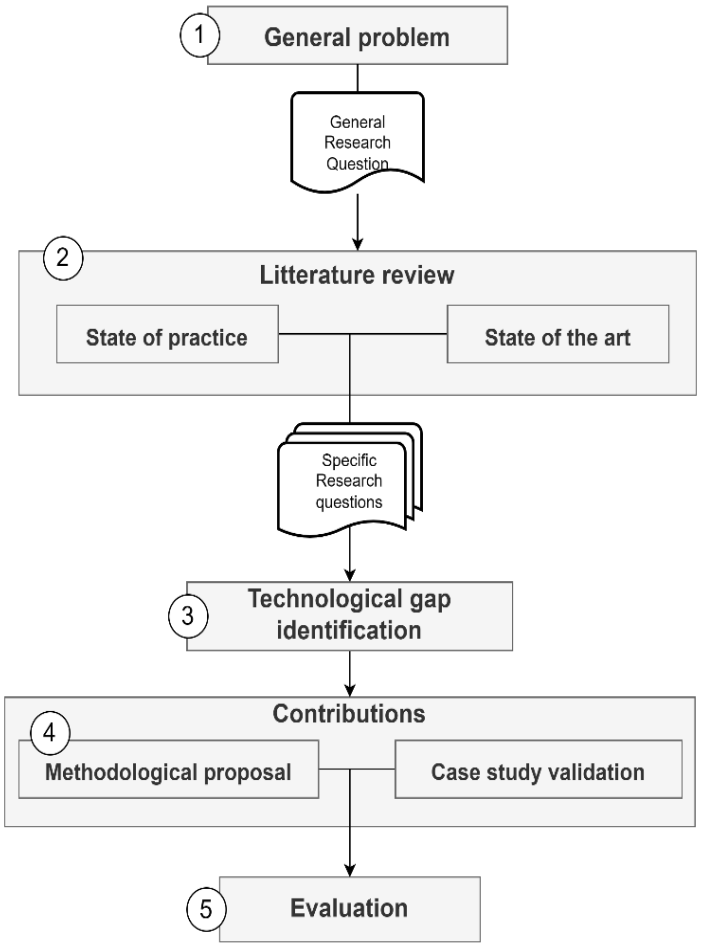

Figure 1. Research method

Hence, an outline of our research method, consisting of 5 main steps, is provided in Figure 1. In the first step, the problem that the thesis aims to address is enunciated going from the general problem to general research questions. In the second step, a literature review consisting of two parts (state of the industrial practice & state of the art) is conducted. In the third step, a comparison between state of the art and practices is conducted to identify some gaps; it justified the need of a new methodology.

The comparison also allowed refining the general problem into sub-problems, and thus to induce sub research questions. From this comparison, specific research questions are elaborated. In comparison to the general problemata and research questions elaborated in the 1st step (that are based on the needs), the specific research questions elaborated in the 3rd step also take into consideration the limitations identified in the state of the art. In the 4th step (identified by "contributions"), proposals are made to address the previously elaborated specific research questions. These proposals are then validated on two case studies. In the final step (5th), both the theoretical aspects of the methodological proposals and the practical results from the case studies are discussed and evaluated to show how 1) the methodological proposal applies to a real system and 2) that the contributions successfully address the methodological and technological gap identified in the 3rd step. From this discussion, perspectives for improvement were defined.

Step1: General Problem

One of the main challenges that hinders the widespread adoption of the MBSA approach is the lack of a methodological support. As in MBSE, an MBSA approach requires a method, a well-structured language and a tool that supports the application of the method. While the language and tool are usually chosen and provided by tool manufacturers, the method is often left to the discretion of the practitioner. Moreover, methods from literature are often found to advocate conflicting principles. For instance, while some practitioners are fervent advocates of a safety dedicated model approach for MBSA, others believe a safety extended MBSE model to be more practical. In addition, most of the current languages, methods and tools that are described in literature are often systems oriented, thus more suitable for MBSA at system level instead of software level. Hence, the main question that arises is how, if any, these methods, languages and tools can be applied to the practice of safety analysis at software engineering level especially in the context of automotive embedded software safety. This question can further be broken into secondary questions which are: 1) How to model the dysfunctional behavior of the elements of the software architecture and 2) How to model failure interaction (structural and functional) between these elements.

Weighing on the arguments in the literature to determine which of the several approaches (dedicated model, extended model or a mix) is most suitable for the application of the MBSA approach at software engineering level, our choice has been to use the dedicated model approach. It advantages in the long term include aspects such as more formal safety analysis based on dedicated safety modeling languages and formal model checking, the separation of concerns [22], independence between systems development and safety assessment processes [10] that are important principles taken into consider in the certification context of critical systems. However, with this choice, a few issues arise. First, while the dedicated model approach has clear advantages (such as safety analysis independence from system or software design or being more formal due to better structured modeling language) it also has limitations. In particular, our observation is that dedicated model dysfunctional architecture modeling can be challenging especially for non-experts. From our point of view, the lack of clear methods and methodological support associated to the need to learn a new safety modeling language, are non-

negligeable issues that further hinder the application of the MBSA approach. Furthermore, the dedicated model approach often requires manual modeling, which is limited in the context of a growing complexity of software architectures. Thus, the main questions that arise are how we can apply the dedicated model approach in a way that better masters the growing complexity and how can we make dysfunctional modeling less complex and easier for non-experts.

Another important aspect to consider in the choice and application of an MBSA approach is its ability to integrate with a broader MBSE framework. In such a case, consistency between the MBSE and MBSA models must be established and maintained throughout the development cycle. Consistency is important because changes resulting from the continuous evolution of design models can result in obsolete or erroneous assumptions in safety analysis models if proper measures are not implemented. Therefore, one need in the adoption of the MBSA approach is to ensure that the MBSA models and the system of software design models remain coherent with one another as the system design evolves throughout the development cycle. The current methods to ensure such interoperability include techniques such as model synchronization through the S2ML language [23]. Such MBSA consistency methods are also system oriented with no focus on the consistency between safety analysis models at software engineering level and software design models. S2ML for instance allows transformation between system models described in SysML and MBSA models described in AltaRica 3. Therefore, one important question is how to improve the integration of software safety analysis with software development process and ensure a better consistency of safety analyses with ISO 26262 recommendations.

Step 2: Literature review

To conduct our research, address the problems and answer the research questions, we used a research method based on systematic literature adapted from [9] and outlined in Figure 2. The literature review process starts with searching through a variety of peer-reviewed relevant research papers using search engines (such as ScienDirect, Springer, Wiley, IEEE, HAL, ACM Digital Library, Google Scholar, ResearchGate). To conduct the search, we start by using basic keywords such as Model-Based Safety Analysis and MBSA. The obtained search results are filtered using inclusion criteria (such as Subject areas: engineering, computer science, peer-review status: reviewed, Full text availability etc.). The abstracts resulting from the search are gone through to check their relevance to the MBSA topic. Additionally, the abstract reading also allows us to identify new relevant keywords that are fed back to the search engines. After abstract reading, if a paper is found irrelevant, the paper is excluded. The papers that are not excluded are read further and new relevant papers from their reference are selected for abstract reading. The final papers are read and analyzed.

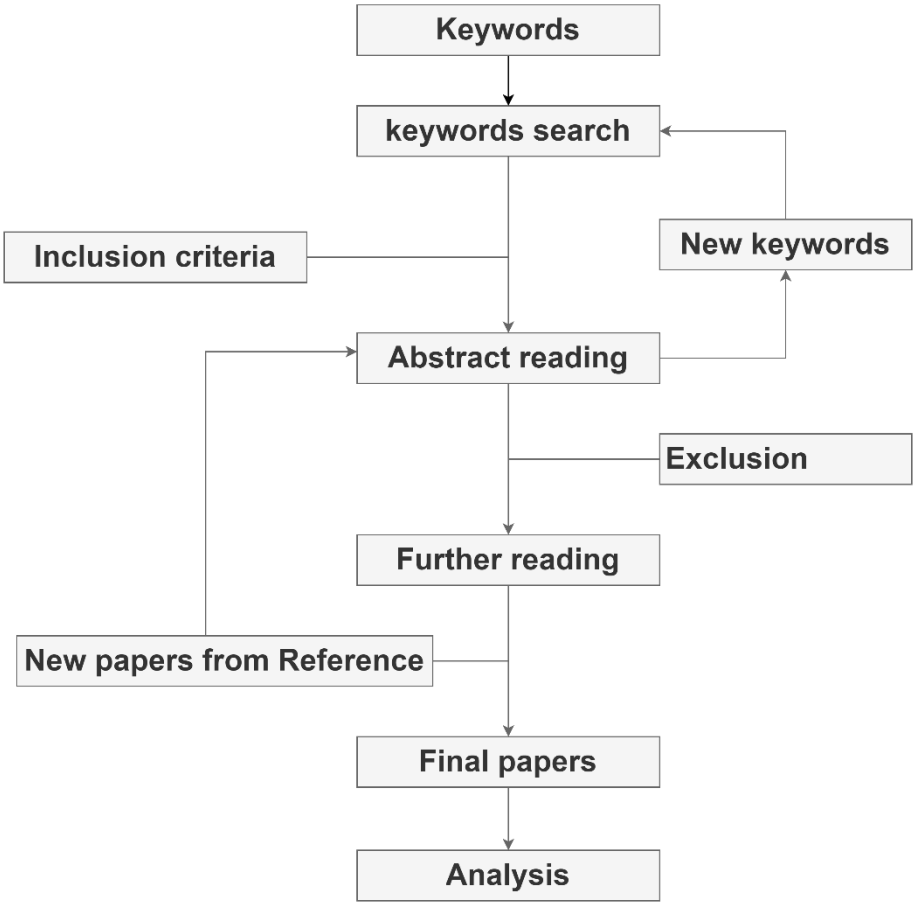

Figure 2. Literature review process

Step 3: Technological gap identification

To identify the gap between current practices and state of the art methods, the papers were read, analyzed, and classified according to 3 categories including methodological papers describing state of the art MBSA methods, case study papers describing state practice and tooling support, and literature review papers that analyze MBSA methodologies, tools and language. A review of survey papers allowed us to identify already known challenges and limitations related to the application of MBSA. Furthermore, a comparison between the state of the art and the state of practices (by studying the recommendations the ISO 26262 standard and MBSA related case studies) enabled us to identify several challenges (declined into research questions) that hinder the application of MBSA methods as described earlier in the “Problem statement” section.

The first challenge results from the fact that most MBSA methods focus on system level safety analysis. Consequently, it remains unclear how such MBSA approaches can be applied to software safety analysis. Hence the first question resides in how to Apply MBSA to automotive software. To be also noted is the lack of modeling methodologies for MBSA in general and software-oriented safety analysis in particular. In fact, in most MBSA approach, while the language and tools are defined, the modeling method is often left at the discretion of the practitioner with conflicting methodological advocacy in the literature. The

additional challenge associated to this is the choice of an appropriate modeling approach. Hence a secondary question is asked: what modeling approach for automotive software MBSA (for instance dedicated or extended model?).

Research question 1

How can the current MBSA methods, tools & languages be applied to the automotive software safety analysis?

&

What modeling approach for software Model-Based Safety Analysis (dedicated or extended model approach)?

The second challenge is related to the growing complexity of software architecture making safety analysis and MBSA modeling challenging especially in the case of a dedicated MBSA model. Hence the question we asked was “How to better master the growing complexity software and make its safety modeling less challenging”.

Research question 2

How to better master the growing complexity software architecture and make its safety modeling less challenging

The third challenge came from the observation from the current practices that automotive software safety analysis suffers of poor integration with software engineering process with no rigorous means to ensure the consistency between software architecture design and safety analyses. Furthermore, the analyses do not necessarily comply with the ISO 26262 recommendations. Hence the challenge is “how to ensure a better compliance with ISO 26262 and improve the integration of software safety analysis with software development process to ensure a better consistency”.

Research question 3

How to ensure a better compliance with ISO 26262

&

How to improve the integration of software safety analysis with software development process to ensure and a better consistency

Step 4: Proposal

Going from the identified gaps, we made a proposal to address the specific issues we identified from the literature review. The goal of the proposal is to provide a methodology that adapts the concepts, principles, and methods of MBSA for the purpose of improving the practice of software safety analysis, taking into consideration the current state of practices in software development in MDE approach.

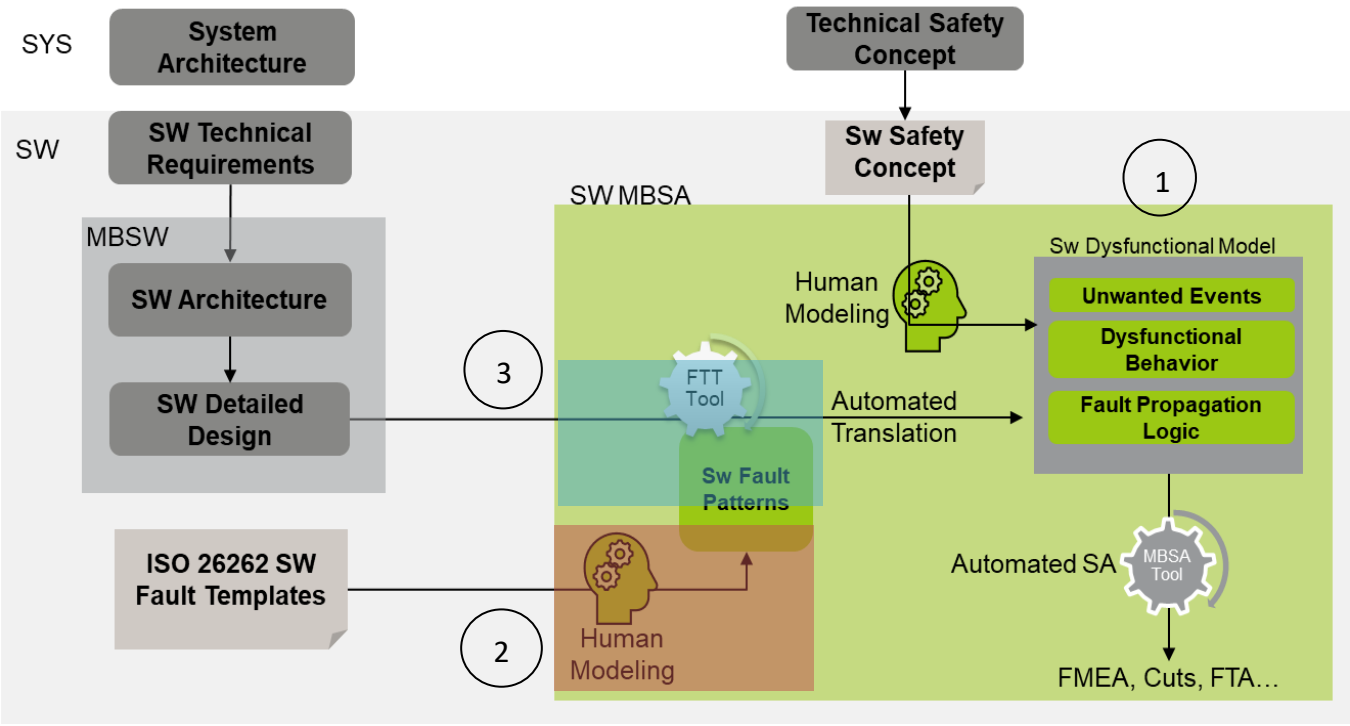

Figure 3. Overview of the methodological proposal

To this goal, we first must acknowledge that both existing software engineering processes and system safety analyses are very far from being fully model based. Indeed, the state of advancement of MDE adoption remains partial as document artifacts continue to be used.

An overview that illustrates our proposal and its interaction with the existing processes is provided in the green frame at the bottom right in Figure 3. At the top, the figure partially shows the systems engineering and system level safety assessment processes in parallel. In the middle left, it outlines a part of the software engineering process going from SW technical requirement analysis to SW detailed design. The proposal and how it relates to these existing processes. In this frame three contributions (labeled 1, 2 and 3 on the figure) can be identified.

The first element of proposal consists of a methodology covering all the steps required to perform safety analysis on automotive software architectures using the model-driven approach while addressing the challenges presented by the lack of inadequate inputs brought by the use of document centric artifacts in some parts of the software engineering process.

The second element of the proposal, also methodological, consists of using software fault patterns based on ISO 26262 software fault templates to ease the construction of the dysfunctional model.

The third element of our contribution is the definition of a tool to partially automated and ease construction of software component's failure behavior and propagation.

Step 5: Evaluation

To check that the three elements of our proposal are suitable for bridging the identified gap and addressing the issues, two complementary evaluations are made. First, our methodological proposal (including its extension consisting of the use of software fault patterns) is applied to a practical case study to show its usefulness and potential limitations. Indeed, our first 2 contributions being methodology-oriented, one valid way to show that they works is to apply the methodology as is to a real system. This application shows how from a dysfunctional software architecture, it is possible to conduct software-oriented safety analyses. Secondarily, the results from the first evaluation are analyzed and discussed. From this analysis and discussion, a perspective for future work is expressed

1.5. Outline of the thesis proposals

A review of the current state of the art on MBSA suggests that most of these approaches are systems oriented and lack clear methodological support. Moreover, some of them (especially those relying on a dedicated model) require deep understanding (in terms of modeling paradigm) and can be challenging to implement in the case of complex systems.

In addition, another point tackled in this PhD that comes from an analysis of the current state of industrial practices is that these practices for safety analysis at software level suffer from poor integration with the software development process, which can result in inconsistent safety analyses.

To address these issues, this thesis makes a methodological proposal aimed at using an MDE approach that adapts the concepts, principles, and methods of MBSA for the purpose of improving the practices of software safety analysis, taking into consideration the current state of practices (in the existing software development process) to conduct automotive software-oriented safety analysis. It defines a method for assessing the safety of automotive software architectures, and to make a tooling proposal. Our work stems from feedback obtained from case studies conducted at Renault Software Labs. The case studies were aimed at deploying the methodology and its tools and getting them evaluated by company experts.

Hence, our main contribution consists of a methodology covering all the steps required to perform safety analysis on automotive software architectures using the model-driven approach while addressing the challenges presented by the lack of inadequate inputs brought by the use of document-centric artifact in some parts of the software engineering process. Through this contribution, we propose a step-by-step methodology for defining the safety analysis context, constructing the software dysfunctional architecture, and using it for safety analyses relying on a dedicated model approach.

Contribution 1

Step by step methodology for software MBSA including a step for modeling formalism choice

Extensions to this main contribution have also been proposed. Thus, a first complementary proposal, also methodological, aims to address some challenges related to complexity brough by the limitations of a dedicated model approach. It consists of using software fault patterns based on ISO 26262 software fault templates to ease the construction of the dysfunctional model. Through this proposal, prototypes of common software fault patterns are developed and reused to build the dysfunctional model.

Contribution 2

Use of software fault patterns iteratively built on failure truth tables to ease the construction of the dysfunctional model and improve reuse

The last contribution is a tooling proposal to partially automated and ease the construction of software component's fault behavior and propagation through functional to dysfunctional logic translation. It aims to ensure a better consistency of software safety analyses with the software development process constantly with ISO 26262 recommendations.

Contribution 3

Functional to dysfunctional logic translation tooling proposal to support the methodological proposal

1.6. Thesis outline

The remainder of this report is outlined as follows.

Chapter 2 presents the thesis project background and motivations, the industrial context, and practices, as well as some key foundations related to systems engineering and systems safety. It explores current practices in light of current development of model-based approaches.

Chapter 3 provides a state of the art of both traditional and model-based safety assessments techniques in order to identify the knowledge gap that exists in the application of these different techniques to support automotive software safety analyses.

Based on the identified gap, Chapter 4 makes the research contributions. They consist of a methodological proposal that includes a methodology adapting MBSA to improve the current practices of software safety analysis in the automotive context, and the use of fault patterns to facilitate dysfunctional modeling.

Chapter 5 first shows the application of the methodology on a case study extracted from a Renault Software Labs project and demonstrates the interest and efficiency of the proposal, as well as the current limitations. To apply the methodology proposal, a choice of tools (based on the AltaRica language and on the SimfiaNeo software) has been made. Then, based on the observed limitations (relative to the methodological efficiency and to the propagation logics correctness) this chapter introduces a complementary tooling proposal aiming at automatically generating failure propagation logics.

Finally, Chapter 6 concludes, by recalling the problem, the contributions and how they address the current methodological gap identified in literature. It also analyzes and discusses the outcomes of the case study, the advantages and limitations of the methodological proposal, as well as how it addresses the current practices of safety analysis in the automotive domain. The chapter finally indicates avenues for future improvements.

Chapter 2. Software engineering and safety analysis practices in the automotive industry

Abstract: *This chapter considers current industrial practices in light of the growing trend for companies to adopt model-based engineering. More specifically, it examines software engineering and safety evaluation practices in the automotive industry, particularly in the Renault Group. It then identifies improvements that need to be implemented to make safety assessment practices compatible with model-based approaches. The chapter ends with the formulation of the general research question considered in the thesis.*

2.1. Industrial context

Starting with the introduction of driving assistance systems, the development of autonomy in vehicles is nowadays gaining popularity as major car manufacturers, tech companies and research laboratories are continuously working on developing further the technologies behind autonomous cars. It is widely said that there are important benefits to the automation of vehicles. Indeed, it is claimed by autonomous vehicle enthusiasts that fully autonomous vehicles will help to minimize the driver distraction that causes the majority of highway deaths. According to the US National Highway Traffic Safety Administration (NHTSA), 94% of highway deaths can be attributed to human error or poor driver decisions. Hence, despites being initially met with skepticism, the majority of car makers now widely believe that the future of automobiles resides in electrification, autonomous driving and connectivity. To some extent, the notion of the autonomous and connected car as a shared service is also gaining popularity.

However, despite these hopes of some self-driving enthusiasts, most car manufacturers agree that fully autonomous cars are quite a few years, or possibly decades, away as the delegation of driving responsibility to automated systems still faces many challenges, especially mastering complexity, minimizing cost, ensuring safety, achieving social acceptance in a context characterized by legal unknowns. To keep up with this rapid technological change underlying the development of autonomous vehicles, and face the associated challenges, automotive companies are adopting new methods to work more effectively and efficiently. For instance, this can be seen in the adoption of agile methods for more effective project management, or the implementation of continuous integration within the software development cycle through the automation of verification and validation tasks for instance. Also, to better master complexity we are witnessing the use of Model Driven Engineering (MDE) as an alternative to the classical systems engineering methods for the promise of better efficiency, effectiveness, cost reduction, and communication.

Due to its important reliance on embedded software, the software engineering process is an important part of autonomous vehicle development. In current industrial practices, the automotive software development process follows a well-defined life cycle that is an integral part of the system development life cycle governed by well-defined standards. Like the system development process, the software development process is subject to safety assessment as required by the ISO 26262 standard. However, in the automotive domain, software safety analyses are currently based on traditional manual techniques. Generic quality-oriented standards are used as references, and the quality of the analyses mostly depends

on the experience of safety experts. Safety analyses are not really formalized, do not allow even a partial reuse and sometimes offer approximate guarantees of safety. With regard to the evolving context, it is therefore necessary to improve current industrial practices in order to better respond to societal and economic issues.

2.2. Model-Based Systems and Software Engineering

Regardless of the sector considered, a general trend has been noted for some time in the evolution of industrial practices towards model-based approaches, particularly in systems engineering (MBSE stands for Model Based Systems Engineering). MBSE is a systems engineering practice aiming at describing, through models, concepts and languages, both a problem (need) and its solution [24].

Before being adopted in systems engineering, model-based approaches appeared in software engineering [25]. They help to ensure a certain continuity between the different stages of system and software development (requirements modeling, logical architecture, physical architecture, up to implementation), by ensuring traceability during the transitions between models.

They often rely on the use of certain high-level languages and tools. For example, the Unified Modeling Language (UML) [26] and the Systems Modeling Language (SysML) [27] languages are commonly used to model functional and organic architectures, and the behavior of software and systems. Structural diagrams (class and package diagrams) are used to model the organic architecture; use case, sequence and activity diagrams are used to model behavioral scenarios.

There are also architecture description languages that specifically support the design stage. One example is the Architecture Analysis and Description Language (AADL) [28], a standard of the Society of Automotive Engineers (SAE) initially designed for avionics, which allows the design and analysis of the architecture of embedded systems. Some architecture description languages are domain specific. In the automotive domain, this is for example the case of EAST-ADL [29]. It is a meta-model that allows to model the environment of the system, the system itself at 5 different levels of detail, and that offers extensions allowing to model the properties of the system at each level.

Still in design, semantics closer to the code, such as those of the SIMULINK [30] and SCADE [31] models, are used for the rapid prototyping of detailed software architectures and for code generation. As an example, SCADE, which is widely used in avionics, allows the modeling of synchronous systems such as flight control systems with real-time constraints and thus anticipates certain safety issues specific to concurrent systems; it also allows test automation. In other sectors, such as the automotive industry, Simulink models are increasingly used [32]

It appears that the use of domain specific models in systems engineering is well mastered as various models and languages are used at different stages of the development cycle of a system or software, depending on the point of view that we want to represent. Indeed, modeling is a well-established and successful discipline that has been practiced for decades [33]. A design supported by models offers several advantages, such as better communication, efficiency and reuse. However, many challenges remain as outlined by several survey on the challenges of the adoption of MDE [34]. One such challenge is how to ensure continuity between these different design artifacts in a true MDE fashion, for example by model transformations, in order to maintain global consistency [35]. Another difficulty, which is becoming major

in the current context of increasingly autonomous critical systems, consists in knowing how to weave these design models with specific analysis models [33], such as safety analyses.

2.3. Software engineering practices in the automotive industry

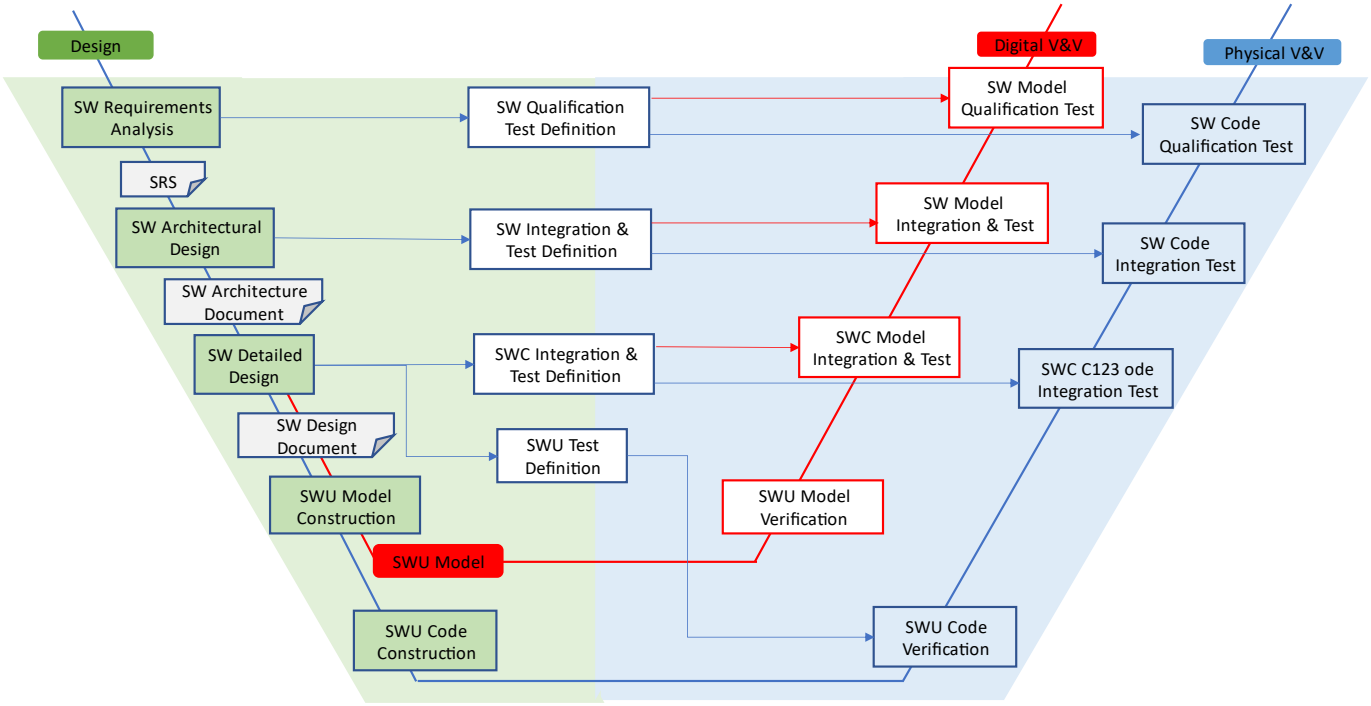

Figure 4. Software engineering process and safety assessment in the automotive industry

Software engineering practices in the automotive industry are governed by several standards. The two main ones are ASPICE (Automotive Software Process Improvement and Capability dEtermination) [36], which defines the use and evaluation of engineering processes, and ISO 26262 [12, p. 26262], which addresses safety aspects in system, hardware and software development. In Europe, the whole automotive industry must adhere to the engineering processes defined by the ASPICE standard, while the safety assessment activities to be conducted are those recommended by ISO 26262.

Based on ASPICE, Renault has defined a business process to implement these procedures and activities called Alliance Software Process (ASWP), shown in Figure 4. The ASWP includes the classic steps of a V-model process: from software requirements elicitation and architecture analysis and design, up to coding on the so-called design phase (on the left in Figure 4) and testing and integration on the Verification and Validation (V&V) phase (on the right in Figure 4). The activities produce different artifacts (mostly document-centric) that can be used to establish traceability links between the different stages of the cycle, e.g., between the architectural design and the detailed design where the main artifact is the architecture document. In the V&V phase of the cycle and in parallel with the physical V&V steps, we observe Digital V&V activities (in red in Figure 4) that start from the detailed architecture and continue throughout the V&V phase. These activities are based on the use of models as main artifacts (model-centric). In addition, a model-based horizontal traceability (even if only partial) is implemented on the same perimeter (red lines). Nevertheless, in the design phase of the cycle—from requirements elicitation to detailed architecture—the main artifacts remain document-based (SW Architecture Document, SW

Design Document, Test plan) as shown in Figure 4. It is precisely at this level, in the design phase, that the ISO 26262 standard recommends performing various safety analyses to evaluate the architecture as early as possible. The current practice does not guarantee rigorous, accurate and traceable safety analyses since the input data for the analyses (which are in the form of architecture documents) are not formal, and therefore subject to interpretation by the analysts.

On one side, the facts are clear: documentary artifacts continue to be used to link the different stages of the development process—including the stages where safety analyses are conducted—although considerable effort is being invested in the wider use of models and significant progress is being made. Thus, with the exception of the low-level processes of the design phase, where automatic tests and code generation can be performed on the basis of a software model, the link between the remaining stages (requirements, architecture) of the design phase remains document centric. Assisted design tools are used to support development activities. For example, the DOORS tool is most often used to save and manage requirements. However, the requirements that are produced and transferred to the design team are generally in Excel format (often large and difficult to use). Similarly, the modeling tool MagicDraw, which is based on the UML and SysML languages, is used to support the architectural design. The hardly structured generated models (in syntax and semantics), however, are used rather as artifacts to communicate design ideas without allowing a direct use for safety analysis. At the stage of the detailed architectural design, the Simulink tool (which implements the MATLAB language) is used to build the detailed architecture model in software components. This model, unlike the one from the higher stages, is well structured and executable.

In conclusion, the current trend in engineering is to adopt model-based approaches. They enable formalizing analyses, better communication and collaboration between interdisciplinary teams, rapid prototyping and simulation, and improved reuse. Using model-based approaches to assess software safety thus seems promising as it would help addressing the current issues that are related to safety-critical software analysis.

2.4. Safety assessment practices in the automotive industry

In the automotive industry, safety assessment is an integral part of the system and software development process. This activity is governed by the ISO 26262 standard, which calls for various safety-oriented studies and analyses throughout the development cycle—from the concept phase to final validation—in order to identify and mitigate risks. To this end, the ISO 26262 standard introduces the notion of ASIL (Automotive Safety Integrity Level) which enables risk classification. ASIL is an attribute useful for specifying the stringency level (a total of four) to be applied to a safety requirement, ranging from A (the least stringent) to D (the most stringent). In addition to these four levels, ISO 26262 includes an additional QM (Quality Management) level, applicable to items that do not have any safety requirement and that do not impose any constraint to comply with ISO 26262. For the QM level, nevertheless, the general recommendations of the applicable quality must be observed, such as those of ASPICE or those relative to software quality (coding rules, verifications, inspections, etc.).

In accordance with the ISO 26262 recommendations, safety analyses are performed at different levels of abstraction (functional, system, hardware and software) during the concept and development phases. The objective of these analyses is to identify whether and how feared events can occur in order to ensure

that the risk of their occurrence is sufficiently low. Depending on the application, this can be achieved by identifying scenarios that can lead to the violation of a safety goal using Fault Tree Analysis (FTA) [13]. The scenarios, presented in the form of a logical tree structure, can then be used to evaluate the probability of occurrence using minimal cuts or sequences [37]. Unlike the minimal cuts, which do not integrate the order of events, the minimal sequences yield the smallest combinations of events that can lead—in a precise order—to the feared event.

In addition to the use of fault trees, safety analyses can also be conducted using the Failure Modes and Effects Analysis (FMEA) method [14], which can analyze the impact of a particular failure on the entire system. An FMEA is presented in the form of a table that lists the failure modes of all components, the subsystems or components that these modes affect, the feared events to which these modes contribute and their criticality, as well as the prevention or control measures to be implemented.

Complementary to these well-known classic safety analysis methods, ISO 26262 also recommends other types of specific safety analyses, including Dependent Failure Analysis (DFA) [10, Part 6. Annex E informative]. Application of safety analyses and analyses of dependent failures at the software architectural level] and ASIL-oriented analyses [10, Part 9: ASIL-oriented and safety-oriented analyses]. However, the standard emphasizes that the latter can also be performed on the basis of fault trees [10, Part 9, page 12]. Thus, in the context of ISO 26262, the role of fault trees is not only limited to the calculation of minimal cuts, but they also constitute the entry points for DFA and ASIL-oriented analyses.

More generally, beyond identifying violation causes of safety objectives and defining control measures, the results of the analyses (whether in the form of tables or trees) also serve as support for the verification and testing activities that will take place further down the development cycle. The results are also relevant to the construction of the safety case (the safety evidence file), initiated during the concept phase and updated throughout the development cycle.

The safety life cycle according to ISO 26262 can be described in 3 main phases: the concept phase, the development phase and the post release-for-production phase as shown in Figure 5. In the concept phase, the system is defined, preliminary risk analyses are conducted, and a safety concept is built. The development phase covers the development of the vehicle at the system (block 4 in Figure 5), hardware (5) and software (6) levels, among other aspects. It also covers the safety validation and assessment (4-9 and 4-10) at the system level, as well as the production and operation planning activities (7-5 and 7-6). In the following sections, we will discuss the safety activities that take place in the concept and development phases at the system level (prior to the development activities at the software level) before focusing on the development at the software level (which is the scope of our interest). We will not discuss part 5 because it focuses on hardware development and neither part 7 because it is outside the scope of system development.

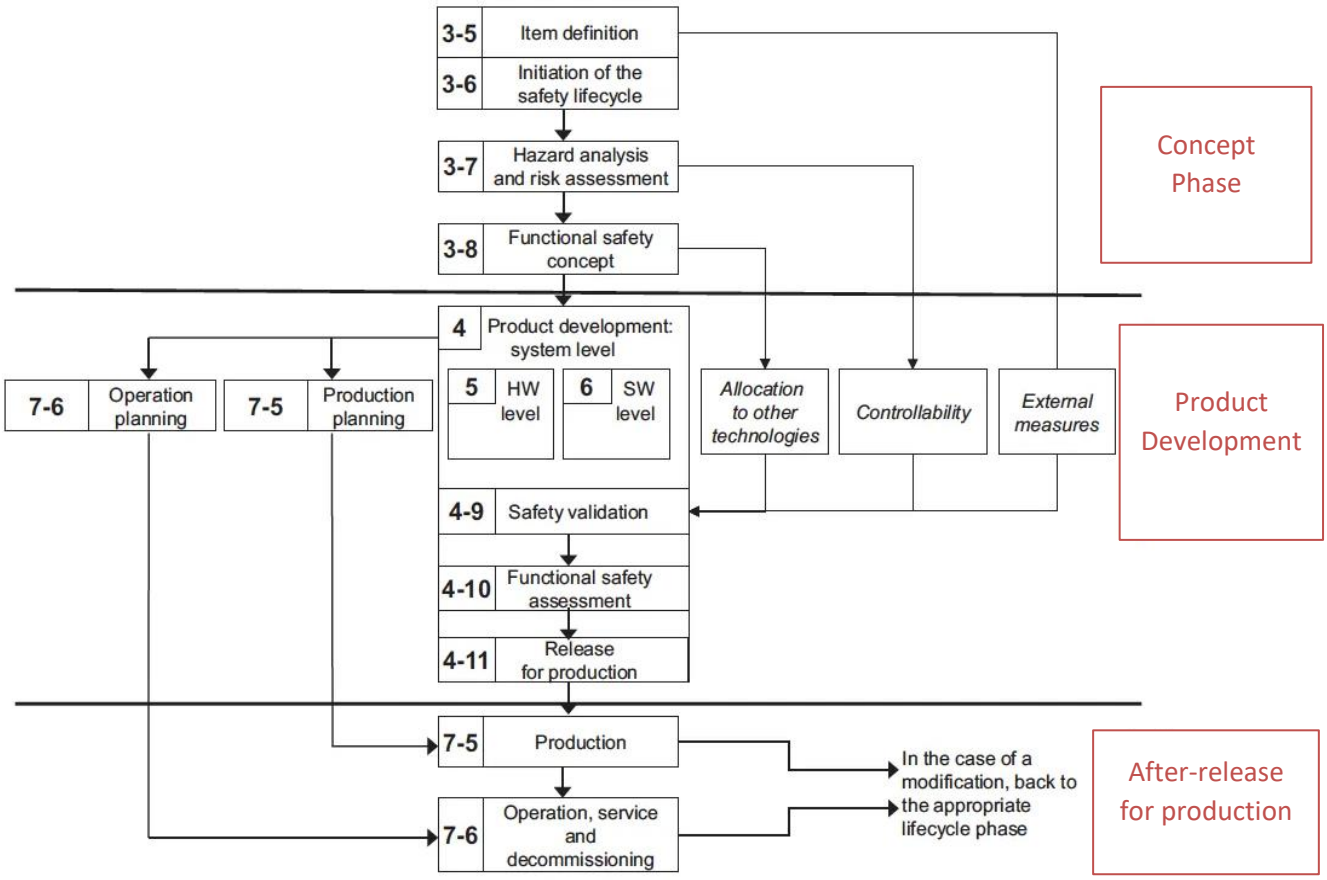

Figure 5. Safety assessment process according to ISO 26262 (ISO, 2018)

2.4.1. Concept phase

The safety life cycle starts in the concept phase with the item definition (as shown by 3-5 and 3-6 in Figure 5), which includes the description of functionality, dependencies and interactions with the vehicle driver, the environment and other system elements at the vehicle level. The item definition is followed by a preliminary risk analysis called HARA (Hazard Analysis and Risk Assessment), as shown by 3-7 in Figure 5. The HARA method can be conducted using techniques such as FMECA or HAZOP (HAZard and OPerability study). When applied to items, HARA identifies and classifies feared events and the failures that can lead to them.

The identified feared events are classified according to the ASIL levels described above, taking into account three factors: severity, exposure and controllability. The severity represents an estimate of the potential gravity of the feared event in a given driving situation, while the probability of exposure quantifies the risk. Controllability, on the other hand, estimates the relative ease or difficulty for the driver or other road users to avoid the feared event.

The results of the HARA analyses, along with their corresponding ASILs, are used to formulate safety goals; these goals are linked to the prevention or mitigation of the feared events. The results are used as input for the construction of the functional safety concept (indicated by 3-8 in Figure 5), which describes, in a

document, the measures and mechanisms necessary to implement the elements of the item architecture, as well as the safety requirements to be specified.

2.4.2. Development phase: System and software

In the system level development phase (as indicated in the “Scope of Part 4” in Figure 6), the safety assessment activities continue with the construction of the technical safety concept in 4-6. The latter describes the technical safety requirements allocated to the hardware and software (e.g., disabling a driver assistance function due to the occurrence of a failure) and the corresponding system architecture, thus justifying the suitability of the architectural design of the system to satisfy the safety requirements stemming from the activities described in the concept phase (item definition, HARA).

The technical safety requirements—often described in textual format—specify the technical implementation of the functional safety requirements (described in the concept phase). They take into consideration the item definition and the architectural design of the system (of the technical solution selected at the system level and implemented by a technical system), dealing with failure detection and prevention through the implementation of safety mechanisms.

Depending on the criticality of the ASIL level, ISO 26262 recommends conducting safety analyses not only at the system level but also at the software level [8]. This approach is used to ensure that the proposed architecture provides evidence of the adequacy of the system design to perform the specific safety-related functions and properties, and also to identify the causes of failures and the effects of faults. At the system level, these analyses are based on the architectural design of the system implementing the Technical Safety Requirement (TSR). The artifacts resulting from the requirements specification and analysis activities, the architectural design of the system and the safety analyses performed at the system level are well defined by ISO 26262. They include the Technical Safety Concept document (TSC document in Figure 3), the system architecture specification document and the report of the safety analyses performed at the system level. These documents are transferred to the development phase at the software level in accordance with the relevance of their allocation to the software components.

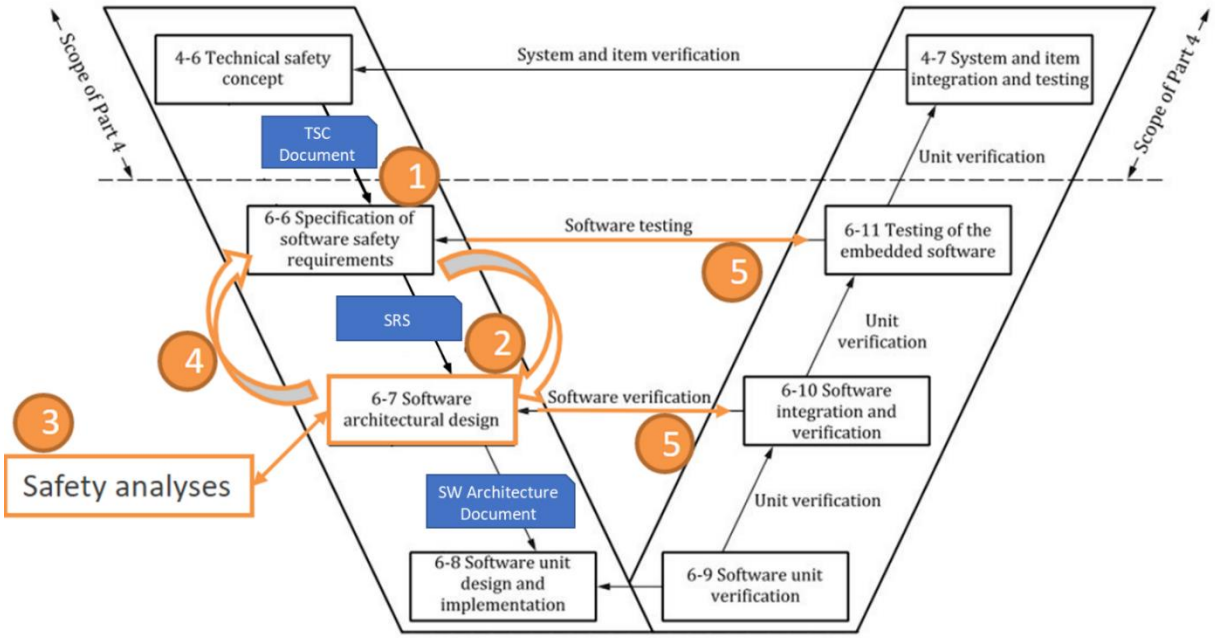

Figure 6. Safety assessment of the software

The details in Figure 6 corresponds to the activities in the block 6 in Figure 5. This figure provides further detail on the transition of the safety life cycle from the system level (indicated by Scope 4) to the software level.

In the software development phase (see Figure 6), the requirements from the technical safety concept (4-6) are translated into software safety requirements (6-6). They are transferred to and implemented in the software architecture in 6-7 through the software safety requirements specification document (SRS in Figure 3). Thus, at the software level, the safety assessment life cycle continues from the software safety requirements specification (① in Figure 6) and the transition of the feared events identified at the system level into feared events at the software level in accordance with the involvement of software functions in the associated failures. A software architecture implementing these safety requirements is then developed, as shown in ②. In order to ensure that this architecture meets the safety requirements specified in the TSC and to identify weaknesses in the design, safety analyses are conducted on the software architecture (③ in Figure 6); depending on the weaknesses found, the architectural design and requirements are updated to address them (④). Similarly, the test cases to be used in the V&V stages of the cycle are updated to ensure that the measures taken to address the identified weaknesses are adequate (⑤). Artifacts resulting from this step (including the software architectural design document and the safety analysis report) will be transferred to the detailed architectural design step for further refinement. These analyses can be done on the basis of classic FTA and FMEA methods. However, due to the specific nature of software (e.g., lack of random failures due to wear and tear or the lack of a robust probabilistic method), ISO 26262 states that the methods established for safety analyses at the system or hardware level can seldom be transferred to the software level without modification, or they might provide inconclusive results. Still consistent with ISO 26262 recommendation, less conventional safety analysis methods such as Critical Path Analysis (CPA) or software Dependent Failure Analysis (DFA) are used by analysts. While the minimal cuts from fault trees whose purpose is to calculate probabilities are

less suitable for the software perimeter, these more software-oriented analyses recommended by ISO 26262 (such as DFA and CPA) can still be conducted at this level using fault trees. However, the least is to say that these latter methods are also manually conducted, crucially lack methodological support and are not well mastered by practitioners.

As far as safety analysis is concerned, it can be stated that we observe the use of document-oriented safety analysis practices that are well consistent with ISO 26262 recommendations to perform safety analysis at software level. But we observe some challenges in translating system level safety analysis practices to software due to complexity. Additionally, the other software-oriented safety analyses lack well-defined methods to make them accessible to safety practitioners.

2.5. Required changes in current industrial practices

The critical embedded systems industry—including the automotive industry—is not immune to the MDE trend, as it is also considering with great interest these model-based approaches. In view of the aforementioned industrial practices, it appears that significant efforts are being made by automotive companies to develop the use of models in the system design and safety assessment processes as illustrated by recent proposals studies such as in [38]. In the context of automotive software engineering, the use of models has also become a common practice. This is notably the case in the lower part of the development cycle, where code generation based on the detailed software architecture. Likewise, it is possible to perform simulations and tests on this model in the context of virtual verification and validation.

Nevertheless, document-centric design practices still persist—especially during the earliest design phases of the development cycle, both for safety evaluation and design activities. This situation is explained by the absence of well-structured, executable models of the system at these stages. Thus, safety analyses (even when performed on the basis of the software architecture) are limited to representations in the form of diagrams, without a well-defined syntax or formal semantics. These practices, although compliant with the recommendations of ISO 26262 (which advocates conducting analyses on the basis of the architectural design and the requirements defined during the requirements specification phase), are essentially founded on the classic methods of fault trees and FMECA, either manual or occasionally supported by computer tools. In addition, and in accordance with the automotive reference standards (ASPICE and ISO 26262), vertical traceability (from requirements to the components implementing them) and horizontal traceability (from tests to requirements) must be maintained between the artifacts produced at the different stages of the development cycle. In current practices, however, the traceability mainly relies on textual documents, even if these are supported by tools such as Excel, DOORS or MagicDraw. With the increasing complexity of software and the growing volume of associated documents, it is no longer possible today to guarantee quality, maintain traceability and comply with standards using current methods. Moreover, the specificity of software enabled systems requires the carrying of safety-oriented analyses to ensure the soundness of the software in addition to usual V&V activities.

As a conclusion, it can be stated that the current state of software engineering is not completely model driven to the extent of allowing for instant seamless integration of safety analyses. Hence, the general research question that can be asked is: How can the current traditional safety analysis practices, that are document centric, be improved to better master complexity, promote communication between experts and teams and improve capitalization through reuse? The position of this thesis is that the clear answer

to this problem still resides in the adoption and mastering of MDE. A complete paradigm shift is therefore necessary: from document-based approaches to model-based methods for system and software development activities, as well as for their safety assessment. This potential solution is comforted by the ISO 26262 standard that presents MBE as an alternative way to achieve its recommendation. Through this possibility offered by the standard, software engineers and safety analysts have the reglementary basis if they wish, to use MDE methods to fulfill ISO 26262 safety objectives. Moreover, on this basis, we can thus consider using an MDE approach for the safety assessment of automotive software.

Chapter 3. State of the Art

Abstract: *This chapter revisits the most known classical safety analysis techniques. It discusses the current state of the art of model-based safety analysis techniques in order to characterize them. Through this chapter, we aim to provide a state of the art of both traditional and model-based safety assessments techniques. By doing so we will be able to identify the knowledge gap that exists in the application of these different techniques to support the automotive software-oriented safety analyses.*

3.1. Introduction

Since their introduction around the sixties, classical safety analysis techniques such as Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMECA) have been consistently used in various contexts (space, military, aeronautics, nuclear, naval) to assess and ensure safety of critical systems. The primary goal of these safety analyses is to identify and correct design errors that, if not addressed, could lead to failures that can cause human harm or property damage. Furthermore, the results of these analyses serve as a means to demonstrate evidence of compliance to regulation authorities.

However, as systems continue to grow in complexity, the use of these known classical safety analysis methods poses new challenges. Some of the concerns with these classical safety models is that they do not have visual commonalities with the real systems, are difficult to maintain when system design evolves and difficult to be understood by non-experts. Furthermore, today's system complexity makes their practice less efficient due to the fact that they are often manually performed.

As a result, with systems becoming more and more complex, more focus has been turned to model based approaches in support of the classical techniques. Such model-based approaches are already well practiced and utilized in systems engineering where they are referred to as MBSE (Model Based Systems Engineering). They have permitted through languages such as UML (Unified Modeling Language) and SysML (Systems Modeling language) to model different views of the same systems through the use of various diagrams (structural, behavioral, parametric) and are now becoming de facto standards throughout the industry. However, in systems safety and reliability engineering, model-based approaches remain a subject of research even if the use of models such as Petri Nets and Markov Chains (that are safety models of a sort) were known since the early beginnings of the disciple in the sixties. Model Based Safety Analysis (MBSA) aims at providing, through the use of models that are closer to systems design specification, an alternative way as a basis for safety analyses.

Today the benefits of adopting model-based approaches (in communication and quality of analysis) are recognized and acclaimed both in systems engineering and to some extent in systems safety. However, as different methods, languages and tools have been developed throughout the years, different trends with diverging views regarding the techniques used have now emerged. Also, as uncovered through the state of practices in the previous chapter, it was shown that traditional and model driven methods continue to cohabitate for both system development and safety assessment. Hence, one legitimate question that arises is how, if any, these methods, languages and tools can be applied to the practice of safety analysis at software engineering level especially in the context of automotive embedded software

safety. This question can further be broken into secondary questions which are: 1) How to model the dysfunctional behavior of the elements of the software architecture and 2) How to model failure interaction (structural and functional) between these elements. Therebefore is necessary to study them in order to define which ones are convenient to which field of application (embedded automotive software safety in our case). Hence, the goal of this state of the art is to study these techniques to characterize them.

3.2. Classical safety analysis techniques

Various safety analyses are performed as part of the design process at different development stages of critical systems. Their role is to enable identification and correction of design errors that may lead to unsafe situations. To this goal, the safety engineers draw their conclusions from traditional models such as respectively FMEA's or FTA's that they manually elaborate from the available system specification and design artifacts resulting from the early design stages. These analyses techniques can be classified depending on various factors. Depending on whether the goal is to identify the cause or consequence of a known failure, the analysis can either be inductive or deductive (whether the cause of failure is known, and the goal is to identify the consequences or vice versa). A deductive or top-down analysis method seeks to identify the causal factors of a known hazard (the hazard is known, and the goal is to identify the contributing cause) whereas an inductive or bottom-up approach seeks to identify the consequences of a known failure. This subsection presents the most common classical safety analysis techniques.

Preliminary Hazard Assessment

Preliminary Hazard Assessment (PHA) is a qualitative hazard analysis technique that is performed on the preliminary design at the earliest stage in the life cycle (concept phase). Its purpose is to identify safety critical areas, provide an initial assessment of hazards and define control measures. To carry out a PHA, a team of people that are familiar with the system conduct a brainstorming around the preliminary design of the product. Although various methods can be used, PHAs are often carried out with the help of checklists. Typically, the result of the study is tabular and contains the list of identified hazards, their effect, severity, co-effectors, probability, and controllability. PHA is an inductive technique.

Functional Hazard Analysis

Functional Hazard Analysis is a qualitative and deductive safety analysis technique that is used to assess the top-level design from a functional viewpoint. It aims to identify which functions of the system contribute to hazards. Like PHA, the result of an FHA study is tabular. An FHA table contains the list of the functions for which are defined several attributes such as the failure description and type (omission, commission), the severity, means of detection, recovery plan and design recommendations, verification means etc. Different FHA methods exist depending on the field of application (automobile, aeronautics). Although they may slightly differ by name or the content of their table, they share the same objective which is a function-by-function analysis.

Hazard and Operability Study

HAZard and OPerability study (HAZOP) [39] [40] is a systematic and structured approach that relies on a multidisciplinary team effort to identify potential hazards that could result from deviations from design

or operating intentions. To analyze safety using the HAZOP method, a system or process is broken down into parts or steps respectively. Then a competent team is assigned to each part/step the method proceeds through 4 major steps. First the scope, objectives and responsibility are defined, and a team is selected. Secondly, the study is planned. Third, the study is carried out. To do so, the system of interest is divided into parts. For a selected part a design intent is defined. Then the analysis is carried out by identifying deviations from the design intent though the use of guide words repeated for each element. The last step consists in documenting and following up the results of the analysis, what differentiate HAZOP from the other previously discussed high-level analysis techniques is that it focusses on safeguarding from deviation from the design or operation intention. Thus, the intention here is different as the technique focused on foreseeing risks related to abnormal use of the system.

Failure Modes and Effects Analysis

Failure Modes and Effects Analysis (FMEA) is an inductive and systematic safety analysis technique that is used to study the effects of individual component failure modes on a system. Using FMEA, one can check whether a proposed design including components and their known failure modes fulfill the system safety requirements. To conduct a FMEA study, the failure modes of components are first defined. Then knowing these failures, one seeks to establish whether they lead to the failure of other components, subsystems or the whole system. Like in most other inductive approaches, the result of an FMEA is tabular. Although the content may slightly differ depending on the application, an FMEA table usually contains the list of components, their individual failure modes, and the effects of these failure modes on other parts of the system. In some cases, it can contain a parameter that specifies the criticality of the failures. In this case, it becomes an FMECA (Failure Modes, Effects and Criticality Analysis). The result of FMEA can serve in two possible ways. First, due to its systematic nature, the results of an FMEA can serve as a proof that a proposed system design fulfills the system safety requirement and thus accepts the proposed design. Secondly, it can serve a basis for recommendations for changes, additional measures (further verification) or procedures to follow (operation or maintenance).

Fault Tree Analysis

Fault Tree Analysis (FTA) [13] is a deductive (top-down) safety analysis technique used to determine the combination of causes called basic events that may lead to a known feared event called top event. In the FTA approach, a top event corresponding to the violation of a system safety requirement is known and serves as an entry point of the analysis (it is assumed that a feared event previously described by the PHA occurred). Thus, the goal of the analysis is to determine the combination of preconditions leading to the top event. To determine all the possible causes for the condition to occur, all necessary preconditions are described at lower levels with logical gates AND or OR as shown on Figure 7. This process is repeated until arriving at basic events that have computable probabilities and do not require any further development. To simplify the analysis, the initial tree is converted to an equivalent reduced tree using Boolean logic equivalency. The reduced tree is also equivalent to a minimal cut which represents the smallest combinations that are necessary to lead to the feared event. Various secondary analyses such as sensibility analysis (to determine the most contributing factors) or probabilistic calculations (to determine the occurrence probability of the top event) can then be conducted based on the reduced FTA.

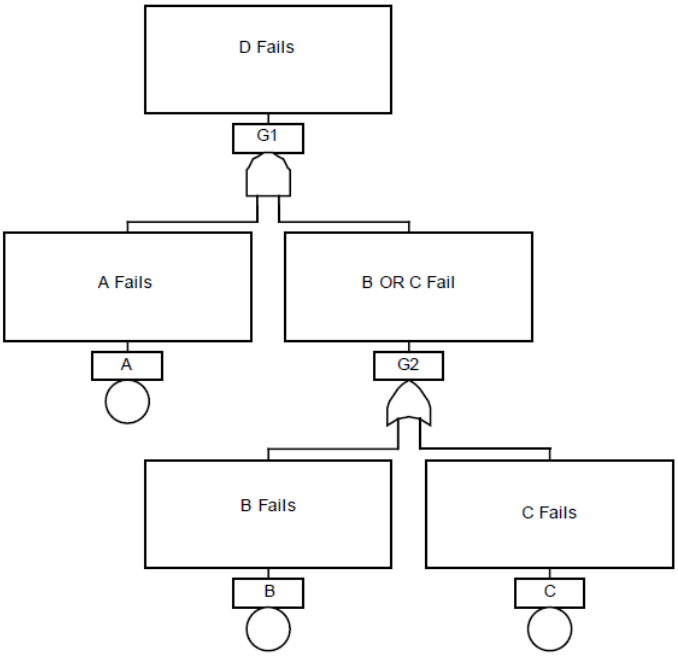

Figure 7. A simple Fault Tree

Software Fault Tree Analysis (SFTA) is a version of FTA dedicated to software that has been proposed by Leveson in [41]. SFTA is defined as a technique that interfaces with hardware (physical system) fault trees to allow the safety of the entire system to be maximized. Its goal is to find failure modes or failure scenarios which could lead to the specified safety failures, or alternatively, to show that the software logic contained in the design is not likely to produce any safety failures. SFTA requires a representation of the program logic such as a detailed design and a list of safety failures (or failure conditions) to be analyzed (that can be derived from the safety requirements). Methodologically, SFTA proceeds in a similar manner to system or hardware FTAs. Hence, SFTA begins with an assumption that a loss of a function or feature has occurred in the bounds of software logic. The code responsible for the output is considered as the starting place for the analysis. Specifically, it is assumed that the failure event occurred in the bound of an if-then-else, assignment, function call or while statements. Then working backwards as in hardware FTA, one deduces how the program got to the part of the problematic code (where the failure occurred). For instance, as explained by Leveson in [41], in the case of an if-then-else structure, the code is divide into 3 parts: the condition, the then part and the else part. If the failure event occurred in the then-part of the structure, then one deduces that the entry condition to the if-block must have been true or the then-part must have failed leading to the failure.

3.3. Model-Based methods languages and tools in safety

The adoption of model-based approaches to conduct safety analyses has led to practices grouped under the acronym MBSA (Model Based Safety (and) Assessment) which allow, through specific formalisms and languages, to capture an "authoritative model of the system" on the basis of which different types of

analyses can be made [17] [30]. MBSA can be defined as a set of safety analysis technique based on the use of safety models that are similar to or based on design models to conduct safety analysis. MBSA shares with MBSE the same vision which is to move away from document-based processes, in order to adopt a model-based approach where a model is the principal artifact in the development cycle. However, whereas MBSE models the nominal (non-failure) functional behavior of a system, MBSA models its fault (dysfunctional) behavior. Indeed, safety analyses aim at identifying whether the system, as modeled, has weak points. They can be performed either to analyze if a system component failure can induce a serious failure at the system level, or to determine what are the possible root causes of a system failure.

Adopting a model-based approach in safety engineering consists in building a dysfunctional model of the system that shows the system behavior in case of a failure (reasoning by failure propagation) from which the traditional analyses minimum cuts, can be derived [16]. This notably allows an easy and quick generation of new safety analyses in case the system architecture evolves, therefore reduces the cost and improves the quality of the safety analysis process. Additionally, MBSA also seeks to integrate different safety analysis models (FMEA, FTA) in a single authoritative model shared (between system and safety) or single model (dedicated to safety). In this subsection we discuss MBSA methods, languages, and associated tools as well as Architecture Description Languages (ADL) that support model-based safety analysis features.

3.3.1. Figaro

Figaro [43] is one of the earliest safety modeling languages. It was created by Electricité De France (EDF) in the late 80s early 90s. Figaro is a modeling language that aims at playing a unifying role as its models can be transformed into conventional models, such as fault-trees, Markov chains and Petri Nets through a variety of compilers and translators [43]. Figaro is object-oriented; it allows the definition of types organized through inheritance relations. A Figaro model consists of two parts: the knowledge base consisting of a declaration of object types written in a generic form [44, p. 3], and the list of objects of the system to be studied [45]. In Figaro, the systems elements referred to as interacting objects; they have no hierarchy (any object can interact directly with all the other objects in the model in various ways). Figaro is supported by the KB3 [44] tool developed by EDF for its internal use.

3.3.2. AltaRica

AltaRica [46] is a high-level modeling language dedicated to risk analysis that supports safety, reliability and performance analyses. It was created at the end of the 90s by LaBRI (Laboratoire Bordelais de Recherche en Informatique, Bordeaux). The initial version of the language was born from an effort to create a language that stands at a higher level that is both formal with a well-defined semantic and graphical that can be compiled into lower-level formalisms such as fault trees, Petri nets or Markov graphs. In this sense the intent behind AltaRica is similar to Figaro described earlier. The semantics of the initial AltaRica was defined in terms of constraint automata (represented by a states/transitions system with input and output flows) [46]. In each state, so-called a mode, the automaton computes the values of output flows from the values of input flows to realize a transfer function.

The second version referred to as AltaRica Dataflow is a generalization of both Petri nets and block diagrams [47]. It has inherited from the first version the notions of states, events and transitions and relies on the notion of state automata. In the AltaRica dataflow, model elements are expressed in terms of

nodes. Each node is composed of states, events, transitions and assertions [48]. States are declared using domains (an enumerate comprising several states). An AltaRica model describes a system through domains, variables, and a hierarchy of nodes where each component can incorporate multiple sub nodes. A domain describes a set of Boolean, integer, enumerated or abstract values that a variable can take. The nodes describe the behavior of system components through state variables, events, transitions, and assertions while the transitions describe changes in state. An illustration of an AltaRica Node is provided in Figure 8. As it can be seen on the figure, a node interacts with the environment in two ways: 1) through events and 2) through flow variables. Like a transfer function, assertions describe the relationships between input flows, state variables and output flows. AltaRica dataflow also introduces the notions of hierarchical description and events synchronization that make it possible to represent remote interactions between components. However, it cannot handle looped systems and bidirectional flows natively. Despite these limitations the AltaRica Dataflow is by far the version that is found in most AltaRica-based tools.

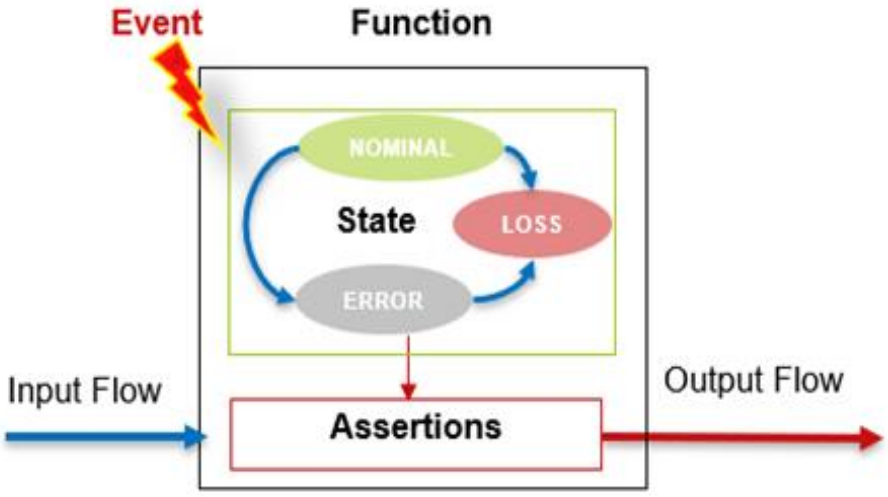

Figure 8. An AltaRica node

To improve the language, a third version (AltaRica 3.0) [49, p. 3] was introduced. Its underlying mathematical model is based on Guarded Transition Systems (GTS) formalism that makes it possible to design components that allow simultaneous bi-directional failure propagations and to handle looped systems. AltaRica 3.0 is prototype oriented [50, p. 3]. Its semantics is based on the notion of reusable hierarchical patterns in contrast to the previous versions that are object oriented and based on the notion of reusable hierarchical components. AltaRica 3.0 aimed to improve the expressive power of its predecessors by making possible the modeling of looped systems and bidirectional flows. Moreover, a timed extension of the language has been proposed in [51] and [52] to extend the capability of AltaRica to real time systems that have strong timing constraints.

Today several tools currently support AltaRica: they include Cecilia Workshop [53] (developed by Dassault Aviation), SimfiaNeo [54] by Apsys-Airbus, and AltaRica Studio [55] (developed by LaBRI), based on the dataflow version of the language. More recently, we can also find Open AltaRica [56] from SystemX, based

on the more recent 3.0 version of the language. Moreover, AltaRica has been used in a couple of large-scale European projects aimed at the enhancement of systems safety analysis such as ESACS [57] or ISAACS [58].

3.3.3. xSAP / NuSVM-SA

xSAP [59], previously FSAP [60], is a tool for safety assessment of synchronous finite-state and infinite-state systems based on symbolic model checking techniques. It consists of a graphical user interface (FSAP) and a verification engine (NuSMV-SA) based on the NuSMV model checker [61]. xSAP provides library-based specification of failures in order to automate model extension for safety analyses including FTA, FMEA, minimal cuts or Common Cause Analysis (CCA). As explained by Bozanno [62] and outlined in Figure 9, the starting point of the xSAP methodology is a formal model of the system called system model written in some formal language and including only the nominal behavior of the system. Then safety engineers proceed to enrich the behavior of the system model by injecting failure modes retrieved from a library of generic failure modes to produce an extended system model. To make it possible to verify the extended system model behavior with respect to the desired functional (nominal behavior) and safety requirements (degraded behavior), design and safety engineers define Requirements that are written using temporal logic formulas or loaded from a generic safety requirement library. The resulting model can now be used to assess the behavior of the system against the functional and safety requirements, by running the NuSMV-SA model checker which is a formal verification engine.

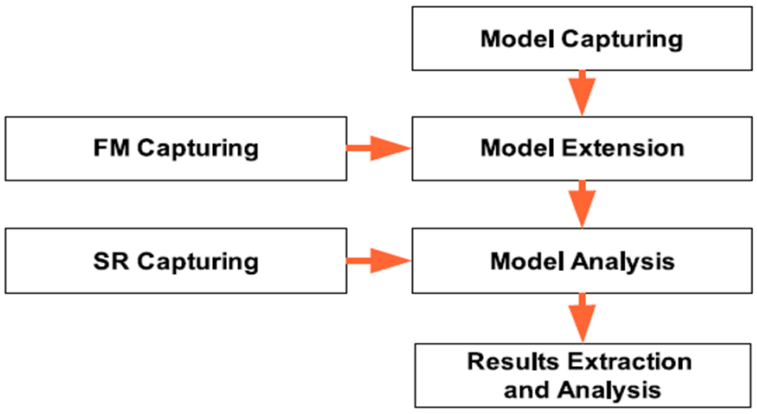

Figure 9. 4 xSAP Methodology [62]

3.3.4. Safety Analysis Modeling Language (SAML)

SAML (Safety Analysis Modeling Language) [63] is a formal modeling language dedicated to safety. It was originally developed as a specification language that is independent of modeling tools and paradigms and model checking tools. In particular SAML aims to unify the quantitative and qualitative approach to safety [64] . In this respect, SAML is presented as an intermediate language between MBSA tools and model-checking tools as shown in Figure 10.

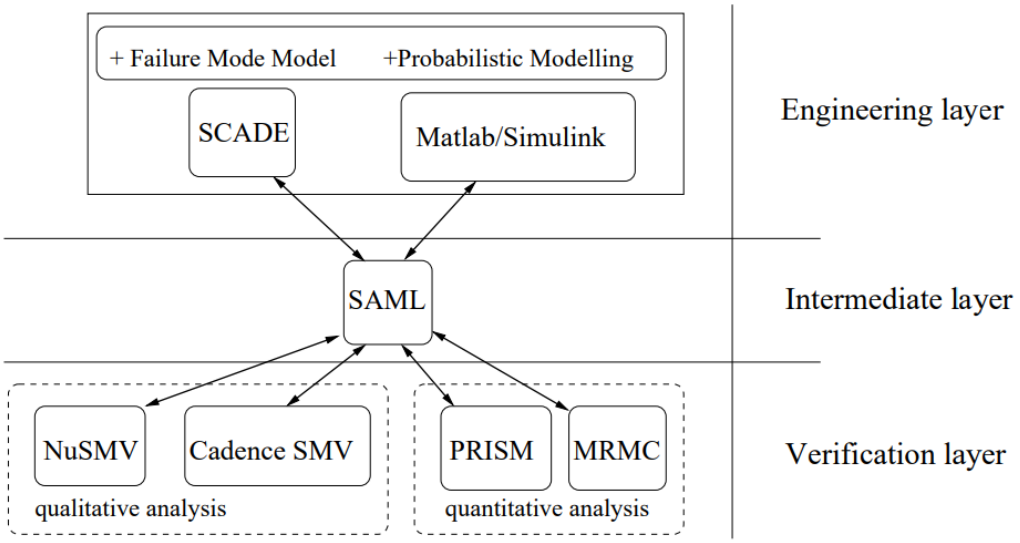

Figure 10. SAML as intermediate safety modeling language [63]

3.3.5. Electronics Architecture and Software Technology – Architecture Description Language

EAST-ADL (Electronics Architecture and Software Technology – Architecture Description Language) [65] is an architectural description language specific to automotive embedded systems. It is the result of an effort by various European research projects (ITEA, MAENAD,) starting around the early 20's. EAST-ADL is a metamodel that can be used to describe automotive electronic systems through a standardized data model. The EAST-ADL metamodel consists mainly of three parts: the system model, the environment model and extension packages as shown in [66] [67] [68]. The system model is defined in 5 levels of abstraction at which software and hardware features are modeled with different levels of details. The extensions packages include requirements, variability, safety, behavior, timing, and generic constraints. Each extension package can reference the core elements (system model) at all abstraction levels. The aspects covered by EAST-ADL include vehicle features, requirements, analytics functions, hardware components, software components as well as communication.

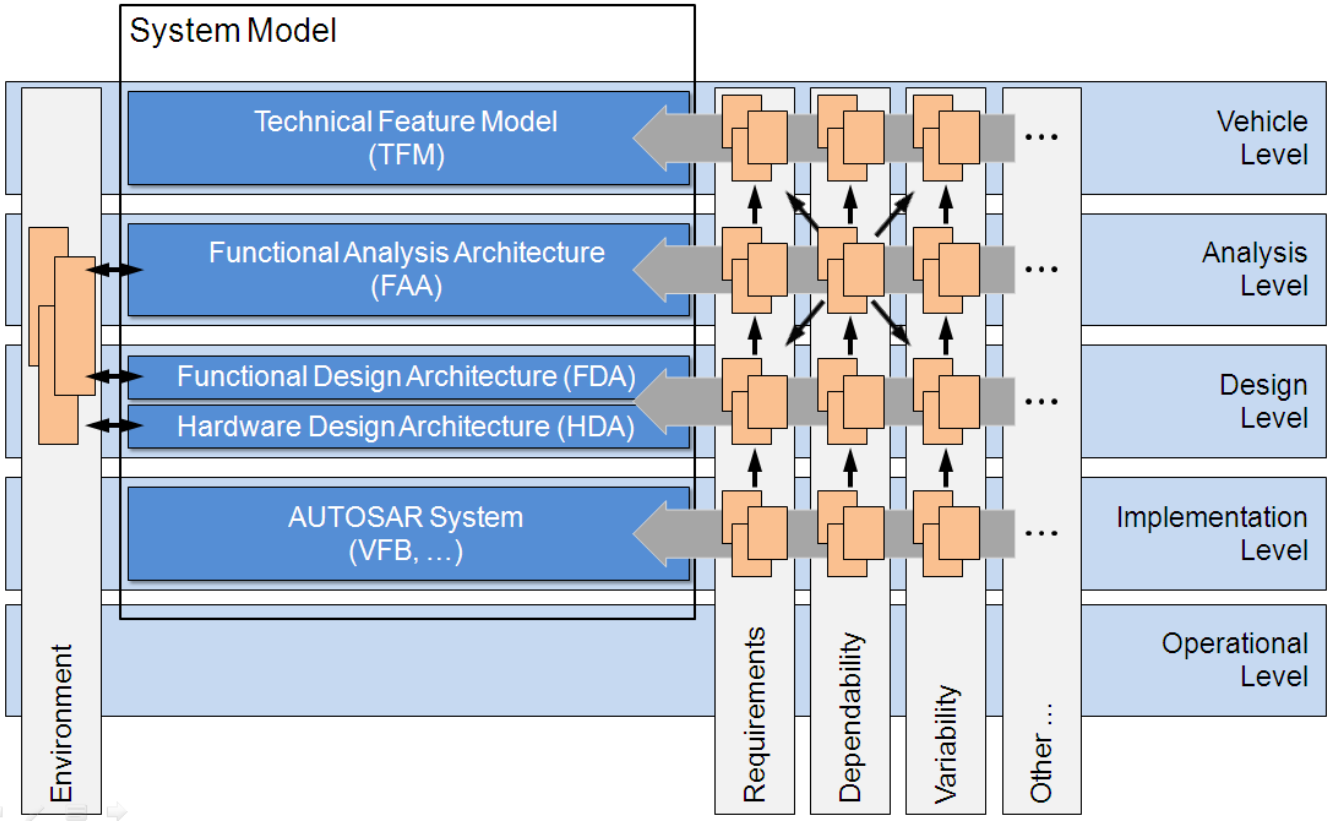

Figure 11. EAST-ADL Model Organization Overview [69]

3.3.6. Architecture Analysis and Design Language (AADL)

AADL (Architecture Analysis and Design Language) [28] is a SAE standardized architecture description language. It was originally developed for avionics (Avionics Architecture Description Language). But now, the AADL language is a standard that allows modeling the software and hardware architecture of embedded real time in various domains including space [70], train control [71], medical devices [72] [73] or automotive [74]. The Language consists of a precise semantics that allows the user to model the hardware and software components and their interaction. The AADL standard has been extended with an error model annex to support architecture fault modeling and automated safety analysis [75]. Thus, thanks to its error model extension EMV2 (Error Model Version 2), AADL can be considered as an architecture description language that is compatible with MBSA. In [76] and [77], Delange et al. described how AADL supports safety analysis following the ARP 4761 process.

There are a couple of tools that support the AADL language. They include OCARINA and OSATE (Open Source AADL Tool Environment) [78] [79]. OSATE is the official open-source modeling tool platform for the language and is based on Eclipse. In this environment, software architects can design and analyze models, and then generate some of the implementation code.

3.3.7. Failure Propagation and Transformation Notation (FPTN)

Failure Propagation and Transformation Notation (FPTN) [80] is a graphical method for expressing the failure behavior of systems with complex internal structures (including software). It is a notation that aims to reflect both system architecture and the way in which failures within the system interact. FPTN notation is analogous to traditional data flow-based design notations. However instead of showing normal data flow between elements in a system, it describes the propagation and transformation of failures. The basic element of an FPTN model is a module as shown on Figure 12. From a semantic point of view, a software module in FPTN is represented by a simple box with a set of input and output failure modes. Inside the box are listed a set of predicates (equivalent to AltaRica assertions) describing the relationship between the input and output failure modes of the module. These predicates correspond to the sum of minimal cuts of fault trees for each output failure modes to form what Fenelon et al. describe as a forest of horizontal fault trees [80]. FPTN also provides representations for failure modes which can arise inside a module classified into various categories (including timing failures, value failures, commission failures and omission failures) and for exceptions handled by modules (analogous to AltaRica Internal failures). From a methodological standpoint, FPTN is a hierarchical and bottom-up technique. For each module, a software fault tree is constructed. Although it aims to be graphical, the FTPN method relies on a subset of the FTA technique called HFTA (Hierarchical-FTA) based on hierarchical views of the system (systems, software, etc.). It starts with a top-level FTA based on the top-level view (system perspective) and proceeds to top level view (software perspective) and lower architecture levels. The HFTA methodology relies on seeking, from a systems point of view, what can cause a function to fail doing what it is supposed to do correctly. Potential root causes may include communication with resources such as memory or processor, internal failure due to software, or physical defects.

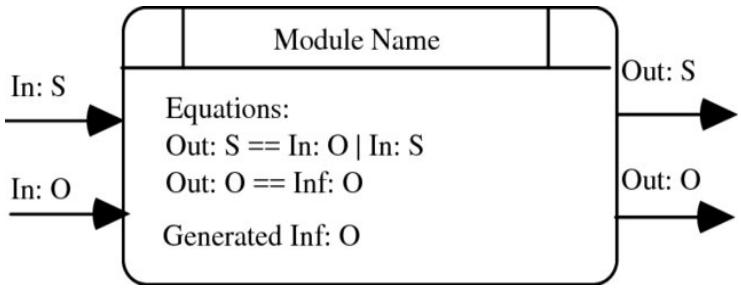

Figure 12. Elements of the FPTN Notation [81]

3.3.8. Hierarchically Performed Hazard Origin and Propagation Studies

HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) [82] is a safety analysis method that integrates and extends classical techniques such as Functional Failure Analysis (FFA), FMEA and FTA to enable integrated assessment of complex systems from functional level through low level of component failure modes. An overview of the safety analysis process in HiP-HOPS is shown in Figure 13. In HiP-HOPS as it can be seen on the figure, all safety analyses are performed on a consistent hierarchical model of the system. The types of safety analysis covered by HiP-HOPS include FFA (exploratory functional failure analysis), failure behaviors at component level and FTA.

As explained by Papadopoulos et al. in [82], the first step consists of a FFA which he defines as an exploratory functional failure analysis of a conceptual design of the system. FFA proceeds through a systematic function by function examination of potential failure modes (including loss of function, the unintended delivery of function and malfunctions such as early or late deployment). FFA is conducted on the basis of an abstract and conceptual functional model constructed as a block diagram following the determination of the effects, criticality and the potential for detection and recovery of each failure and the identification of plausible combinations as well as their effects and criticality. The second step in the HiP-HOPS method consists of an analysis of failure behavior at component level. At this level, an extension of the FMEA method called IF-FMEA (Interface Focused-FMEA) is used to describe the failure behavior of the basic hardware and software components resulting from the decomposition and refinement of the hierarchical model. Papadopoulos et al. argue that IF-FMEA advances traditional FMEA by providing a systematic way to examine the detection, mitigation and propagation of failure across the component input and output interfaces.

HiP-HOPS is supported by the Safety Argument Manager (SAM) [82] [83], a tool that was developed by the University of York to support the production of safety cases. HiP-HOPS is also supported by a tool by the same name.

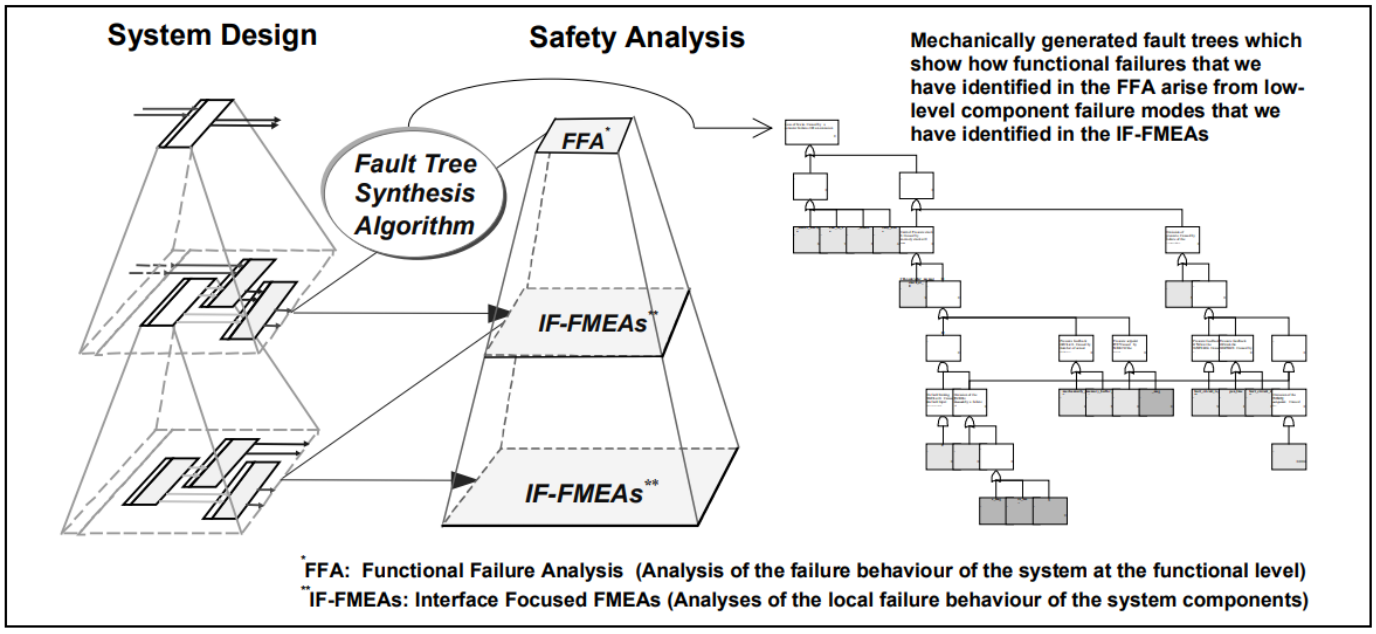

Figure 13. Overview of Design and Safety Analysis in HiP-HOPS [82]

3.3.9. Systems Theoretic Process Analysis (STPA)

STPA (System-Theoretic Process Analysis) is a relatively new hazard analysis technique based on STAMP (System-Theoretic Accident Model and Processes), an extended model of accident causation [84]. STAMP is an accident model that is based on system theory. In classical safety analysis techniques such as FTA or FMEA, the assumption is that accidents are caused by the failure of individual system components. However, in system theory on which STPA and STAMP are based, the system is treated as a whole, and

more focus is put on its emerging properties resulting from the interaction of the system components. As a consequence, STPA assumes that accidents can also be caused by unsafe interactions of system components (including humans), none of which may have failed. Therefore in STAMP and STPA, safety is treated as a dynamic control rather than as a failure prevention problem [84] .The STPA method proceeds in 4 basic steps as shown in Figure 14.

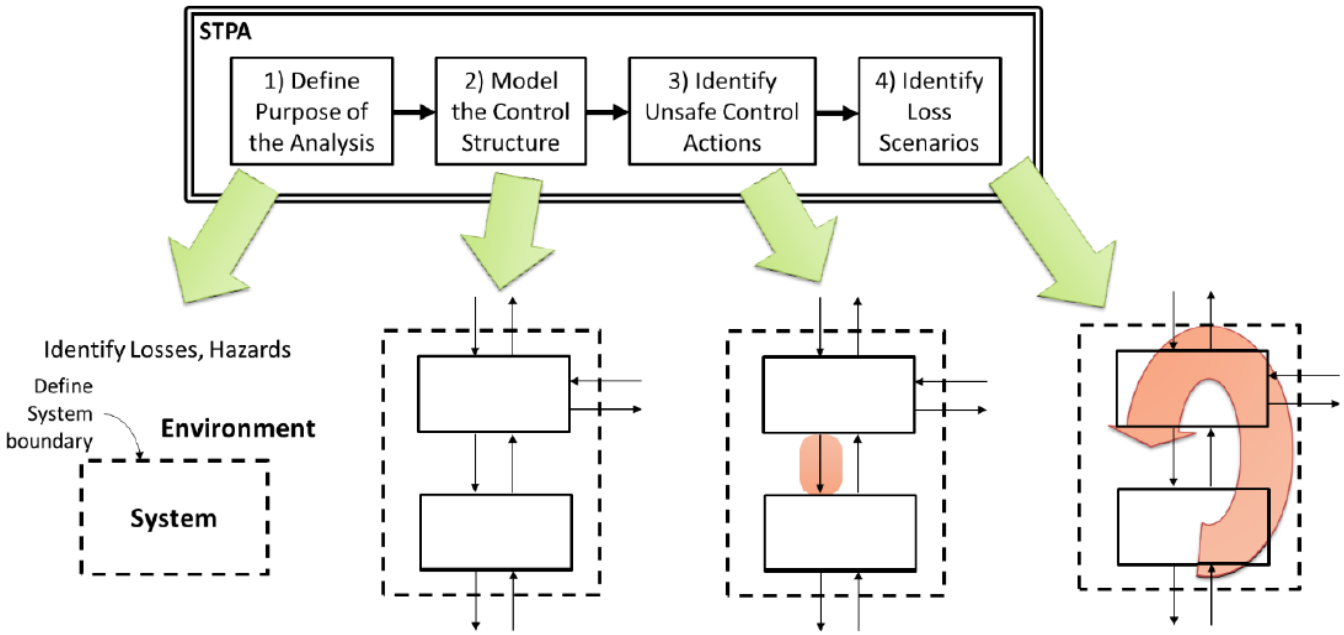

Figure 14. Overview of the basic STPA Method [84]

The first step is to define the purpose of the analysis. It also identifies hazards, losses and defines the boundary of the system. The second step consists of building a model of the system that captures functional relationships and interactions through a set of feedback control loops. This model is called a control structure. The third step is to identify and analyze Unsafe Control Action (UCA) in the control structure to examine how they could lead to the losses identified in the first step. The fourth step identifies the causes of the identified unsafe control actions through the creation of various scenarios leading to a loss.

STPA based studies has been used in a variety of industrial contexts such as in aeronautics [85] [86], but also in automobile [87] [88] as well as other critical systems fields. What most of these studies have in common is that they show that STPA helps in identifying unsafe scenarios that are not identified using the traditional safety analysis techniques. Such evidence can be found in [85]. In this regard, if compared to other classical and MBSA safety analysis methods, STPA is indeed a complete change of paradigm as it has been argued by Leveson. Furthermore, STPA allows taking into account human factors.

3.4. Analysis of the different MBSA Methods

Since their apparition starting in the early 90s several MBSA methods such as FPTN or HiP-HOPS have been proposed. Today several modeling languages support MBSA. They share some commonalities and have some differences. As such, some such as AltaRica and Figaro are language based, others like xSAP are tool based, while others such as FTPN are more methodical. What are their characteristics?

Some of these languages such as AltaRica, SAML or Figaro are dedicated to safety. Others such as AADL or EAST-ADL are Architecture Description Languages (ADL): they extend their core semantics and syntaxes to support safety analyses through profiles and error annexes. The use of AADL for safety analysis in industrial context has been reported in many works and can be found in [43]. The error annex allows the practice of MBSA through the use of the model extension approach. However dedicated AADL models (built for the purpose of safety analysis) have also been used. One advantage of the language is that it has semantics to fully cover the real time aspects of embedded software like that one can be concerned with in the automotive software (scheduling, memory, communication). In this regard we believe that AADL can be suitable for identifying safety issues coming from the lower level of the software architectures because the language is embedded system oriented and rich in its real time semantics (scheduling, priorities, timing). Multipurpose modeling languages such as UML or SysML are also used through profiles and meta models to describe the safety behavior of systems. Tools such as Cecilia OCAS [18], FSAP [23] or Safety Architect [26], have been developed to support MBSA. Today, it can be stated that some of these techniques have been introduced and successfully applied in different industrial contexts. AltaRica for instance has been applied in various European industrial research projects such as the ESACS (Enhanced Safety Assessment for Complex Systems and ISAAC (Improvement of Safety Activities on Aeronautical Complex Systems) projects. Both were European community funded projects launched in the early 20's that sought to support the safety assessment of complex embedded systems. Among the earlier dedicated model approaches, FPTN can be credited as one of the earliest graphical safety analysis approaches that sought to integrate both FTA and FMEA in systems like architecture. Although the notation was not as structured as most of the more recent MBSA language, FTPN had all the ingredients to make it a pioneering MBSA method. First it proposed to adopt a graphical system-like architecture as the basis for safety analysis. Secondly, it introduced the modeling paradigm that described system components using abstract states and flows, which was important as this paradigm will be later adopted by most other prominent MBSA techniques such as AltaRica. Finally, FPTN was introduced as a safety analysis method focused on complex intensive software systems, which makes the approach interesting to consider in embedded systems. However, very little is known about the tooling support of the method. Nevertheless, we can generally state that, apart from STPA which focused more on unsafe interactions between systems components, all the described MBSA approaches focus on failures related to intrinsic behavior and interaction of systems components. Leveson et al. compared the safety analysis process of ARP 4761 with STPA, using the wheel brake system example in ARP 4761. Their results show that STPA identified hazards that were omitted by the ARP 4761 process, particularly those associated with software, human factors, and operations. Similarly, in [89], Placke et al. performed a case study involving several driver assistance systems including advanced brake controls, advanced engine control, and advanced adaptive cruise control. Their result showed that potential conflicts that would prohibit safe and successful operation are also efficiently identified thanks to STPA, allowing engineers to develop suitable controls that prevent these conflicts. The described MBSA methods also differ in their approach. According to Lisagor et al. in [42], these MBSA methods can be classified according to two main criteria, the dysfunctional model construction and the semantics of component interfaces.

According to the first criterion related to the process for defining the MBSA model and its relationship with the system design model, the MBSA model can either be an extension of the design model or a

dedicated model [42]. We find in [90] an example of an extended model, where a nominal functional model is first constructed during design to which failure modes are added for the purpose of safety analysis. The key advantage of the model extension approach is the consistency, by construction, of the safety analyses and the design model of the system. Furthermore, development and safety processes can share a common modeling environment, languages and tools [42]. However, it has some drawbacks. One is that it does not allow independence between the system and safety models which is argued by some practitioners to be an important principle. In the case of a dedicated model, a distinct 'standalone' dysfunctional model is built by the safety engineer based on his understanding of available information from design documents and functional models. The key advantages of this approach are that it is more pragmatic to implement, as it ensures independence and separation of concern (between safety and engineering disciplines). However, one of its drawbacks is that it requires supplementary means for ensuring consistency with the design model.

The second criterion is related to the dysfunctional model semantics (components behavior) and the type of information that is conveyed through the component interfaces, either nominal or failure flows [14]. This criterion leads to distinguishing Failure Logic Modeling (FLM) [80], [91] (that uses failure flows) and Failure Effect Modeling (FEM) [80] (that uses nominal flows). Dependency between the components of the model is defined based on the type of flows. In the case of FLM, the dependencies are captured in terms of deviations of their behavior from design intent and failure modes exhibited by other components of the system. In the case of FEM, the model components carry the functional behavior of the system. The interfaces between components are captured through abstracted real flow of data, matter or energy [42]. Early proposals adopted the FEM approach [90] but, as the discipline evolved, FLM has gained prevalence and most of the pioneering MBSA methods such as FPTN [80], HiP-HOPS [82] and AltaRica [46] rely on it. In addition to these methods, a number of hybrid methods have been proposed [42]. They utilize the architecture of the design model whilst tasking safety engineers with characterizing behavior of individual components for the purpose of the safety assessment. Examples are the integration of HiP-HOPS with Matlab/Simulink proposed by Papadopoulos and Maruhn [92] [42] and the Error Modelling Annex of the Architecture Analysis and Design Language (AADL) [93].

Another aspect that is less discussed in the state of the art is the essential characteristic of the failure mechanism modeling within the elements of an MBSA model. In regard to the previously discussed methods, it can be observed that, although their semantics differ, all the associated MBSA methods tend to focus on modeling, on one hand, the cause of failures at component level, and on the other hand their immediate consequences. This can be seen in AltaRica nodes where state transition and events are used to describe the causal factor of components failure behavior whereas assertions are used to describe the failure effect. In FPTN we found a similar construct with causal factors being described using categories of possible failures and exceptions whereas the failure effects including propagation and transformation are described using the so-called forest of fault trees. The same pattern can be found in the xSAP methodology, the use of its generic fault library, and HiP-HOPS with its IF-FMEAs that capture input/output relationships. The overall failure propagation at the architecture level is established through dependency links depending on the chosen type of failure dependency semantic as described earlier (FEM or FLM). In conclusion, it can be said that in terms of modeling most of the MBSA approach focuses on modeling the cause and effects of failures at component level.

Consequently, it can be argued that such MBSA approaches adopt a bottom-up approach. Far from being a disadvantage, the bottom-up approach in modeling is what enables the automation of safety analysis thanks to the causal factors of failures and their propagation modeled at component level.

3.5. Conclusion

In this chapter, several MBSA approaches were discussed and categorized. Each approach offers advantages and disadvantages. It can be observed that most of the described methods and tools are system oriented. Thus, in this context, one interrogation this bring is whether the current MBSA methods, tools & languages can be applied automotive software safety analysis if we were to adopt an MDE approach. This question is easy to answer as many of the analyzed MBSA approach, despite having effective tooling support, have some considerations for safety analysis at software level. Furthermore, language such as AltaRica, despites being initially designed for high level systems safety analysis, are generic and semantically structured enough to enable the representation of the dysfunctional software architectures.

With this choice to apply MDE to automotive software safety, several questions arose. Among those, we chose to focus on the application of MBSA to the automotive software. In fact, in the safety domain the term MBSA includes all safety assessment practices that rely less or more on models. Therefore, if we want to explore MDE approaches for safety analysis, the starting point is MBSA. One question that can be asked is that “In the automotive software context, if we were to apply the MDE approach for safety analysis, what modeling approach for automotive software MBSA (dedicated or extended model)?”.

Specific research question 1

How can the current MBSA methods, tools & languages be applied to the automotive software safety analysis?

&

What modeling approach for software Model-Based Safety Analysis (dedicated or extended model approach)?

As we argued, an extended model ensures consistency between the system model and the derived safety model, without the need for additional mechanisms; in addition, the designer and the safety expert can use the same modeling environment and tool. However, if the ultimate goal (as far as coherence is concerned) is to have a single model able to integrate functional and dysfunctional elements and sufficiently structured to derive safety analyses, in practice the semantics of the design models are often poorly structured and not formal enough. Consequently, it can be argued that although such approaches model-based their semantics are not formal to the extent of enabling an MDE approach. Moreover, such an approach would call into question the independence between system design and safety assessment,

which is a key principle in certain fields such as aeronautics. A satisfactory alternative in this case is to use a dedicated model. One of the advantages of this approach is that it enables the independence between design and safety analysis activities, which can be an important asset in a certification context, for example. Among the dedicated approaches, AltaRica appears to be the most compatible and promising. The main appeal of AltaRica lies in its semantics, which are both formal and close to the systems it describes. This duality, difficult to obtain with other approaches, plays in favor of the AltaRica language.

However, if we choose to use a dedicated model approach, it is necessary to construct such model. Therefore, this implies another question: “How to model the dysfunctional architecture of the automotive software in question in order to enable the automation of safety analyses?” An aspect related to this question we discussed was the essential characteristic of the failure mechanism modeling within the elements of an MBSA model. In particular, we observed that to enable the automation of analysis, virtually all the described MBSA methods tend to rely on modeling the cause of failures at component level, and on the other hand their immediate consequences (propagation at component level). Hence, the secondary questions that can be asked following the first are: 1) “How to model the cause of failures (or dysfunctional behavior) at component level of components?” and 2) “How to model failure propagation (that can be structural or functional interaction)?”.

Refined specific research question 1

1) How to model the dysfunctional behavior of software components?

&

2) How to model fault propagation between software components (structural and functional interaction between components)

Secondly, another question that arose from the choice for a dedicated model approach related to mastering the complexity (in modeling) is how to best address this complexity in the context of MBSA or dysfunctional modeling. As explained in 3.4, a dedicated model requires that such models are purposedly build. However, when dealing with complex software architectures, constructing such model can be challenging and time costing

Specific research question 2

How to better master the growing complexity and make dysfunctional modeling easier?

Finally, there was also the question of the poor integration of safety analyses into the software development process prompting the question of how to better integrate safety analyses with the software development process so that they are carried out as smoothly as possible and gradually and at a lower cost. Indeed, software architectures are growing in complexity. This can make safety modeling challenging even in a model-based approach especially if a dedicated model approach is used.

Specific research question 3

How to ensure compliance of safety analyses with ISO 26262 recommendations and improve the integration of software safety analysis with the software development process?

The goal of the next chapter is to address these questions through a proposal consisting of several contributions. The next chapter will address these questions as well as the challenges related to our choice for a dedicated model approach.

Chapter 4. Proposal for a model-driven methodology for automotive software safety analysis

Abstract: *In the state of the art, we discussed the different MBSA methods, tools, and languages. One of the gaps we identified was the lack of methodological support for applying the MBSA approach. Another identified gap was the specificity of the current methods being systems oriented making their application to software-level safety analysis more challenging. In this chapter, we make a proposal for a MBSA methodology that adapts the concepts, principles and methods of MBSA for the purpose of improving the current practice of software safety analysis in the automotive software context. Two elements of our proposal are discussed. The first element consists of a methodology for applying MBSA In the automotive embedded software context. The second element consists in relying on the use of fault patterns that are constructed and easily reused for dysfunctional model construction to make safety modeling less challenging.*

4.1. Motivation

In critical systems, critical embedded software take part or assume various critical (safety related) functions. One specificity of such software lies in the safety of the systems they contribute to, that must be ensured and proven. Consequently, they are required by applicable standards to be developed with a certain level of rigor depending on their level of criticality. This brings the notion of the Safety Integrity Level (SIL) for any software that commands, controls, or monitors safety-critical functions. The rating provides a target to attain for each safety function.

In automotive, the ISO 26262 standard makes a specific recommendation to perform safety analyses not only at system level but also at software level. This differs from most of the other critical systems industry where safety analyses are conducted at system level, with software safety ensured by considerations that mostly focusing on correctness and with the means of ensuring this safety limited to verification and validation. The specificity of automotive embedded software is that in addition to common system level safety analyses, software-oriented safety analyses must be conducted at software level according to ISO 26262 recommendations.

The aim of this proposal is to provide a methodology for building and exploiting a dedicated software model for the need of safety analysis in the automotive context. As described earlier in the state of practice, one current trend in the industry to improve the quality of safety analyses is to rely on MBSA. Nevertheless, as we suggested in the state of the art, most of the proposed approaches are systems oriented and whether they can be applied to software in the automotive domain remains a question to answer. In fact, one tendency shared across the current system-level MBSA is to conduct safety analyses based on a probabilistic failure approach through failure ratios annotated in the dysfunctional model. Generally, the goal is to generate minimal cut sets based on probabilistic calculations. However, due to the deterministic aspect of software logic, it remains unclear how such probabilistic approaches can be applied to automotive software safety analysis. Another limitation is the specificity of the critical component's library developed library of critical components as well as their associated logics provided within some MBSA tools such as Cecilia OCAS. They are often dedicated to physical system components

(such as pumps, electrical motors, valves, or control units). However, in the automotive context, ISO 26262 specifically recommends conducting safety analyses not only at system level, but also at software level [12, p. 26262]. While these libraries can be used for modeling physical systems at system level in automotive, they are less suitable for modeling dedicated safety architectures of the embedded software. The problem description, research questions and the associated contributions that derive from this problem is outlined in Figure 15 below. Indeed, as it can be seen in the figure, one interrogation the current development state of MBSA method is how, if any, can the existing MBSA methods be applied to software safety analysis. This question can be refined further into more detailed considerations such as which approach (dedicated or extended model) is more suitable for automotive software safety analysis, or which modeling formalism (failure effect or failure logic modeling) is more suitable. Also associated with these questions is the lack of clear methodological support for the existing MBSA approaches in terms of model construction, as well as modeling formalism choice which hinders the adoption of MBSA.

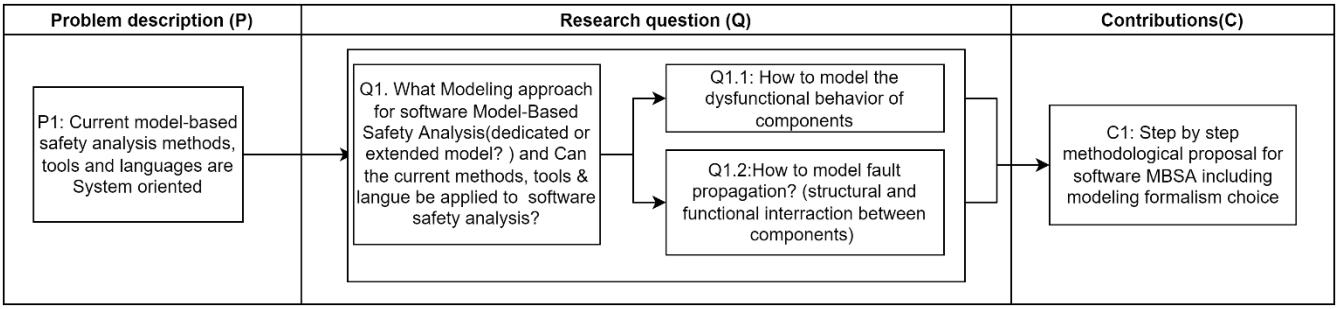

Figure 15. First contribution's relation to the problem description and research question

To apply MBSA to safety analysis at software level in the automotive context, our choice has been to use the dedicated model approach coupled with dedicated languages such as AltaRica as described in 3.3.2. However, as stated earlier, building a dysfunctional model can be challenging especially for complex systems. To address this concern, our solution has been to focus on selecting and including in the dysfunctional model only components that are safety related. Even then, the failure propagation logic of these components must be manually written by the modeler (which can be time consuming for large systems).

To ease the construction of dysfunctional models to conduct safety assessment, efforts have been done mostly using generic libraries of safety-related system elements. An example is the Safety Architecture Pattern (SAP) approach proposed by Kheren in [94]. In this approach, a library of SAP (components that highlights useful system's attributes from a safety point of view) are developed and coded using the AltaRica language. The generic library is then reused to easily prototype safety-oriented systems architectures that can be reused to perform safety analysis using tools such as the OCAS Workshop [53]. However, the developed libraries are mostly dedicated system components (such as pumps, electrical motors, valves, or control units). While these libraries can be used for modeling physical systems at system level in automobiles, they are less suitable for modeling dedicated safety architectures of the embedded software. Hence, the proposed approaches are mostly systems oriented. In this context, if the model-based approach is to be used for safety analysis at software architecture level as the commended by

ISO26262, safety analyses can be made easier if patterns or libraries of safety related components (such as safety mechanisms used in software) can be developed drawing from the same principles as those described in the case of systems SAP-oriented approach.

Another aspect that motivates the need for software fault patterns lies in the requirement to comply with the ISO 2626 in terms of the type of software failures that the standard recommends considering for safety analyses. In fact, for safety analysis at software architecture level, the ISO 26262 standard recommends using guide words to systematically examine the possible deviations from a specific design intent and to determine their possible consequences. In the software context, the guide words are associated to signal or data attributes and include words such as after/late (signal too late or out of sequence), less (signal value falls bellows the permitted range), more (signal value exceeds permitted range), other than (values of signal is inconsistent) etc. When using guide words, the safety-oriented analyses of the specific functions or properties for each design element are repeated with each guide word, until the predetermined guide words have all been examined. Although, it clearly appears that the use of guide words is a means to conduct such analyses systematically and to support the argument for completeness, the analysis process can be challenging when faced with complexity (to its repetitive aspect). Additionally, ISO 26262 provides software fault models for consideration when analyzing interference between software elements, for example with respect to 'timing and execution', memory, or 'exchange of information'. The fault models (described in textual format) consist of examples of faults that can cause interference between software elements (e.g., software elements of different software partitions). Moreover, the annex D provides examples of possible mechanisms that can be considered for the prevention, or detection and mitigation, of the listed faults such as memory protection, parity bits, error-correcting code (ECC), cyclic redundancy check (CRC), redundant storage, restricted access to memory, static analysis of memory accessing software and static allocation etc.

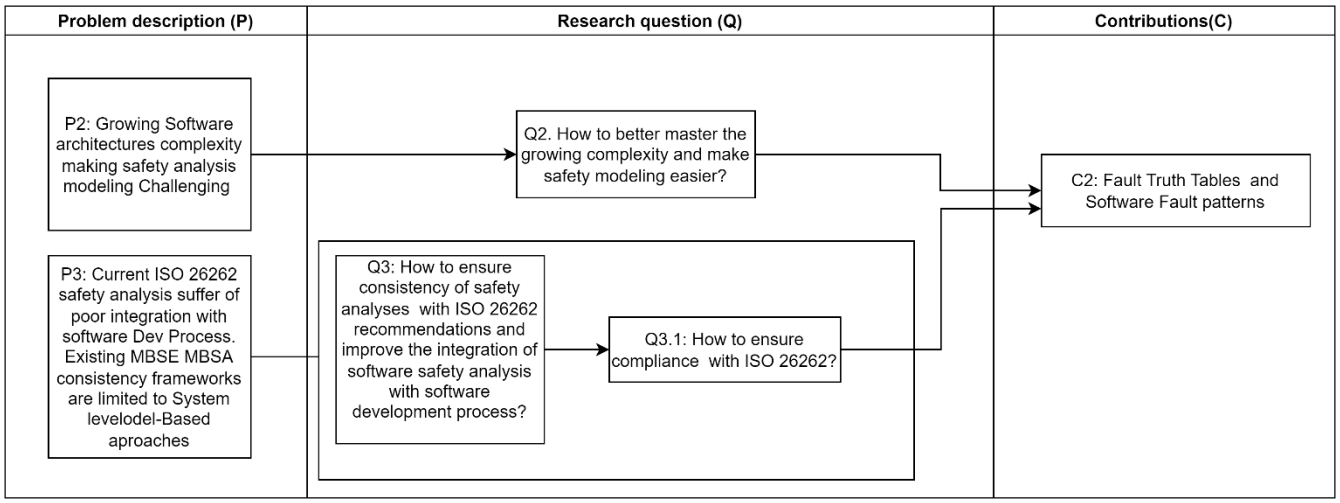

Figure 16. Second contribution's specific problem description and research questions

To address the described 2 issues, the aim of the second contribution is to develop and integrate the ISO 26262 recommended software fault categories and software safety mechanisms into specific dysfunctional software fault patterns to facilitate dysfunctional modeling and to ensure that the failure

modes described by ISO 26262 are fully accounted for when building the software dysfunctional architecture. The problem description and research questions associated with this contribution contributions out outlined in Figure 16.

4.2. Three-steps methodological proposal including the use of fault patterns

To recall, the need that motivated this proposal was the necessity to adopt a model-based approach for automotive software safety-oriented analyses. However, based on the identified methodological gaps in the current application of MBSA, we identified that engineers need to be provided with clear methodologies which can improve the current traditional techniques be beneficial to the adoption of MBSA.

Therefore, based on the review of current MBSA approaches, our proposal is to rely on a dedicated dysfunctional model of the software architecture and using it as the basis for conducting safety analyses. Unlike a functional model of the software that conveys the nominal behavior of the software, the dysfunctional software model focus on the failure behavior (in presence of faults). Although such model can be derived in certain circumstances from the functional models (extended model approach), our choice has been to use a dedicated dysfunctional model (dedicated model approach). Indeed, such model must be built, and this will be the object of the second contribution.

The proposal proceeds in three steps: 1) software safety analysis context definition, 2) software dysfunctional modeling relying on the use of fault patterns and 3) software safety analysis. An overview of the methodology is given in Figure 17. The first step (software safety analysis context definition) is a preliminary step; it focuses on the selection of input data (consisting in identification of the safety related components to be included in the dysfunctional model), the identification of the functional mode (necessary to the understanding of the nominal behaviors) and the identification of the associated safety mechanism proposed for the prevention, or detection and mitigation of faults. This step is necessary as it will enable the construction of a more synthetized and pragmatic dysfunctional model of the software. Indeed, we hypothesize that limiting the MBSA model to safety related components is sufficient to carry out meaningful safety analysis and can improve both efficiency and the quality of safety analysis. The second step (software dysfunctional architecture modeling) therefore consists in the construction of the dysfunctional model of the software based on the elements defined in the first step. Lastly, the third step (Software model-based safety analysis) consists in exploiting the dysfunctional model using commonly used safety techniques to conduct analyses such as minimal cuts, fault trees or FMEAs that can be automatically generated from the dysfunctional model. More detailed descriptions of the steps will be provided in the next subsections.

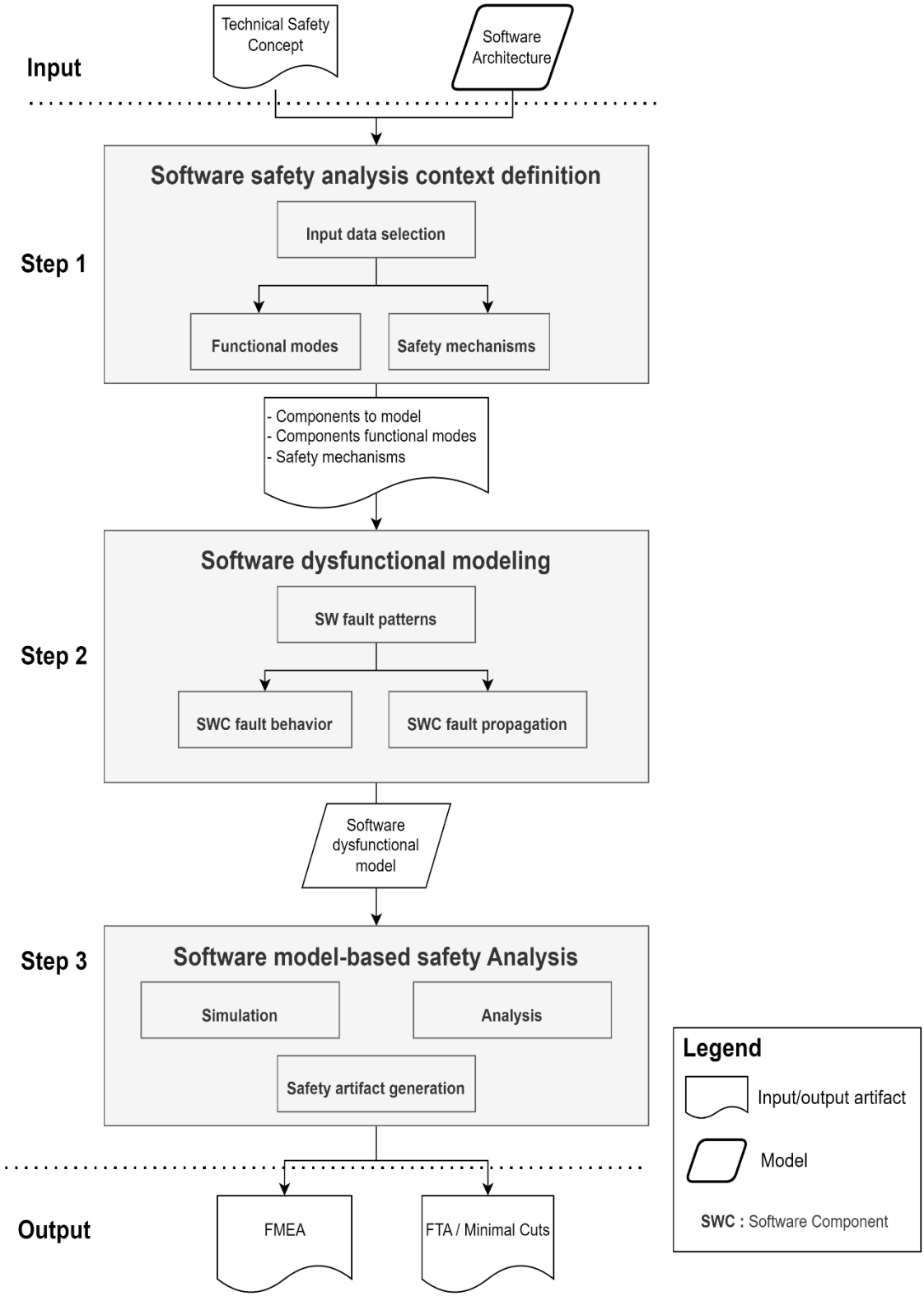

Figure 17. Steps of the methodological proposal

Our proposal is based on the use of a dedicated model and Failure Logic Modeling (FLM) techniques by means of the applicable languages and semantics. As explained in the state of the art, in FLM, the dependency between the components of the model is defined in terms of deviations of their behavior from design intent and how they are affected by the failures of other components of the system. In our method we chose to use the dedicated model approach. As stated in the state of the art, the dysfunctional model can either be an extension of the design model or a dedicated model. While the model extension enables consistency, by construction, of the safety analyses and the design model of the system, the dedicated model is more pragmatic to implement, as it ensures independency and separation of concern (between safety assessment and system design). Weighing on the arguments for and against the dedicated or extended model approach, we have chosen in our proposal to adopt the dedicated approach. In particular, we argue that in context of critical systems development, the independence between system design and safety analysis is indeed an important principle that should be maintained. Furthermore, we argue that safety models that are systematically derived from design models are often cumbersome to the point of not necessarily yielding meaningful safety analysis results. Another argument against design model extension for safety analysis is the lack of design models that are well structured enough to allow such extension in the first place. In fact, the automatic deriving of safety models from design models requires having design models that are well structured, executable, and simulate-able at the proper level of detail and stage of the development process.

In our proposal, we have chosen to explore the AltaRica language among other possible language choices. Indeed, modeling languages such as AltaRica, AADL and EAST-ADL can be useful at different levels and contexts. However, despite being initially designed for system, we favor AltaRica for various reasons including its simplicity, well-structured and rich semantics. The main appeal of AltaRica lies in its semantics, which are both formal and close to the systems it describes. This duality is difficult to obtain with other approaches. Furthermore, AltaRica has proven its effectiveness usefulness in several large-scale European systems safety projects as well as in many critical systems certification contexts as we mentioned earlier in the state of the art. Furthermore, AltaRica is compatible with the dedicated model approach (the approach that we chose to use) and FLM (thanks to the definition abstract failure flows). Although the other prominent MBSA approaches such as FPTN, AADL, HiP-HOPS ort EAST-ADL have interesting features they either are more relevant to the model extension approach or lack up to date effective tooling support or availability for public use. AADL and EAST-ADL for instance are more compatible with model extension (through error model annex for AADL and packages for EAST-ADL).

Our choice to use a dedicated model approach implies that such model must be purposely constructed by the safety engineer, which introduces a new issue: the challenge of the model construction due to the complexity of the software architecture. Therefore, goal of the second contribution which is part of the second step of the methodological proposal (software dysfunctional architecture modeling) is to construct a library of reusable software fault patterns and safety mechanisms that are commonly found in safety related software components, drawing from the Safety Architecture Pattern(SAP) oriented approach [94]. To address the challenge that represents dysfunctional modeling associated with complexity, our hypothesis is that making the MBSA model construction easier can both benefit its adoption by companies as well as improve the quality of safety analysis. Therefore, the position of this contribution is to make

the construction of MBSA models less painful for safety analysist by proposing a set of predefined reusable libraries of software fault models to ease the dysfunctional model building process. As indicated in Figure 18, our software fault patterns are constructed based on software fault templates defined by ISO 26262. Hence, the contribution helps in improving compliance with ISO 26262 in terms of recommended failure to consider for software-oriented safety analysis. The contribution provides two types of fault patterns: 1) generic patterns for software components depending on the categories of fault they are subject to and 2) generic patterns for software safety mechanisms commonly used in automotive software architecture to prevent or mitigate failures.

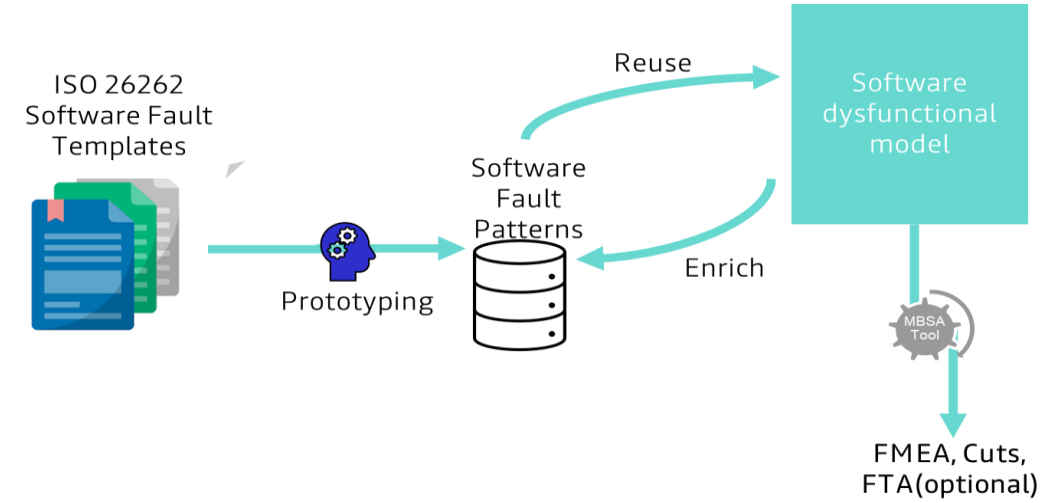

Figure 18. Software fault patterns prototyping to improve reuse and facilitate modeling

4.2.1. Step 1: Software safety analysis context definition

The main objective of this step is to 1) select which components to model in the dysfunctional model depending on the Unwanted Software Events (USE) identified in the Technical Safety Context (TSC) and 2) select the information on which the dysfunctional modeling will be based. The step is motivated by two main issues. First with the rising complexity in automotive software architecture, the manual modeling of the dysfunctional software architectures can become challenging and more error prone. To address this issue, our hypothesis is to limit dysfunctional modeling to only software components that assume safety functionalities. We believe that only modeling components associated with safety functions will help address the complexity issue (in modeling) while allowing us to build a small but pragmatic dysfunctional model that is both less complex and sufficient for safety analysis needs. Secondly, in a development context where MBSE is not well mastered, well-structured functional design models that would normally support the dysfunctional model building are either of poor quality (informal semantics, lack of structure) or even nonexistent. This adds to the challenge associated with the use of a dedicated model approach which is the need to ensure that the dysfunctional model is built consistently with the associated design model (which are software architecture models in our case). In the case of well-developed MBSE approaches, well-structured executable design models are available and can serve as a basis for automatically deriving or manually constructing the dysfunctional model. However, this is not always the case, as in most cases where the MBSE adoption is still in its early stages and where a mix of informal

models and document-centric artifacts are used together throughout the development cycle as described in the review of practices. In this context, selecting which information to use for building the dysfunctional model becomes necessary.

The main inputs required for this step consist of the Technical Safety Concept (TSC) resulting from safety assessment performed at system level and the software architecture model and/or the software architecture document. The TSC is an aggregation of safety requirement specifications (often in textual and tabular format) from the system, as well as their allocations to hardware and software components and associated information (text, diagrams, or sketches), which justify that safety measures and mechanisms are in place. In the TCS we can find the description of system-level safety goals to consider for analysis, their ASIL ratings as well as their declination to software components in the form of Unwanted Software Events (USE) which are events occurring at the software level that violate the system-wide safety goals. Additionally, the TSC also contains the description of safety mechanisms that are proposed to ensure that the safety goals are not violated. Safety mechanisms are technical solutions (software functions in our case) implemented in the architecture, whose objective is to detect, mitigate, and tolerate faults through isolation or reconfiguration in order to maintain critical functionality or to bring the system to a safe state. Information contained in the TSC include data such as ASIL ratings specifically describing to which components such safety is allocated, allowing us to identify which software components are safety related. After identifying the safety-related software components and their proposed safety mechanisms, one can seek to know how they are intended to be implemented in the software architecture in order to model their exact behavior. This is done by consulting the software architecture and/or the software architecture document (if useful models are not available). In the software architecture model/document, we find the details of the implementation proposed to meet the safety requirements contained in the safety concept such as the functional behavior of the specified safety related components as well as detailed technical descriptions of the safety mechanisms. The approach followed in this first step is illustrated in Figure 19.

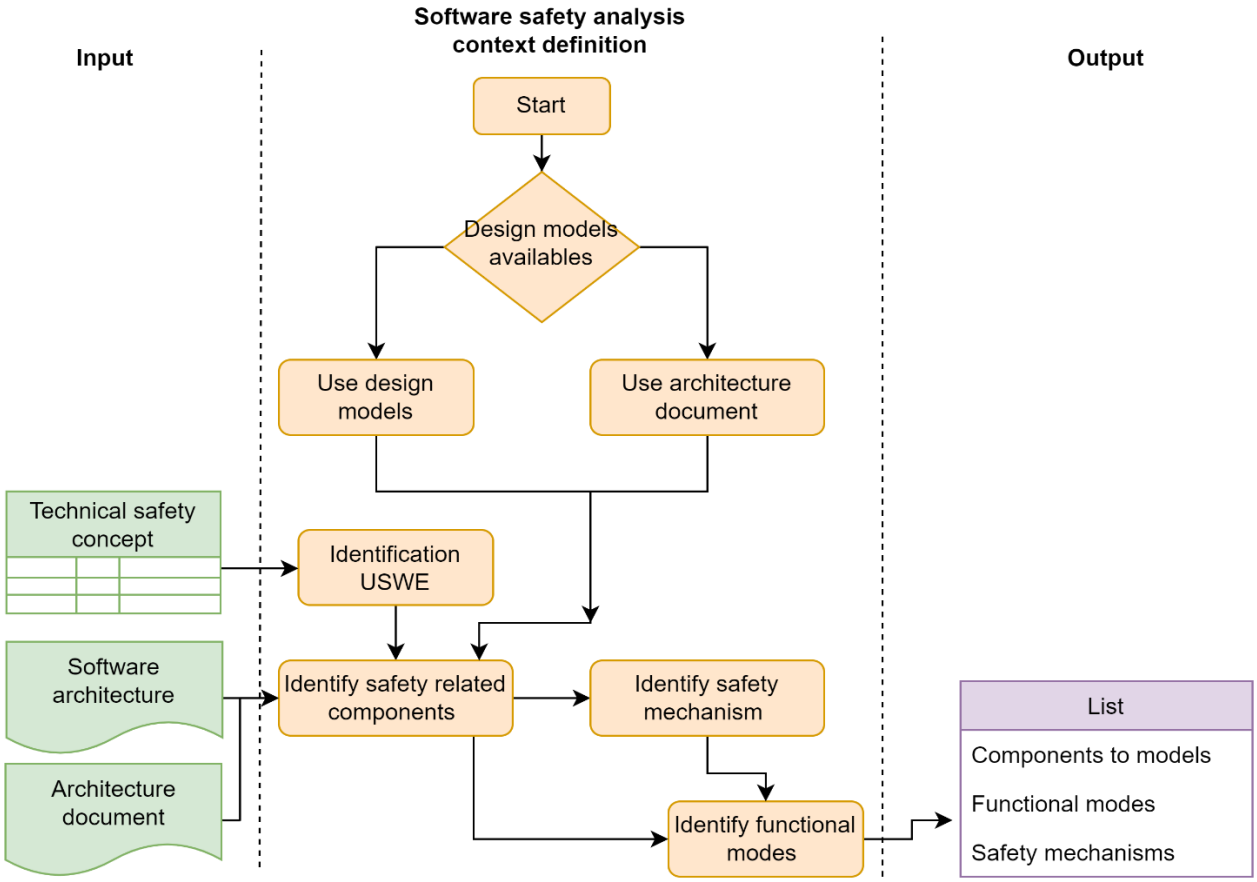

Figure 19. Step 1: Software safety analysis context definition

As it can be seen in the flowchart, this step aims to have a certain flexibility in the selection of data input for the construction of our dysfunctional model by prioritizing the use of well-structured software models if they exist while being still open to architecture documents if such models are not available. This is necessary since well-structured models of the software are not always available at the right stage of the development cycle as we pointed out earlier in review of practices in section 2.4. While this step can appear to be trivial, its advantage is that we aim to not limit ourselves to a fixed single approach as far as the input data for the construction of the dysfunctional model is concerned. Instead, we aim to adapt the choice of input data depending on their availability. Through the decision three shown in Figure 19, we encourage prioritizing the use of formal models of the software if they exist and are sufficiently structured to enable their unambiguous interpretation. This is important because the use of formal models is not always the preferred choice for safety engineers that are used to getting their inputs from architecture documents or through informal discussions with the system or software designer. If necessary, the understanding can be complemented (if the elements of the design document do not provide this information) by consulting the designer (software architect) or by exploiting more detailed design models (that are however one step later in the development cycle). For example, Simulink functional models can be used to understand the functional logic and to determine the manner in which safety mechanisms are implemented.

The input data selection specifically addresses the difficulty of obtaining good inputs such as software architecture models necessary for a more pragmatic construction of the software dysfunctional model. The functional mode identification aims to help in the understanding of the expected basic behavior of the software components in the absence of faults. The safety mechanism identification sub step aims to identify the logic behind the safety mechanism that are proposed in the architecture design to fulfill the safety goals specified in the safety concept. By introducing this preliminary step, the second step will be simplified thanks to the collected data.

4.2.2. Step 2: Software dysfunctional architecture modeling

As pointed out earlier in the state of the art, one aspect that all the prominent MBSA methodologies such as HIP-HOPs or AltaRica share in common is that there are 2 considerations in the modeling of an MBSA model. First, they all tend to model the isolated failure behavior of components using precise semantics that will translate into basic failure modes or failure events in the safety analysis. Secondly, they all model the interaction between these components to capture the failure propagation behavior between the components. Thus, to successfully model a dysfunctional architecture that can be automatically derived into safety analysis, one must model both the dysfunctional behavior of single components as well as their interaction among each other in the form of failure propagation. Drawing from this understanding, we propose to divide the modeling of the software dysfunctional architecture into two parts: the dysfunctional behavior of single components and the failure propagation logic between the components.

Thus, the software dysfunctional modeling consists in modeling the dysfunctional behavior (in the presence of failures) of software components and their interactions. As illustrated in Figure 20, it takes place in 3 sub steps: 1) software fault pattern modeling, 2) behavior modeling of the components using state machines, and 3) failure propagation modeling. We will describe these in the next subsections respectively.

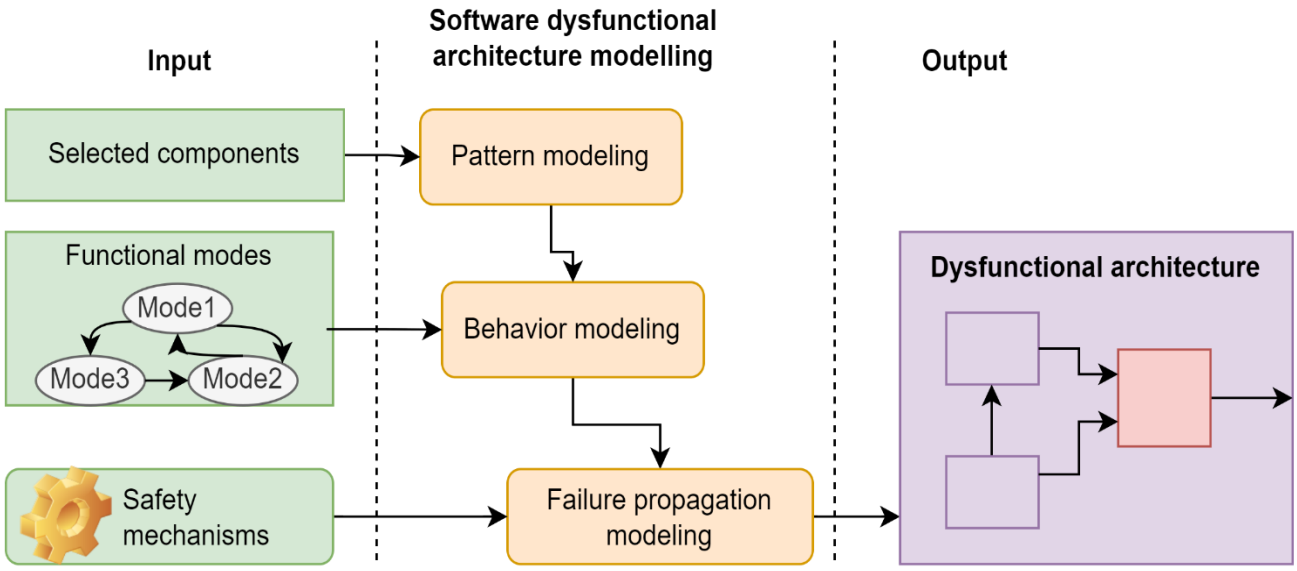

Figure 20. Step 2: Software Dysfunctional Modeling

Software fault pattern modeling

We start by describing the software components' behavior through a generic abstract template using states and transitions. An example of such a template consisting of a software component with generic failure modes is provided in Figure 21. It shows 4 states (Inactive, Nominal, Erroneous and Failed), representing the execution status of the software component linked by several possible transitions (Minor fault, Major fault, Recovery etc.). The inactive state is the idle or initial state preceding the initialization where the software component is not solicited or does not provide its function. From this state, the 3 outgoing transitions labeled "Activation", "Major fault" and "Minor fault" will lead the component to the "Nominal", "Failed", or "Erroneous" states respectively. In the nominal state, the software component executes and delivers its intended function. From this state, the component can revert to the Inactive state as indicated by the transition label "Deactivation" or move to the "Failed" or "Erroneous" states if a major or minor fault occurs as indicated by the transitions. In the erroneous state, the software component provides erroneous results while, in the "Failed" state, it fails to provide its intended function. From the generic pattern, more specific patterns related to specific fault categories as described by ISO 26262 can be easily derived. As an illustrative example, let us consider a piece of software that reads some critical data from memory, performs some critical calculations, and stores the result back to memory. Based on its function, the failure modes of this software component could include memory access related faults (such as read and write errors) as well control flow and timing related failure modes (such as execution failure, or untimely execution). Depending on the exact safety requirement, safety mechanisms to prevent the failure of this software component's function could include a protection against unwanted writing and a watchdog timer. Based on this information, the elements that will be necessary to model the safety related behavior of this software component, are the failure mode related to the memory access and the two safety mechanisms that will be modeled through states and transitions within this software component.

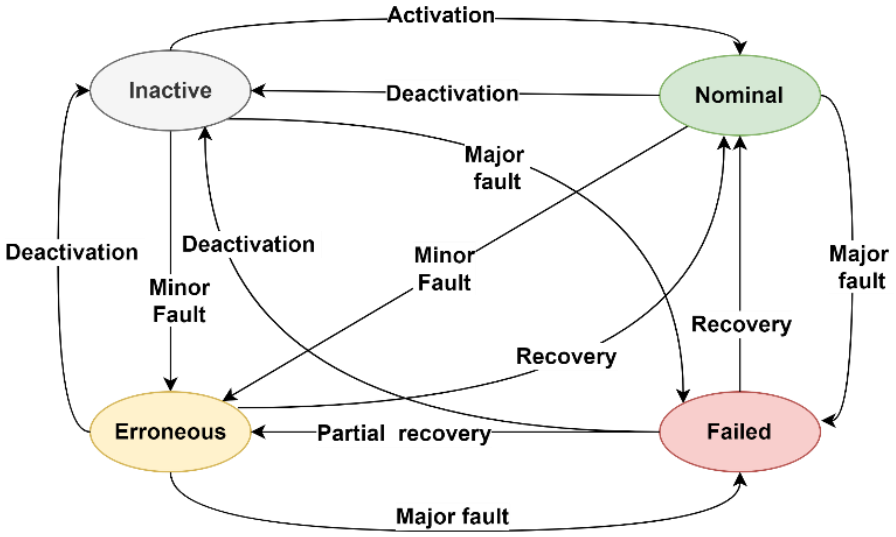

Figure 21. Failure modes of a generic software component

The pattern presented in Figure 22 shows the execution of a software component functioning based on received data can subjected to the 'data exchange category' fault. Like the pattern shown in Figure 21, it

has 4 states: “Init,” “Nominal”, “Erroneous”, and “Failed”. From the initial state (indicated by the blue state on Figure 22), the function will either move to the “Nominal” or “Erroneous” states as indicated by the outgoing transitions depending on a successful or unsuccessful data transmission initialization. In the nominal state, the software component executes normally and fulfills its function. If a data transmission initialization fault has led the function to the “Erroneous” state, an execution of a safety mechanism such as CRC can bring the function to nominal if successful or to the “Failed” state (red state on Figure 22) if unsuccessful. The function can also move from the nominal state to the “Erroneous” state with the occurrence of inconsistencies of the transmitted data (such as corruption, incorrect data value, out of range data values or incorrect sequence of data). In the “Failed” state, the software component fails to provide the expected function due to missing or loss of transmitted data or due to the safety mechanism failure to recover from the “Erroneous” state. As it can be seen Figure 22, this pattern is built on the generic pattern presented in Figure 21. However, it differs by the specificity of its failure modes expressed in the transitions that are specific to the “Data Exchange” fault category. Based on this pattern, another related pattern was derived to cover the specificity of other data exchange related faults such as delayed data transmission that will cause the function’s execution to be delayed. In such case, the “Erroneous” state was further split into several states depending on the specificity of the software component.

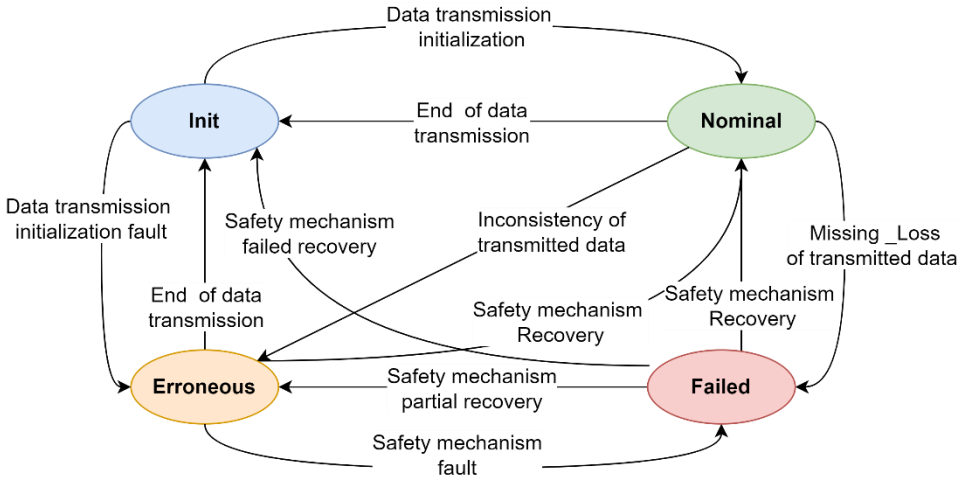

Figure 22. Data exchange fault pattern

The next pattern shown in Figure 23 aims to capture the behavior of a generic software safety mechanism. The 4 states (Nominal Inactive, Nominal Active, Misleading and Failed) represent the execution state of the generic safety mechanism. In the “Nominal inactive” state, the safety mechanism is in its nominal execution state with no fault detected. When a fault is detected, it moves to the “Nominal active” state. In this state, the safety mechanism is successful in reacting and correcting the effect of the fault. From these states, an erroneous or failed reaction will lead the safety mechanism to “Misleading” or “Failed” states respectively.

The behavior of the software fault patterns is completed by writing failure propagation logic knowing the functions of the software component and considering the associated fault modes identified earlier. To do so, we first need to study the software components and safety mechanism’s function in order to identify their basic behavior, what their inputs are, and the result they produce in normal or faulty execution. This allows writing the failure propagation logic of the identified software component and safety mechanisms.

Depending on the expression of the safety requirements that the identified mechanisms are to fulfill, logical operators such as “or, and” and program control structures such as “if-then-else” can be useful to express the function or the combination of several mechanisms within a software component. While

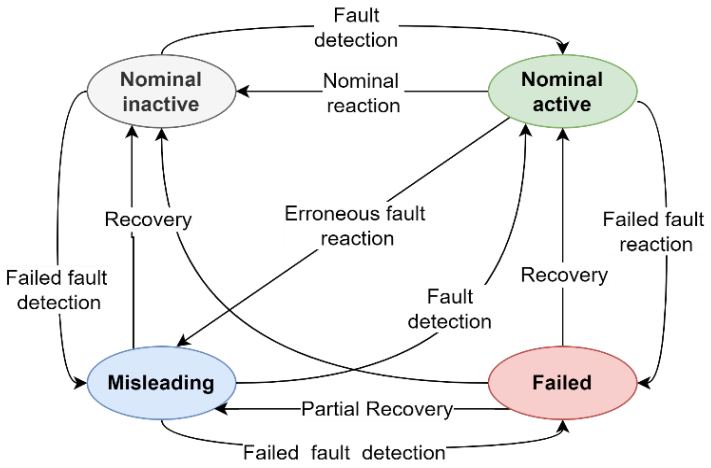

Figure 23.Generic safety mechanism fault pattern

these operators are already part of the AltaRica semantics and can be expressed in assertions, modeling them in virtual bricks allows to graphically associate the components without rewriting the propagation logic. Therefore, we also need to write and add to the library the failure propagation logic through these operators and control structures. This will result in a library of software fault patterns, safety mechanisms, operators and control structures that will be reused to construct the dysfunctional model of the software.

To complete the patterns behavior, we need to express the dependencies between the component inputs and outputs, that is how failures can propagate through the software architecture. To this end we use Failure Truth Tables (FTT) that we introduced in our early work [95]. FTTs are dysfunctional failure propagation tables consisting of discrete input and output variables whose possible values are defined depending on the failure behavior of the components function. FTTs can be used to capture the dysfunctional logic of a function based on its inputs. FTTs allow us to systematically map the normal flows (from the functional logic) into failure flows (dysfunctional logic). To build the failure truth tables, we first analyze the functional logic of a component. Then using the states (nominal, erroneous, loss), defined in the first step, we set the inputs (nominal, erroneous or loss) and deduce the corresponding output (nominal, erroneous, loss). Repeating this process allows building the FTT with all the possible combinations of the inputs and the corresponding outputs in dysfunctional flows.

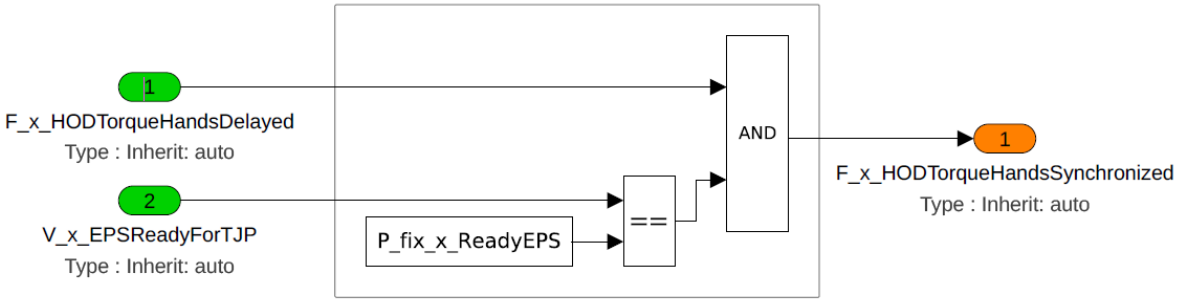

Figure 24. Simple functional logic of a software component

To illustrate the construction of an FTT, Figure 24 presents a simple example of functional logic of a software component (in the rectangle). It is a simple function that returns an output based on the value of two inputs labeled 1 and 2. Using our proposal, we set the internal states to the inputs and report to the table the corresponding output states. We reiterate this for all the possible combinations of the input's states.

Table 1. Simple Failure Truth Table

Input 2 / Input 1	Erroneous	Loss	Nominal
Erroneous	Erroneous	Loss	Erroneous
Loss	Loss	Loss	Loss
Nominal	Erroneous	Loss	Nominal

Output

This results in the FTT shown on Table 1. In the case of this example as it can be reflected by the values of the output in the red frame, if both inputs are 'nominal', then the output is 'nominal'. If either one of the inputs is 'loss' then, the output state is 'loss'. Otherwise, the output state is 'erroneous'. In order to obtain a computable expression of this logic, we then translate the information of the FTT into a dysfunctional logical expression of outputs:

```
if (input_1 = Nominal AND input_2 = nominal)
then
        {Output := Nominal ;}
else if (input_1 = Loss OR input_2 = loss) then
        {Output := loss;}
```

The advantage of this FTT is that it allows changing the point of view (from functional to dysfunctional) and expressing the failure behavior in a syntactic and formal manner that can be used to write logical expressions of the output (for software engineers) as well as in natural language (to communicate). Furthermore, when the design logic evolves, the FTT can be updated accordingly, helping in this way to remain consistency. They could also be reused in the long run to capture more complex logics.

Software component's dysfunctional behavior modeling

The software dysfunctional behavior modeling consists in modeling the internal dysfunctional behavior of the architecture's software components based on the modeled software fault patterns using the data collected in step 1 (operating modes, safety mechanisms), focusing on the components and signals related to safety. In our approach we chose to model the internal behavior of the components using state machines (states and transitions). The states in the state machines are deduced from the operating modes that we identified in step 1. Through the transitions, we express the logical conditions controlling the transition from one state to another. These conditions can be linked to deterministic (e.g., a development error) or stochastic (random events associated with a probability law) events, but also be dependent on the input state. This modeling results in software components that are reduced to state machines. To facilitate modeling, reduce modeling time, the patterns described in the previous step elements, if stored in a library, can be easily reused to construct the dysfunctional model. This step requires having a tool that offers library support such as OCAS or SimfiaNeo. Using the elements stored in the library in an appropriate MBSA tool, one can easily model the dysfunctional architecture of a system through reuse. This is possible since the safety mechanisms self-contain their propagation logics as well as the associated fault modes. Therefore, there is no need to write the failure propagation logic code in this step.

Software component's fault propagation logic modeling

Software fault propagation modeling complements the component behavior modeling with failure propagation logics. It involves modeling the dysfunctional interactions between components (i.e., how failures propagate from one component to another). As stated in the state of the art, the two main modeling paradigms for representing failure interaction between components of a dysfunctional architecture include the "failure effect modeling "(interaction through real flow of data) and the "failure logic modeling" (interaction through abstract flow of data). In our approach we chose to use the failure logic modeling approach. Hence, to write the failure dependencies, abstract failure flows are used instead of the usual nominal flows found in dataflow architectures. Our choice to use the failure logic modeling is first motivated by the fact that from a safety standpoint, failure propagation through software the architecture is not always the result of direct interaction but rather a cascading effect resulting from the unfulfillment by some software components of their intended functions. Moreover, it is also our argument that using real flow of data results in higher complexity in the dysfunctional architecture. Formalisms that support the failure logic modeling approach include languages such as AltaRica described earlier in the state of the art.

To proceed, we first model the input and output signals of each component by assigning them discrete states representing the classes of values they can take. For example, from a dysfunctional point of view, a given input data can be nominal (correct), erroneous (outside of the expected range), late, or absent

(see annex E of part 6 of ISO 26262, which lists possible errors related to the execution and the exchange of information between software components). In a second step, we write logics that enable us to deduce the output states in accordance with the internal state of the component and its inputs. For example, let us consider a simple ADD function that adds two variables (*input1* and *input2*). For this function, we define a state variable that can take three values: nominal, erroneous or failed. In the *nominal_state*, the component is operational and produces good outputs; in the *erroneous_state*, the component is operational but can potentially produce bad results; in the *failed_state*, the component is not operational anymore. For the two inputs and the output variables (*input1, input2, output*), we define a flow variable that can also take three values: *nominal_flow* (good data), *erroneous_flow* (potentially bad data), or *lost*_flow(no data).

We can then write the expression of the output as a function of the inputs and the state of the function as outlined below:

```
if (state = nominal_state) then
        if ((input1 = nominal_flow) and (input2 = nominal_flow)) then
                output: = nominal_flow
        else if ((input1 = lost_flow) or (input = lost_flow)) then
                output: = lost_flow
        else
                output: = erroneous_flow
        end if
else if (state = erroneous_state) then
        output: = erroneous_flow
else //state== failed_state
        output: = lost_flow
end if
```

By repeating this approach for all components, we end up with a well-defined dysfunctional architecture and within formal semantics; this constitutes a necessary basis to support safety analyses. Again, this step is made easier thanks to the failure propagation logic already defined in FTT's or implemented in the software fault patterns.

4.3. Step 3: Software model-based safety analyses

The objective of step 3 is to conduct safety analyses or extract classic safety models of interest from the dysfunctional model we built. To do so, it is first necessary to add Unwanted Software Events (USWEs) to the dysfunctional model. To this end, it is necessary to mathematically express the USWEs through predicates or Boolean combinations and to associate them to the relevant components. The resulting model can then be used to produce Failure Modes and Effect Analysis (FMEAs) and fault trees for analysis by means of model checking. Moreover, the addition of USWEs to the architecture allows to associate them or not to certain components by following the modeled propagation logic. Several USWEs can be added and evaluated on the basis of the same dysfunctional architecture.

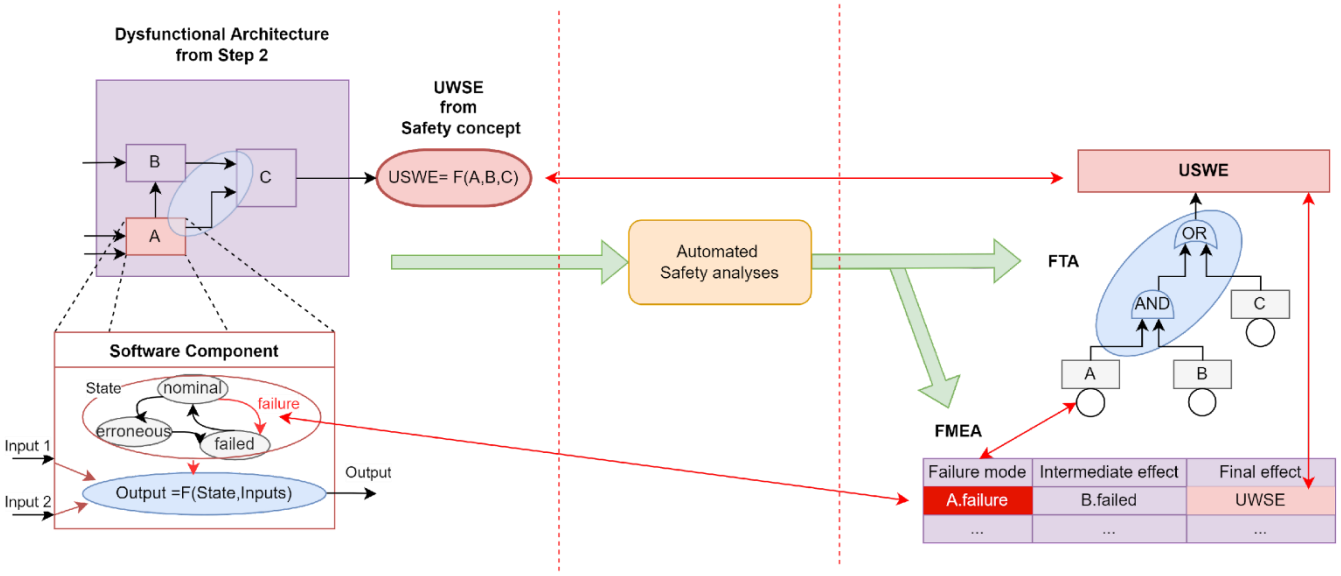

Figure 25. FMEA and FTA deduction from the dysfunctional architecture

Figure 25 illustrates the automatic generation of FMEAs and fault trees and correspondences between the elements of the dysfunctional model and those of the FMEAs and fault trees. On the left, is a synopsis of a simple dysfunctional model consisting of three software components (A, B and C), and a detailed view of one software component as an example. At the output of the dysfunctional model is a predicate expressing one USWE (as defined in the safety concept). Depending on the description in the safety concept, this predicate can be as simple as output_of _C = failed or more complex combinations such as output_of _C = erroneous combined with other conditions (such as failed state of A or B, or erroneous values of the input flows). The red double arrows show the equivalency between the dysfunctional architecture and the generated FTA and FMEA. On the basis of the dysfunctional architecture, we can generate fault trees having as a top event one of the UWSEs modeled, and whose branches derive (according to the propagation logic) from the transitions to the failed states, as modeled individually by the state machines at the component level. This same logic can be applied to FMEAs, where transitions to failed states will become failure modes while USWEs will become final effects

4.4. Discussion

In the traditional approach, the safety analyst spends much of their effort interpreting various design documents to manually construct classical model's safety models such as FTAs or FMEAs. Using our methodological approach, it is possible to unify these different classic safety models into a single dysfunctional model. Furthermore, the approach offers a huge reusability potential. Numerous USWE's can be analyzed on the basis of the unique dysfunctional model whereas in the traditional classical method, If the system design evolves, the safety analyst must, for instance for each UWSE, 1) manually construct a fault tree; and 2) manually update the system's FMEA. In such a case, the analyst would need to manually construct as many FTAs as there are feared events. If the design evolves, they will need to individually update all the fault trees and the FMEAs. To evaluate new USWEs through our approach, it will suffice to decompose them into logical combinations and to associate them to the dysfunctional

architecture already modeled. The associated new fault trees will be automatically generated and the systems FMEA automatically updated. Thus, it can be argued that this methodological proposal thus improves current practices. Following the described approach, the task of the safety expert can be shifted from the manual construction of FMEAs and fault trees to the construction of a dysfunctional architecture. By focusing its efforts on dysfunctional modeling, the method brings the safety models as close as possible to the design models, thereby improving the quality of the analyses. In the traditional approach, errors and discrepancies in safety analysis are often the result of an erroneous conceptual model of the system (how the analyst perceives the system model in its mind). Through our approach, such errors are minimized since the dysfunctional model is an accurate implementation of what the technical safety concept aims to achieve given that we selected the elements to model based on the element of this safety concept. In other words, what is analyzed is what is expected to be implemented from a safety function standpoint. In addition, representing the behavior of the system without ambiguity is possible through formal semantics such as AltaRica, turning it into a possible candidate in a certification context.

Through the second contribution we show that using fault patterns and the adequate tool, the dysfunctional model construction can be made easier. In Particular, we argue that the specificity of the software-oriented fault patterns that are based on the ISO 26262 software fault consideration can benefit the application of MBSA in the automotive software safety context enabling us to fulfill adherence to the standard. Associated with modeling tools such as SimfiaNeo, the use of pattern relieves the safety expert of manually modeling all the components of the software architecture. Instead, it allows them to concentrate their energies on designing the fault patterns that can be reused. Once the fault patterns are built, they can be reused to build the dysfunctional model and make it possible to conduct analyses with different parameters for many USWEs based on the same model. The use of patterns can be also seen as a way for the safety expert to put their knowledge and experience in models that can be reused by non-expert to build dysfunctional architectures more easily. Hence, the benefits can be reflected in terms of knowledge sharing, reusability of the models and time saving based on reuse. All these elements make this safety analysis method an interesting alternative that has the potential not only to improve current practices but to contribute to the adoption of the model-based approach by making dysfunctional modeling easier for non-experts.

Admittedly, it can be argued that this methodological proposal requires a certain modeling effort due to our preference for a dedicated model approach. However, thanks to the appropriate selection of inputs in the first step of the methodology, and the use of the software patterns, the modeling effort is minimized. Furthermore, the dedicated model approach appears to be the appropriate avenue given the current context of our MDE adoption. In fact, at a company level, our approach fits in a global effort to adopt MDE along with other ongoing MDE related efforts that are not yet mature. While the chosen dedicated model approach fits the current context, avenues for a better integration of all these efforts into a more seamless MDE approach must be envisioned. In addition, we acknowledge that our choice for a dedicated model approach brings additional challenges. Indeed, one of the difficulties brought about by a dedicated dysfunctional model is maintaining its consistency with the design model when the latter evolves. Implementing additional measures is therefore necessary to guarantee consistency. One proposal, that will be discussed in the perspectives aims to address this issue.

4.5. Conclusion

This chapter made a methodological proposal based on MDE to improve the state of current practices of software safety analyses in the automotive industry. It defined a step-by-step methodology for defining the safety analysis context, constructing the software dysfunctional architecture, and using it for safety analyses relying on a dedicated model approach. The proposal discussed in this chapter aims to demonstrates that it is possible to apply an MBSA approach to evaluate software safety, especially in automotive applications. It also highlights the benefits of basing safety analyses on a dysfunctional model reflected in terms time saving, analysis quality, and reusability. The chapter also made a proposal based on fault patterns that can be used to easily build the software dysfunctional model through prototyping and reuse, and from which it is possible to derive classic safety models.

We argued that using the proposed methodology, it is therefore to unify different classic safety models into a single dysfunctional model which offers improved safety analysis quality, time saving a potential for reuse. We explained that following the described approach, the task of the safety expert can be shifted from the manual construction of FMEAs and fault trees to the construction of a dysfunctional architecture. By focusing its efforts on dysfunctional modeling, the method brings the safety models as close as possible to the design models, thereby improving the quality of the analyses.

However, to be validated, the proposal needs to be applied to a case study. Therefore, the next case study section, will show how, safety analyses can benefit from this alternative new model construction method using adequate modeling languages and tools. Since the proposals described in this chapter are based on a dedicated dysfunctional model, it will also be essential to supplement the method with additional mechanisms that ensure consistency between the system design and safety models. This mitigation measure of the object of our future work.

Chapter 5. Longitudinal control case study

Abstract: *In the previous chapter, we made a proposal, consisting in a methodology for applying the model-based approach and in the use of fault patterns to improve the current state of practice of software safety analysis in the automotive context. In the current chapter, this proposal is integrated and tooled with a modeling language to illustrate its application on a concrete case study chosen at Renault. To this goal, the chapter explains how to apply the proposed methodology step by step and evaluates the obtained results through a discussion where its advantages and disadvantages are weighed as well as its adequation with the tooling choice. Based on the observe limitations, the chapter also outlines avenues for future improvement through a perspective of a tooling proposal aimed at improving inter model consistency.*

5.1. Introduction

In Chapter 4, we made a proposal, consisting in a methodology for applying the model-based approach to improve the current state of practice of software safety analysis in the automotive context. It proceeded in three steps: 1) software safety analysis context definition, 2) software dysfunctional modeling relying on the use of fault patterns and 3) software safety analysis. In this chapter, we present a case study of an automotive driver assistance system to show how to apply the methodology in association with the use of software fault patterns. Initially, the case study was intended to include third application of a third contribution consisting of a tooling proposal (aimed at improving consistency between system design and safety models) in addition to the two contributions that were described in Chapter 4. However, we were unable to include this last contribution to the case study due to a delay in the reception of a software license. Furthermore, although it has solid theoretical foundations and preliminary results, this third contribution (consisting of tooling proposal to improve consistency) is still ongoing development. Therefore, it will only be discussed a perspective in this chapter based on the limits identified in the results.

The case study assumes that the Item definition, HARA, and system level safety analyses have been priorly conducted as part of the scope 1 and 3 of ISO 262622 and are not discussed (considered out of the scope of the present study).

5.2. System Presentation

The system for this case study is the Longitudinal Control (LC) system. It is a function of the ADAS (Advanced Driver Assistance Systems) technology whose purpose is to assist the driver by ensuring speed and braking control in autonomous or assisted driving modes. In concrete terms, the ADAS longitudinal control system aims to help the driver to drive at a desired speed, keep a safe distance to preceding objects (other road users) and more importantly avoid longitudinal collision. To that end, the longitudinal

control can be activated/deactivated by the driver. When activated, it can perform automatic acceleration or braking based on the relative distance to the preceding vehicle to ensure a secure distance. In emergency situations, it can also perform automatic steering and braking to avoid longitudinal collision and performs automatic throttle control to limit or sustain speed.

A synopsis of the control scheme of the LC control system is shown on Figure 26. As it can be seen on the figure, the LC receives the relative distance to the preceding vehicle through the vehicle's onboards sensor data fusion algorithm (combining radar and camera sensor data). Based on these data and the input set by the driver, the LC will send commands to the powertrain (composed of engine and gearbox) and the braking systems to achieve the desired speed. Depending on the level of automation, numerous versions of LC implementations are available. In this case study, we focused on the Adaptive Cruise Control (ACC) consisting mainly of the speed and distance regulation components of longitudinal control system.

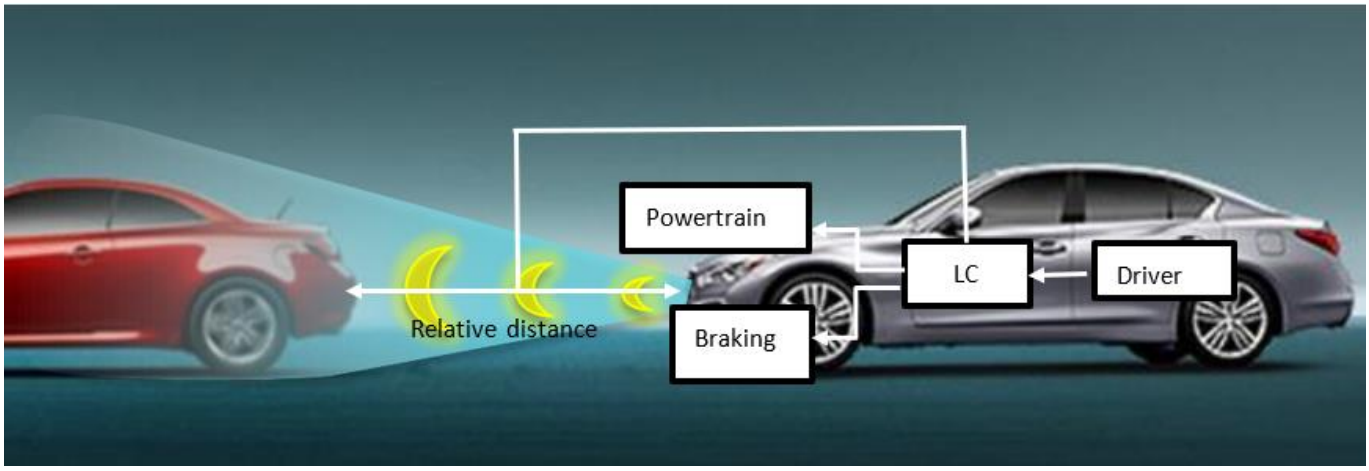

Figure 26. Longitudinal control context

As shown on Figure 27, the ACC relies on a distance and speed control software that calculates how fast the vehicle can travel while remaining in a safe situation with respect to certain predefined events (turns, traffic jams, stop signs, etc.). To do so the ACC adjusts the vehicle speed at the value set by the driver if there are no other vehicles driving ahead in the same lane. If slower vehicles are detected ahead of the vehicle by the frontal radar alone, or front Radar and frontal camera together, the system considers the distance relative to the preceding vehicle (called ACC target) and adjusts automatically the speed, acting through powertrain and braking systems to keep a preset distance of follow-up time.

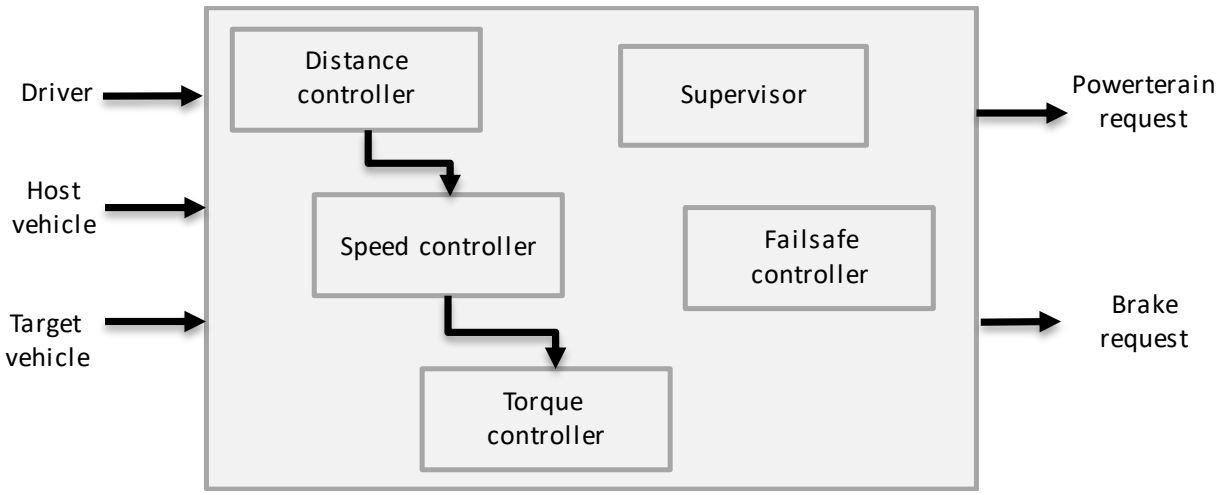

Figure 27. Adapted Cruise Control (ACC) feature of the longitudinal control

To ensure its safe behavior, the ACC relies on an internal safety protection in the form of a failsafe controller implemented as a software sub-component. The failsafe controller performs failure detection to manage limitation/adaptation of powertrain and brake request) and places the longitudinal control in some predefined safe states when certain faults or conditions occur. In addition to the failsafe controller, the ACC is equipped with an internal supervisor, a software subcomponent that manages its states (activation, stopping sequence…) as well as the driver requests (speed and distance setting). Several Unwanted Software Events (USWE) such as untimely braking or uncontrolled acceleration were derived from the system safety requirements and allocated within the ACC (because they possibly could come from the ACC). Other constraints (such as the ability of the driver to always be able to deactivate the ACC) are also taken into consideration.

The system was modeled taking into consideration how these requirements are implemented in the proposed software architecture of the longitudinal control while examining closely the safety mechanisms proposed in the software architecture (acceleration limits, ACC activation conditions…). This case study focuses on one single UWSE related to the occurrence of an unintentional acceleration above the permissible acceleration limit (defined by ISO 22179) during travel ($v \neq 0$ km/h) entailing longitudinal control. The first step of the methodology described in the following section will formally describe the identification of the software components related to the longitudinal controller based on the information collected from the Technical Safety Concept (TSC) and the system architecture.

5.3. Step by Step Application of the methodology

To apply our methodological proposal, we used SimfiaNeo, an MBSA tool based on the AltaRica language developed by APSYS-Airbus. It offers a graphical modeling interface based on Eclipse and implements the dataflow version of the AltaRica language. SimfiaNeo allows to graphically build the AltaRica model of a system and to directly generate cuts, fault trees and FMEAs from the AltaRica model. SimfiaNeo was therefore compatible with our approach; for our study, we used version 1.4. It must be noted that other AltaRica editors can be used, such as Cecilia Workshop (developed by Dassault Aviation) or AltaRica Studio (developed by LaBRI), both of which are based on the dataflow version of AltaRica. More recently, we can also find Open AltaRica from SystemX, based on the more recent 3.0 version of the language.

As a reminder, the major of the methodology thereafter in are outline in

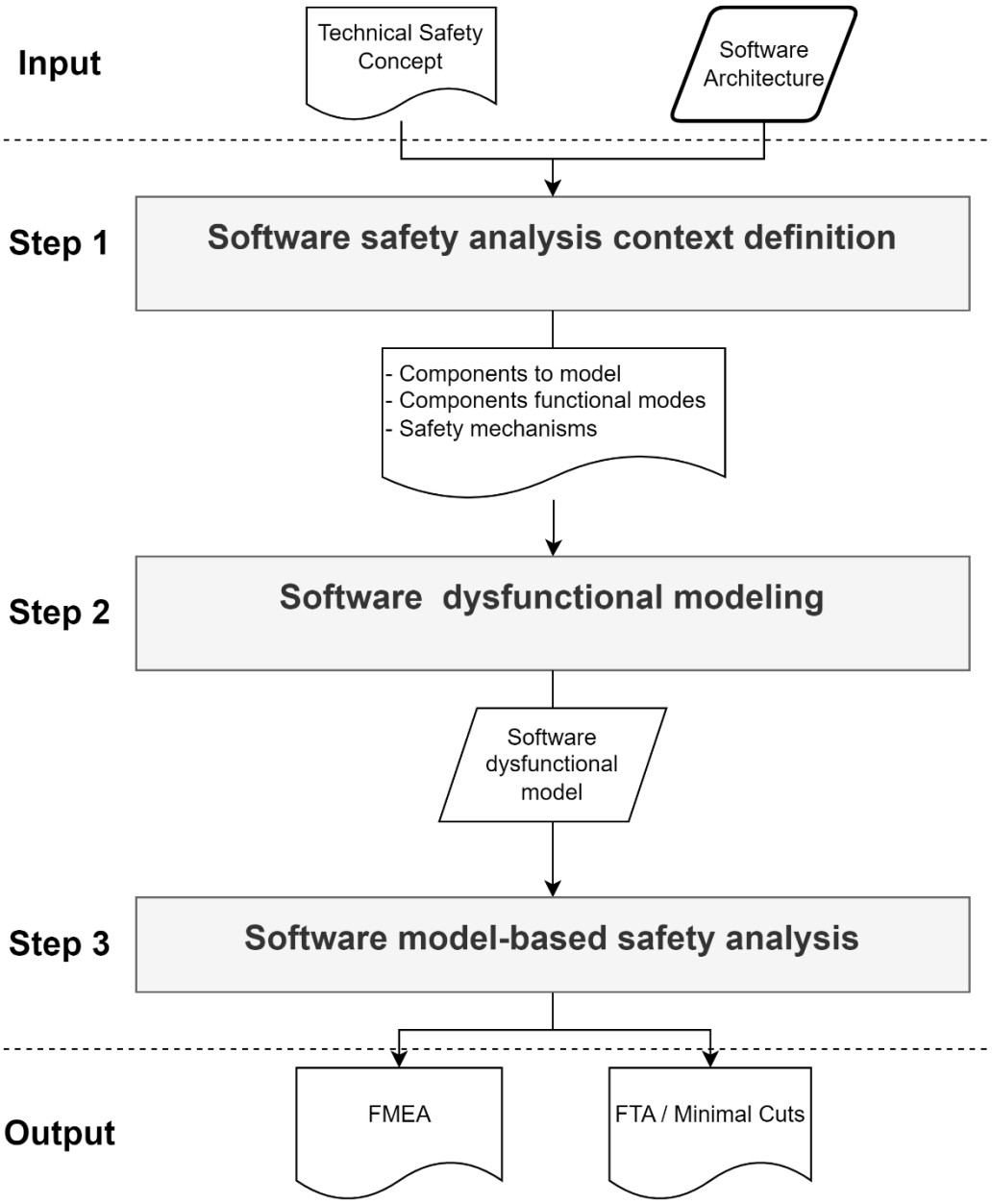

Figure 28. Major steps of the methodological proposal

5.3.1. Step 1: Software safety analysis context definition

In order to analyze the ACC, it is necessary to consider not only the internal elements but also the other elements (other software components) the Longitudinal Control (LC) interacts with in order to see how they can globally affect safety. This enabled delimiting the perimeter of the system and favoring a better interpretation of the interfaces. The interaction of the LC with other software components is outlined in Figure 29. As it can be seen, the LC is connected to the vehicle Status Input (VSI) for incoming data (such as vehicle speed) or messages from other ECU (fusion). In addition, the LC is also connected to the Failsafe Activation component from which it receives fault messages such as hardware failure or unavailability

messages. Finally, the LC is connected to the Braking and Engine Software component through which it sends output messages (brake requests, HMI outputs, …) to the ADAS ECU Vehicle status Output (VSO).

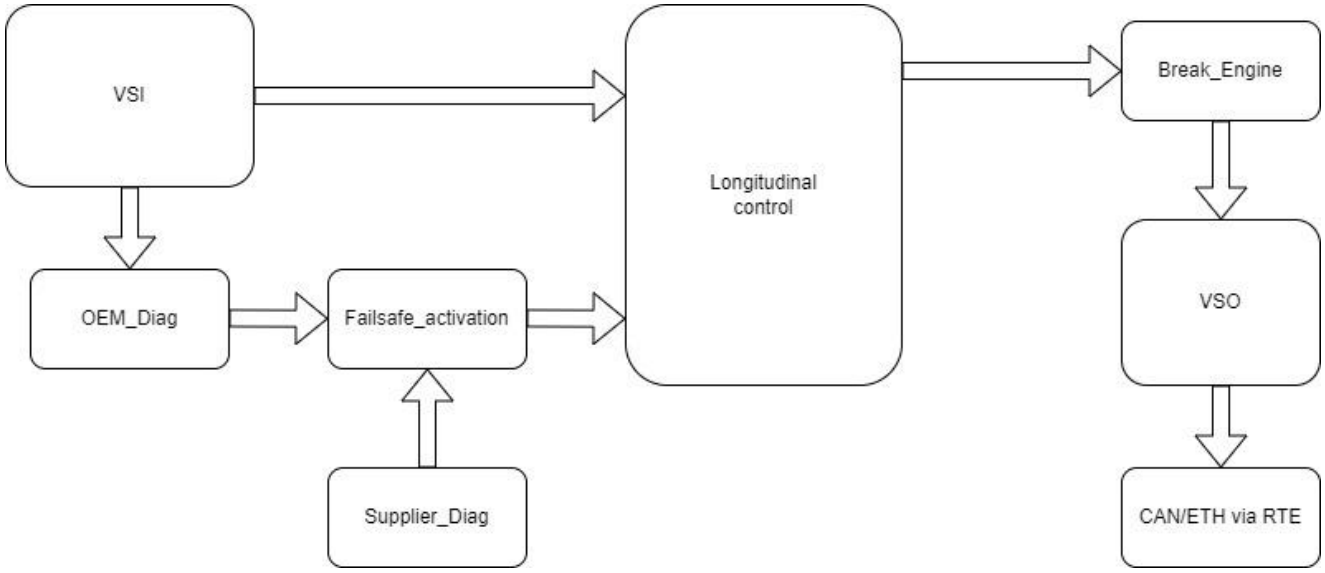

Figure 29. Longitudinal Control (LC) interfaces with other components

Having opted in our approach for a manual construction of a dysfunctional model, we first had to collect a set of design information for the purpose. To this end, official project deliverables were used. They consisted of the software architecture documents as well as the longitudinal control software safety concept. Contrary to the traditional safety analysis where such information is directly exploited to conduct safety analyses, they serve as the input to the dysfunctional model construction in our approach. Consistently with ISO 26262 requirements, the safety requirements associated with the longitudinal control are documented in a safety concept through safety goals that are declined to Unwanted Software Events (UWSE) at the software level. Using the elements contained in the safety concept (presented in the form of a multi-tab Excel file), we were able to identify the safety goals, the components (and subcomponents) to be modeled, and the associated input signals to take into consideration.

An extract of this safety concept is shown in Table 1. This extract gathers the elements that come into play in the case of the specified safety goal “REQ_TRV_SdF_TJP_HS_SG_004”. The safety goal prevents the occurrence of unintentional acceleration above the permissible acceleration limit (defined by ISO 22179) during travel ($v \neq 0$ km/h) entailing longitudinal control. The first column of the table lists all signals that contribute to this safety goal, while columns 2–4 and 6 specify the safety parameters and mechanisms to be implemented. Column 5 shows the allocation of these mechanisms to the software components and column 7 gives further details on the measures to be implemented. Other elements of this safety concept (not shown in the extract) detail the derivation of several software-level feared events from each safety goal, also specifying the associated ASIL levels. Based on all this data, we know exactly which signals and components (limited to those related to the safety goals) to model in our dysfunctional architecture.

Table 2. Extract from a technical safety concept

REQ_TRV_SdF_TJP_HS_SG_004						
No unintended Acceleration > ISO 22179 acceleration limit while travelling (v ≠ 0 km/h) requested by the longitudinal TJP feature						
	Checks	**Values**	**Proposed SM**	**SW allocation**	**Action (if an error is detected)**	**Remarks**
Acceleration target (V_req_mps2_AccMdlLim)	Limit	ISO22179 Acceleration	Range check	swcControlLongi	ADAS ECU shall switch into the Safe State SAST_TRV_SdF_TJP_004.	Output range check [ISO 22179 limit] on signal after limit.
Vehicle speed		Vehicle speed from 2 wheel (FR and FL or RR and RL) and maximum speed of 4 wheel speed	Plausibility check	Calculation in swcVehiclestatus_In and comparison in swcControlLongi	ADAS ECU shall switch into the Safe State SAST_TRV_SdF_TJP_004.	Used to derive limit of PWTWheelTorqueRequest
4 Wheel Speed						Checks are also performed on wheel speed CAN parameters received by the ADAS for unavailable, absent or invalid.
ADAS ECU Internal Failure				SUPDIAG FS_Act ControlLongi	ADAS ECU shall switch into the Safe State SAST_TRV_SdF_TJP_004.	Fault signal will be communicated to FS_Act and FS_Act will sent signal 'V_mes_x_ACCFailOut' to swcControlLongi.

We also identified the component operating modes from the design document, as well as their activation conditions. A summary of the available modes for the ACC software component is presented in Figure 30. We can see five possible states—including OFF and FAILED—and 3 substates (ACTIVE, CANCELED, and WAITING) grouped under a macro-state (ON). Also represented are the possible changes from one state to another (arrows), which gives us an idea of the possible transitions to model. Nevertheless, the architecture document did not provide enough details to allow us to describe these transitions.

Therefore, due to the lack of clarity in the expression of the high-level models mostly consisting of informal diagrams, we had to complement our understanding of the system by interpreting the Simulink models of the ACC. However, the Simulink models of the software components are considered to be part of the detailed architecture. Using the set of detailed architecture level artifacts, we were able to identify expected transition conditions (involved data, signals or conditions) related to the change between the operating modes shown on Figure 30.

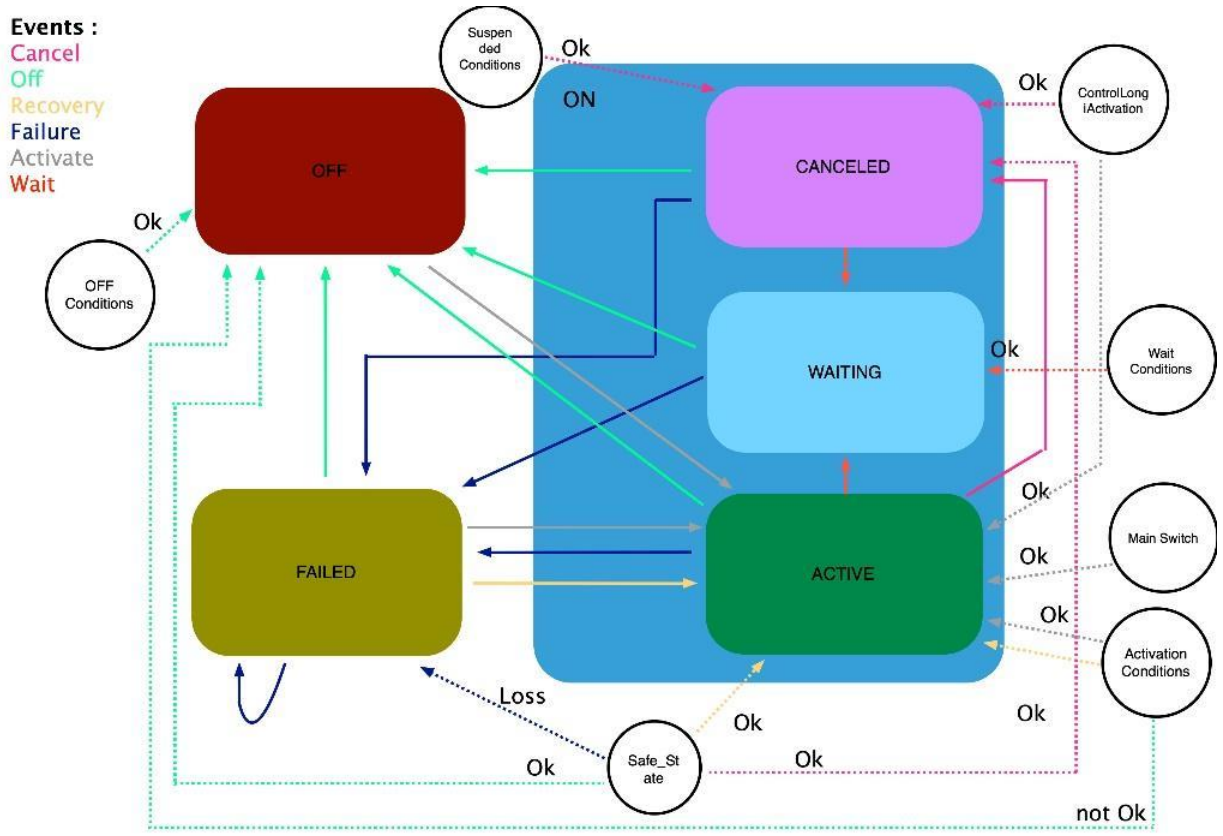

Figure 30. Operating modes of the longitudinal control

5.3.2. Step 2: Software dysfunctional architecture modeling

We used the SimfiaNeo tool to model the dysfunctional behavior and fault propagation logic of the software components taking part in the functioning of the ACC, proceeding as described in the second step of our proposed methodology. First, software fault patterns are modeled and stored in the SimfiaNeo library. Secondly, they are reused through instantiation to model software components and build the overall architecture of the software.

Software fault patterns modeling

We used the readily available model bricks to model the patterns, their dysfunctional behavior through states as well as their failure propagation logic in the AltaRica language using the SimfiaNeo tool. To model the patterns, we first declared the necessary AltaRica domains based on the fault categories describing generic software component's fault behavior as they related to ISO 26262 software fault categories (encompassing data integrity, data exchange, memory).

We also modeled generic prototypes of safety mechanisms, generic software component state as well as flow domains (possible dysfunctional values) for generic data. Using these prototypes of domains, we modeled bricks of components representing the elements of a software safety architecture intervening in the ACC's architecture. For instance, such elements included a rate limiting function which is a safety mechanism that is present in the speed and torque controllers within the ACC. For each pattern, the dysfunctional behavior is modeled, using abstract states (depending on the function of the pattern),

complemented with the transition crossing conditions based on the faults identified in the ISO 26262 software fault categories as previously presented in the previous chapter. Consistently with the AltaRica semantic, the modeled transitions are characterized by an event (occurrence of a fault potentially associate with a distribution law such as exponential or Dirac), a guard (condition to fulfill before triggering the effect), an effect (action resulting from the state change). For each event, the SimfiaNeo tool allows to specify a probability that will be used during calculations. However, given in the context of software faults, such values are irrelevant and a probability of 1 was used instead. For software based on different patterns, different AltaRica domains and events were used to model the dysfunctional behavior. Added to the dysfunctional behavior is the failure propagation logic linking the state of its output to its input(s). For each pattern, we wrote not only the fault propagation logic linking the inputs to the output but also the fault propagation logic linking the internal states to the outputs. The input-output relationship is established by examining the functional dependency of the component from its input(s).

Software component's dysfunctional behavior modeling

The basic behavior of instantiated components is inherited from the patterns. However, they can be modified, if deemed necessary, by dissociating them from their parent fault pattern. This can be necessary if the failure behavior or propagation logic of the components need to be refined further to accommodate the specificity of the functions they implement or to create new patterns. We used the readily available model bricks to model the components and their states in the tool, assigning the previously created domains to them.

Through the creation of AltaRica events in SimfiaNeo, we then modeled the transition crossing conditions using the equations identified in the state machines. An AltaRica event is characterized by a guard (condition to fulfill before triggering the state change), an effect (action resulting from the state change), and potentially a law (exponential, Dirac etc.). Traditionally, in AltaRica, the guards only show the state in which the component must be in order for the transition to occur; the event is then triggered according to the value of a specified probability law. By associating the inputs with the guards, however, we were able to go beyond this paradigm and ensured that the guards also depended on the inputs. This resulted in a state change that no longer depended solely on law-driven events (probabilistic or deterministic) but also on the state of the inputs.

Software component's fault propagation logic modeling

As for the state behavior, the fault propagation logic of each component is inherited from that of the pattern it is instantiated from and can be modified for the same reason. Hence, for each we wrote or updated the failure propagation logics linking the corresponding inputs and outputs. To this end, we studied Simulink files that specify the implemented functional logics. We also sorted and selected the elements to be considered since not all inputs and outputs were useful for our dysfunctional model, as they do not affect the associated safety mechanisms. The failure logic between components was mostly modeled using a generic flow pattern consisting of 3 literals (possible values) including nominal, erroneous, loss. However, depending on the function of the software component other literal such as delayed (for data exchange) or out of range (for a range limiting function for instance) were added to

generic flow pattern. Using these states, we were then able to model the dysfunctional information flows between components.

An extract of the Simulink logic of a subcomponent that manages safe states within the ACC component and the resulting AltaRica propagation logic code is presented in as an example. We can see that the State of the output F_x_SafeStateTJP004 (which is a status) is a logical OR of two inputs (which are also statuses). The first (F_x_FSAct_SafeStateTJP004) originates from the Failsafe Activation software ware component, and the second (F_x_Longi_SafeStateTJP004) is internal to the ACC. The corresponding AltaRica code that reflects this logic lies at the bottom of the figure.

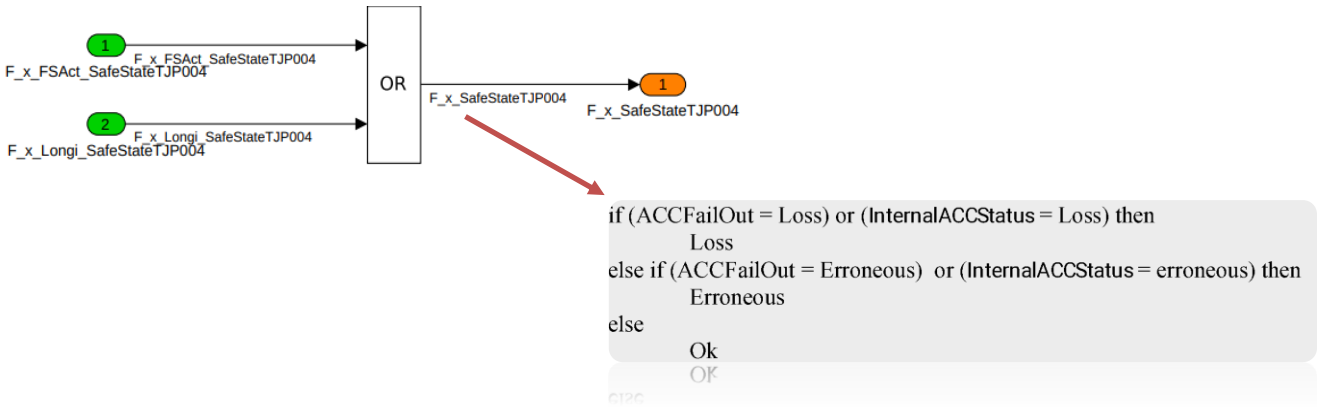

Figure 31. Transition from Simulink logics to AltaRica assertions

Thanks to the fault patterns, we were able to model the behavior of software components using the previously described fault categories and safety mechanisms. Using the elements stored in the library in an appropriate MBSA tool, one can easily prototype the dysfunctional architecture of a system through drag and drop. This is possible since the safety mechanisms self-contain their propagation logics as well as the associated fault modes.

We constructed the model presented on Figure 32 using the patterns stored in the library as shown by the library icon on some of the components (e.g., component identified as ETH at the bottom left of the models). As an example, the components labeled CAN and ETH in Figure 32 were both modeled through an instantiation of the data exchange fault pattern described earlier. Such was also the case of as well as subcomponent in the “vehicle-status-input” component that receive these data. Similarly, the “memory” component shown on Figure 32 as well as some subcomponents in the “longitudinal controller” component that read data from the memory were modeled through the instantiation of the data integrity fault pattern. Through these reusable libraries, we can model the dysfunctional behavior and failure propagation without having to manually rewrite their AltaRica code from scratch.

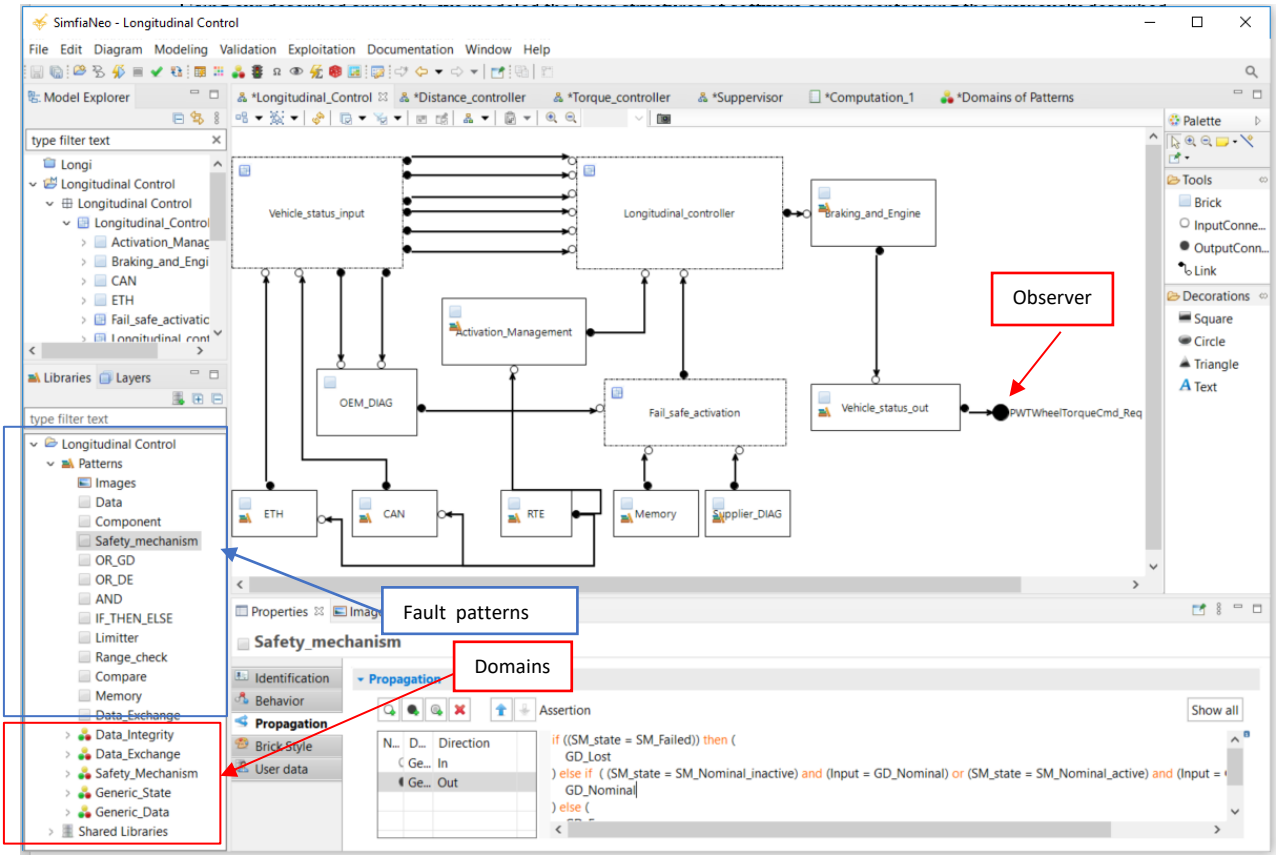

Figure 32. Pattern prototyping and dysfunctional model construction with SimfiaNeo

The resulting dysfunctional architecture will be used in the next step to conduct safety analysis in the SimfiaNeo tool.

5.3.3. Step 3: Software model-base safety analyses

The objective was to perform safety analyses from the dysfunctional model of the longitudinal control software component including the components that it interacts with using the safety analyses features available in the SimfiaNeo Tool. We performed various safety analyses including step by step simulations, minimal cuts, FTA and FMEA based on the dysfunctional model constructed using the previously developed fault patterns. Having completed the modeling of the longitudinal control software component and the components that interact with it.

To conduct safety analysis on a dysfunctional architecture, failure conditions (or in our case USWEs) must be added to the dysfunctional architecture. In AltaRica, this is done through observers. An AltaRica observer is an indicator consisting of a logical expression that expresses the dysfunctional logic of a failure condition that we wish to capture. For this purpose, we set up AltaRica observers on the outputs we were interested in (as shown in Figure 32). For example, let us consider the UWSE that we have chosen for our case study related to an “unintended Acceleration > ISO 22179 acceleration limit while travelling (v ≠ 0 km/h) requested by the longitudinal control feature”. In our model, we identified that the acceleration target and request in the speed controller subcomponent (Speed-Ctrl) are limited to 0.2G until vehicle

speed is above 10 km/h. We also identified that the final value of the acceleration target is transmitted to the engine through the engine management command 'PWTWheelTorqueCmd' (Powertrain Wheel Torque Command). Thus, any erroneous value of 'PWTWheelTorqueCmd' can result in the violation of the safety goal and the occurrence of the UWSE. Therefore, the observer's predicate to capture the occurrence of this UWSE can simply be 'PWTWheelTorqueCmd =Erroneous'. Having added the expression of the observer to the model, the objective was to verify, by means of the simulation, FMEAs, minimal cuts and fault trees, whether we could determine the events or faults that could lead to the transmission of this erroneous command that can result in the violation of the chosen UWSE.

Simulation

We ran a series of simulations where we observed the propagation of failures from individual components to the observer through the visualization of components in different colors (red: in the presence of a failure; orange: in error state; green: ok), as shown in Figure 33. Blocks containing several components appeared without color during the simulation.

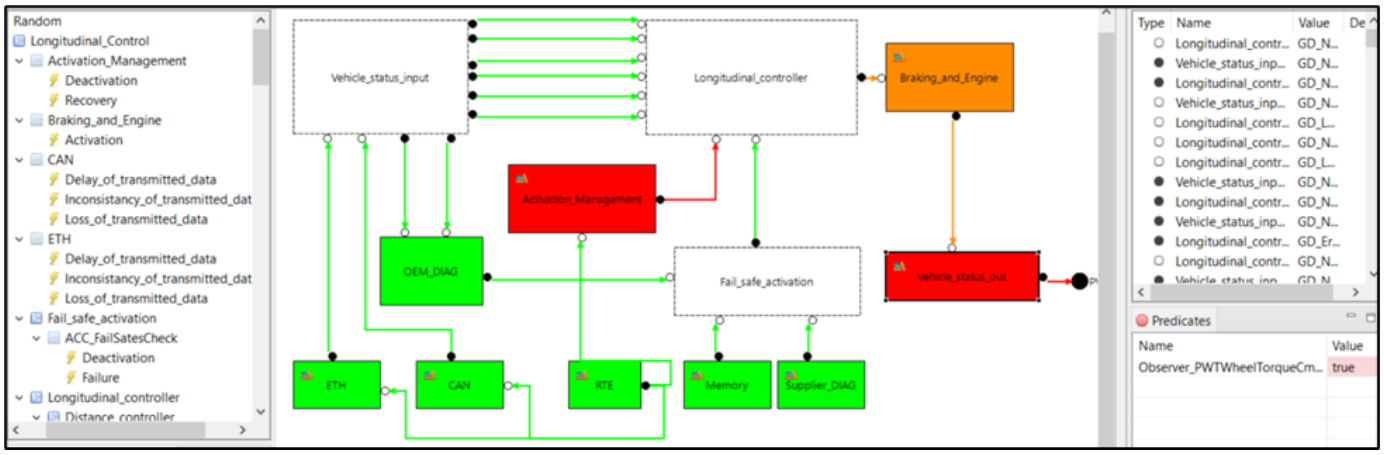

Figure 33. Simulation

This step—although not overly formal—allows the model to be verified as it is being built. The analyst can then use it to quickly evaluate the dysfunctional architecture by visualizing how all the components and observers react to the presence of one or more failures at specific locations. The simulation can be used to confirm and demonstrate (for communication purposes) the feared scenarios identified with the classic methods (FMEA and fault trees) that we will discuss in the following subsections. In addition to that, simulation can be useful to execute a use case or replicate

Failure Mode and Effects Analysis

We used SimfiaNeo to generate the FMEA tables from the dysfunctional model we had built. The FMEAs list all the events resulting in the violation of a safety goal (or of a created observer), doing so for each component of the model.

Table 3 shows an excerpt from an FMEA, containing a certain number of elements typically found in these tables. The first column (Event) lists the events causing the violation. In the following columns, we can find the Local Effect (effect of the event on the output of the initial component), the Intermediate Effect (effect of the event on all intermediate components between the initial component and the final observer), and the Final Effect (effect on the output of the model; in here, the effect on the observers described previously). For instance, in relation to our chosen UWSE, the excerpt from Table 3 shows how faults in the vehicle status input component related to vehicle speed can affect other components and the final observer.

Table 3. Generated FMEA table

Event	Local effect	Local effect val...	Intermediate ef...	Intermediate ef...	Final effect	Final effect value
Vehicle_status_input.Vehicle_speed_estimation.Loss_of_transmitted_data	Vehicle_status_input.Vehicle_speed_esti...	GD_Lost	Vehicle_status_i...	GD_Lost	PWTWheelTorq...	GD_Lost
Vehicle_status_input.Vehicle_speed_estimation.Delay_of_transmitted_data	Vehicle_status_input.Vehicle_speed_esti...	GD_Erroneous	Vehicle_status_i...	GD_Erroneous		
Vehicle_status_input.Vehicle_speed_estimation.Inconsistancy_of_transmitted_...	Vehicle_status_input.Vehicle_speed_esti...	GD_Erroneous	Vehicle_status_i...	GD_Erroneous		
Vehicle_status_input.ACCEngin_Perf_Status.Loss_of_transmitted_data	Vehicle_status_input.ACCEngin_Perf_Stat...	GD_Lost	Vehicle_status_i...	GD_Lost		
Vehicle_status_input.ACCEngin_Perf_Status.Delay_of_transmitted_data	Vehicle_status_input.ACCEngin_Perf_Stat...	GD_Erroneous	Vehicle_status_i...	GD_Erroneous		
Vehicle_status_input.ACCEngin_Perf_Status.Inconsistancy_of_transmitted_data	Vehicle_status_input.ACCEngin_Perf_Stat...	GD_Erroneous	Vehicle_status_i...	GD_Erroneous		
Vehicle_status_input.Parking_Brake.Loss_of_transmitted_data	Vehicle_status_input.Parking_Brake.Output	GD_Lost	Vehicle_status_i...	GD_Lost		
Vehicle_status_input.Parking_Brake.Delay_of_transmitted_data	Vehicle_status_input.Parking_Brake.Output	GD_Erroneous	Vehicle_status_i...	GD_Erroneous		
Vehicle_status_input.Parking_Brake.Inconsistancy_of_transmitted_data	Vehicle_status_input.Parking_Brake.Output	GD_Erroneous	Vehicle_status_i...	GD_Erroneous		
Vehicle_status_input.Status_input_swc.Failure	Vehicle_status_input.Status_input_swc.O...	GD_Lost	Vehicle_status_i...	GD_Lost	PWTWheelTorq...	GD_Lost

SimfiaNeo can export this document as an Excel spreadsheet, allowing for better data processing and sharing. This is an important asset of the tool, considering that MBSA tools are not necessarily used by many, but every engineer manipulates Excel files. Note, however, that Table 3 represents only a very small excerpt from the initial FMEA, which has more than 18,000 lines. Therefore, if the goal is to obtain usable details or to make a synthesis, this representation is not ideal.

To analyze the usefulness of this FMEA, a comparison with a manually performed FMEA would have been interesting. However, in the context of current practice in our case, there are no software FMEAs performed using the traditional approach. In contrast to fault trees that focus on one feared event, FMEAs are systematic and constitute a great way of showing that all failure modes have been accounted for within the system. This remains a difficult task for the safety analyst especially in the software context where failure modes can be plethoric. Despite the absence of a comparison with a manually performed FMEA, we argue that our approach allows performing this type of analysis that is otherwise difficult to perform manually.

Minimal cut sets and fault tree

The generation of the minimal cuts is achieved through the configuration of the cut set and sequence calculation engine for a given observer. Thus, for an observer we can choose the maximum order of the cuts, the filter type (minimal cut or minimal sequence) and the generation type (combination, permutation or stochastic) which will be used during the cut set or sequence calculations. The maximum order corresponds to the maximum number of primary events in a sequence.

To choose the maximum order, we experimented with values ranging from 2 to 5. We observed that SimfiaNeo load on the processor and memory consumption remained relatively constant (respectively close to 30% and 700 Megabyte) regardless of the maximum order value, while the computation time increased exponentially from under 2 minutes for order 2 to 7 hours for order 5.

Table 4. Generated minimal cut set

	Elements	Order ▲	Probability
1	RTE.Memory_access_fault	1	1.0
2	Longitudinal_controller.Speed_controller.Failure	1	1.0
3	Longitudinal_controller.Fail_safe_controller.Upper_range_value.Memory_access_fault	1	1.0
4	Longitudinal_controller.Torque_controller.Preprocessing.Loss_of_transmitted_data	1	1.0
5	Longitudinal_controller.Fail_safe_controller.Upper_range_value.Nominal_deactivation	1	1.0
6	RTE.Nominal_deactivation	1	1.0
7	Longitudinal_controller.Fail_safe_controller.Range_check.Failed_reaction	1	1.0
8	Longitudinal_controller.Torque_controller.Engine_torque.Failure	1	1.0
9	Longitudinal_controller.Fail_safe_controller.lower_range_value.Nominal_deactivation	1	1.0
10	Vehicle_status_out.Loss_of_transmitted_data	1	1.0
11	Vehicle_status_input.Target_speed.Loss_of_transmitted_data	1	1.0
12	Longitudinal_controller.Torque_controller.Dynamique_saturation.Failure	1	1.0
13	Longitudinal_controller.Fail_safe_controller.lower_range_value.Memory_access_fault	1	1.0
14	Braking_and_Engine.Failure	1	1.0
15	Vehicle_status_input.Status_input_swc.Failure	1	1.0
16	Vehicle_status_input.Vehicle_speed_estimation.Loss_of_transmitted_data	1	1.0
17	Longitudinal_controller.Distance_controller.Vstop_Vmin_Strategies.Failed_reaction	1	1.0
18	Longitudinal_controller.Distance_controller.Limitter.Failed_reaction	1	1.0
19	ETH.Loss_of_transmitted_data & CAN.Loss_of_transmitted_data	2	0.9999

Meanwhile the maximum order in the resulting minimal cut set remained equal to 3 even if we consider sequences of size 4 or 5. We chose order 3 for our case study—the higher the order, the longer the generation of the cut will take. An order of 3 is therefore a good tradeoff between computation time and accuracy. The choice of the filter type is also important; we have chosen the “minimal cuts'' option since it makes fault tree generation possible. Lastly, the choice of the generation type specifies the combinatorial or stochastic sequence (based on random simulations) used during the generation of the cut.

As an example, we considered the chosen UWSE linked to the transmission of an erroneous torque command to the engine. For this UWSE, we generated a minimal cut by choosing “order 3” as value of the maximum order, the “minimal cut” filter and “permutation” as the generation type. The generated minimal cut is shown in Table 4. It shows combinations (of order 1, 2) of basic events that could cause the specified UWSE, as well as their associated probabilities (added by default). We can see that the cut highlights the events and the hierarchical components, enabling traceability of the components at high level (as shown in Table 4). For dysfunctional models where several subcomponents have identical nomenclature, this traceability allows to clearly identify the origin of each event.

During the execution of these calculations, however, several compilation errors occurred, some of them due to the presence of loops in the failure propagation chain. This is a known issue related to the dataflow version of the AltaRica language. To solve this problem, we modified the assertions of the failure propagation involved in these loops. In the case of a redundant evaluation of a variable (where one of the assertions is part of the loop), removing the redundant evaluations of the involved variable allowed to break the loop. For this purpose, we considered a loop and identified the self-dependent variables in the chain of assertions constituting this loop. One possibility was to remove this variable from one of the assertions if it was already considered in another assertion. If this was not possible without modifying the validity of the assertion, the second possibility was to remove the dependency link and successively assign to the state variable all possible values and perform the calculations with each scenario. In the latter case, it was necessary to manually change the value of the variable to include the scenarios which were excluded by assigning it a fixed value. In both cases, the dysfunctional logic of the assertion remains valid.

For a defined feared event, SimfiaNeo allows the generation of FTAs from the equivalent minimal cut. Through its tree structure and logical combinations, FTAs illustrate how basic events (located at the bottom of the tree) can lead to the feared event (at the top of the tree). In other words, FTAs highlight the causal chain between the basic events at the component level (at the bottom of the tree) and the high-level feared event (at the top of the tree) through a tree structure represented in graphic form. In SimfiaNeo, it was possible to generate an equivalent reduced fault tree from minimal cuts for a defined feared event. Nevertheless, we did not identify any added value through this generation as the minimum cuts in our opinion present the same information in a more concise and readable format. Furthermore, generating FTAs from minimal cuts can be considered counter intuitive as in practice safety engineers use the reverse process (they use FTAs to compute minimal cuts).

5.4. Discussion

This chapter presented a case study of the ACC driver assistance system to show how to apply our methodological proposal consisting in a step-by-step methodology associated with the use of software fault patterns. The application of the methodological proposal to the case study makes it possible to underline its advantages but also to identify its limitations and identify avenues for improvement. It demonstrates that it is possible to apply an MBSA-type approach (which has been system-oriented until now) to the embedded automotive software. Through the appropriate selection of input data and modeling effort, our approach provides a pragmatic dysfunctional model representative of the real system, thus improving the quality of safety analyses and addressing complexity. Furthermore, the use of a tool such as SimfiaNeo relieves the safety expert of manual calculations while allowing him to concentrate on dysfunctional modeling. The benefits are reflected in terms of time saving, analysis quality and reusability of the same dysfunctional model for different types of safety analyses and for a large number of UWSEs. AltaRica for instance, once the dysfunctional model is built, it will be possible to conduct analyses for different USWEs by simply specifying as observers the conditions of their occurrence through predicates. In addition, the use of a formal semantics such as AltaRica allows representing the dysfunctional behavior of the system without ambiguity which is both beneficial to the quality of the safety analyses as well as the demonstration of safety evidence (in certification context for instance).

The application to the case study also shows that using fault patterns and the adequate tool, the dysfunctional model construction is made easy and safety analyses can benefit from this alternative new model construction method. In addition to the general benefits of adopting a model-based approach, the benefits of using the fault patterns include facilitated model construction, reuse and adherence to the ISO 26262 recommendation (in relation to the type of software faults to consider in software-oriented safety analyses). Once the fault patterns are built, they can be reused to build the dysfunctional model and making the dysfunctional model constructions easier and more efficient. Furthermore, the use of patterns can be beneficial to the adoption of the MBSA approach for the automotive software safety. For instance, the fault patterns can be developed by a safety expert and reused by non-experts, hence making the methodology more accessible. The specificity of the case study (automotive software component) also demonstrates that the use of software-oriented fault patterns can benefit the application of MBSA in the automotive software safety context.

All these elements contribute to the closure of identified the methodological gap related to the application of MBSA in the context of automotive software safety and they can significantly improve current practices in safety analysis that rely on traditional techniques. Furthermore, tools such as SimfiaNeo relieve the safety expert of manual calculations while allowing him to concentrate on dysfunctional modeling.

Nevertheless, we noted some limitations on our proposal. One of the difficulties brought about by a dedicated dysfunctional model is maintaining its consistency with the system design model when the latter evolves. This is not a new issue as the continuity between MBSE and MBSA remain a topic of interest in research. Implementing additional measures is therefore necessary to guarantee consistency in our approach.

5.5. Work in progress

This case study validated our methodological proposal on the ACC driver assistance system. In our approach, we mentioned a limitation related to consistency brought about by the use of a dedicated dysfunctional model approach in our methodology. To address this limitation, we specified a tool named Raphael, a simple Python-based tool used to automatically generate Failure Truth tables and AltaRica nodes of identified safety functions. The tool is based on Failure Logic Thinking (FLT), a novel concept that relies on a defined custom data type to enable automated calculations on failure logics. Through its ability to translate functional logic into failure logic, the tool aims to provides a better consistency between the functional and associated dysfunctional model.

As illustrated in Figure 34, Raphael is built around the definition of a custom data type named the Failean type representing discrete values of failure flows (such as nominal, failed or misleading). It is designed with its own set of operation rules based on the FLT concept that allow operations on the type such as addition, multiplication or comparison. As illustrated on Figure 34, the Failean type is implemented in Raphael as a new data type class. It has its own operating rules defined in its methods. The set, comprising the data type definition and the operators, allows to automate the generation of the failure logic of a given function specified. For instance, the tool can generate and display the AltaRica node of a specified

function. The user can export the generated node in a text file that can later be imported and reused in an AltaRica editor. Raphael also relies on FTT's discussed earlier for its internal usage (storage of already known failure logics). Furthermore, the tools allow the generation of the FTT's in a human readable format for review or sharing purposes. Currently basic functions such as addition, multiplication, comparison or negation on the defined dysfunctional datatype are implemented within the tool. The generation of FTTs and AltaRica nodes based on these simple functions have been successfully experimented.

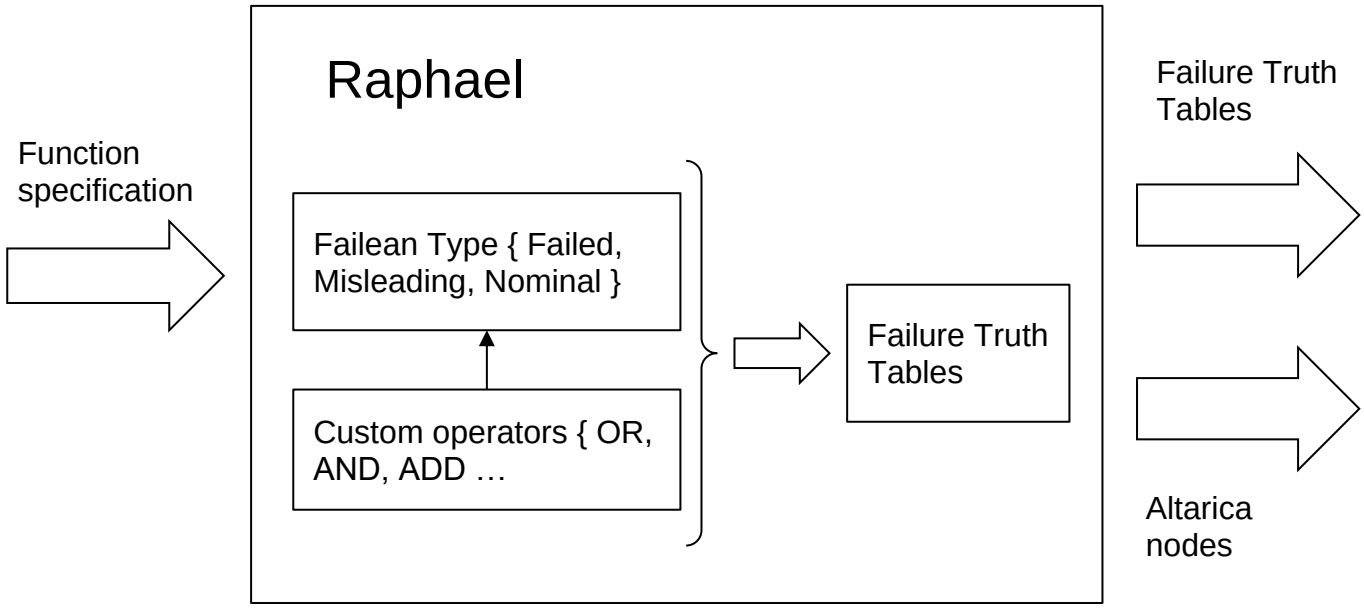

Figure 34. Overview of Raphael

5.6. Conclusion

This chapter applied our methodological proposal based on model-driven engineering that can be used to build a dysfunctional model in three steps, and from which it is possible to derive classic safety models. It recounts how we performed software safety analyses based on our methodology and compared the results to those obtained using the traditional approaches.

Using the SimfiaNeo tool and the AltaRica language, we applied the methodology on a case study, building a dysfunctional model of a software from which we were able to generate FMEAs, minimal cuts and fault trees. These results are encouraging and demonstrate that it is possible to apply an MBSA approach to evaluate software safety, especially in automotive applications. They also highlight the benefits of generating safety analyses from a dysfunctional model (time saving, analysis quality, and reusability).

This chapter also made a methodological proposal based on fault patterns that can be used to build a dysfunctional model, and from which it is possible to derive classic safety models. Using the SimfiaNeo tool and the AltaRica language, we applied the methodology on a case study, building a dysfunctional model of a software from which we were able to generate FMEAs and minimal cuts. These results are

encouraging and demonstrate the usefulness of patterns to facilitate MBSA model construction. More generally, they show that it is possible to apply an MBSA approach to evaluate software safety, especially in automotive applications. They also highlight the benefits of generating safety analyses from a dysfunctional model (time saving and reusability). However, since our proposal is based on a dedicated dysfunctional model, it will also be essential to supplement the method with a mechanism that ensures consistency between the design models and the safety model

As a perspective this chapter introduces Raphael, a simple python-based tool that can be used to automatically generate Failure Truth tables and AltaRica nodes of identified safety functions. The chapter also introduces Failure Logic Thinking (FLT), a concept to assist in writing dysfunctional logic implemented in the tool as a new data type specific to failure domain enabling the representation and operation on dysfunctional literals such as erroneous or failed. We argued that the implementation of this novel concept in the new tool can enable automated functional to dysfunctional logic translation, resulting in better quality dysfunctional models as well as a better consistency between functional and associated dysfunctional models. The new method can be used in complement to our previous proposal that it aims to improve.

However, although the new proposal has a clear potential to improve the consistency between functional and dysfunctional logics through logic translation, we are still far from a seamless integration of software MBSA and the software model-based engineering process. Nevertheless, the logical choice appears to first close the methodological gap before attempting to establish consistency with the software model-based development process.

Chapter 6. Conclusion and perspective

Abstract: *This chapter concludes our work. It starts by recalling the need and the context that motivated our thesis as well as the elaboration of the research questions. The chapter then recalls the steps that we took to solve the identified issues and summarizes the different contributions. Finally, it outlines the results obtained through the application of our proposal to the case study and their significance, before indicating some perspectives aiming at solving the identified limitations.*

6.1. Motivation reminder

This research work was motivated by the need of improvement of the current practices of safety analyses in the automotive embedded software context, where safety assessment practices are a recommendation of ISO 26262 (Road vehicles – Functional safety). In particular this standard recommends that safety analyses be conducted throughout the development cycle (including on the system, hardware and software architectures). However, a study of the state of current practices in terms of MDE and safety analyses revealed that, in the automotive sector in general, the current practices are essentially relying on time-consuming manual methods that are prone to human error and very limited in the context of evolving complexity. Because these analyses are done manually on the basis of the experience of the engineers, there is no guarantee that they will always be accurate. In addition, safety engineers base their analyses on their personal interpretation of data that was retrieved from documents of various sources. Not only is the quality of the data and input documents for these analyses are problematic, but their interpretation also due to the fact that they are subject to human error. In addition, these practices result in a lot of wasted time spent getting the information needed for the safety of the analyses. Under these conditions, and with the growing complexity in the automotive industry characterized by the increasing use of embedded software assuming sometimes critical functionalities, it was deemed necessary to seek for new ways to improve the state of current practices of safety analysis in order to be able to ensure more efficient, complete, correct and formal safety analyses. To address this issue related to the inadequacy of the current safety analysis practices in the context of embedded automotive software, our overall proposal was to rely on Model Driven Engineering (MDE) to conduct safety analyses no longer from design documents but from a model of the software architecture.

6.2. Research methodology and contributions

To address concerns described above and improve the state of current practices, we **first formulated the general problem** we addressed “How can current embedded software safety analysis practices be improved in automotive?” declined it into several problem descriptions. For instance, one problem description states that “the current model-based safety analysis methods, tools and languages are system

oriented". This problem description was in turn **declined into a research questions** taking into consideration the specificities of the industrial context and needs of the PhD, for instance "whether the current MBSA methods, tools & language can be applied to automotive software safety analysis". We then proceeded to a literature review focused on analyzing the current MBSA approaches to explore if they have indeed been applied in such context or whether this problem has already been addressed and how. Based on this **analysis of the state of the art**, more specific research questions are formulated (cf Q1.1 and Q1.2 in Table 5); and to answer these research question, contribution proposals including a methodology and a choice of modeling formalisms.

Table 5. General problem, research questions and contributions

<table>
<tr><th>General problem (P0)</th><th>Problem Description (P)</th><th>Research Questions (Q)</th><th>Contributions (C)</th></tr>
<tr><td rowspan="3">P0: ISO 26262 software safety analysis current practice relies on manual traditional safety analysis techniques resulting in subjective, inefficient, poor quality, error prone analyses.

General question

Q0: How can the current safety analysis practices be improved?

General Answer

A0: Apply Model Driven Engineering</td><td>P1: Current model-based safety analysis methods, tools and languages are system oriented</td><td>Q.1.1: How can the current MBSA methods, tools & languages be applied to the automotive software safety analysis?

Q1.2: What modeling approach for software Model-Based Safety Analysis (dedicated or extended model approach)?</td><td>C1: Step by step methodological proposal for software MBSA with including a modeling formalism choice and a step context definition to reduce complexity</td></tr>
<tr><td>P2: Growing Software architectures complexity making safety analysis modeling challenging</td><td>Q2: How to better master the growing complexity?</td><td>C2: Use Software Fault Patterns and Fault Truth Tables to facilitate modeling</td></tr>
<tr><td>P3: Current ISO 26262 safety analysis suffers from poor integration with software Dev Process. Existing MBSE MBSA consistency frameworks are limited to System level</td><td>Q3: How to ensure better integration of software safety analysis with software development process?</td><td>C3: Proposal for an automatic functional to dysfunctional logic translation tool based</td></tr>
</table>

We then **developed and experimented an analysis methodology based on MDE** to conduct the safety analyses of automotive embedded software. Among the possible approaches reported in the state of the art, we chose to explore that of a dedicated dysfunctional model, based on the Failure Logic Modeling and using the AltaRica language. Our methodological proposal consists in modeling a dysfunctional architecture of the software that can be used to automatically perform safety analyses and generate

safety models such as minimum cuts, fault trees. The methodology consists in: 1rst step) defining the software safety analysis context, 2d step) establishing a software dysfunctional modeling thanks to the help of fault patterns and 3rd step) analyzing the software safety. **The originality of the contribution lies in the adaptation and transfer of the concepts, principles and methods of Model-Based Safety Analysis (MBSA), generally applied at system level, to software level**.

Let us emphasize that **a notable element of contribution consists in the very first step of the methodology** which could seem trivial. Indeed, due to the difficulty sometimes to have the relevant data for building the dysfunctional model, the first step of our methodology is crucial as it allows defining a framework of elements to be considered in the construction of the dysfunctional model. This preliminary step addressed two main concerns. First, the choice of input data specifically addresses the difficulty of having good inputs for the construction of the dysfunctional model. Secondly, the step makes it possible to better manage complexity by choosing as elements to be modeled only those related to safety. This was necessary because one limitation of the manual approach to modeling the dedicated dysfunctional architecture was how to deal with complex software architectures for which the manual modeling approach would not be efficient. To this end, **we proposed to adopt a more pragmatic modeling approach based on a good definition of the architecture elements** to be modeled in order to better manage complexity. Indeed, for the case of complex software architectures, limiting the elements to be modeled to those that impact safety as we have proposed makes the modeling approach less complex and more pragmatic. Indeed, we believe that limiting the MBSA model to safety related components is sufficient to carry out meaningful safety analysis and can improve both efficiency and the quality of safety analysis. Indeed, models being abstractions of real systems, a common practice in high level systems modeling is that not all aspects of the systems are modeled but only certain the representation of interest for the application domain. In alignment with this same principle, it is logical in the safety domain that dysfunctional models are limited to safety related items.

In our modeling approach **we first relied on the use of Failure Truth Tables** (FTT) to facilitate writing the failure propagation logics. To recall, FTTs are dysfunctional failure propagation tables consisting of discrete input and output variables whose possible values are defined depending on the failure behavior of the software component's function. Through these FTTs, we were able to systematically map the nominal flows (from the functional logic) into failure flows (dysfunctional logic). FTTs helped us to write the dysfunctional logic of a function based on its input flow variables. **We then improved the proposal by** completing the use of FTTs thanks to **the adoption of a pattern-oriented approach** that relies on patterns that integrate the failure logics we initially defined using the FTTs. Compared to FTTs, these fault pattens offer a more powerful feature. They encapsulate the same logics as the FTTs but can be reused to model relevant software component without any additional modeling effort. Whereas in the case of FTTs, they

are used as knowledge support and the failure logic must be manually re written for every new component. Patterns offer several interests. They **make modeling easier and faster**; they can be reused to model the dysfunctional behavior of any software components. They also **allow reducing human modeling errors** because the propagation logics are defined upstream. Through the use of reusable patterns, the MBSA model construction is easier and **can be beneficial to the adoption of the proposal by automotive companies**. Last but not least, due to the consistency brought by the use of patterns, the **quality of safety analysis is also improved**.

6.3. Results

We demonstrated the efficiency and efficacy of the proposal on a case study of an automotive software component, the ADAS longitudinal control system which is a driver assistance system that manages acceleration and braking. We carried out a safety analysis of the software components following the prescribed steps of the methodology, incorporating the use of the fault patterns. Using the SimfiaNeo MBSA tool and the AltaRica language semantics, we were able to show how the proposed approach made it possible to automate safety analysis as well as automatically generate classic safety models including minimum cuts, FTAs and FMEA intended safety analysis and demonstration purposes. Although it may not be always necessary in some circumstances, the generation of the classical safety models through MBSA can serve as a way to fulfill regulatory requirements. For instance, in the automotive context, FTAs or FMEAs can be included in the safety case to be used as proof of compliance with the standard's requirements.

Thanks to the case study results, we concluded on the implementation and usefulness of the methodology. Combined with the languages and tools we used, we showed that the methodology adds value and improves current manual safety analysis practices in time, quality of analysis and reuse. We also showed that, thanks to the steps of our methodology, the software dysfunctional model construction can be made easier and more efficient through reuse.

The set of contributions mentioned above allowed us to position our work in relation to existing processes (software development process, system-level safety assessment processes) and in compliance with the requirements of ISO 26262 as indicated in Figure 35 below. Indeed, one of the ongoing efforts at the company level (Renault) is the integration of various ongoing works into a broader context of the adoption of model-based approaches in the system and software level development process as well as its application in a project context through a common case study in a project context. Figure 35 shows how our contributions, outlined in the green rectangle, connects to existing processes outlined in gray (on the left, the model-based software engineering, and at the top the system level safety activities).

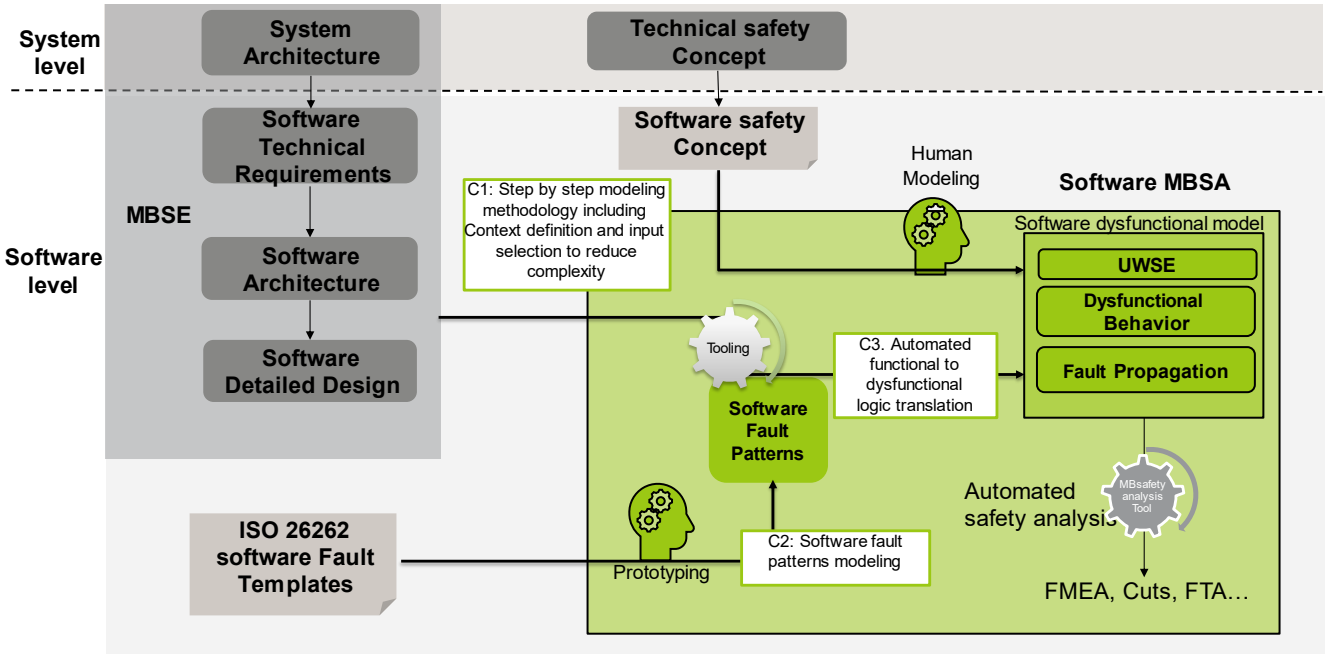

Figure 35. summary of the contributions

6.4. Perspectives

In line with the limitations identified in our proposal (notably the challenge related to the use of manual modeling in a dedicated model approach), we are currently working on a semi-automation of logic translation based on our proposed dysfunctional truth tables and a new concept, the Failure Logic Thinking (FLT). We defined FLT as a concept to assist in writing dysfunctional logic implemented through the definition of a dedicated data type specific to software failure domain enabling the representation and manipulation of dysfunctional literals. The principle is that if all the truth tables and fault patterns are integrated in a tool, they can enable the automatic generation of the dysfunctional behavior of software components based on the implemented functions. As a result, the construction of fault propagation logics will be facilitated and improved. This last work has therefore essentially aimed at improving the construction of dysfunctional architectures by facilitating it (through reuse) and reducing the risk of modeling errors (thanks to the propagation logics convened in advance and stored in FTTs). This is necessary because in the context of safety analyses based on dedicated models, modeling can be tedious and prone to human error as outlined in the limitations of our methodology.

The proposal will be implemented through a tool (Raphael) that is still undergoing development and improvement. Through automated functional to dysfunctional logic translation the proposal aims to save the safety analyst from tedious dysfunctional logic interpretation. The tool also aims to ensure a better consistency of software safety analyses with the software development process and help comply with the ISO 26262 recommendations by basing failure modeling on the category of software failures specified by

the standard. Through the tooling proposal, the fault pattern prototyping process can also be made easier if based on the component nodes generated through the application of FLT in Raphael. The integration of our proposal in the new tool should provide a better consistency resulting in better quality of dysfunctional models as well as a better consistency between functional and associated dysfunctional models which can benefit the overall adoption model driven approach at company level.

Beyond the work in progress, another perspective that is more strategical could consist of assessing the maturity of the proposed methods and tools envision their deployment. Such assessment could include testing the methodology and tooling proposals with on a targeted audience in order to assess their acceptability. To have feedbacks and seek further avenues for improvement, a wide test on different projects can also be envisioned. Finally, the tooling choice can be re appreciated to take into consideration the evolution of the tool's new dissemination policy (prices, wider distribution, training) in regard to the evolution of the other tools that are available on the market.

References

[1] Y. Argotti, “Study of Qualimetry essentials applied to embedded software development,” phdthesis, INSA de Toulouse, 2021. Accessed: Dec. 14, 2022. [Online]. Available: https://theses.hal.science/tel-03738148

[2] O. Guetta, E. Coutenceau, and K. Ishigami, “Renault Nissan new Software Strategy,” in *ERTS 2018*, Toulouse, France, Jan. 2018. Accessed: Nov. 18, 2019. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02156078

[3] A. Yasmine, R. Ameur-Boulifa, P. Guitton-Ouhamou, and R. Pacalet, “Automatic Support for Requirements Validation”.

[4] P. Koopman, U. Ferrell, F. Fratrik, and M. Wagner, “A Safety Standard Approach for Fully Autonomous Vehicles,” in *Computer Safety, Reliability, and Security*, vol. 11699, A. Romanovsky, E. Troubitsyna, I. Gashi, E. Schoitsch, and F. Bitsch, Eds. Cham: Springer International Publishing, 2019, pp. 326–332. doi: 10.1007/978-3-030-26250-1_26.

[5] N. G. Leveson, “Safety as a system property.” Association for Computing Machinery, Nov. 01, 1995. Accessed: Apr. 20, 2020. [Online]. Available: https://doi.org/10.1145/219717.219816

[6] S. Goetsch, “Safeware: System Safety and Computers, by Nancy Leveson,” *Med. Phys.*, vol. 23, no. 10, pp. 1821–1821, 1996, doi: 10.1118/1.597766.

[7] “J3016_202104: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles - SAE International.” https://www.sae.org/standards/content/j3016_202104/ (accessed Dec. 07, 2022).

[8] “IEC 61508: Functional Safety - Standards.” https://www.iec.ch/functionalsafety/standards/page2.htm (accessed Jun. 10, 2020).

[9] DoD, “MIL-STD-882E “Department of Defense Standard Practice System Safety".” May 12, 2012. Accessed: Oct. 16, 2020. [Online]. Available: https://www.dau.edu/cop/armyesoh/DAU%20Sponsored%20Documents/MIL-STD-882E.pdf

[10] “ARP4754A/ ED-79A - Guidelines for Development of Civil Aircraft and Systems.”

[11] S-18 Aircraft and Sys Dev and Safety Assessment Committee, “ARP4761 - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment,” SAE International. doi: 10.4271/ARP4761.

[12] International Organization for Standardization, “ISO 26262 2018 — Road vehicles — Functional safety,” ISO, Edition 2, Dec. 2018.

[13] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, “NUREG-0492, ‘Fault Tree Handbook’.,” p. 209, Jan. 1981.

[14] SAE, “Potential Failure Mode and Effect Analysis Reference Manual.” Feb. 2015. Accessed: Apr. 20, 2020. [Online]. Available: https://www.lehigh.edu/~intribos/Resources/SAE_FMEA.pdf

[15] NDIA, “Final Report of the Model Based Engineering Subcommittee,” Feb. 2011. Accessed: Oct. 16, 2020. [Online]. Available: https://www.ndia.org/-/media/sites/ndia/meetings-and-events/divisions/systems-engineering/modeling-and-simulation/reports/model-based-engineering.ashx

[16] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice: Second Edition*, 2nd ed. Morgan & Claypool Publishers, 2017.

Reference

[17] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, Inc, 2007. [Online]. Available: https://onlinelibrary.wiley.com/doi/book/10.1002/9780470249260

[18] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep. 2003, doi: 10.1109/MS.2003.1231146.

[19] D. C. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *Computer*, vol. 39, no. 2, pp. 25–31, Feb. 2006, doi: 10.1109/MC.2006.58.

[20] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Sci. Comput. Program.*, vol. 89, pp. 144–161, Sep. 2014, doi: 10.1016/j.scico.2013.03.017.

[21] H. Snyder, "Literature review as a research methodology: An overview and guidelines," *J. Bus. Res.*, vol. 104, pp. 333–339, Nov. 2019, doi: 10.1016/j.jbusres.2019.07.039.

[22] A. B. Rauzy and C. Haskins, "Foundations for model-based systems engineering and model-based safety assessment," *Syst. Eng.*, vol. 22, no. 2, pp. 146–155, 2019, doi: 10.1002/sys.21469.

[23] M. Batteux, T. Prosvirnova, and A. Rauzy, "System Structure Modeling Language (S2ML)," Dec. 2015. Accessed: Dec. 07, 2022. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01234903

[24] C. Baron and V. Louis, "Towards a continuous certification of safety-critical avionics software," *Comput. Ind.*, vol. 125, p. 103382, Feb. 2021, doi: 10.1016/j.compind.2020.103382.

[25] J. A. Estefan, "Survey of Model-Based Systems Engineering (MBSE) Methodologies," p. 70, 2008.

[26] "Welcome To UML Web Site!" https://www.uml.org/ (accessed Apr. 23, 2021).

[27] "SysML Open Source Project - What is SysML? Who created it?," *SysML.org*. https://sysml.org/index.html (accessed Apr. 23, 2021).

[28] P. H. Feiler, B. Lewis, and S. Vestal, "The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering:," Defense Technical Information Center, Fort Belvoir, VA, Apr. 2003. doi: 10.21236/ADA612735.

[29] P. Cuenot *et al.*, "The EAST-ADL Architecture Description Language for Automotive Embedded Software," in *Proceedings of the 2007 International Dagstuhl Conference on Model-based Engineering of Embedded Real-time Systems*, Berlin, Heidelberg, 2010, pp. 297–307. Accessed: Nov. 13, 2019. [Online]. Available: http://dl.acm.org/citation.cfm?id=1927558.1927574

[30] "Simulink - Simulation and Model-Based Design." https://www.mathworks.com/products/simulink.html (accessed Apr. 23, 2021).

[31] J. Colaço, B. Pagano, and M. Pouzet, "SCADE 6: A formal language for embedded critical software development (invited paper)," in *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, Sep. 2017, pp. 1–11. doi: 10.1109/TASE.2017.8285623.

[32] M. Staron, "Detailed Design of Automotive Software," in *Automotive Software Architectures: An Introduction*, M. Staron, Ed. Cham: Springer International Publishing, 2017, pp. 117–149. doi: 10.1007/978-3-319-58610-6_5.

[33] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio, "Grand challenges in model-driven engineering: an analysis of the state of the research," *Softw. Syst. Model.*, vol. 19, no. 1, pp. 5–13, Jan. 2020, doi: 10.1007/s10270-019-00773-6.

[34] M. Chami and J.-M. Bruel, "A Survey on MBSE Adoption Challenges," p. 16.

[35] R. Van Der Straeten, T. Mens, and S. Van Baelen, "Challenges in Model-Driven Software Engineering," in *Models in Software Engineering*, Berlin, Heidelberg, 2009, pp. 35–47. doi: 10.1007/978-3-642-01648-6_4.

Reference

[36] VDA QMC, “Automotive SPICE Process Assessment / Reference Model.” Jul. 16, 2015. Accessed: Jun. 10, 2020. [Online]. Available: http://www.automotivespice.com/fileadmin/software-download/Automotive_SPICE_PAM_30.pdf

[37] Center for Chemical Process Safety, “Appendix D: Minimal Cut Set Analysis,” in *Guidelines for Chemical Process Quantitative Risk Analysis*, John Wiley & Sons, Ltd, 2010, pp. 661–670. doi: 10.1002/9780470935422.app4.

[38] P. Mauborgne, S. Deniaud, É. Levrat, É. Bonjour, J.-P. Micaëlli, and D. Loise, “The Determination of Functional Safety Concept coupled with the definition of Logical Architecture: a framework of analysis from the automotive industry,” *IFAC-Pap.*, vol. 50, no. 1, pp. 7278–7283, Jul. 2017, doi: 10.1016/j.ifacol.2017.08.1400.

[39] P. Fenelon and B. D. Hebbron, “Applying HAZOP to Software Engineering Models,” 2008.

[40] J. A. McDermid, M. Nicholson, D. J. Pumfrey, and P. Fenelon, “Experience with the application of HAZOP to computer-based systems,” presented at the COMPASS - Proceedings of the Annual Conference on Computer Assurance, 1995, pp. 37–48.

[41] N. G. Leveson and P. R. Harvey, “Software fault tree analysis,” *J. Syst. Softw.*, vol. 3, no. 2, pp. 173–181, Jun. 1983, doi: 10.1016/0164-1212(83)90030-4.

[42] O. Lisagor, T. Kelly, and R. Niu, “Model-based safety assessment: Review of the discipline and its challenges,” in *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*, Guiyang, China, Jun. 2011, pp. 625–632. doi: 10.1109/ICRMS.2011.5979344.

[43] M. Bouissou, H. Bouhadana, M. Bannelier, and N. Villatte, “Knowledge Modelling and Reliability Processing: Presentation of the Figaro Language and Associated Tools,” *IFAC Proc. Vol.*, vol. 24, no. 13, pp. 69–75, Oct. 1991, doi: 10.1016/S1474-6670(17)51368-3.

[44] M. Bouissou, S. Humbert, S. Muffat, and N. Villatte, “KB3 TOOL: FEEDBACK ON KNOWLEDGE BASES,” p. 6.

[45] M. Bouissou, “Automated Dependability Analysis of Complex Systems with the KB3 Workbench: the Experience of EDF R&D.” 2005.

[46] G. Point and A. Rauzy, “AltaRica: Constraint automata as a description language,” 1999. Accessed: Nov. 28, 2019. [Online]. Available: http://www.altarica-association.org/ressources/ARBib/PointRauzy1999-AltaRicaConstraintLanguage.pdf

[47] M. Boiteau, “The AltaRica data-flow language in use: modeling of production availability of a multi-state system.,” *Reliab Eng Syst Saf*, vol. 91, no. 7, pp. 747–755, 2006, doi: 10.1016/j.ress.2004.12.004.

[48] M. Bozzano *et al.*, “Symbolic Model Checking and Safety Assessment of Altarica models,” vol. 35, p. 16, 2010.

[49] M. Batteux, T. Prosvirnova, A. Rauzy, and L. Kloul, “The AltaRica 3.0 project for model-based safety assessment,” in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, Jul. 2013, pp. 741–746. doi: 10.1109/INDIN.2013.6622976.

[50] M. Batteux, T. Prosvirnova, and A. B. Rauzy, “AltaRica 3.0 in ten modelling patterns,” *Int. J. Crit. Comput.-Based Syst.*, vol. 9, no. 1–2, pp. 133–165, Jan. 2019, doi: 10.1504/IJCCBS.2019.098809.

[51] F. Cassez, C. Pagetti, and O. Roux, “A timed extension for AltaRica,” p. 43.

[52] M. B. Batteux, T. Prosvirnova, and A. Rauzy, “Enhancement of the AltaRica 3.0 stepwise simulator by introducing an abstract notion of time,” in *28th European Safety and Reliability Conference ESREL (ESREL 2018)*, Trondheim, Norway, Jun. 2018. Accessed: Jun. 03, 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01826656

[53] P. Bieber, C. Bougnol, C. Castel, J.-P. H. Christophe Kehren, S. Metge, and C. Seguin, “Safety Assessment with Altarica,” in *Building the Information Society*, Boston, MA, 2004, pp. 505–510. doi: 10.1007/978-1-4020-8157-6_45.

[54] M. Machin, L. Sagaspe, and X. de Bossoreille, “SimfiaNeo, Complex Systems, yet Simple Safety,” p. 4.

[55] “AltaRica Project | MEthods and Tools for AltaRica Language.” https://altarica.labri.fr/wp/ (accessed May 27, 2020).

[56] “OpenAltaRica.” https://www.openaltarica.fr/ (accessed May 27, 2020).

[57] M. Bozzano *et al.*, “ESACS: an integrated methodology for design and safety analysis of complex systems,” p. 8.

[58] O. Akerlund *et al.*, “ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects,” in *Conference ERTS’06*, Toulouse, France, Jan. 2006. Accessed: Apr. 19, 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02270466

[59] B. Bittner *et al.*, “The xSAP Safety Analysis Platform,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2016, vol. 9636, pp. 533–539. doi: 10.1007/978-3-662-49674-9_31.

[60] M. Bozzano and C. Jochim, “The FSAP/NuSMV-SA Safety Analysis Platform,” *Softw. Tools Technol. Transf.*, vol. 9, p. 2007.

[61] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, “NUSMV: a new symbolic model checker,” *Int. J. Softw. Tools Technol. Transf. STTT*, vol. 2, no. 4, pp. 410–425, Mar. 2000, doi: 10.1007/s100090050046.

[62] M. Bozzano and A. Villafiorita, “The FSAP/NuSMV-SA Safety Analysis Platform,” *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 1, pp. 5–24, Feb. 2007, doi: 10.1007/s10009-006-0001-2.

[63] F. Ortmeier, M. Güdemann, M. Lipaczewski, and S. Struck, “Unifying Probabilistic and Traditional Formal Model Based Analysis,” in *MBEES*, 2012.

[64] M. Lipaczewski, S. Struck, and F. Ortmeier, “Using Tool-Supported Model Based Safety Analysis – Progress and Experiences in SAML Development,” in *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*, Oct. 2012, pp. 159–166. doi: 10.1109/HASE.2012.34.

[65] P. Cuenot *et al.*, “The EAST-ADL architecture description language for automotive embedded software,” in *Proceedings of the 2007 International Dagstuhl conference on Model-based engineering of embedded real-time systems*, Dagstuhl Castle, Germany, Nov. 2007, pp. 297–307.

[66] D. Chen *et al.*, “Integrated safety and architecture modeling for automotive embedded systems*,” *E Elektrotechnik Informationstechnik*, vol. 128, no. 6, pp. 196–202, Jun. 2011, doi: 10.1007/s00502-011-0007-7.

[67] S. Diampovesa, A. Hubert, and P.-A. Yvars, “Designing physical systems through a model-based synthesis approach. Example of a Li-ion battery for electrical vehicles,” *Comput. Ind.*, vol. 129, p. 103440, Aug. 2021, doi: 10.1016/j.compind.2021.103440.

[68] P. Cuenot, C. Ainhauser, N. Adler, S. Otten, and F. Meurville, “Applying Model Based Techniques for Early Safety Evaluation of an Automotive Architecture in Compliance with the ISO 26262 Standard,” in *Embedded Real Time Software and Systems (ERTS2014)*, Toulouse, France, Feb. 2014. Accessed: Nov. 13, 2019. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02271308

[69] “EAST-ADL Association.” https://www.east-adl.info/ (accessed May 25, 2020).

[70] M. Fernández and M. Michela, “Model-based systems engineering with the Architecture Analysis and Design Language (AADL) applied to NASA mission operations,” May 2014, Accessed: Nov. 18, 2019. [Online]. Available: https://trs.jpl.nasa.gov/handle/2014/45524

Reference

[71] A. Baouya, O. Ait Mohamed, D. Bennouar, and S. Ouchani, “Safety analysis of train control system based on model-driven design methodology,” *Comput. Ind.*, vol. 105, pp. 1–16, Feb. 2019, doi: 10.1016/j.compind.2018.10.007.

[72] B. Larson, J. Hatcliff, K. Fowler, and J. Delange, “Illustrating the AADL error modeling annex (v.2) using a simple safety-critical medical device,” in *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology - HILT ’13*, Pittsburgh, Pennsylvania, USA, 2013, pp. 65–84. doi: 10.1145/2527269.2527271.

[73] H. Thiagarajan, B. Larson, J. Hatcliff, and Y. Zhang, “Model-Based Risk Analysis for an Open-Source PCA Pump Using AADL Error Modeling,” in *Model-Based Safety and Assessment*, Cham, 2020, pp. 34–50. doi: 10.1007/978-3-030-58920-2_3.

[74] J. Hudak and P. Feiler, “Developing AADL Models for Control Systems: A Practitioner’s Guide:,” Defense Technical Information Center, Fort Belvoir, VA, Jul. 2007. doi: 10.21236/ADA472931.

[75] J. Delange and P. Feiler, “Architecture Fault Modeling with the AADL Error-Model Annex,” in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, Aug. 2014, pp. 361–368. doi: 10.1109/SEAA.2014.20.

[76] J. Delange, P. Feiler, D. P. Gluch, and J. Hudak, “AADL Fault Modeling and Analysis Within an ARP4761 Safety Assessment:,” Defense Technical Information Center, Fort Belvoir, VA, Oct. 2014. doi: 10.21236/ADA610294; http://web.archive.org/web/20200426050610/https://apps.dtic.mil/docs/citations/ADA610294.

[77] J. Delange and P. Feiler, “Supporting the ARP4761 Safety Assessment Process with AADL,” p. 11.

[78] “AADL and OSATE: A Tool Kit to Support Model-Based Engineering.”

[79] “Welcome to OSATE — OSATE 2.7.1 documentation.” https://osate.org/ (accessed May 27, 2020).

[80] P. Fenelon and J. A. McDermid, “An integrated tool set for software safety analysis,” *J. Syst. Softw.*, vol. 21, no. 3, pp. 279–290, Jun. 1993, doi: 10.1016/0164-1212(93)90029-W.

[81] P. Fenelon, T. Kelly, and J. A. McDermid, “Safety Cases for Software Application Reuse,” Apr. 2003, doi: 10.1007/978-1-4471-3054-3_29.

[82] Y. Papadopoulos and J. A. McDermid, “Hierarchically Performed Hazard Origin and Propagation Studies,” in *Computer Safety, Reliability and Security*, Sep. 1999, pp. 139–152. doi: 10.1007/3-540-48249-0_13.

[83] S. P. Wilson, “The Safety Argument Manager: An Integrated Approach to the Engineering and Safety Assessment of Computer Based Systems.,” pp. 198–205, 1996, doi: 10.1109/ECBS.1996.494529.

[84] N. Leveson, *STPA Handbook*.

[85] N. Leveson, C. Wilkinson, C. Fleming, J. Thomas, and I. Tracy, “A Comparison of STPA and the ARP 4761 Safety Assessment Process,” p. 72.

[86] D. Arterburn, B. R. Abrecht, D. C. Horney, J. Schneider, B. R. Abel, and N. G. Leveson, “A New Approach to Hazard Analysis for Rotorcraft,” 2016.

[87] H. S. Mahajan, T. Bradley, and S. Pasricha, “Application of systems theoretic process analysis to a lane keeping assist system,” *Reliab. Eng. Syst. Saf.*, vol. 167, pp. 177–183, Nov. 2017, doi: 10.1016/j.ress.2017.05.037.

[88] D. Suo, S. Yako, M. Boesch, and K. Post, “Integrating STPA into ISO 26262 Process for Requirement Development,” presented at the WCX™ 17: SAE World Congress Experience, Mar. 2017, pp. 2017-01–0058. doi: 10.4271/2017-01-0058.

Reference

[89] S. Placke, J. Thomas, and D. Suo, “Integration of Multiple Active Safety Systems using STPA,” presented at the SAE 2015 World Congress & Exhibition, Apr. 2015, pp. 2015-01–0277. doi: 10.4271/2015-01-0277.

[90] A. Joshi, S. P. Miller, M. Whalen, and M. P. E. Heimdahl, “A proposal for model-based safety analysis,” in *24th Digital Avionics Systems Conference*, Oct. 2005, vol. 2, p. 13 pp. Vol. 2-. doi: 10.1109/DASC.2005.1563469.

[91] O. Lisagor, “Failure logic modelling: a pragmatic approach,” phd, University of York, 2010. Accessed: Nov. 13, 2019. [Online]. Available: http://etheses.whiterose.ac.uk/1044/

[92] Y. Papadopoulos and M. Maruhn, “Model-based synthesis of fault trees from Matlab-Simulink models,” in *2001 International Conference on Dependable Systems and Networks*, Jul. 2001, pp. 77–82. doi: 10.1109/DSN.2001.941393.

[93] J. Delange and P. Feiler, “Architecture Fault Modeling with the AADL Error-Model Annex,” in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, Aug. 2014, pp. 361–368. doi: 10.1109/SEAA.2014.20.

[94] C. Kehren *et al.*, “Architecture Patterns for Safe Design,” *AAAF 1ST COMPLEX SAFE Syst. Eng. Conf. CS2E 2004 21-22 JUIN 2004*, [Online]. Available: http://130.203.136.95/viewdoc/summary;jsessionid=AF8F5506E5BFE636FDEC5DACA3E7DD02?doi=10.1.1.77.488

[95] Y. Sirgabsou, L. Pahun, C. Baron, L. Grenier, C. Bonnard, and P. Esteban, “Investigating the use of a model-based approach to assess automotive embedded software safety,” p. 9, 2020.

Publications

[1] Y. Sirgabsou, C. Baron, C. Bonnard, L. Pahun, L. Grenier, and P. Esteban, “Investigating the use of a model-based approach to assess automotive embedded software safety,” presented at the 13th International Conference on Modeling, Optimization and Simulation (MOSIM20), Nov. 2020. [Online] Available: https://hal.laas.fr/hal-02942695

[2] Y. Sirgabsou, C. Baron, L. Grenier, L. Pahun, and P. Esteban, “L’ingénierie dirigée par les modèles pour assurer la sécurité des logiciels embarqués en automobile,” Grenoble, France, May 2021. Accessed: Sep. 29, 2021. [Online] Available: https://hal.laas.fr/hal-03232108

[3] Y. Sirgabsou, C. Baron, L. Pahun, and P. Esteban, “Software fault propagation patterns for model-based safety assessment in autonomous cars,” presented at the ERTS 2022, Jun. 2022. Accessed: Jun. 23, 2022. [Online] Available: https://hal.laas.fr/hal-03699226

[4] Y. Sirgabsou, C. Baron, L. Pahun, and P. Esteban, “Model-driven engineering to ensure automotive embedded software safety. Methodological proposal and case study,” *Comput. Ind.*, vol. 138, p. 103636, 2022. doi: https://doi.org/10.1016/j.compind.2022.103636.