

Model-driven engineering to ensure automotive embedded software safety. Methodological proposal and case study

Yandika Sirgabsou, Claude Baron, Laurent Pahun, Philippe Esteban

▶ **To cite this version:**

Yandika Sirgabsou, Claude Baron, Laurent Pahun, Philippe Esteban. Model-driven engineering to ensure automotive embedded software safety. Methodological proposal and case study. Computers in Industry, 2022, 2022-02, 138, pp.103636. 10.1016/j.compind.2022.103636 . hal-03590899

HAL Id: hal-03590899

https://laas.hal.science/hal-03590899

Submitted on 10 Mar 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Model-driven engineering to ensure automotive embedded software safety. Methodological proposal and case study.

Yandika Sirgabsou[abce] , Claude Baron[bce*], Laurent Pahun[a], Philippe Esteban[bde]

[a] *Renault Software Factory, Toulouse, France*
[b] *LAAS-CNRS, Toulouse, France*
[c] *INSA Toulouse, Toulouse, France*
[d] *Université de Toulouse III, Toulouse, France*
[e] *Université de Toulouse, Toulouse, France*

Abstract

The development of driver assistance and autonomous driving systems for vehicles has started to revolutionize the transportation sector, offering comfort and safety. While significant technological progress has already been made in this area, the road ahead is littered with many challenges. Among these challenges, ensuring driver safety has become even more critical due to the increasing use of complex, communicating and reconfigurable embedded software. Current approaches to document-based safety analysis have reached their limit and the time has come to rethink them. To this end, we propose to rely on model-driven engineering to conduct safety analyses. This paper makes a methodological proposal that improves current practices in terms of time, analysis quality and reusability, and that has been validated on the study of an automotive software component.

Keywords: ISO 26262, Embedded Software, MBSE, MBSA, Safety

1. Introduction

With the advent of autonomous vehicles, the automotive industry is increasingly relying on embedded software to enable various features. This technology has profound implications for safety, i.e., the ability of the system to prevent failures that could lead to injuries and damages [1], and which must therefore be rigorously ensured. The ability to provide safety guarantees will be key to the acceptance of autonomous vehicles by society. The ISO 26262 standard (ISO 26262-1,2018), which governs functional safety for road vehicles, already requires that safety analyses be performed and proven; however, the current complexity of embedded systems in vehicles means that guaranteeing this safety is extremely difficult. 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. Moreover, the complexity of these architectures currently requires highly collaborative work between several teams comprising specialists from different business sectors, who need to use efficient methods and means to work together. 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.

Faced with the 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, it is now necessary 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 [3] [4] [5]. 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. In this paper, we propose to use MDE to define 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.

Following a state of the art of current industrial practices in software and safety engineering (Section 2), the paper explores MDE-based methods for system and software design, as well as for safety assessment (Section 3). Section 4 introduces a methodological proposal to help safety experts build, from engineering artifacts, so-called dysfunctional models from which to conduct safety analyses. To illustrate the proposal, Section 5 applies the step-by-step methodology on a case study in the automotive domain, i.e., an embedded application of longitudinal control that helps to manage vehicle speed according to traffic signs and conditions. Section 6 discusses the results obtained. Lastly, the paper outlines some interesting avenues for further development of the methodology and tools given the observed limitations, as well as alternative tooling options.

* Corresponding author. Tel.: +33 06-80-31-03-77
E-mail address: claude.baron@laas.fr

2. Context and problematic: a necessary evolution of industrial practices in software and safety engineering

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) [6], which defines the use and evaluation of engineering processes, and ISO 26262 (ISO 26262-1,2018), which addresses safety aspects in system, hardware and software development.

This section takes stock of current practices in MDE-related software engineering and safety assessment within the framework of the Renault Group. Then, it identifies what improvements must be implemented to make safety assessment practices compatible with the model-based approach.

2.1. Industrial practices in automotive software engineering: The example of Renault

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. Renault has defined a business process to implement these procedures and activities called Alliance Software Process (ASWP), shown in Figure 1.

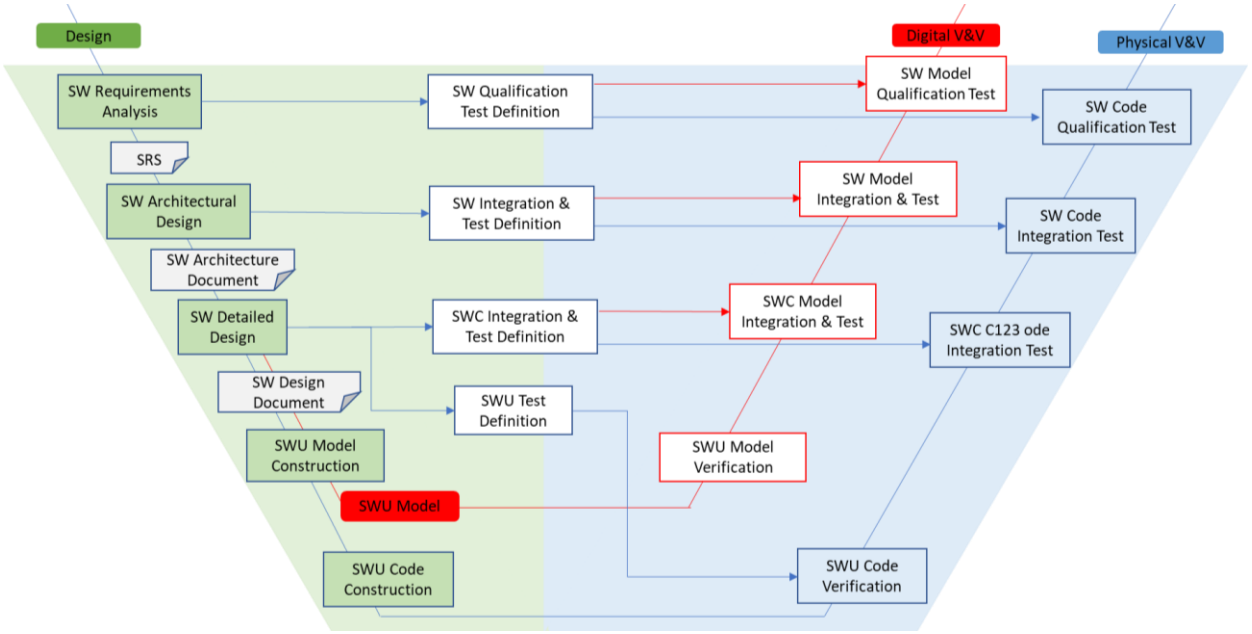


Figure 1. Software engineering process and safety assessment in the automotive industry

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 1) and testing and integration on the Verification and Validation (V&V) phase (on the right in Figure 1).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 1) 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 (dotted 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 1. 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.

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.

2.2. Industrial practices in safety analysis in the automotive industry: The example of Renault

In the automotive industry, safety assessment is an integral part of the 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)† [5] .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 [8]. 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, analyses can also be conducted using the FMECA‡ method (Failure Modes, Effects and Criticality Analysis) [9], which can analyze the impact of a particular failure on the entire system. A FMECA 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.

In addition to these well-known classic methods, ISO 26262 also recommends other types of safety analyses, including Dependent Failure Analysis (DFA) [8. Part 6. Annex E (informative). Application of safety analyses and analyses of dependent failures at the software architectural level] and ASIL-oriented analyses [8, Part 9:Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses]. However, the standard emphasizes that the latter can also be performed on the basis of fault trees [8, 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.

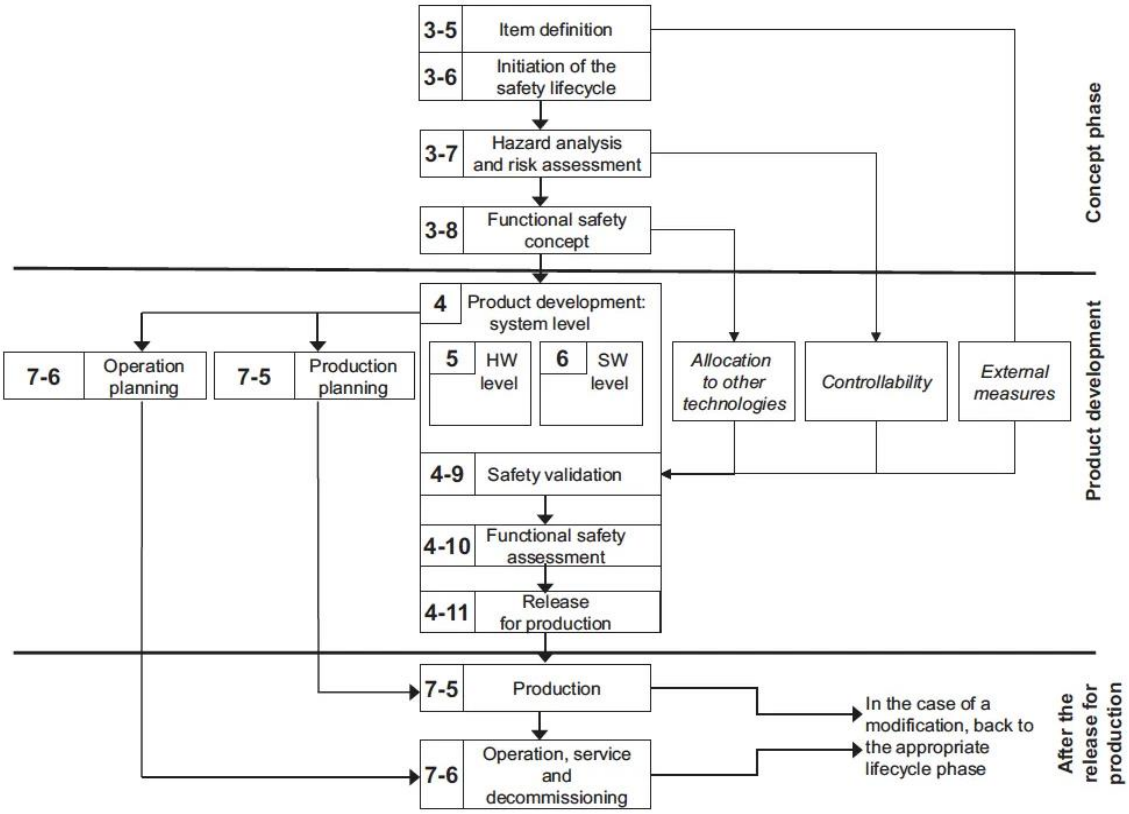


Figure 2. Safety assessment process according to ISO 26262 (ISO, 2018)

† The Fault Tree Analysis is a deductive (top-down) safety analysis technique used for determining the combinations of possible causes (basic events) that can lead to a feared event (top event) [7].

‡ The FMECA is a systematic, bottom-up safety analysis technique that used for determining the effect of individual component failures on the whole system [7] .

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 2. 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 2), 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 parts 5 and 7 because they are outside the scope of our study.

2.2.1. 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 2), 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 2. 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 2), 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.

2.2.2. Development phase: System and software

In the system level development phase (as indicated in the “Scope of Part 4” in Figure 3), 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 to conduct safety analyses not only at the system level but also at the software level [10]. 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 3 corresponds to the activities indicated by 6 in Figure 2, providing 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 (Figure 3), 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 3) 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 3); 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 FMECA 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. For example, the minimal cuts from fault trees whose purpose is to calculate probabilities are less suitable for the software perimeter, whereas the more specific analyses recommended by ISO 26262 (such as DFA and ASIL-oriented analyses) can still be conducted at this level using fault trees.

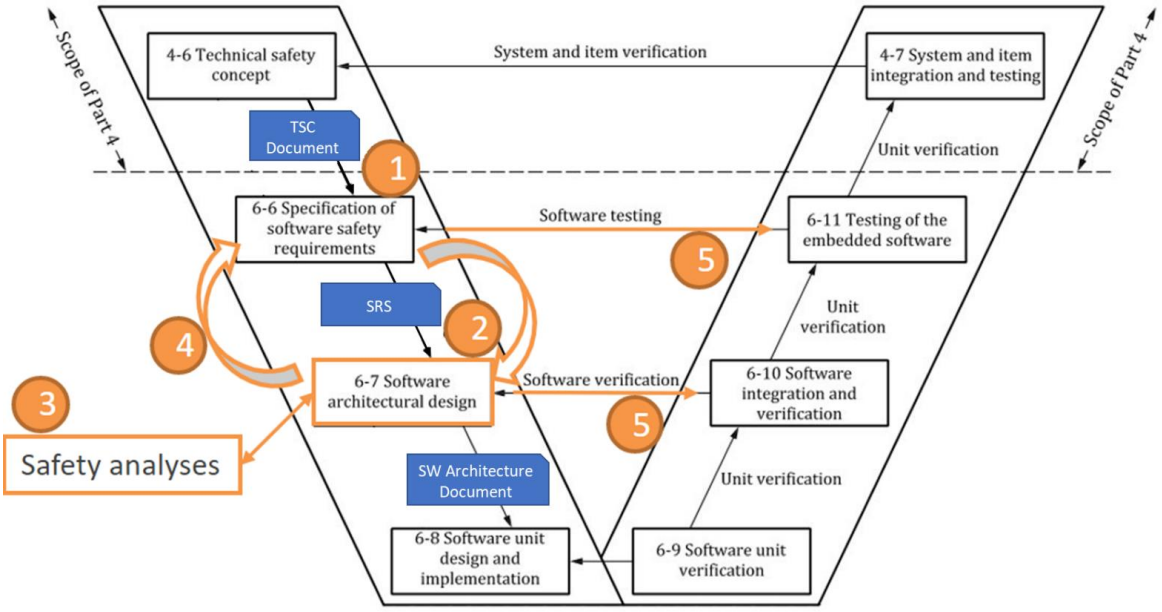


Figure 3. Safety assessment of the software

2.2.3. Required changes in current industrial practices

In view of the aforementioned industrial practices, it appears that significant efforts are being made by companies to develop the use of MDE in the system design and safety assessment processes. In the context of automotive software engineering, this is notably the case in the lower part of the development cycle, where code generation based on the detailed software architecture model has become a common practice. 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 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.

A 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. These approaches promote communication between experts and teams, and improve capitalization through reuse.

The critical embedded systems industry—including the automotive industry—is not immune to this trend, as it is also considering with great interest these model-based approaches. It will soon be necessary to be able to meet the certification requirements for autonomous vehicles; this must be anticipated now. Moreover, the ISO 26262 standard gives designers the freedom to use the MDE methods or not (ISO, 2018, Part 6, Annex E). On this basis, we can thus consider using an MDE approach for the safety assessment of automotive software.

Based on this review, the current state of practice shows that model-based artefacts are used at later stages of the development process. However, early stages such as requirement analysis, software architectural design and the safety concept, and thus all related safety analyses, still rely on document centric artifacts. Safety analyses conducted in such circumstances are prone to error because they are manually performed and subject to the interpretation of the document artefacts by the analyst. Furthermore, in the context of complex systems, manual analysis can be inefficient (time consuming, lack of reuse). Therefore, given this current state of practice in safety analysis dominated by document-oriented methods, a general question one can ask is how the current practices can be improved by taking advantage of a model-based approach. To answer this question, the next section looks at the current MBSA methodologies in order to determine which methods, languages or modelling approach might be appropriate if we were to apply MBSA to software in the automotive context.

3. Literature review on model-based approaches

Regardless of the sector considered and for some time now, a general trend has been observed in the evolution of industrial practices towards model-based approaches, particularly in systems engineering. MBSE (Model-Based Systems Engineering) is a systems engineering practice aimed at describing both a problem (need) and its solution through models, concepts and languages [11]; this practice is also called MBSSE (Model-Based System and Software Engineering). The adoption of model-based approaches to conduct safety analyses has given rise to a number of practices grouped under the acronym MBSA (Model-Based Safety [and] Assessment). MBSA practices enable the capture (through specific formalisms and languages) of an "authoritative system model", on the basis of which different safety analyses can be made [10] [13].

3.1. Model-Based Systems and Software Engineering

Prior to their adoption in systems engineering, model-based approaches appeared in software engineering [14]. They ensure a certain continuity between the different stages of system and software development (modeling of requirements and of the logical and physical architecture up to implementation), by providing traceability during the transitions between models.

Model-based approaches often rely on certain high-level languages and tools. For example, the UML (Unified Modeling Language [15]) and SysML (Systems Modeling Language [16]) languages are commonly used to model functional and organic architectures, as well as the behavior of software and systems. Structural diagrams (class and package diagrams) are used to model the organic architecture, while use case, sequence and activity diagrams are used to model behavioral scenarios. Architecture description languages that specifically support the design stage also exist; one example is the Architecture Analysis and Description Language (AADL) [17], a standard of the Society of Automotive Engineers (SAE) initially designed for avionics, used for designing and analyzing the architecture of embedded systems including space [18], train control [19], medical devices [20] [21] or automotive [22]. Some architecture description languages are domain-specific. Such is the case of EAST-ADL [23] in the automotive domain, a meta-model used for modeling the environment of the system and the system itself at five different levels of detail. EAST-ADL also offers extensions that can model the properties of the system at each level [24] [25] [26].

Semantics closer to coding languages—such as those of the Simulink [17] and SCADE [18] models—are used in design for rapidly prototyping detailed software architectures and for code generation. SCADE is widely used in avionics to model synchronous systems such as flight control systems with real-time constraints, thus foreseeing certain safety issues specific to competing systems; it also makes test automation possible. In other sectors, such as the automotive industry, Simulink models are increasingly being used [29].

Through this brief review of MDE from Systems to Software Engineering, we observe that in practice, various and often heterogeneous domain models are used depending on the point of view of interest. As an example, UML diagrams are often used to model software architectures at higher level of abstraction, whereas more precise semantics that can generate executable code such as Simulink, SCADE or AADL are preferred for detailed design and software unit construction. While a model-based design offers several advantages, such as better communication, efficiency, and reusability, ensuring continuity between these different types of models (e.g., through techniques such as model transformations) within the development process remains a difficulty. Likewise, the global consistency (keeping the different domain models of a system coherent with each other as the system evolves through the development process) remains challenging. While the advocacy for the adoption of a single model approach as a solution to this problem is prevalent in the literature, various significant counter arguments make a case for keeping different domain models separated. A resulting difficulty—ever more important in the current context of increasingly autonomous critical systems—consists in knowing how to integrate these design models with specific analysis models such as safety analyses.

3.2. Model-Based Safety Assessment

As with MBSE, MBSA results from the trend in engineering to abandon document-based approaches in favor of model-based ones, with several factors justifying its adoption. The modeling formalisms most commonly used to conduct safety analyses (such as fault trees, Petri nets or Markov chains) either lack expressiveness or are semantically too distant from the associated system specification models; these models are therefore difficult to develop, to share with stakeholders, and to maintain throughout the system life cycle. To mitigate this, Domain-Specific Languages are built to enable a more comprehensive and efficient modeling of systems safety related behavior.

MBSA can be used to model systems in a more structured manner by means of precise mathematical semantics, thus reducing the gap between system specifications and safety models [30]. MBSA replaces the traditional models on which safety analyses are based, such as fault trees [5], whose main objective is to determine the minimal cuts [8], and FMECAs [9]. Furthermore, MBSA seeks to unify the classic models of safety analysis, considered in a broad sense (including reliability, maintainability and availability), into a single dysfunctional model [13].

Several MBSA approaches exist. Some of them, such as the Failure Propagation and Transformation Notation (FPTN) [21] or the Hierarchically Performed Hazard Origins and Propagation Studies (HiP-HOPS) [22] are methodological; others, such as FSAP/NuSVM [23] or Safety Architect [34], are tools; yet others, such as Figaro (implemented in EDF's KB3 tool [35]), SAML (Safety Analysis Modeling Language) [26] or AltaRica[37] are languages. In addition to these native MBSA approaches, some of the MBSE approaches mentioned earlier in 3.1 extend their semantics to support safety analysis through annexes or safety related packages. .Without going into the details, these approaches are mainly distinguished by the manner the dysfunctional model is built: it can be extended from the functional model (called nominal) resulting from the design, or it can be distinct [13]. In the case of an extended model, the system model will be improved with failure information, while a dedicated model will be built manually

by the safety expert. Another distinction criterion that can be used to characterize the current MBSA practice is the type of information the dysfunctional model carries. In [13], Lisagor explained that the dependency between the components of an MBSA model is defined based on the type of flows that is exchanged. This leads to distinguishing between Failure Logic Modeling (FLM) (that uses abstract flows) and Failure Effect Modeling (FEM) that uses real flows. In the case of FLM, often associated with the dedicated model approach described earlier, the dependencies between the components are captured in terms of deviations of their behaviour from design intent and failure modes exhibited by other components of the system. In the case of FEM (more suited to the extended model approach), the model components carry the functional behaviour of the system and the dependency between the components are captured through real flow of data, matter, or energy.

Each approach offers advantages and disadvantages. 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 modelling 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. 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 approaches mentioned earlier, AltaRica [37] is of particular interest to us. Currently a pioneering approach in the European critical systems ecosystem, AltaRica is a high-level language designed to perform qualitative and quantitative system-level safety analysis. 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 (see Figure 4), while the transitions describe changes in state. A node interacts with the environment in two ways: 1) through events and 2) through flow variables. Just like a transfer function, assertions describe the relationships between input flows, state variables and output flows.

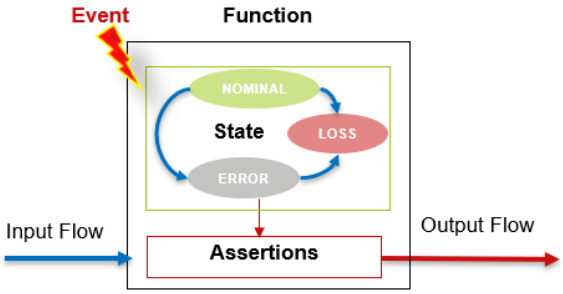


Figure 4. An AltaRica node

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, justifies choosing AltaRica for our approach.

One common characteristic the above-mentioned approaches share is that they were all created with system level application in mind. This way, the objective behind AltaRica was to create a language that is close to the way that systems elements are described [37]. The same applies to EAST-ADL [23] in the automotive domain where a meta model of the vehicle serves as a blueprint for modeling all relevant aspects of a vehicle and its systems component including safety properties. In EAST-ADL, modeling starts at system level while software level safety considerations are simply declined to AUTOSAR layer of the model. Nevertheless, although it is widely acknowledged in critical systems that safety is a systems issues, the ISO 26262 requires that specific safety analyses be conducted at software level.

One can make an argument that more embedded systems domain specific approaches such as FPTN or AADL address software level safety considerations more precisely. Unfortunately, both approaches have limitation that may make them not to be an ideal choice. For FPTN, as of today, no formal description of the semantics nor up-to-date tool implementation of the approach can be found in the literature. As far as AADL is concerned, the approach is well suited for real time embedded systems independently of the domain but relies on model extension for safety analysis, which might not be the preferred approach for all systems safety practitioners. In fact, adopting an extended model approach only makes sense if systems design and safety assessment adopt the same description language (for instance both system and safety models are described in AADL instead of a situation where the system model is in UML while the safety model is in AADL).

Based on this review, it appears that the most prominent and commonly used MBSA approaches, such as HIP HOPS, AltaRica, or EAST-ADL, are all systems oriented. Domain specific approaches such as FPTN and AADL exist, but they either lack supporting tool or their application requires a certain continuity in methodological choice for systems design and safety assessment. The question this paper addresses is how these approaches can be applied to embedded software level where safety analyses are required by the applicable standards. Indeed, applying MBSA to software would require creating software models that can support dysfunctional (failure related) modeling. Given the different possibilities in terms of modeling paradigm, several sub questions can be asked. First, how can such models be created and used? Secondly, what information should these models represent to allow automatic generation of meaningful safety artifacts? Lastly, what gain would justify the adoption of such an approach?

4. Methodological proposal

We aim to provide a methodology that adapts the concepts, principles and methods of MBSA for the purpose of improving the practice of software safety analysis.

Among the possible approaches, in this paper we have chosen to explore a dedicated dysfunctional model approach based on a failure propagation logic modeling method, using the AltaRica language. Other approaches such as the AADL language [15] and its error annex (EMV2), or EAST-ADL [23], can also be considered as alternatives.

This methodology extends and adapts a first proposal, described in [38], which enabled building a dysfunctional model of the software architecture and using it as the basis to conduct safety analyses in three steps: dysfunctional modeling, logic translation and safety analysis. This new proposal introduces a preliminary step of input data selection that specifically addresses the difficulty of obtaining good inputs for the construction of the model, essential to avoid resorting to logic translation (deemed complex after experimenting with the first proposal). By introducing the preliminary step, the new proposal simplifies and reorganizes the main steps of the methodology. As illustrated in Figure 5, the methodology therefore consists in three new steps: 1) the selection of input data from which the dysfunctional model will be built, 2) the construction of the dysfunctional model of the software, from which 3) the minimal cuts, fault trees and FMECAs can be generated automatically.

Our proposal is based on the use of a dedicated model and failure propagation modeling techniques by means of the applicable languages and semantics, as described in section 3.2.

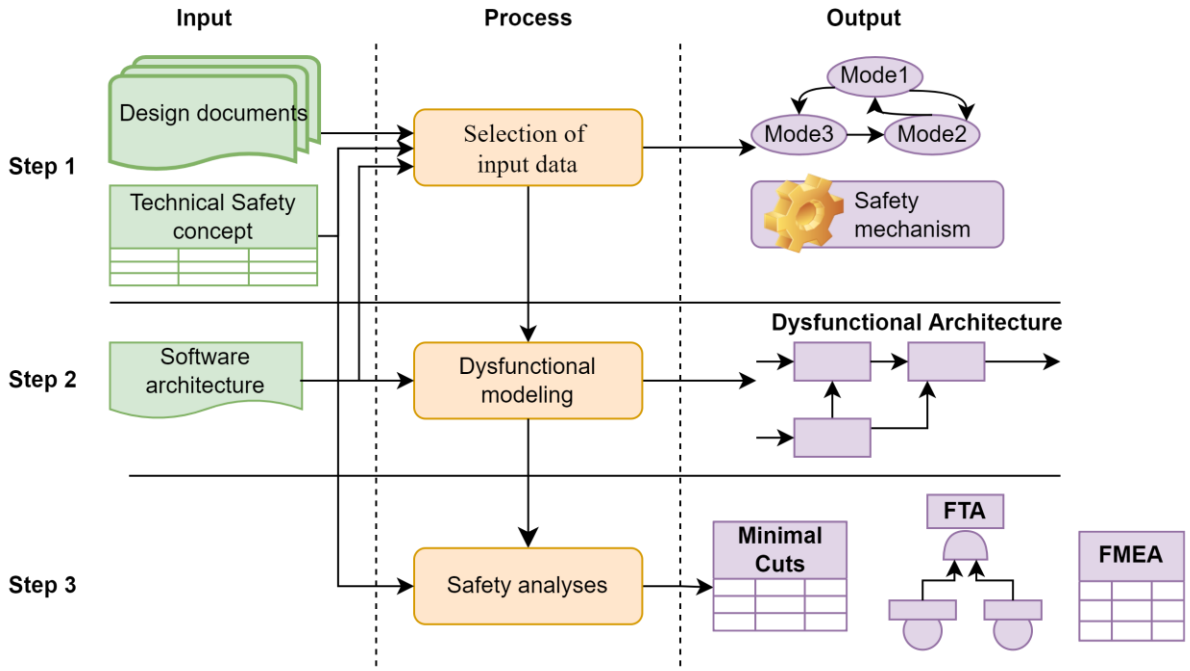


Figure 5. The three main steps of our method

In our method we chose to use the dedicated model approach because we adhere to the principle that there should be an independence between System design and safety analysis. We also believe that safety models that are systematically derived from design models do not necessarily yield meaningful safety analysis results. Furthermore, the automatic deriving off 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 order to be more pedagogical, an overview of the methodological approach is first illustrated through Figure 5 then more detailed descriptions of the steps are given in the following subsections.

4.1. Step 1: Selection of input data

As mentioned earlier, one of the difficulties of using a dedicated model for safety is to ensure its consistency with the system design model. In principle, in the context of a robust and well-established MBSE approach, an executable design model exists and should be readily available to the architect for a functional evaluation of the system architecture under development [12]. This model can then be transferred to the safety expert, who can either extend it or use it to build a dedicated model according to the principle described in section 3.2.

However, in the context of current practices (as described in section 2) it is not possible to apply this principle ‘as is’, since design models are often non-existent or poorly structured. Building the dysfunctional model on the basis of available information thus becomes necessary, whether contained in models or documents.

Having opted for an approach based on a dedicated model, we must first identify the information necessary for its construction, starting with the software architectural design documents and the TSC resulting from the system level safety assessments. As introduced in section 2.2.2, the Technical Safety Requirements resulting from the requirement analysis performed at the system level and contained in the TSC are translated into Software Safety Requirements (SRS) in part 6.5 of the development phase at the

software level. Likewise, system-level feared events (resulting from the violation of safety goals) are declined into Unwanted Software Events (USWE) at software level. The software architectural design implementing these requirements represents the software architectural elements and their interactions in a hierarchical structure. Different aspects are also described, including static (e.g., the interface between software components) and dynamic aspects (e.g., process sequences and synchronization behaviour) [10]. 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. Safety measures are activities or technical solutions performed during development aimed at preventing the occurrence of a particular failure (preventive action); 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 and reconfiguration in order to maintain a critical functionality or to bring the system to a safe state. Thus, based on the TSC and the definition of the items, we can identify the components and interfaces to model, as well as the requirements and safety mechanisms to evaluate in the context of the dysfunctional architecture. In this way, we can represent in the dysfunctional architecture only those components that impact the safety goals, which are high-level safety requirements resulting from the preliminary risk analysis at the vehicle level (see ISO 26262-1 3.139). This will also avoid overloading the dysfunctional model with elements unnecessary for safety.

In the architecture documents, we find the details of the implementation proposed to meet the safety requirements contained in the safety concept. Using this information, we can identify the operating modes (nominal or faulty states) of the system and the state change conditions of its components, thus obtaining an understanding of its operation. The approach followed in this first step is illustrated in Figure 6.

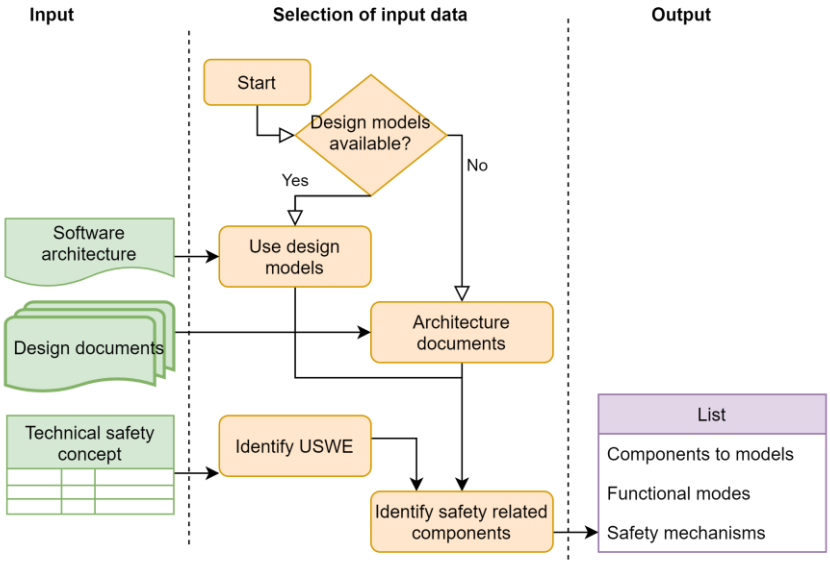


Figure 6. Step 1: Selection of Input Data

Through this step, our aim is 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 section 2. 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 6, 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 document 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 if they exist. For example, Simulink functional models can be used to understand the functional logic and to determine the manner in which safety mechanisms are implemented.

4.2. Step 2: Dysfunctional modeling

Dysfunctional modeling consists in modeling the dysfunctional behavior (in the presence of failures) of software components and their interactions. As illustrated in Figure 7, it takes place in 2 sub steps: 1) behavior modeling of the components using state machines, and 2) failure propagation modeling. We will describe these in the next two subsections respectively.

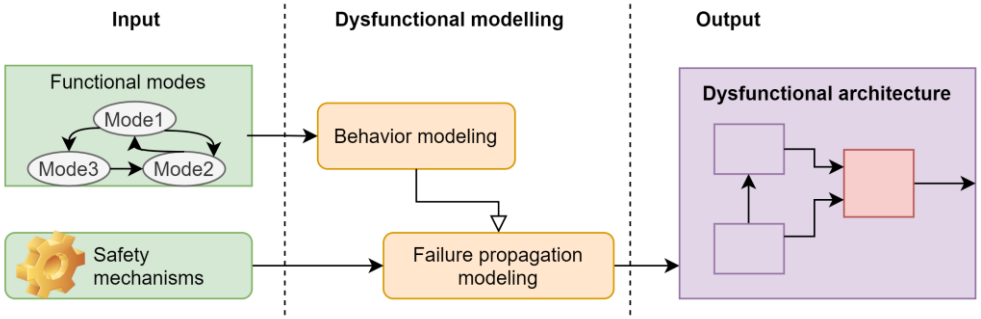


Figure 7. Step 2: Dysfunctional Modeling

4.2.1. Modeling the behavior of the components

Behavior modeling consists in modeling the internal dysfunctional behavior of the system components using the data collected in step 1 (operating modes, safety mechanisms), focusing on the components and signals related to safety. We model the internal behavior of the components using state machines (states and transitions). The states 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.

4.2.2. Interface modeling

Interface modeling complements 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).

First, we 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, 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: ok, erroneous or failed. In the *ok* state, the component is operational and produces good results; 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, we define a flow variable that can also take three values: *nominal* (good data), *erroneous* (potentially bad data), or *lost* (no data).

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

if (state = ok) then
    if ((input1 = nominal) and (input2 = nominal)) then
        output: = nominal
    Else if ((input1 = lost) or (input = lost)) then
        output: = lost
    else
        output: = erroneous
    end if
else //state not ok
    output: = lost
end if

To write these expressions, let us note that failure propagation logics (abstract failure flows) are used instead of the usual nominal flows found in dataflow architectures. At present, there are formalisms that support such an approach, such as AltaRica or the AADL error annex.

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.

4.3. Step 3: Safety Analysis

The objective of step 3 is to extract the 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 8 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.

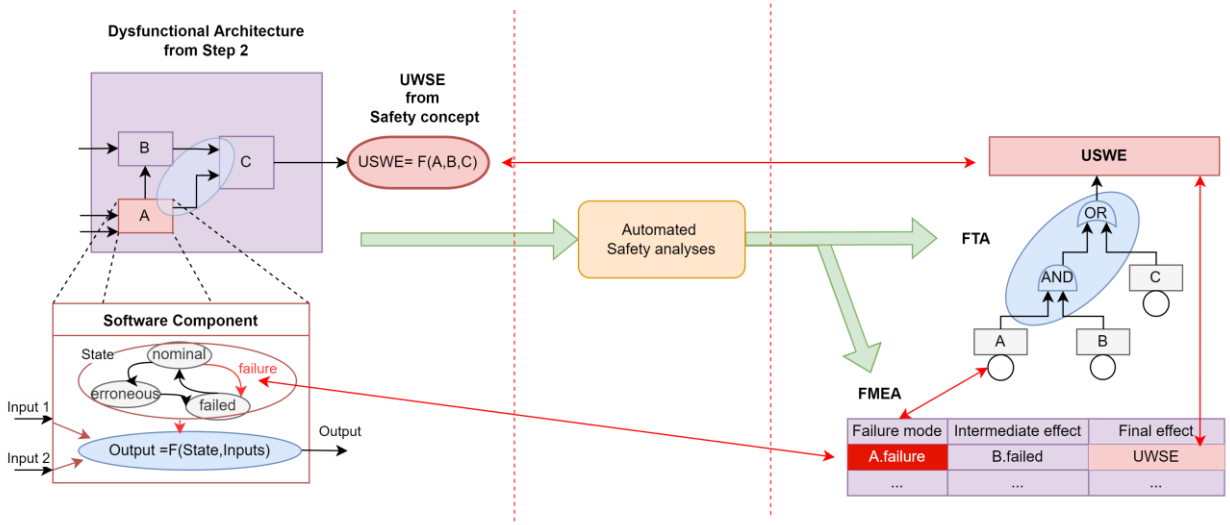


Figure 8. Step 3: FMEA and FTA deduction from the dysfunctional architecture

4.4. Discussion

Using this approach, it is therefore possible to unify 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, the safety analyst must, for instance for each UWSE, 1) manually construct a fault tree; and 2) update the system's FMEA. 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.

Our 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.

Admittedly, it can be argued that this methodology requires a certain modeling effort due to our preference for a manual modeling approach. However, thanks to the appropriate selection of inputs in the first step of the methodology, the modeling effort is minimized. Furthermore, a manual modeling 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 manual modeling approach fits the current context, there are envisioned avenues for better integration of all these efforts into a more seamless MDE approach.

5. Case study

We adapted the method described above to the SimfiaNeo software and applied it to a case study—the analysis of a software component that ensures the longitudinal control of a vehicle. It is complementary to the study on lateral control described in [38]. The objective was to evaluate the software architecture through a dysfunctional model, to verify whether it satisfied a certain number of safety requirements. At the same time, we evaluated the performance of the SimfiaNeo tool. The scope of the new study included a safety concept incorporating informal models that explain the proposed safety mechanisms, allowing us to simplify the modeling process of the dysfunctional architecture. Using our proposed methodology, we modeled a dedicated dysfunctional architecture of the longitudinal control software and used it as basis for safety analyses. In this case study, we focused on a single

USWE related the acceleration target limit exceedance by the longitudinal control. The modeling, simulation and safety analyses were conducted using a regular business laptop equipped with an Intel i5-63000 dual processor clocking at 2.40GHz, a 8 Gigabytes memory and running under a 64 bits Windows 10.

5.1. Presentation of SimfiaNeo

SimfiaNeo is 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 [39] (developed by Dassault Aviation) or AltaRica Studio [40] (developed by the Laboratoire Bordelais de Recherche en Informatique, LaBRI), both of which are based on the dataflow version of AltaRica. More recently, we can also find Open AltaRica [41] from SystemX, based on the more recent 3.0 version of the language.

5.2. Presentation of the system

Longitudinal control is a function of the ADAS (Advanced Driver Assistance Systems) technology. In concrete terms, it is a software component whose purpose is to ensure speed and braking control in autonomous driving mode. It is built around the ACC (Adaptive Cruise Control), a speed and distance control system 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 ensure its safe activation, the longitudinal control relies on an internal supervisor (a subcomponent that manages its states) and a failsafe controller (a subcomponent that places the longitudinal control in some predefined safe states when certain faults occur). A number of safety requirements are associated with the longitudinal control. Among them are those related to USWEs, such as untimely braking and uncontrolled acceleration from the ACC. Other constraints also exist (e.g., the user must be able to deactivate the ACC at all times).

We modeled the system with these requirements in mind, while paying special attention to the safety mechanisms proposed in the software architecture for their implementation (acceleration limits, ACC activation conditions). In order to model the ACC, it was necessary to consider not only the internal elements but also the other elements with which the ACC interacts. This enabled delimiting the perimeter of the system and favoring a better interpretation of the interfaces. In the following section, we will identify the components related to the longitudinal controller, starting from the TSC and the general documentation of the ADAS.

5.3. Step by Step Application of the methodology

5.3.1. Selection of input data

Having opted in our approach for the manual construction of a dysfunctional model, we first had to collect a set of design information. To this end, we consulted the project deliverables: design documents, official presentation boards, Excel tables of requirements, and the safety concept. Due to the lack of clarity in the expression of the high-level modes, we had to complement our understanding of the system by interpreting the Simulink models of the ACC.

Table 1. 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.

The first input we used in this step is the technical safety concept defined earlier. 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.

The second input we used in this case study is the official architecture document. While in the traditional approach the architecture document is used by the safety analyst to directly conduct the safety analysis, we only use it to understand the functioning of the system if formal models describing the behavior of the system are not available. Thanks to the description in the architecture document, we were able to identify the components' operating modes. From the functional mode, we deduced the states and possible transitions that are necessary to model de dysfunctional behavior of the software components. A summary of the available functional modes for the software component "swcControlLongi" and the corresponding dysfunctional states is presented in Figure 9. From the functional modes, we deduced 4 dysfunctional states (Inactive, Nominal, Failed and Safe State). They will be necessary to model the dysfunctional behavior of the longitudinal control component. Nevertheless, as it can be seen on Figure 9 on the left, the architecture document did not provide enough details to allow us to describe these transitions. To overcome this limitation, we used Simulink models of the software components (which are, however, part of the detailed architecture). Thereby, we could determine the AltaRica domains (set of possible states to attribute to components and interfaces) necessary to model the dysfunctional behavior of the components and the signals on which they depend.

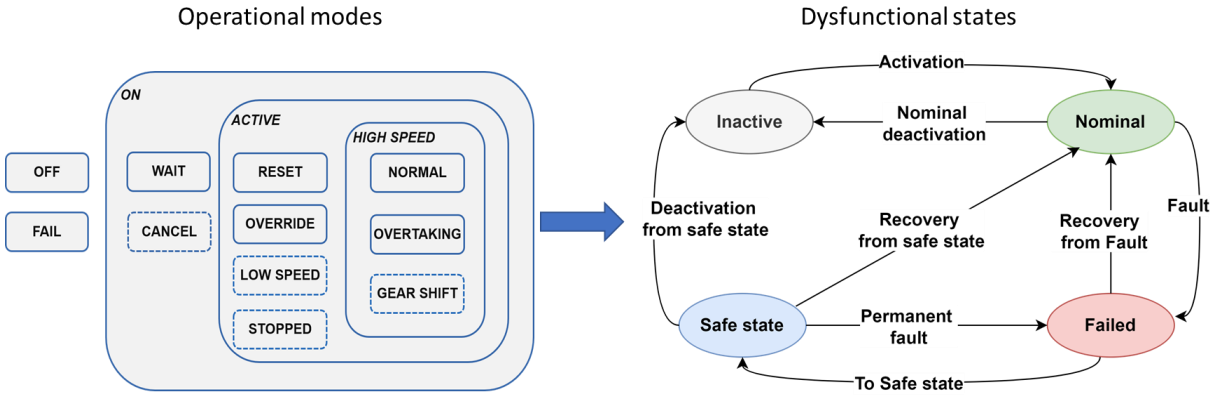


Figure 9. Dysfunctional states from the operating modes of the longitudinal control

Having identified the states and transitions necessary to model the longitudinal control software component, we also needed to do the same for some of its subcomponents as well as the other components interacting with it. After studying these components, we created AltaRica domains consisting of several states referred to as literal as show in Table 2. As show in the table, the Longi_Sup (longitudinal supervision) domain is used to describe the behavior of the longitudinal control according to 4 states (Inactive, Nominal, Failed and Safe State) that were described earlier in Figure 9. Similarly, other software components interacting with longitudinal control are described using a generic state domain (Generic_State) consisting of 3 literals (Nominal, Failed and Erroneous). To model the behavior of these other components, we were not interested in going into details as for the longitudinal control but to model their basic fault behavior just to be able to capture how they may affect the longitudinal control function. To complement the behavioral modeling, the failure flows between the components are modeled using a generic data domain (Generic_Data) consisting of 5 literals (Nominal, Loss, Delay, Out_of_Range and Erroneous). Using these domains, we were able to build state machines that describe the dysfunctional behavior of all the components.

Table 2. List of domains

Domain	Longi_Sup	Generic_State	Generic_Data
Literals	Nominal	Nominal	Nominal
	Failed	Failed	Loss
	Safe state	Erroneous	Delay
	Inactive		Out_of_Range
			Erroneous

5.3.2. Modeling

We used the SimfiaNeo tool to model the dysfunctional behavior of the components taking part in the functioning of the ACC, starting from the previously defined domains. Since the domains have already been defined in the previous step through a study of the operating modes, we only had to declare them in SimfiaNeo by relying on the operating modes retained in the previous step. In this way, we were able to create the domains representing not only the state of the components but also those the possible values of their inputs and outputs flows. This information allowed for easily modeling the behavior and the interfaces of each component.

Modeling the behavior of the components

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.). 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 default values were used instead. 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.

Modeling of component interfaces

For each component, we wrote 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 (based on the descriptions found in the safety concept) since not all inputs and outputs were useful for our dysfunctional model, as they do not affect the associated safety mechanisms. Next, we described the states of the inputs and outputs using the AltaRica domain that we named “Generic_Data” and that incorporates four states: Nominal (the data is good), Loss (the data is lost), Delay (the data arrives late), Out_of_Range (the value is outside the acceptable limits). 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 and the resulting AltaRica propagation logic code are presented in Figure 10 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 FSAct 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 10. Transition from Simulink logics to AltaRica assertions

Because of the advanced level of detail required to model the behavior of the system, we obtained a 3-layer model as shown in Figure 11. Layer 1 represents the longitudinal controller and the components interacting with it. Examples of elements descripted in the safety concept that can be seen in this model include the vehicle status input where the vehicle speed along with other input data is computed (cf. column 4 of the safety concept in Table 1.) and the fail-safe activation (FS_ACT) were various faults are checked and the information transmitted to the longitudinal control component. Layer 2 comprises the internal components of the longitudinal controller. It contains 3 controllers that manage distance, speed, and torque as well as the longitudinal fail-safe controller that implements most of the safety mechanisms descripted in the safety concept such as a range check and limits on the vehicle speed and the acceleration target. Some subcomponents in this layer are further modeled in detail in a third layer (as labeled layer 3 in Figure 11). is the longitudinal control supervisor where the dysfunctional sates and transitions previously described in the first step are modeled.

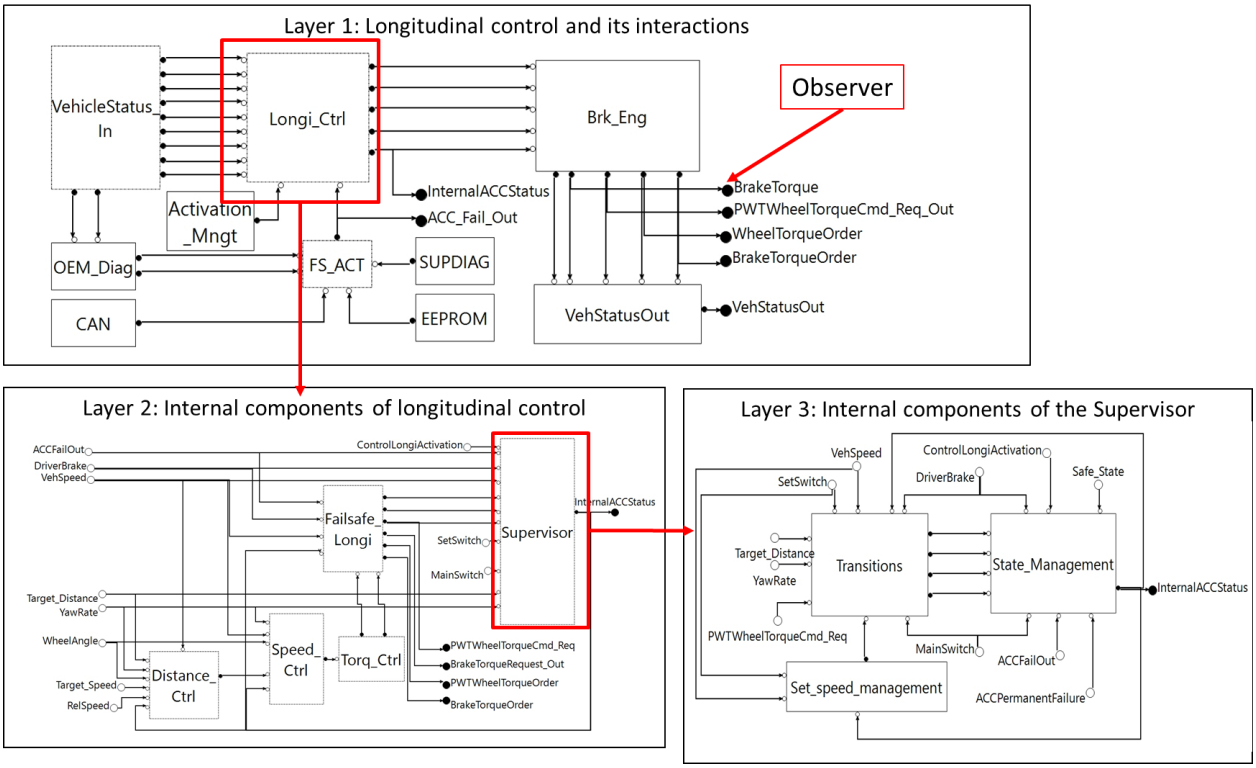


Figure 11. Modeling in 3 layers

5.3.3. Safety Analyses

Having completely modeled the ACC and the components that interact with it, the objective was to perform safety analyses from the dysfunctional model. For this purpose, we set up AltaRica observers on the outputs we were interested in (as shown in Figure 11). An AltaRica observer is an indicator that can be associated with a failure that we wish to observe. For example, let us consider, the USWE that we have chosen for our case study related to the violation of the safety goal 'REQ_TRV_SdF_TJP_HS_SG_004' that states: "No unintended Acceleration > ISO 22179 acceleration limit while travelling (v ≠ 0 km/h) requested by the longitudinal TJP feature". In our model, identified that the acceleration target and request in the speed controller subcomponent (Speed-Ctrl) are limited to 0.2G until vehicle speed vehicle 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 USWE. Therefore, the observers predicate to capture the occurrence of this USWE 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 failures that could lead to the transmission of this erroneous command that can result in the violation of the chosen UWSE. In addition, other safety goals can be affected by this scenario where an erroneous acceleration value is sent, including: 1) untimely acceleration or engine braking, 2) application of an engine command when the ACC or PWTWheelTorqueOrder status do not allow it, and 3) application of a command that conflicts with the braking command Therefore, more elaborates observers describing these scenarios can be associated with the PWTWheelTorqueCmd and other outputs form the dysfunctional model. This makes our model suitable for analyzing a vast range of safety goals without additional modeling effort.

5.3.3.1. Simulations

Having run a series of simulations, we observed the propagation of failures to observers 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 12. Blocks containing several components appeared without color during the 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.

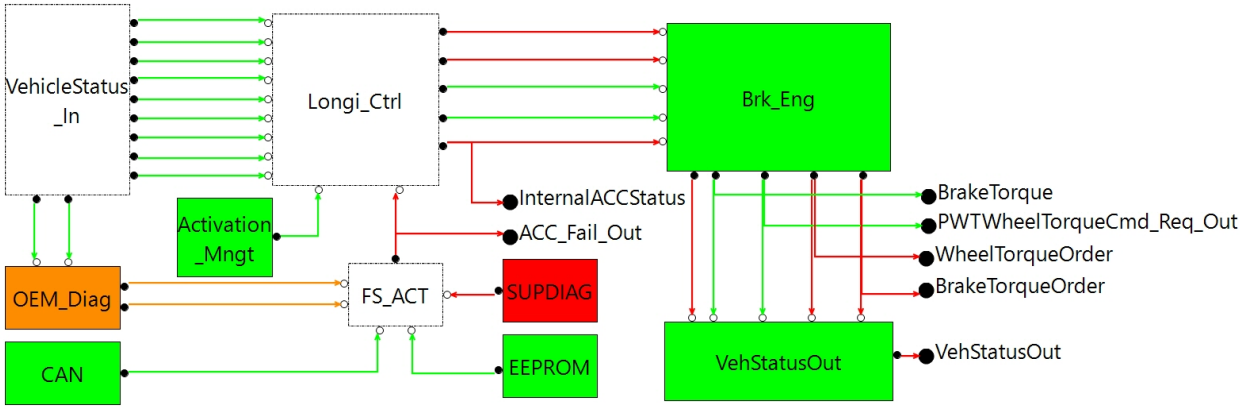


Figure 12. Simulation

5.3.3.2. 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 extract from an FMEA, containing a certain number of elements typically found in these tables. The first column (Scenario) 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 some of the observers described in the previous parts). For instance, in relation to our chosen UWSE, the excerpt from Table 3 shows how faults in the vehicle status input component (VehicleStatus_In) related to vehicle speed and wheel angle can affect other components and the final observer

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.

Table 3. FMEA

Event	Local effect	Local effect value	Intermediate effect	Intermediate effect value	Final effect	Final effect value
VehicleStatus_In.WheelAngle.error	VehicleStatus_In.WheelAngle.WheelAngle	Erroneous	VehicleStatus_In.WheelAngle.WheelAngle	Erroneous	PWTWheelTorqueCmd_Req_Out	Erroneous
			Brk_Eng.PWTWheelTorqueCmd_Req_Out	Erroneous	WheelTorqueOrder	Erroneous
VehicleStatus_In.V_kph_VehicleSpeed_Est_comp.failure	VehicleStatus_In.V_kph_VehicleSpeed_Est_comp.VehSpeed	Loss	VehicleStatus_In.V_kph_VehicleSpeed_Est_comp.VehSpeed	Loss	InternalACCStatus	Erroneous
			VehicleStatus_In.VehSpeed	Loss	VehStatusOut	Loss
VehicleStatus_In.V_kph_VehicleSpeed_Est_comp.error	VehicleStatus_In.V_kph_VehicleSpeed_Est_comp.VehSpeed	Erroneous	VehicleStatus_In.V_kph_VehicleSpeed_Est_comp.VehSpeed	Erroneous	PWTWheelTorqueCmd_Req_Out	Erroneous
			VehicleStatus_In.VehSpeed	Erroneous	WheelTorqueOrder	Erroneous
			Brk_Eng.PWTWheelTorqueCmd_Req_Out	Erroneous	BrakeTorque	Erroneous
Brk_Eng.failure	Brk_Eng.BrakeTorqueRequest_Out	Loss	Brk_Eng.BrakeTorqueRequest_Out	Loss	VehStatusOut	Erroneous
	Brk_Eng.WheelTorqueOrder_Out	Loss	Brk_Eng.WheelTorqueOrder_Out	Loss	BrakeTorqueOrder	Loss
	Brk_Eng.BrakeTorqueOrder_Out	Loss	Brk_Eng.BrakeTorqueOrder_Out	Loss	WheelTorqueOrder	Loss
	Brk_Eng.PWTWheelTorqueCmd_Req_Out	Loss	Brk_Eng.PWTWheelTorqueCmd_Req_Out	Loss	BrakeTorque	Loss

To analyses 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 con 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.

5.3.3.3. Minimal cut sets

The generation of the minimal cuts is achieved through the configuration and generation of the calculation sequences that offer several possibilities to the same 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 generation of calculations. The maximum order corresponds to the maximum number of possible combinations between elementary events leading to the feared event. After experimenting with a range of maximum order 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. Meanwhile the maximum order in the resulting minimal cut set remained equal to 3 for maximum orders above 3. 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 compromise 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 used during the generation of the cut.

As an example, we considered the chosen (PWT_Torque_Order =Erroneous) 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 and 3) of basic events that could cause the specified USWE (PWT_Torque_Order =Erroneous), as well as their associated probabilities. The probabilities are 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.

Table 4. Minimal cut

	Elements	Order	Probability
1	FS_ACT.ACC_OutputGen.error	1	1.0E-5
2	Longi_Ctrl.Torq_Ctrl.Brake_Torque.error	1	1.0E-5
3	Longi_Ctrl.Failsafe_Longi.SafeState.TempFailure	1	1.0E-5
4	Activation_Mngt.error	1	1.0E-5
5	Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error & VehicleStatus_In.WheelAngle.error	2	1.0E-10
6	Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error & VehicleStatus_In.V_kph_VehicleSpeed_Est_comp.error	2	1.0E-10
7	Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error & VehicleStatus_In.YawRate.error	2	1.0E-10
8	Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error & VehicleStatus_In.RelSpeed.error	2	1.0E-10
9	Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error & Longi_Ctrl.Speed_Ctrl.FFD_Loop_FBK_Loop_Corrections.error	2	1.0E-10
10	Longi_Ctrl.Distance_Ctrl.Vstop_Vmin_Strategies.error & Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error	2	1.0E-10
11	Longi_Ctrl.Distance_Ctrl.Rate_Limiters.error & Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error	2	1.0E-10
12	Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error & VehicleStatus_In.Target_Distance.error	2	1.0E-10
13	Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error & VehicleStatus_In.Target_Speed.error	2	1.0E-10
14	Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error & Longi_Ctrl.Speed_Ctrl.Internal_Model_Accelerator_Ctrl.error	2	1.0E-10
15	Longi_Ctrl.Distance_Ctrl.Feedback_Module.error & Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error	2	1.0E-10
16	Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error & Longi_Ctrl.Torq_Ctrl.Preprocessing.error	2	1.0E-10
17	Longi_Ctrl.Distance_Ctrl.Cut_In_Cut_Out.error & Longi_Ctrl.Distance_Ctrl.Following_Module.error & Longi_Ctrl.Torq_Ctrl.DynamicSaturation.error	3	1.0E-15

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 order to break them. For this purpose, we considered a loop and identified the self-dependent variable in the chain of assertions constituting this loop. We could then choose to remove this variable from one of the assertions if it was already taken into account in another assertion, or to give the variable a fixed value. In the latter case, it was necessary to manually change the value of the variable to include the scenarios which were excluded by giving it a fixed value. In both cases, the dysfunctional logic of the assertion remains valid.

The wide range of choices offered by SimfiaNeo through different configuration options (as described earlier) makes it possible to refine the calculations and simulations according to the needs of the analysis. By modifying the logic of the assertions and the generation options, producing cuts for various dysfunctional scenarios is therefore possible. However, a limitation related to high processing times was observed with higher maximum orders for the calculations. Although this appears to be a recurrent issue in safety analysis in general, it is an aspect that can certainly be improved in the tool. Regardless of the tool performance, this aspect also reminds us why it is important to keep safety analysis models simple and why a more pragmatic approach to safety modeling should be privileged

5.3.3.4. Fault trees from minimal cuts

For a defined feared event, it is possible to generate fault trees on the basis of minimal cuts: for example, we were able to generate the fault tree shown in

Figure 13 from the minimal cut presented in Table 4. Like the cut from which it is derived, this tree is specific to the UWSE (PWT_Torque_Order =erroneous) linked to the transmission of an erroneous torque command to the engine. Through its tree structure and logical combinations, the fault tree illustrates how the basic events (located at the bottom of the tree) can lead to the feared event (at the top of the tree). As shown in

Figure 13, fault trees 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.

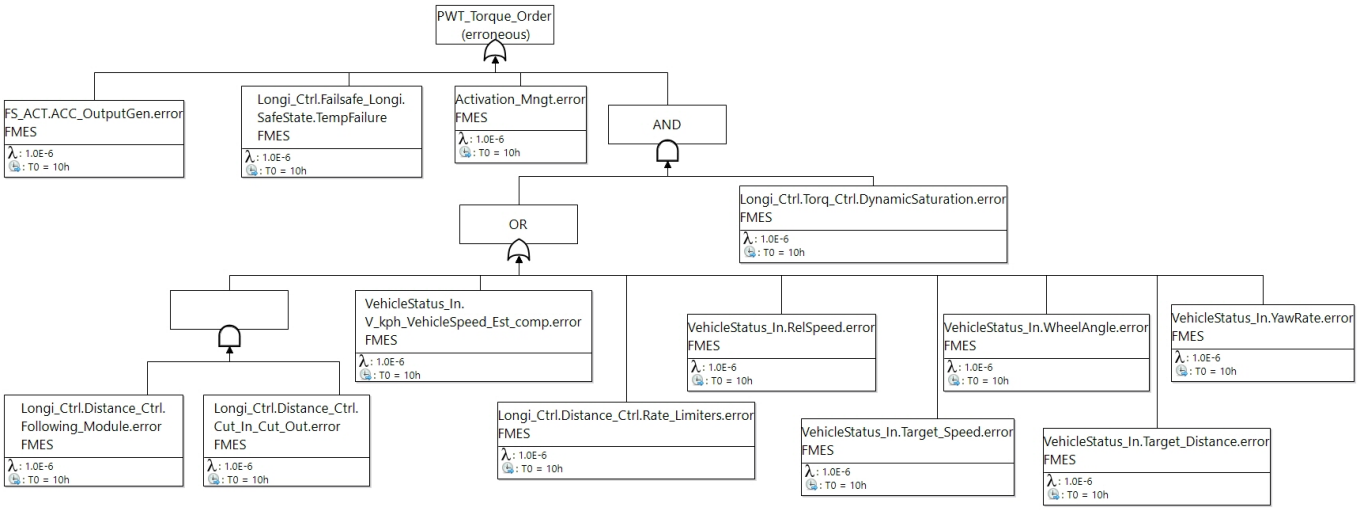


Figure 13. Fault tree

As is the case with the cuts, the generated tree can be updated by modifying the assertions of the observers and the generation parameters of the associated cut. Moreover, the observers can combine several basic events, thus enabling the reproduction of feared events linking several components.

To evaluate these results, we performed a comparison with another fault tree of the same USWE manually obtained through the traditional approach that can be found in Figure 14 in the annex section. To evaluate the occurrence of the USWE presented at the top of the tree, the manual fault tree in Figure 14 develops two scenarios: the non-detection of the USWE and the misleading reaction to the detection of the USWE. These two scenarios are then developed into further basic contributing causes. However, although the descriptions of the scenario are more humanly readable, the fault tree lack precision, is abstract, and difficult to relate to the architecture of the software. Contrary to the tree automatically generated though our approach, the manual fault tree does not show in which component the faults occur. In addition to being more precise, our fault tree uncovers more basic events resulting from computing errors along the processions path to the final acceleration command to the motor. To be useful, manually constructed fault tree are generally reduced through cuts to produce the smallest combination of basic events required to cause the top event to occur. This constitutes an additional effort that the analysist must make. Through our approach, the generated fault tree is already reduced to the minimal possible since its computation is based on a minimal cut.

6. Discussion

The application of the methodological proposal to the case study demonstrates that it is possible to apply an MBSA-type approach (which has been system-oriented until now) to software. Through the appropriate selection of input data and modeling effort, our approach provides a dysfunctional model representative of the real system, thus improving the quality of safety analyses. Compared to safety analysis results obtained through the traditional approach, our method proves to produce various types of analyses with relatively minimal efforts. For instance, through the traditional approach, only fault trees analyses were conducted by the safety analyses while trough our approach various types of analyses are possible without additional effort. Another issue with the manually conducted fault tree is that it was done based on the detailed architecture of the software as stated in the safety analysis report document. This raises the question of late safety analysis. Our approach enables early safety analyses as modeling is done at the architecture level and further refined at detailed architecture

Furthermore, tools such as SimfiaNeo relieve the safety expert of manual calculations while allowing him to concentrate on dysfunctional modeling. The benefits are reflected in terms of time, analysis quality and reusability of the models. Once the dysfunctional model is built, it will be possible to conduct analyses with different parameters for a large number of USWEs because it will be enough to express these parameters using appropriate Boolean combinations. 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 fault trees or FMEAs. The analyst would need to manually construct as many fault trees as there are feared events. If the design evolves, they will need to individually update all the fault trees and the FMEAs. Through our proposal, the dysfunctional model is easy to maintain. If the system design evolves, the dysfunctional model can be updated and updated fault trees or FMEAs can be automatically derived. In addition, representing the behavior of the system without ambiguity is possible through the formal semantics of AltaRica, turning it into a possible candidate in a certification context.

All these elements can significantly improve current practices in safety analysis. Nevertheless, we noted some limitations on the method, the language, and the tool during the case study. Modeling can be time consuming, since the right level of detail in the representation of the system is needed. Here, we propose to clearly define the input elements (components, signals, safety mechanisms) to be studied in step 1. Although Step 1 and 2 are manual, they allow a pragmatic approach to construct the dysfunctional model thanks to the limited number of elements to model based on the content of the technical safety concept. This results in a succinct but sufficient dysfunctional model that is constructed based on the needs of the analysis which considerably

the time needed to perform the analyses. One of the difficulties brought about by a dedicated dysfunctional model is maintaining its consistency with the design model when the latter evolves; this problem was already true when working on analyses based on fault trees. Implementing additional measures is therefore necessary to guarantee consistency. The difficulty of translating certain safety requirements (such as those with time constraints) into AltaRica expressions is a limitation of our methodology, albeit linked to the version of the language used: it is, in fact, a specific problem of the version of AltaRica (dataflow) implemented in the SimfiaNeo tool. Another limitation, still related to the AltaRica dataflow, is the inability of our approach to manage loops (note that this problem also concerns fault trees); our solution was to modify some assertions in order to remove these loops. A last limitation, related to the performance of SimfiaNeo in this case, is its high resource requirements (in terms of memory and processing power); we encountered the unexplained loss of a portion of the source file of our model, and we had to rebuild it manually. This raises questions about the reliability of the tool. Finally, as the case study involves a system of reduced complexity, the scaling up of our methodological proposal must yet be evaluated. Yet we remain optimistic and believe that our pragmatic approach can adequately handle complexity through the proper selection of inputs and components to model. While this might be a small step, within our industrial context in our company, our proposal integrates a broader technological transformation towards MDE together with various other works that aims to make the current practices evolve.

7. Conclusion

This paper made a 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.

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).

Building on these results, the study must now continue to evaluate the complexity of the systems for which the methodology and the tooling can be reasonably applied. Since the proposal of this paper 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 models.

8. Acknowledgements

The authors would like to thank Christophe Frazza of the DGA TA (Direction Générale de l'Armement Techniques Aéronautiques) for his proofreading and advice.

9. References

[1] 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

[2] “ISO 26262-1_2018_Ed2.”

[3] 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

[4] M. Brambilla, J. Cabot, and M. Wimmer, “Model-Driven Software Engineering in Practice, Second Edition,” *Synth. Lect. Softw. Eng.*, vol. 3, no. 1, pp. 1–207, Mar. 2017, doi: 10.2200/S00751ED2V01Y201701SWE004.

[5] S. Kelly and J.-P. Tolvanen, “Domain-specific modeling: enabling full code generation,” p. 446.

[6] 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

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

[8] 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.

[9] 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

[10] ISO, *ISO 26262 2018 Ed2 — Road vehicles — Functional safety*. ISO, 2018.

[11] V. Louis and C. Baron, “Vers une certification continue des logiciels critiques en aéronautique,” *undefined*, 2019, Accessed: Jun. 02, 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02372069/

[12] O. Akerlund *et al.*, “ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects,” p. 12, 2006.

[13] 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.

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

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

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

[17] 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.
[18] 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
[19] 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.
[20] 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.
[21] 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.
[22] 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.
[23] 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.
[24] 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.
[25] 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.
[26] 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," Toulouse, France, Feb. 2014. Accessed: Nov. 13, 2019. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02271308
[27] "Simulink - Simulation and Model-Based Design." https://www.mathworks.com/products/simulink.html (accessed Apr. 23, 2021).
[28] J.-L. Colaco, B. Pagano, M. Pouzet, and M. Pouzet, "Scade 6: A Formal Language for Embedded Critical Software Development," p. 10, Sep. 2017.
[29] 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.
[30] 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.
[31] 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.
[32] 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.
[33] 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.
[34] M. Sango, F. Vallée, A.-C. Vié, J.-L. Voirin, X. Leroux, and V. Normand, "MBSE and MBSA with Capella and Safety Architect Tools," in *Complex Systems Design & Management*, Cham, 2017, pp. 239–239. doi: 10.1007/978-3-319-49103-5_22.
[35] 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.
[36] M. Gudemann and F. Ortmeier, "A Framework for Qualitative and Quantitative Formal Model-Based Safety Analysis," in *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*, Nov. 2010, pp. 132–141. doi: 10.1109/HASE.2010.24.
[37] G. Point and A. Rauzy, "AltaRica: Constraint automata as a description language," 1999. Accessed: Nov. 28, 2019. [Online]. Available: https://altarica.labri.fr/wp/wp-content/uploads/2013/05/PR99b.pdf
[38] 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.
[39] 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.
[40] "AltaRica Studio | AltaRica Project." https://altarica.labri.fr/wp/?page_id=286 (accessed Apr. 26, 2021).
[41] "OpenAltaRica." https://www.openaltarica.fr/ (accessed May 27, 2020).

10. Annex

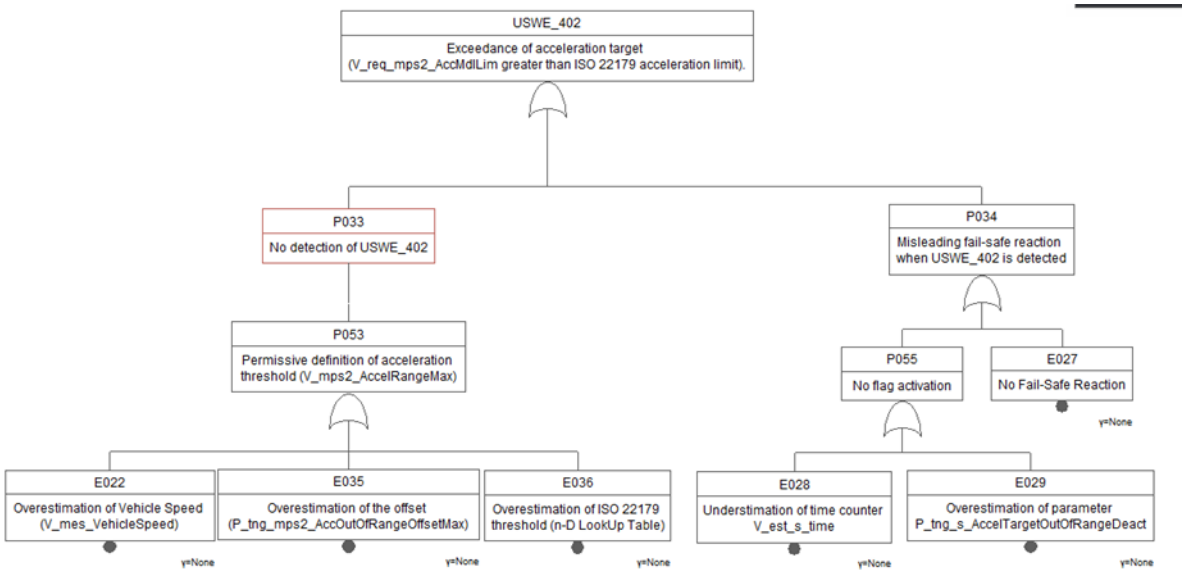


Figure 14. Manually constructed fault tree