Investigating the use of a model-based approach to assess automotive embedded software safety

Yandika Sirgabsou, Claude Baron, Cyril Bonnard, Laurent Pahun, Lorenzo Grenier, Philippe Esteban

INVESTIGATING THE USE OF A MODEL-BASED APPROACH TO ASSESS AUTOMOTIVE EMBEDDED SOFTWARE SAFETY

Yandika SIRGABSOU
LAAS-CNRS, Université de Toulouse, CNRS, INSA Toulouse
Renault Software Labs
Toulouse, France
yandika.sirgabsou@renault.com

Claude BARON
LAAS-CNRS, Université de Toulouse, CNRS, INSA Toulouse
ISAE-Supaéro
Toulouse, France
claude.baron@laas.fr

Cyril BONNARD
Groupe Renault
Renault Software Labs
Toulouse, France
cyril.bonnard@renault.com

Laurent PAHUN
Groupe Renault
Renault Software Labs
Toulouse, France
laurent.pahun@renault.com

Lorenzo GRENIER
INSA Toulouse
Université de Toulouse
Toulouse, France
grenier@etud.insa-toulouse.fr

Philippe ESTEBAN
LAAS-CNRS, Université de Toulouse,
CNRS, Université Toulouse 3
Toulouse, France
philippe.esteban@laas.fr

ABSTRACT: *With autonomous driving, vehicles are undergoing tremendous and multiple innovations in a variety of areas of automotive expertise. In particular, the amount of software used in embedded safety-critical systems is increasing at a rapid rate to implement new features. It is therefore essential today to guarantee the safety of software by carrying out safety analyses in accordance with automotive standards. These analyses allow engineers assessing the design with regard to safety and to determine the modifications if needed to meet safety objectives. However, the traditional approach to perform these analyses is cumbersome and limited when faced with the complexity of today's automotive software architectures. Safety analyses are currently performed manually, and the results are dependent on the experience of the safety expert. As a result, they are highly subjective and are not guaranteed to be exhaustive and error-free. To overcome these issues, this paper explores the use of a model-based safety approach in the context of safety-critical automotive embedded software. It makes a methodological proposal that relies on the software architecture model to build a dedicated safety model from which safety analyses can be automatically derived. The method is experimented on an automotive case study, an embedded software that assists the driver in following the lane.*

KEYWORDS: *Automotive, embedded software, safety-critical systems, model, software engineering, MBSA, MBSE*

1 INTRODUCTION

In our society undergoing deep technological and societal changes, vehicles (cars, drones…) are becoming more and more autonomous. Their architecture involves several cyber-physical systems that massively embed software components in order to allow this autonomy. In this context, there is a societal need to ensure and guarantee the vehicles safety, therefore the systems and software safety.
In the automotive domain, software safety analyses are currently based on traditional manual techniques. Generic quality-oriented standards are used as references, and the quality of the analyses mostly depends on the experience of safety experts. Safety analyses are not really formalized, do not allow even a partial reuse and sometimes offer approximate guarantees of safety. With regard to this evolving context, it is therefore necessary to improve current industrial practices in order to better respond to the societal and economic issues.
On another side, the current trend in engineering is to adopt model-based approaches. They enable formalizing analyses, a better communication and collaboration between interdisciplinary teams, rapid prototyping and simulation, and improve reuse. Using model-based approaches to assess software safety thus seems promising as it would help addressing the current issues that are related to safety-critical software analysis.
Therefore, the objective of this paper is to explore the possibility of using a model-based approach to perform safety analyses on embedded automotive software. To this goal, this paper first gives an overview of model-based methods and underlines their interest in software engineering and safety assessment. It compares the state of scientific knowledge with respect to the state of industrial practices in the field of automotive software engineering in order to identify areas for improvement, in the practice of software safety analysis. We make a methodological proposal, which consists in automating the production of software safety analyses from a purposely built safety model. Then we give the results obtained from the application of the proposal on a real-world case study, the lateral control software component which is part of the autonomous driving software and whose role is to keep the vehicle in lane. We finally indicate some interesting research avenues to further develop this work.

2 MODEL-BASED APPROACHES IN SOFTWARE ENGINEERING AND SAFETY

The growing systems complexity requires the implementation of development methods to keep costs, time and quality under control. Traditional, document-centric and

test-based approaches are not sufficient enough for the development of multi-disciplinary and distributed smart systems. Model-based (or model-driven) approaches address this complexity with a model-centric, frontloaded engineering methodology that focuses on creating and exploiting domain models as the primary means of information exchange between engineers, rather than on document-based information This section quickly reminds the principles and interests of model-based approaches in software engineering then presents their use in the context of safety analyses.

2.1 Model-Based Software Engineering

Model-Based Systems Engineering (MBSE) is defined as "the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases [1][2]. This definition is also applicable to software engineering, where MBSE[1] has been proposed as a promising software development methodology to overcome limitations of traditional programming-based methodology [3]. MBSE promotes the use of modeling languages for describing software in an abstract way; it is used for modeling requirements, functional and physical architectures, but it also supports simulation, code generation and verification by providing means for automatically generating different development artifacts, e.g. code and documentation, from models.

2.2 Model-Based Safety Assessment

According to Joshi et al. [4], Model Based Safety Analyses (MBSA[2]) is the application of the MBSE techniques to support safety analysis. However, whereas MBSE models the nominal (non-failure) functional behavior of a system, MBSA models its fault (dysfunctional) behavior. Indeed, safety analyses aim at identifying whether the system, as modeled, has weak points. They can be performed either to analyze if a system component failure can induce a serious failure at the system level, or to determine what are the possible root causes of a system failure. To this goal, the safety engineers draw their conclusions from traditional models such as respectively FMEA[3]s or FTA[4]s that they manually elaborate from the available system specification and design artefacts resulting from design stage.

Adopting a model-based approach in safety engineering consists in building a dysfunctional model of the system that shows the system behavior in case of a failure (reasoning by failure propagation), from which the traditional analyses (minimum cuts[5], FTAs and FMEAs) can be derived [6]. This notably allows an easy and quick generation of new safety analyses in case the system architecture evolves, therefore reduces the cost and improves the quality of the safety analysis process.

2.2.1 MBSA Methods

MBSA methods can be classified according to two main criteria, the dysfunctional model construction and the semantics of component interfaces.

The first criterion is related to the process for defining the safety model and its relationship with the system design model: the dysfunctional model can either be an extension of the design model or a dedicated model [7]. We find in [4] an example of an extended model, where a nominal functional model is first constructed during design to which failure modes are added for the purpose of safety analysis. The key advantage of the model extension approach is the consistency, by construction, of the safety analyses and the design model of the system. Furthermore, development and safety processes can share a common modeling environment, languages and tools [7]. However, it has some drawbacks. One is that it does not allow independence between the system and safety models. In the case of a dedicated model, a distinct 'standalone' dysfunctional model is built by the safety engineer based on his understanding of available information from design documents and functional models. The key advantages of this approach are that it is more pragmatic to implement, as it ensures independence and separation of concern (between safety and engineering disciplines). However, one of its drawbacks is that it requires supplementary means for ensuring consistency with the design model.

The second criterion is related to the dysfunctional model semantics (components behavior) and the type of information that is conveyed through the component interfaces, either nominal or failure flows [7]. This criterion leads to distinguishing Failure Logic Modeling (FLM) [6][8] (that uses failure flows) and Failure Effect Modeling (FEM) [6] (that uses nominal flows). Early proposals adopted the FEM approach [4] but, as the discipline evolved, FLM has gained prevalence and most of the pioneering MBSA methods such as FPTN [6], HiP-HOPS [9] and AltaRica [10] rely on it.

Several modeling languages support MBSA. Some are dedicated to safety such as AltaRica [13], SAML [14] or Figaro [15]. Others are multipurpose modeling languages (such as UML) and architecture description languages (such as AADL [16] or EAST-ADL [17]) that extend their core semantics and syntaxes to support safety analyses trough profiles and error annexes [12] .

2.2.2 AltaRica language

AltaRica [19] is a high-level modeling language dedicated to risk analysis that supports safety, reliability and

[1] In this paper, the S in MBSE would either stand for System or Software.
[2] The A in MBSA either means Analyses or Assessment.
[3] Failure Modes and Effects Analysis. FMEAs are systematic bottom-up safety analyses to study the effects of every single failure condition (a feared event) on the whole system.
[4] Fault Tree Analysis. FTA is a graphical tool to explore the causes of system level failures. It uses Boolean logic to combine a series of lower level events and it is basically a top-down approach to identify the component level failures (basic event) that cause the system level failure (top event) to occur.
[5] Minimum cuts represent the smallest combination in which basic events can possibly cause a system failure; they are used intermediately in the generation of certain FTAs. Cut sets are the unique combinations of component failures that can cause system failure. A cut set is said to be minimal if, when any basic event is removed from the set, the remaining events collectively are no longer a cut set [5].

performance analyses [10]. In AltaRica, model elements are expressed in terms of nodes. Each node is composed of states, events, transitions and assertions [20]. States are declared using domains. A domain is an enumerate comprising several states. Several tools currently support AltaRica: OCAS [13], Simfia and SimfiaNeo [23], Open AltaRica [24] and AltaRica Studio [25].
In conclusion, among the current MBSA methods, we found that using a dedicated model and FLM approaches reveals to be the most interesting, as it adopts a safety oriented modeling; when combined with a dedicated model, this ensures independence between system design and safety (which is an important criterion in the certification of critical systems). For the case study analysis, we chose AltaRica for its simplicity and proven usefulness in the system context. However, if AltaRica is well-adapted to software failure modeling remains to be clarified.

3 AUTOMOTIVE PRACTICES IN SOFTWARE ENGINEERING AND SAFETY ASSESSMENT

3.1 A document-based software development process

The automotive software development process is centered around two standards: ASPICE [32] (for quality) and ISO 26262 [33] (for functional safety). ASPICE (Automotive Software Performance Improvement and Capability dEtermination) is a standard that provides guidelines to improve software development processes and to assess suppliers. ISO 26262, the "Road Vehicles-Functional safety" is the standard for functional safety of electrical and/or electronic systems in automotive production. It provides a reference for the automotive safety lifecycle. It also defines the Automotive Safety Integrity Level (ASIL), representing the degree of automotive hazard and the degree of rigor to apply in specifying and implementing safety requirements and safety measures. ASILs range from A to D, with D representing the most stringent and A the least stringent level, while QM (Quality Management) is allocated to items that have no impact on safety. ISO26262 is divided into 11 parts; Part 6 addresses the product development at software level (which is more specific to software engineering context).
Figure 1 shows the ISO26262 phase model for the product development at the software level. The left side of the cycle covers the 'Specification of the software safety requirements', the 'Software architectural design', and the 'Software unit design and implementation' phases. The software safety requirements usually are non-formalized, expressed in natural language and managed through tools such as Rational DOORS. At the software architectural design phase, the use of semi-formal modeling language (such as UML) to support design and analysis remains quite immature and unmastered. As a result, the software architectural design process mostly relies on informal models and natural language descriptions. However, at the software unit design phase, tools like Simulink are effectively used for prototyping, simulation and code generation.

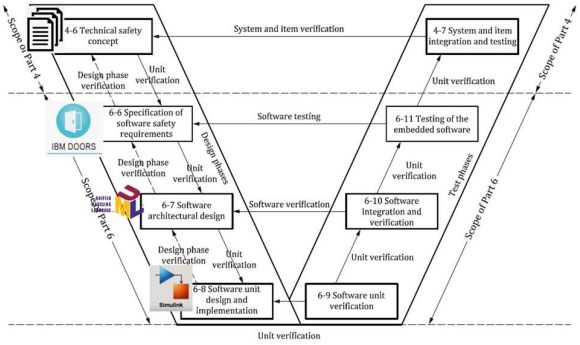

Figure 1. Reference phase model for the product development at the software level from [33]

On the right side of the cycle feature the 'Software unit verification", 'Software integration and verification' and 'Testing of the embedded software' phases. Each of these phases aim to verify and validate if the corresponding left side (design stage) phases are implemented as required. Traceability is maintained between related phases as shown by the arrows on Figure 1.

3.2 A document-based safety assessment process

In the current practices in automobile, a document-based safety assessment process is conducted as the integral part of the design process and is done in compliance with ISO 26262. The process comprises the manual construction of safety cases and safety analyses in document templates (in Excel, Word, PowerPoint format). Unfortunately, these analyses are highly subjective and dependent on the skill of the engineer. Their traceability back to design artifacts is mainly maintained through naming conventions and sometimes hyperlinks in collaborative tools. The final FTAs are often produced through a process of review and consensus building between the system and safety engineers [34]. Even after a consensus is reached, it is unlikely that the analysis results will be complete, consistent, and error free due in part to the informal models used as the basis of the analysis. In fact, the lack of precise models of the system architecture and its failure modes often forces the safety analysts to devote much of their effort to gathering information about the system architecture and system behavior and embedding this information in the safety artifacts such as FTAs. In these conditions, it is difficult to ensure rigorous safety analyses and traceability.
Ideally, this situation could be significantly improved, for instance if engineers produced formal models of the system under development, then extended this model with a dysfunctional model, and then performed safety analyses by deriving FMEAs and FTAs from this latter. However, as currently design models are not formalized enough to be exploited, we think that a first step to bridge the gap towards this ideal situation could consist in a manual but traceable building of the dysfunctional model. The achievement of this objective constitutes the motivation for our methodological proposal.

4 METHODOLOGICAL PROPOSAL

Our proposal consists in building a dysfunctional model of the software from which minimum cuts, FTAs and FMEAs could be automatically generated. Referring to what section 2.2.1 clarified, it relies on the use of a dedicated model [7] and the use of a failure logic modeling technique [6] and supporting languages.
A synopsis of the proposal is presented in Figure 2. It proceeds into 3 steps: ① Dysfunctional Modeling, ② Functional to Dysfunctional Logic Translation and ③ Safety Analysis.

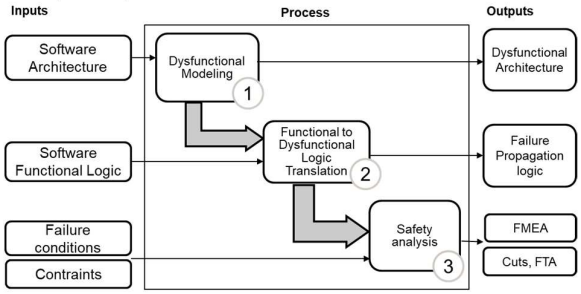

Figure 2. Synopsis of the proposed methodology

Step ① consists in building a dysfunctional hierarchical architecture of the software from the software functional architecture. For each software component, Step ② translates the software functional logic into a failure propagation logic. This step allows writing the output state of each component taking in consideration not only the internal state of the components but also the states of the inputs. Step ② results in a dysfunctional model that is used at Step ③ to perform safety analysis and generate classical safety models (such as minimum cut sets, FTAs and FMEA) from failure conditions and constraints.

4.1 Step 1: Dysfunctional modeling

We propose building a formal dysfunctional model from the software functional architecture in order to capture the fault behavior of the software.
Using various information captured from the software architecture documents (containing informal designs models as well as their functional description), we define and assign abstract states (e.g. active, failed or temporarily failed) and associated transitions[6] (e.g. failure, partial failure, deactivation, cancel) to software components. We do the same for the interface (inputs and outputs) between the components to which we assign abstracts states (e.g. ok, no data, loss, erroneous data). This results in a dysfunctional model featuring the software components and how they are connected by failure dependency links.
Thanks to its graphical representation, this model is easy to understand; as formal, it can be used for simulation, analysis and evidence. To build this model, several formalisms can be used, including state/transition diagrams as well as other dedicated safety modeling languages such as AltaRica or AADL.

To illustrate this proposal, an example is given in Figure 3. It shows three software components A, B and C, to which internal state variables (representing their possible states) are assigned. Each component has 2 states (nominal or failed). The inputs and outputs are also expressed using states (ok, no-data, erroneous). In this example, if the software component B fails (due to its internal state or erroneous input), its output can replicate this failure to the C software component which in turn can relay it to the Failure Condition.

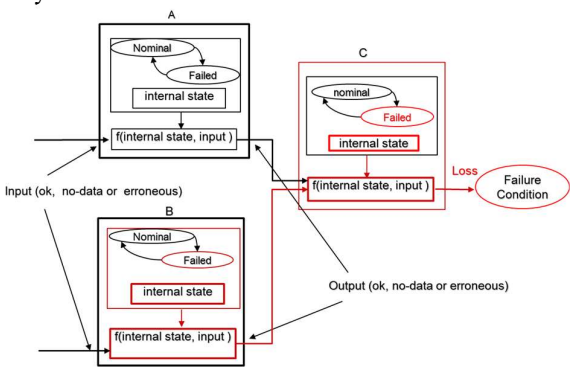

Figure 3. Illustration of a dysfunctional architecture

The completion of Step ① results in a dysfunctional architecture comprising software components (whose behavior is modeled using states) and their interconnections.

4.2 Step 2: Functional to dysfunctional logic translation

To complete the dysfunctional model, the goal of Step ② is to express the dependencies between the component inputs and outputs, that is how failures can propagate through the software architecture. To ensure the consistency between the functional and dysfunctional models, we propose to translate the functional logic into a dysfunctional logic.
Therefore, we proceed by using failure truth tables. We call 'failure truth table' (FTT) one table that systematically maps the normal flows (from the functional logic) into failure flows (dysfunctional logic).
To build the failure truth tables, we first analyze the functional logic of a component. Then using the states (nominal, erroneous, loss), defined in Step ①, we set the inputs (to nominal, erroneous or loss) and deduce the corresponding output (nominal, erroneous, loss). Repeating this process allows building the FTT with all the possible combinations of the inputs and the corresponding outputs in dysfunctional flows.

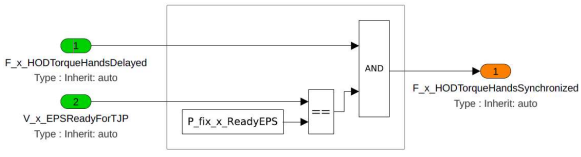

Figure 4. Simple functional logic of a software component

[6] A transition is a passage from one state to another.

To illustrate the construction of an FTT, Figure 4 presents a simple example of functional logic of a software component (in the rectangle). It is a simple function that returns an output based on the value of two inputs labeled 1 and 2. Using our proposal, we set the internal states to the inputs and report to the table the corresponding output states. We reiterate this for all the possible combinations of the input's states.

Table 1. Failure truth table

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

Output

This results in the FTT shown on Table 1. In the case of this example, if both inputs are 'nominal', then the output is 'nominal'. If either one of the inputs is 'loss' then, the output state is 'loss'. Otherwise, the output state is 'erroneous'.

In order to obtain a computable expression of this logic, we then translate the information of the FTT into a dysfunctional logical expression of outputs:

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

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

The completion of Step ② produces a well-expressed failure logic resulting from the FTTs and a complete dysfunctional model that can be used in Step ③ to perform safety analyses.

4.3 Step 3: Safety analysis

This step consists in exploiting the dysfunctional model resulting from Step ② for simulation and generation of safety analyses such as FMEAs, Minimum cut sets and FTAs. To this goal, failure conditions, that represent the violation of safety software associated goals, are added to the architecture.

In conclusion, Step ③ can help in the evaluation of the system safety through simulations and the generation of classical models (Minimum cut sets, FTAs, FMEAs). This saves time and helps the safety engineers in their task.

4.4 Discussion

The proposal offers several interests and significantly improves current practices, mainly thanks to the adoption of a model-based approach. By automating the generation of safety analyses from a dysfunctional model of the software, it allows quickly processing new analyses if ever a modification occurs on the model, at low effort. Assuming that the dysfunctional model is correctly established from the functional model, this methodology prevents from the introduction of bias in manually building FTAs or FMEAs due to potential interpretations of the safety analyst. To perform the functional to dysfunctional model transformation at best, the consistency and traceability between models is ensured thanks to the failure truth tables. In addition, the methodology allows maintaining the safety analyses whenever the design evolves.

However, the proposal still has some limitations. First the manual modeling could be improved if some behavioral attributes of the software component had already been elucidated in the design model. In this case, a partial import could be considered. This highlights the need of improving model-based practices on the design side. A second limitation lies in the manual logic translation through failure truth tables, that could be improved if a form of automation was used, based on a kind of automatic model transformation.

5 CASE STUDY

In an automobile, the Advance Drivers Assistance System (ADAS) and Autonomous Driving (AD) implement software-based functions that assist the driver and improve their driving experience. The chosen case study is a subsystem of the AD software system. As presented in Figure 5, it consists of three main software components: The Hands-Off Detection (HOD), the Longitudinal Control and the Lateral Control. This case study being part of a broader system, we also consider other components of the system (such as the Status Input) that must be included to perform the analysis and to position the software components with respect to their environment.

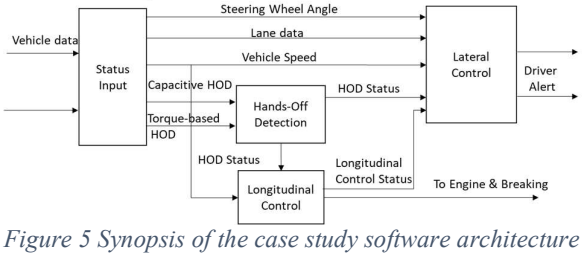

Figure 5 Synopsis of the case study software architecture

At autonomy level 1 (corresponding to assisted driving[7]), the drivers are required to keep their hands on the wheels even though AD features are activated. In this context, the role of the Hands-Off Detection (HOD) is to detect if the

[7] There are 6 levels of autonomy ranging from 0 (no automation) to 5 (full automation)

driver has taken their hands off the steering wheel. To ensure an effective and reliable detection, two redundant strategies are used, a capacitive HOD detection based on a capacitive sensor and a torque based HOD based on a measure of the force the driver applies on the steering wheel. Both strategies provide a consolidated HOD status for lateral and longitudinal control. The Longitudinal Control manages the acceleration, deceleration and braking. The Lateral Control keeps the vehicle in lane. It takes part in critical functions, steering and user alert. Its activation depends on both the HOD and the Longitudinal Control statuses as well as on various environmental and vehicle data (lane, steering wheel angle, vehicle speed) through the Status Input (a software component that provides data to the HOD, Longitudinal and Lateral Control).

5.1 Choosing tools to support the methodology

Let us remember that we chose using a dedicated dysfunctional model and the FLM approach. This allows us to have a more safety-oriented point of view of the system and enforces independence and separation of concern (functional features vs safety considerations).
We chose AltaRica based solution for its simplicity but also for its formal aspects. AltaRica allows expressing a dysfunctional behavior through a well-defined semantic. Moreover the language has been proven useful in many contexts (for instance for systems safety assessments in aeronautics) as outlined in [13]. Among AltaRica based tools we chose SimfiaNeo for its more innovative features and friendly eclipse-based user interface. SimfiaNeo has a built-in step-simulator offers an FMEA generator, and a built-in model checker that is used to directly generate minimum cut sets and FTAs from the dysfunctional architecture.

5.2 Applying the methodology

We then applied the different steps of the methodology: first, in Step ①, we defined and modeled the software components states and basic failures, then in Step ②, we modeled the failure propagation using logical expressions linking outputs to inputs as well as internal states; finally in Step ③, we used the resulting dysfunctional architecture to perform safety analysis.

5.2.1 Step 1: Dysfunctional modeling

We used the information from the AD software functional architecture (documents and informal models) to build our basic dysfunctional model. First, using SimfiaNeo graphical modeling interface, we modeled the software components with model bricks.

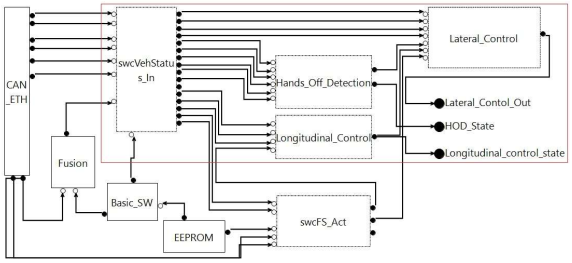

Figure 6. Top level view of the case study model

Then, using AltaRica, we defined domains (cf. section 2.2.2) for components and inputs / outputs, states and associated transitions (expressed as events in AltaRica) for each component. A top-level view of the model is shown in Figure 6. The red rectangle delimits the perimeter of the case study.
We now need to complete this dysfunctional model with a failure propagation logic to be able to perform safety analysis.

5.2.2 Functional to dysfunctional logic translation

A logic translation is used to assist in the building of the failure propagation and ensure a better consistency of the dysfunctional model with the design model. For that, logic retrieved from Simulink models are translated into AltaRica logic using FTTs.
Figure 7 illustrates this transformation. The functional logic is from a subcomponent of the HOD software component. From a functional point of view, this logic returns a status (HODConsolidationHandsOn) based on whether the two signals (HandsOff and HandOff_mirror) are equal. The output (HandsOffState) captures this status.

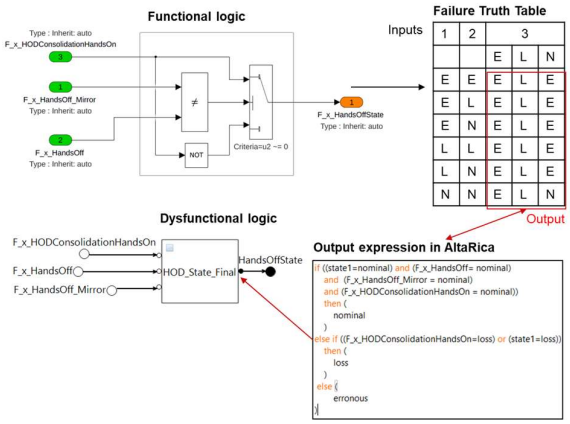

Figure 7. Functional to dysfunctional logic translation using truth failure tables

The goal of the functional to dysfunctional logic translation is to go from this functional point of view to a dysfunctional point of view. For that, we assign the states (predefined in Step ①) to the inputs (on the functional logic), and we write the corresponding output states on the FTT. We reiterate this process for all possible combinations of input states. This creates the corresponding failure table, with all possible output states, as shown in Fig. 7.
Using the FTT, we can now write the output expression (of the corresponding dysfunctional brick) in AltaRica.
This step is repeated for all the software components of AD. The tables can be used as templates to model more complex components and systems. This can help save time through reuse. Overall, Step ② completes the dysfunctional model initiated in Step ①. Thanks to this, the dysfunctional model is more consistent with the functional model and can be used for safety analysis in Step ③.

5.2.3 Step 3: Safety Analysis

This step consists in generating classical safety models such as FMEAs, minimum cut sets and FTAs from the dysfunctional model, using the SimfiaNeo model checking capabilities. To this goal, we used the dysfunctional model from Step ② and the failure conditions identified in earlier stages of the safety lifecycle. We proceeded by adding failures conditions using AltaRica observers[8] via direct connections from the software components outputs or through a combination of assertions expressed in AltaRica. Once this is done, we can now perform safety analysis. First, we performed simulations. Then we directly generate FMEAs from the dysfunctional architecture. After that, we compute the minimum cuts sets, from which we were able to immediately generate FTAs.

To perform simulation, we used the SimfiaNeo built-in step simulator. It allows us to interactively assess the dysfunctional behavior of the system. For that, we manually trigger basic failure events (at component level) and observe their effect, through change in colors, on other components and failure conditions (earlier added through AltaRica observers).

SimfiaNeo offers the automatic generation and export of FMEA tables directly from the software dysfunctional model. Given the complexity of the software architecture and the multiple basic failures (at component level), this results in a plethoric number of failure modes (in FMEA tables). For this reason, we found that in our case, the FMEA tables were less pertinent in comparison to minimum cuts and FTAs, that are more compact and synthetic. However, whereas generating FMEA is immediate, generating FTAs requires a (somehow quick and easy) configuration. Preliminary to the creation of FTAs, minimum cut sets are first generated. To generate a minimum cut, we specify the failure condition for the top event, the maximum order (of the computation) and a few other parameters such as the type of generation (stochastic, permutation or combination). We can also add to the configuration some constraints representing any conditions we want to check (e.g. 'No single point of failure'). Once the computation is launched, it returns a table (as shown in Figure 8) listing the basic failure events ('Elements' column), the order (number of basic events contributing to the top event) as well as their probabilities.

Lateral_Control_loss

	Elements	Order	Probability
1	Lateral_Ctl.failure	1	1.0E-6
2	Status_IN.failure & HOD.HODStateFinal.failure	2	1.0E-11
3	Fusion.failure & HOD.HODStateFinal.failure	2	1.0E-11

Figure 8. Loss of lateral control minimum cut set

FTAs basically contain the same information as minimum cut sets but have the advantage of presenting them in a more visual and intuitive way. Figure 9 shows an example of FTA. It is constructed from the minimum cut sets of events leading to the ADAS 'Lateral control loss' feature.

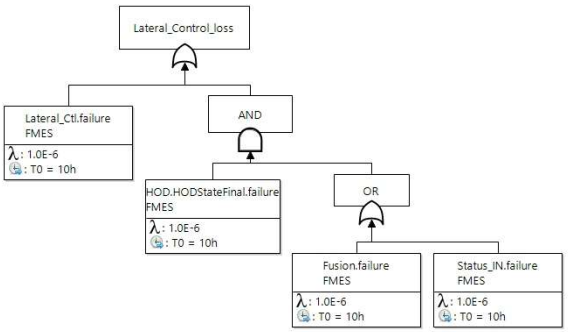

Figure 9. Loss of lateral control status Fault Tree

At the top of the tree stands the top event (Lateral control loss). In the leaves, we see the basic failure events and their probabilities (that are not useful in our case). The FTA graphically shows that the lateral control loss can be caused by the single failure of the lateral control component (Lateral_Ctrl.failure) or a combination of the HOD (HOD.HODStateFinal) with either the Fusion (Fusion.failure) or Statut Input (Status_IN.failure) software component failures. Additionally, the FTA can be useful in capturing inconsistencies in the ASIL decomposition that are not visible through the functional view. For example, the lateral control component is rated ASIL B whereas one of the contributing basic events (Status_IN.failure) in this FTA does not bear any ASIL requirement. So, using FTA can be useful to check if, for a given safety goal, the ASIL of the combination of the triggering event is inconsistent.

This step showed how the classical safety models can be generated from the dysfunctional architecture we built in Step ① and Step ②.

5.3 Case study conclusion

The results from the case study show that using the methodology we propose, it is possible to apply and benefit from a model-based safety analysis approach for software in the automotive industry. It also demonstrates the usefulness and efficiency of using basic and pragmatic tools features such as simulation and failure cause and consequence that are available in tools. Additionally, this approach makes safety analysis more accessible to engineers through a tool-based generation of traditional safety analyses. Overall, the methodology enforces safety by allowing engineers to focus on getting a right model and spending less effort in generating analyses, therefore improving the quality of these automatically generated analyses. The possibility to generate FTA in one click is interesting. Indeed, if the software functional model changes, the dysfunctional model can be updated, and FTAs can be regenerated without additional effort. This means that the dysfunctional model should be maintained. However, given that the dysfunctional model construction and logic translation are done manually, there are some

[8] An AltaRica observer is an indicator that can be associated to failure condition that we want to watch out for.

limitations. First, the manual modeling and logic translation make impractical the application of the methodology to large and complex software systems. This manual modeling approach can certainly be improved to ensure a better consistency of our model with the design model. Secondly, the AltaRica language is system oriented and lacks semantics for certain categories of software failures. It was difficult to express and simulate failures related to timing and values. For example, we were not able to model and observe failures such as the temporary deactivation of lateral control or out-of-range values. In this regard, we consider that the use of AltaRica needs to be reassessed. Nevertheless, the proposed methodology answers our current needs as it is a clear improvement from the manual techniques.

6 PERSPECTIVES AND CONCLUSION

This paper made a methodological proposal for using MBSA for automotive software safety analysis. It consists in building a dysfunctional architecture of the automotive software from the functional high-level software nominal architecture, using compatible semantics and syntaxes, and a translation of a functional to a dysfunctional logic. Then, from this dysfunctional model, the safety engineer can automatically generate minimal cut sets, FTAs and FMEAs for interpretation and as safety proofs if required. We applied this methodology on a real-world industrial case study that involved the use of SimfiaNeo and AltaRica. We concluded on the implementability and usefulness of the methodology. Coupled with the languages and tools we used, it brings added value and improves current manual safety analysis practices. Limitations that we identified will be addressed in future work. This will include improving the dysfunctional modeling for which we will consider a partial automation of the model building and logic translation. We will also explore languages that have better semantics for software failures (AADL EMV2 for example). Finally, we will work on aligning the proposed model-based software safety analysis methodology with model-based software development to ensure better consistency and higher quality of safety analysis.

ACKNOWLEDGEMENTS

This work was sponsored by Renault Software Labs and the ANRT (Association National de la Recherche et de la Technologie). We thank them for their financial support.

REFERENCES

[1] "SEVision2020_20071003_v2_03.pdf." Accessed: Jun. 09, 2020. [Online]. Available: http://www.ccose.org/media/upload/SEVision2020_20071003_v2_03.pdf.

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

[3] van C. Pham, "Model-Based Software Engineering: Methodologies for Model-Code Synchronization in Reactive System Development," phdthesis, Université Paris-Saclay, 2018.

[4] A. Joshi, S. P. Miller, M. Whalen, and M. P. E. Heimdahl, "A proposal for model-based safety analysis," in *24th Digital Avionics Systems Conference*, Oct. 2005, vol. 2, p. 13 pp. Vol. 2-,

[5] D. Kececioglu, *Reliability Engineering Handbook*, 1 edition. Englewood Cliffs, N.J: Prentice Hall, 1991.

[6] 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.

[7] 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.

[8] O. Lisagor, "Failure logic modelling: a pragmatic approach," phd, University of York, 2010.

[9] Y. Papadopoulos and J. A. McDermid, "Hierarchically Performed Hazard Origin and Propagation Studies," in *Computer Safety, Reliability and Security*, Sep. 1999, pp. 139–152.

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

[11] Y. Papadopoulos and M. Maruhn, "Model-based synthesis of fault trees from Matlab-Simulink models," in *2001 International Conference on Dependable Systems and Networks*, Jul. 2001, pp. 77–82.

[12] J. Delange and P. Feiler, "Architecture Fault Modeling with the AADL Error-Model Annex," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, Aug. 2014, pp. 361–368.

[13] 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.

[14] 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.

[15] 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.

[16] P. H. Feiler, B. Lewis, S. Vestal, and E. Colbert, "An Overview of the SAE Architecture Analysis & Design Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering," in *Architecture Description Languages*, Boston, MA, 2005, pp. 3–15.

[17] H. Blom *et al.*, "EAST-ADL: An Architecture Description Language for Automotive Software-intensive Systems in the Light of Recent use and Research," *Int. J. Syst. Dyn. Appl.*, vol. 5, no. 3, pp. 1–20, Jul. 2016.

[19]"AltaRica Association." http://www.altarica-association.org/ (accessed May 25, 2020).
[20] M. Bozzano *et al.*, "Symbolic Model Checking and Safety Assessment of Altarica models," vol. 35, p. 16, 2010.
[21] M. Boiteau, Y. Dutuit, and A. Rauzy, "The AltaRica Data-Flow Language in Use: Modeling of Production Availability of a MultiStates System," p. 22.
[23]M. Machin, L. Sagaspe, and X. de Bossoreille, "SimfiaNeo, Complex Systems, yet Simple Safety," p. 4.
[24] "OpenAltaRica." https://www.openaltarica.fr/ (accessed May 27, 2020).
[25] "AltaRica Project | MEthods and Tools for AltaRica Language." https://altarica.labri.fr/wp/
[26] "Architecture Analysis and Design Language." https://www.sei.cmu.edu/research-capabilities/all-work/display.cfm?customel_datapageid_4050=191439.
[27] "Welcome to OSATE — OSATE 2.7.1 documentation." https://osate.org/# (accessed May 25, 2020).
[28] "EAST-ADL Association." https://www.east-adl.info/Specification.html (accessed Apr. 16, 2020).
[29] B. Bittner *et al.*, "The xSAP Safety Analysis Platform," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2016, vol. 9636, pp. 533–539.
[30] 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.
[31] N. M. Inc, "MagicDraw." https://www.nomagic.com/products/magicdraw (accessed May 27, 2020).
[32] "Automotive_SPICE_PAM_30.pdf." Available: http://www.automotivespice.com/fileadmin/software-download/Automotive_SPICE_PAM_30.pdf.
[33] 14:00-17:00, "ISO 26262-1:2018," *ISO*. https://www.iso.org/cms/render/live/en/sites/iso-org/contents/data/standard/06/83/68383.html (accessed Jun. 10, 2020).
[34] A. Joshi, M. Whalen, and M. P. E. Heimdahl, "ModelBased Safety Analysis: Final Report," 2005.