Software fault propagation patterns for model-based safety assessment in autonomous cars

Yandika Sirgabsou, Claude Baron, Laurent Pahun, Philippe Esteban

Software fault propagation patterns for model-based safety assessment in autonomous cars

Yandika SIRGABSOU[1*], Claude. BARON[1] , Laurent PAHUN[2], Phillipe ESTEBAN[1]

[1]LAAS-CNRS, Toulouse, France
[2]Renault Software Factory, Toulouse, France

*Yandika.sirgabsou@laas.fr

Keywords:
MBSA, MBSE, Safety analysis, Embedded software, Fault propagation pattern

Abstract:
The development of driver assistance and autonomous driving systems for vehicles has started to revolutionize the transportation sector, promising comfort, and safety. While significant technological progress has already been made in this area, many challenges remain. Among these challenges, ensuring safety has become even more critical due to the increasing use of complex, communicating, and reconfigurable embedded software. Current solutions to address safety include the use of model-based approaches for safety analyses instead of the traditional document-based safety analysis that is both informal and inefficient when faced with complexity. To this end, and in the context of automotive embedded software, we propose to rely on the use of fault patterns to improve the construction of software models used to conduct safety analyses. This paper makes a methodological proposal that improves current practices in terms of facilitated model construction and reusability, and that has been validated on the study of an automotive software component.

I. Introduction

The rapid development of embedded systems has led to numerous innovations in various systems in our modern society such as autonomous vehicles and highly computerized systems in airplanes. The technological challenges related with complexity and societal needs for guaranteeing safety induced by this trend opened new avenues for research in systems engineering but also exacerbated existing problems as they relate to the use of critical software and its contribution to systems safety.

To cope with these issues, industrials developing safety critical systems are looking for new methods and tools for designing and sharing design ideas more efficiently while ensuring system safety as required by standards and regulations. In the past decades, most of systems and software engineering development processes relied on document-based methods that relied on informal design documents to convey design ideas and artefacts from one development stage to the other. The informal aspect of these practices (such as manual analysis based on informal documents that are subject to the interpretation of the safety analyst) makes them prone to errors and less efficient in regard of the complexity of today's systems architectures. Although they are still widely used, these methods are now being challenged and model-based are more and more favored. In this context, embedded systems manufacturers are turning towards model driven engineering as part of both their systems and software development as well as systems safety assessment. In Systems Engineering (SE), this had led to the adoption of MBSE (Model-Based Systems Engineering), a systems engineering practice aimed at describing both a problem (need) and its solution through models, concepts and languages [1]. Its adoption can now be considered a success story as we witness that more industrials developing safety critical systems are turning towards the MBSE approach in Systems and Software Engineering. Examples include Dassault with its integrated 3DS MBSE solution or SIEMENS that integrates MBSE within its Product Lifecycle management (PLM) solution. In Systems safety, a similar trend has led to the development of Model based Safety Assessment (MBSA), a practice that enables the capture, through specific formalisms and languages, of a systems safety related model (that describes the failure behavior and unifies all the safety property of a system in a single model), on the basis of which different safety analyses can be made.

However, despite the discipline being an early pioneer in the use of models, the wide adoption of model-based approaches for safety assessment has remained embryonic. In automotive, the ISO 26262 standard [2], titled "Road vehicles – Functional safety", requires performing safety analyses not only at system level but also at software level,

to ensure the safe behavior of the embedded software. Moreover, in the context of autonomous driving, embedded software assumes various critical safety functionalities. Unfortunately, today, the currents practices in safety analyses do not focus enough on embedded software even though the software implements the logic of some of the critical safety mechanisms. As a result, in the software context, safety analyses are either not performed or if performed, only done through traditional document-oriented approaches. Therefore, there is a need for more focused and rigorous methods for safety analyses at software level.

This paper aims to propose a methodology, specially aimed at improving the practice of automotive embedded software safety analysis thought the use of fault patterns within the MBSA approach. In the next section, a state of the art of current MBSA approaches is presented. In section 3, a methodological proposal based on the use of fault pattern is made and applied to a case study in section 4. The results are discussed in section 5 and a conclusion in made in section 6.

II. State of the art

In systems safety, safety assessment has been dominated by document-centric methods and processes since the 60s. Classical methods such as Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) have been complementarily used by experts in the industry for systems safety analyses. In these classical approaches, safety analyses are manually performed based on paper-style artefacts such as systems drawings or spreadsheets that are found in design documents. While these practices use a sort of well-defined semantics (such as Boolean operators and FTA symbols), their representation is often very far from the systems they describe.

With the introduction of Model-Based Safety Analysis (MBSA) starting around early 90s, through the earliest model-oriented safety analysis techniques such as FPTN [3], Figaro [4] or the AltaRica [5] language, the focus has shifted to model-based approaches that no more base safety analyses on paper-style documents, but on a formal model of the system under design. This move has led, nowadays to an academic trend that seeks to address the interdisciplinarity and the consistency of the use of broad models through MBSE and MBSA throughout Systems Engineering and Systems Safety. In this trend one possibility is to directly associate the MBSA safety models with the MBSE system models. This allows conducting safety analysis directly on an extended version of the MBSE model. Examples of methods that are based on this extended approach include the approach included in the xSAP/Nu-SVM [6] [7] safety analysis tool. Another example is the Hierarchically Performed Hazard Origin Studies [8] based on extended SIMULINK models [9]. Although a clear advantage of the extended approach resides in the consistency (between system design and safety) it enables through the use of a shared system and safety model, it is argued that basing safety analysis on an extended model can lead to false assumptions and thus leading to hidden safety flaws (despite this claim not being shared by all systems engineering and safety practitioners). Furthermore, the safety analysis resulting from such model could be difficult to exploit because of the complexity of the extended model. In practice, safety assessment models are primarily used for mathematical or probabilistic calculations such as minimal cuts or monté carlo simulations. The more complex the model, the more computing power is required to effectively perform these calculations. Furthermore, the result of the safety analysis can be difficult to exploit due to the source models being blurred and overloaded (making them lose their ability to support a seamless communication) while the generated formal models are incomplete and uselessly complex [10].

As an alternative to the described first approach, a more appealing trend in academic research is to build a separate MBSA model that needs to be kept consistent with the MBSE model through additional measures such as model synchronization [11]. This last approach is often based on dedicated safety modeling languages such as AltaRica [12], Figaro [4] or SAML [13]. These safety focused languages allow unambiguous representation of systems for the needs of safety analysis using well defined syntax and semantics. However, early experiments feedback suggests that, in the case of the dedicated model approach, the MBSA model construction can be challenging for safety analysts especially for complex systems. In such case, it is imperative to find the right level of details in modeling for fear of having a model too complex that may well be at the limit useless or irrelevant. Furthermore, some analysts don't necessarily see the advantage and gain in time of modeling a separate dysfunctional architecture for safety analyses. However, as argued by Rauzy and Haskins in [10], systems description and safety analysis models are different by nature and efforts to unify such models in one single super model remains unrealistic. Based on this argument, the right direction is to keep a consistent separate MBSA model but make its construction easier. Nevertheless, the separate model construction can be challenging for large systems (what details, what modeling strategy etc.).

To ease the construction of dysfunctional models to conduct safety assessment, efforts have been done mostly using generic libraries of system elements that exhibit some safety properties. An example is the Safety Architecture Pattern(SAP) approach proposed by Kheren in [14]. In this approach, a library of SAP (components that highlights useful system's attributes from a safety point of view) are developed and coded using the AltaRica language. The

generic library is then reused to easily prototype safety-oriented systems architectures that can be reused to perform safety analysis using tools such as the OCAS Workshop[15]. Nevertheless, the proposed approaches are mostly aeronautic systems oriented. The developed libraries are often dedicated to avionics systems (such as pumps, electrical motors, valves, or control units). While these libraries can be used for modeling physical systems at system level in automobile, they are less suitable for modeling dedicated safety architectures of the embedded software. Moreover, although there are ongoing works that aim to apply the MBSA approach to automobile at system level, less focus is being put on embedded software. However, in the automotive context, ISO 26262 recommends conducting safety analyses not only at system level, but also at software level[2]. In this context, if the model-based approach is to be used for safety analysis at software architecture level as the commended by ISO26262, safety analyses can be made easier if patterns or libraries of safety related components (such as safety mechanisms used in software) can be developed drawing from the same principles as those described in the case of systems SAP-oriented approach.

III. Methodological proposal

The goal of the methodological proposal is to construct a fault library of reusable software safety mechanisms that are commonly found in safety related software components, drawing form the SAP library-oriented approach described in the state of the art and given the need for improving safety analysis practices at software level in the automotive context. Our choice has been to use the dedicated model approach coupled with dedicated languages such as AltaRica as described in [16]. However, as stated earlier, building a dysfunctional model can be challenging especially for complex systems. To address this concern, our solution has been to focus on selecting and including in the dysfunctional model only components that are safety-related as described in our previous work [17]. Even then, the failure propagation logic of these components must be manually written by the modeler (which can be time consuming for large systems).To address this concern, our first hypothesis is that making the MBSA model construction easier can both benefit its adoption by companies and improve the quality of safety analysis. A second hypothesis is that limiting the MBSA model to safety related components is sufficient to carry out meaningful safety analysis and can improve both efficiency and the quality of safety analysis. Therefore, the position of this paper is to make less painful for safety analysist the construction of MBSA models by proposing a set of predefined reusable libraries of software fault models to ease the dysfunctional model building process and improve the quality and consistency of the analyses. Its scope is limited to the context of automotive safety analysis at software level consistently with ISO 26262. Our methodology proceeds in 3 basic steps. The first one is to identify the safety mechanisms and their related software failure modes. The second step is to write the failure propagation logic through the safety mechanisms based on their functions and the identified failure modes; and to store the components in a library. The last step consists in reusing the elements stored in the library to build a dysfunctional model and conduct safety analyses.

In the first step we proceed by identifying the software failure modes and associated safety mechanisms found in automotive software architectures. In automobile, these failure modes and associated safety mechanisms are well defined by IO 26262 [13, Annex E, Annex D]. They are further detailed by two annexes in AUTOSAR [18] [19]. These failure modes are clustered into 4 categories. They include “data integrity, initialization & configuration data” “data exchange”, “timing & control flow” and “data processing”. The “data Integrity” category summarizes all failure modes related to corrupted memory or initialization and configuration data related to the corruption of software data at one memory address such as the corruption of memory content, memory partitioning fault or memory access fault. The “Data Exchange” category covers failures related to data transmission between sender and receiver such as between different ECUs (Electronic Control Units) or software components. The “Timing and Control Flow” category covers failure modes related to the timing of execution and scheduling. To model these fault categories, we start by describing the software components’ behavior through a generic abstract template using states and transitions. An example of such a template consisting of a software component with generic failure modes is provided in in Figure 1. It shows 4 states (Inactive, Nominal, Erroneous and Failed), representing the execution status of the software component linked by several possible transitions (Minor fault, Major fault, Recovery etc.). The inactive state is the idle or initial state preceding the initialization where the software component is not solicited or does not provide its function. From this state, the 3 outgoing transitions labeled “Activation”, “Major fault” and “Minor fault” will lead the component to the “Nominal”, “Failed”, or “Erroneous” states respectively. In the nominal state, the software component executes and delivers its intended function. From this state, the component can revert to the Inactive state as indicated by the transition label “Deactivation” or move to the “Failed” or “Erroneous” states if a major or minor fault occurs as indicated by the transitions. In the erroneous state, the software component provides erroneous results while, in the “Failed” state, it fails to provide its intended function. From the generic pattern, more specific patterns related to specific fault categories as described by ISO 26262 can be easily derived.

As an illustrative example, let us consider a piece of software that reads some critical data from memory, performs some critical calculations, and stores the result back to memory. Based on its function, the failure modes of this software component could include memory access related faults (such as read and write errors) as well control flow and timing related failure modes (such as execution failure, or untimely execution). Depending on the exact safety requirement, safety mechanisms to prevent the failure of this software component's function could include a protection against unwanted writing and a watchdog timer. Based on this information, the elements that will be necessary to model the safety related behavior of this software component, are the failure mode related to the memory access and the two safety mechanisms that will be modeled though states and transitions within this software component.

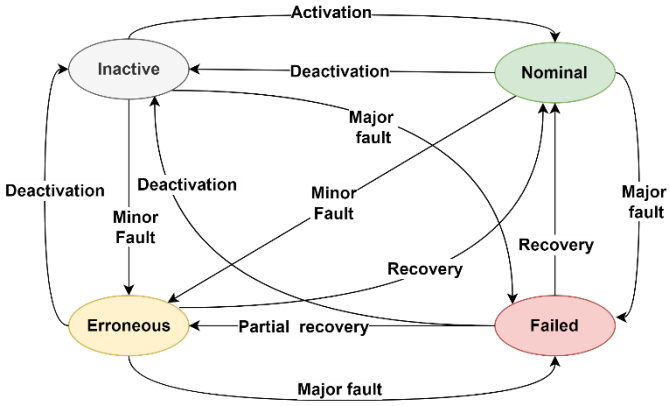

Figure 1. Failure modes of a generic software component

The second step focuses on writing failure propagation logic of the identified safety mechanisms knowing their function and considering the associated failure modes identified in the first step. To do so, we first need to study the safety mechanism and identify their basic behavior, what their inputs are, and the result they produce in normal or faulty execution. Once done, we can write the failure propagation logic of the identified safety mechanisms. To this end we use Failure Truth Tables (FTT) that we introduced in our early work [16]. FTTs are dysfunctional failure propagation tables consisting of discrete input and output variables whose possible values are defined depending of the failure behavior of the components function. FTTs can be used to capture the dysfunctional logic of a function based on its inputs. Depending on the expression of the safety requirements that the identified mechanisms are to fulfill, logical operators such as "or, and" and program control structures such as "if-then-else" can be useful to express the function or the combination of several mechanisms within a software component. Therefore, we also need to write and add to the library, the failure propagation logic through theses operators and control structures. This will result in a library of safety mechanism, operators and control structures that will be used in the last step to construct the dysfunctional model of a system.

The last step consists in using the elements stored in the library to construct the dysfunctional model. This step requires having a tool that offers library support. Using the elements stored in the library in an appropriate MBSA tool, one can easily model the dysfunctional architecture of a systems through drag and drop. This is possible since the safety mechanisms self-contain their propagation logics as well as the associated fault modes. Therefore, there is no need to write the failure propagation logic code in this step.

IV. Implementation in SimfiaNeo and case study

The objective of the case study was to first develop a library of safety mechanisms and used it to model the dysfunctional architectures. To this end, we are using SimfiaNeo [20] , an MBSA modeling tool based on the AltaRica language and 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 perform various safety analyses based on minimal cuts or in the form fault trees and FMEAs directly from the AltaRica model. Furthermore, the tool in its latest version offers library features that make it suitable for our proposal. It allows the modeling, storing and instantiation of custom components in a library.

In the context of Advanced Driver Assistance Systems (ADAS) and Autonomous Driving (AD), we aim to apply the proposed methodology on a practical case study, the longitudinal control software component. The longitudinal control is a function of the ADAS technology. 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 monitor (a subcomponent called supervisor that manages its states) and a failsafe controller (a subcomponent that places the longitudinal control in some predefined safe states when some faults occur). Safety requirements associated with the longitudinal control are documented in a safety concept through safety goals that are declined to Unwanted Software Events (UWSE) at the software level. In this case study, we focus on one single UWSE related to the occurrence of an unintentional acceleration above the permissible acceleration limit (defined by ISO 22179) during travel ($v \neq 0$ km/h) entailing longitudinal control. Other constraints also exist (e.g., the user must be able to deactivate the ACC at all times).

1. Pattern prototyping and dysfunctional model construction with SimfiaNeo

Using our described approach, we modeled the basic structures of software components using the previously described fault categories and safety mechanisms. First, we declared the necessary AltaRica domains in SimfiaNeo. To this end, we declare 5 domains with different literals encompassing data integrity, Data exchange, Safety mechanism, generic state, and generic data. Using these 5 domains we modeled bricks of components representing the elements of a software safety architecture including elements such as generic software components, generic safety mechanisms, Boolean operators, and program control structures.

Description the fault patterns

In automobile, software fault modes and associated safety mechanisms are clustered into 4 categories. Including include “data integrity, initialization & configuration data” “data exchange”, “timing & control flow” and “data processing”. In this subsection we present a fault pattern related to the data exchange as an illustrative example and a generic behavioral pattern for safety mechanisms.

As described by ISO 26262 in the software scope, the “Data Exchange” category covers failure modes related data transmission between sender and receiver such as between different ECUs or software components. The pattern presented on Figure 2 shows how the execution of a receiving software component subjected to this category can be affected. Like the pattern shown in Figure 1, it has 4 states: “Init,” “Nominal”, “Erroneous”, and “Failed”. From the initial state (blue state on Figure 2), the function will either move to the “Nominal” or “Erroneous” states as indicated by the outgoing transitions depending on a successful or unsuccessful data transmission initialization. In the nominal sate, the software component executes normally and fulfils its function. If a data transmission initialization fault has led the function to the “Erroneous” state, an execution of a safety mechanism can bring the function to nominal if successful or to the “Failed” state (red state on Figure 2) if unsuccessful. The function can also move to from the nominal state to the “Erroneous” state with the occurrence of inconsistencies of the transmitted data (such as corruption, incorrect data value, out of range data values or incorrect sequence of data). In the “Failed” state, the software component fails to provide the expected function due to missing or loss of transmitted data or due to the safety mechanism failure to recover from the “Erroneous” state. As it can be seen in Figure 2, this pattern is built on the generic pattern presented in Figure 1. However, it differs by the specificity of its failure modes expressed in the transitions that are specific to “Data Exchange” fault category. Based on this pattern, another related pattern was derived to cover the specificity of other data exchange related faults such as delayed data transmission that will cause the function’s execution to be delayed. It such case, the “Erroneous” state was further split into several states depending on the specificity of the software component.

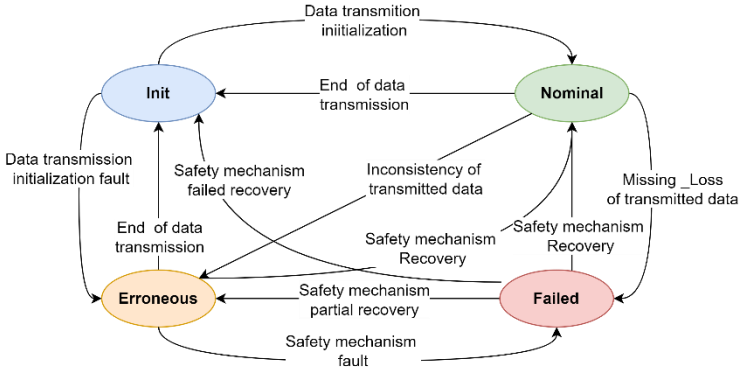

Figure 2. Data exchange fault pattern

The next pattern aims to capture the behavior of a generic software safety mechanism. The 4 states (Nominal Inactive, Nominal Active, Misleading and Failed) represent the execution state of the safety mechanism. In the “Nominal inactive” state, the safety mechanism is in its nominal execution state with no fault detected. When a fault

is detected, it moves to the “Nominal active” state. In this state, the safety mechanism is successful in reacting and correcting the effect of the fault. From this states, an erroneous or failed reaction will lead the safety mechanism to “Misleading” or “Failed” states respectively.

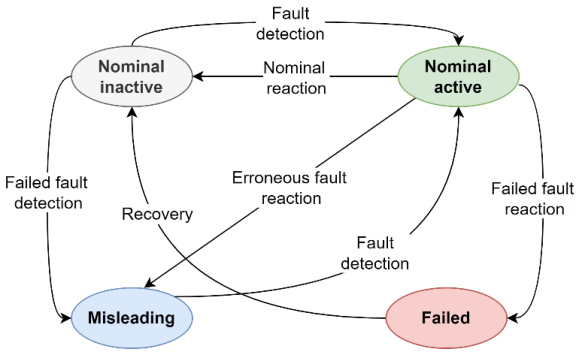

Figure 3. Generic safety mechanism fault pattern

The behavior of the safety mechanism is completed with writing failure propagation logic knowing their function and considering the associated fault modes identified earlier. To do so, we first need to study the safety mechanism’s function and identify their basic behavior, what their inputs are, and the result they produce in normal or faulty execution. Once done, we can write the failure propagation logic of the identified safety mechanisms. Depending on the expression of the safety requirements that the identified mechanisms are to fulfill, logical operators such as “or, and” and program control structures such as “if-then-else” can be useful to express the function or the combination of several mechanisms within a software component. While these operators are already part of the AltaRica semantics and can be expressed in assertions, modeling them in virtual bricks allows to graphically associate the components without rewriting the propagation logic. Therefore, we also need to write and add to the library, the failure propagation logic through theses operators and control structures. This will result in a library of safety mechanism, operators and control structures that will be use in the last step to construct the dysfunctional model of a system.

Modeling

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 Technical Safety Concept (TSC) resulting from the system level safety assessments. 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. Based on the TSC and the definition of the items, we can identify the safety-relevant 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.

We used the readily available model bricks to model the patterns and their states in the tool, assigning the previously created domains to them. An overview of the modeled system along with the fault patterns is presented in Figure 4. Depending on the function of the pattern, different domains were used. Through the creation of AltaRica events in SimfiaNeo, we then modeled the transition firing conditions using the events identified in the state machines previously presented in the methodological proposal section. An AltaRica transition is characterized by a guard (condition to fulfill before triggering the effect), an effect (action resulting from the state change), and potentially a distribution (exponential, Dirac etc.) associated to the event. For each event, the SimfiaNeo tool allows to specify a probability that will be used during calculations. However, given in the context of software faults, such values are irrelevant and a probability of 1 was used instead. For each pattern, we wrote fault propagation logic linking the corresponding inputs (if they exist), internal states and outputs. We described the states of the inputs and outputs using the AltaRica domain that we named “Generic Data” and that incorporates four states: Nominal, Lost, Delayed, Erroneous. Using these states, we were then able to model the dysfunctional information flows between components trough AltaRica expressions. Using the elements stored in the library in an appropriate MBSA tool, one can easily model the dysfunction architecture of a systems through drag and drop. This is possible since the safety

mechanisms self-contain their propagation logics as well as the associated fault modes. Therefore, there is no need to write the failure propagation logic code in this step. We constructed the model presented on Figure 4 using the elements stored in the library as shown by the library icon in some of the components (e.g., component identified as ETH at the bottom left of the models). As an examples the components labeled CAN and ETH in Figure 4 were both modeled thorough an instantiation of the data exchange fault pattern described earlier as well as some internal components in the "vehicle-status-input" component that receive these data. Similarly, the "memory" component shown on Figure 4 as well as some subcomponents in the "longitudinal controller" component that read data from the memory were modeled though the instantiation of the data integrity fault pattern. Through these reusable libraries, we are able to model the dysfunctional behavior and failure propagation without having to manually rewrite their AltaRica code.

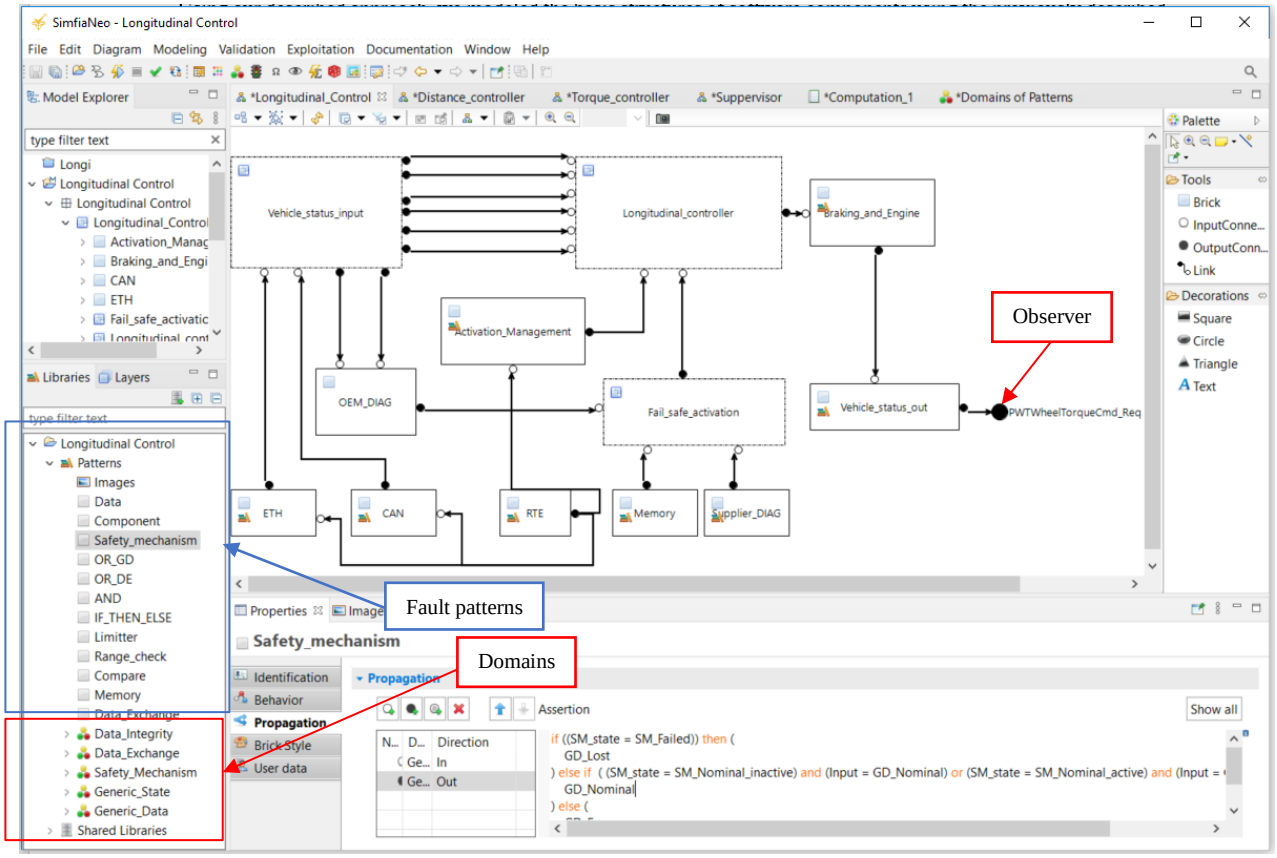

Figure 4. Pattern prototyping and dysfunctional model construction with SimfiaNeo

2. Safety Analyses

Using the SimfiaNeo Tool, we performed various safety analyses including step by step simulations, minimal cuts, FTA and FMEA based on the dysfunctional model constructed using the previously developed fault patterns. Having completed the modeling of the longitudinal control software component and the components that interact with it, 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 4). An AltaRica observer is an indicator that can be associated with a failure condition or feared event that we wish to capture. For example, let us consider, the UWSE that we have chosen for our case study related to an "unintended Acceleration > ISO 22179 acceleration limit while travelling (v $\neq$ 0 km/h) requested by the longitudinal control feature". In our model, we identified that the acceleration target and request in the speed controller subcomponent (Speed-Ctrl) are limited to 0.2G until vehicle speed vehicle is above 10 km/h. We also identified that the final value of the acceleration target is transmitted to the engine through the engine management command 'PWTWheelTorqueCmd' (Powertrain Wheel Torque Command). Thus, any erroneous value of 'PWTWheelTorqueCmd' can result in the violation of the safety goal and the occurrence of the UWSE. Therefore, the observers predicate to capture the occurrence of this UWSE can simply be 'PWTWheelTorqueCmd =Erroneous'. Having added the expression of the observer to the model, the objective was to verify, by means of the simulation, FMEAs, minimal cuts and fault trees, whether we could

determine the events or failures that could lead to the transmission of this erroneous command that can result in the violation of the chosen UWSE.

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: in nominal state), as shown in Figure 5. However, hierarchical components (consisting of blocks containing several subcomponents) 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.

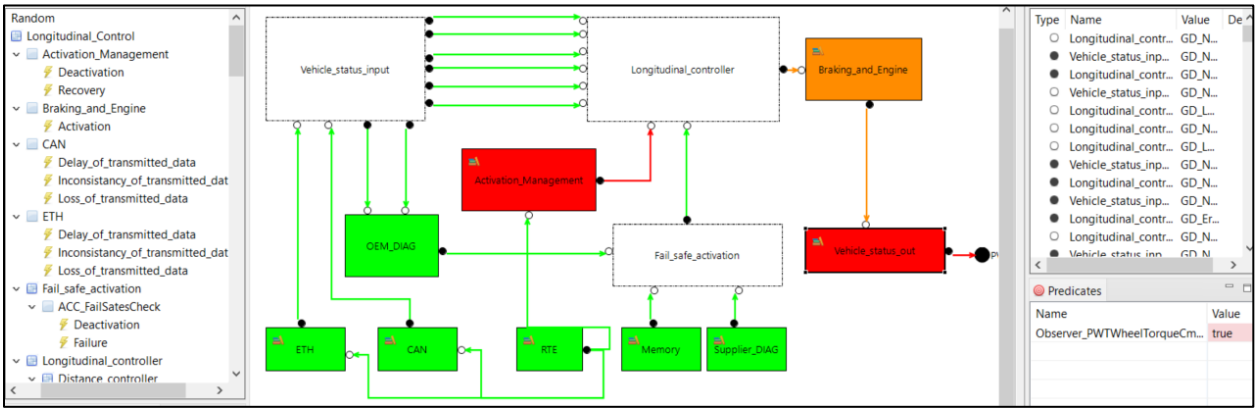

Figure 5. Simulation

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. Figure 6 shows an excerpt from an FMEA, containing a certain number of elements typically found in these tables. The first column (Event) lists the events causing the violation. In the following columns, we can find the Local Effect (effect of the event on the output of the initial component), the Intermediate Effect (effect of the event on all intermediate components between the initial component and the final observer), and the Final Effect (effect on the output of the model; in here, the effect on the observers described previously). For instance, in relation to our chosen UWSE, the excerpt from Figure 6 shows how faults in the vehicle status input component related to vehicle speed can affect other components and the final observer.

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

Figure 6. FMEA

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 Figure 6 represents only a very small excerpt from the initial FMEA, which has more than 18,000 lines. Therefore, if the goal is to obtain usable details or to make a synthesis, this representation is not ideal.

To 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 can be plethoric. Despite the absence of a comparison with a manually performed FMEA, we argue that our approach allows performing this type of analysis that is otherwise difficult to perform manually.

Minimal cut set and fault tree

The generation of the minimal cuts is achieved through the configuration of the cut set and sequence calculation engine for a given observer. Thus, for an observer we can choose the maximum order of the cuts, the filter type (minimal cut or minimal sequence) and the generation type (combination, permutation or stochastic) which will be used during the cut set or sequence calculations. The maximum order corresponds to the maximum number of primary events in a sequence. To choose the maximum order, we experimented with values ranging from 2 to 5. We observed that SimfiaNeo load on the processor and memory consumption remained relatively constant (respectively close to 30% and 700 Megabyte) regardless of the maximum order value, while the computation time increased exponentially from under 2 minutes for order 2 to 7 hours for order 5. Meanwhile the maximum order in the resulting minimal cut set remained equal to 3 even if we consider sequences of size 4 or 5. We chose order 3 for our case study—the higher the order, the longer the generation of the cut will take. An order of 3 is therefore a good tradeoff between computation time and accuracy. The choice of the filter type is also important; we have chosen the "minimal cuts" option since it makes fault tree generation possible. Lastly, the choice of the generation type specifies the combinatorial or stochastic sequence (based on random simulations) used during the generation of the cut.

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

Figure 7. Minimal cut

As an example, we considered the chosen UWSE linked to the transmission of an erroneous torque command to the engine. For this UWSE, we generated a minimal cut by choosing "order 3" as value of the maximum order, the "minimal cut" filter and "permutation" as the generation type. The generated minimal cut is shown in Figure 7. It shows combinations (of order 1, 2) of basic events that could cause the specified UWSE, as well as their associated probabilities (added by default). We can see that the cut highlights the events and the hierarchical components, enabling traceability of the components at high level (as shown in Figure 7). For dysfunctional models where several subcomponents have identical nomenclature, this traceability allows to clearly identify the origin of each event.

During the execution of these calculations, however, several compilation errors occurred, some of them due to the presence of loops in the failure propagation chain. This is a known issue related to the dataflow version of the AltaRica language. To solve this problem, we modified the assertions of the failure propagation involved in these loops. In the case of a redundant evaluation of a variable (where one of the assertion is part of the loop), removing the redundant evaluations of the involved variable allowed to break the loop. For this purpose, we considered a loop and identified the self-dependent variables in the chain of assertions constituting this loop. One possibility was to remove this variable from one of the assertions if it was already considered in another assertion. If this was not possible without modifying the validity of the assertion, the second possibility was to remove the dependency link

and successively assign to the state variable all possible values and perform the calculations with each scenario. In the latter case, it was necessary to manually change the value of the variable to include the scenarios which were excluded by assigning it a fixed value. In both cases, the dysfunctional logic of the assertion remains valid.

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

V. Discussion

The results obtained from the application to the case study shows that using fault patterns and the adequate tool, the dysfunctional model construction is made easy and safety analyses can benefit from this alternative new analysis method. The specificity of the case study demonstrates that the use of software-oriented fault patterns can benefit the application of MBSA in the automotive software safety context. Furthermore, tools such as SimfiaNeo relieve the safety expert of manual calculations while allowing him to concentrate on modeling. The benefits are reflected in terms of reusability of the models. Once the fault patterns are built, they can be reused to build the dysfunctional model and make it possible to conduct analyses with different parameters for many UWSEs based on the same model. In the traditional approach, the safety analyst spends much of their effort interpreting various design documents to manually construct classical model's safety models such as FTAs or FMEAs. The analyst would need to manually construct as many FTAs as there are feared events. If the design evolves, they will need to individually update all the fault trees and the FMEAs. Through our proposal, the dysfunctional model is easy to construct. If the system design evolves, the dysfunctional model can be updated, and updated safety analyses 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 make this safety analysis method an interesting alternative that has the potential not only improve current practices but to contribute to the adoption of the model-based approach. Nevertheless, we noted some limitations on the method. 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 and is not addressed by our proposal. Implementing additional measures is therefore necessary to guarantee consistency. Another limitation, related to the AltaRica dataflow, is the inability to natively manage loops; our solution was to modify some assertions in order to remove these loops. Finally, as the case study involves a system of reduced complexity, the scaling up of our methodological proposal is yet to be evaluated and we will need to perform a comparison with safety analyses results obtained through the traditional manual approach to fully evaluate the proposed approach.

VI. Conclusion

This paper made a methodological proposal based on fault patterns that can be used to build a dysfunctional model, and from which it is possible to derive classic safety models. Using the SimfiaNeo tool and the AltaRica language, the methodology was applied on a case study, building a dysfunctional model of a software from which we were able to generate FMEAs and minimal cuts. These results are encouraging and demonstrate the usefulness of patterns to facilitate MBSA model construction. More generally, they show that it is possible to apply an MBSA approach to evaluate software safety, especially in automotive applications. They also highlight the benefits of generating safety analyses from a dysfunctional model (time saving and reusability). 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.

Reference

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

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

[3] 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.

[4] 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.

[5] A. Arnold, G. Point, A. Griffault, and A. Rauzy, “The AltaRica Formalism for Describing Concurrent Systems,” *Fundam. Informaticae*, vol. 40, no. 2,3, pp. 109–124, 1999, doi: 10.3233/FI-1999-402302.

[6] M. Bozzano and A. Villafiorita, “Improving System Reliability via Model Checking: The FSAP/NuSMV-SA Safety Analysis Platform,” in *Computer Safety, Reliability, and Security*, vol. 2788, S. Anderson, M. Felici, and B. Littlewood, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 49–62. doi: 10.1007/978-3-540-39878-3_5.

[7] B. Bittner *et al.*, “The xSAP Safety Analysis Platform,” in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 9636, M. Chechik and J.-F. Raskin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 533–539. doi: 10.1007/978-3-662-49674-9_31.

[8] 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.

[9] N. Jiang, G. Li, and B. Liu, “Model-based safety analyses of embedded system using stateflow,” in *2016 11th International Conference on Reliability, Maintainability and Safety (ICRMS)*, Oct. 2016, pp. 1–6. doi: 10.1109/ICRMS.2016.8050084.

[10] 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.

[11] A. Legendre, A. Lanusse, and A. Rauzy, “Toward Model Synchronization Between Safety Analysis and System Architecture Design in Industrial Contexts,” in *Model-Based Safety and Assessment - 5th International Symposium, IMBSA 2017, Trento, Italy, September 11-13, 2017, Proceedings*, 2017, vol. 10437, pp. 35–49. doi: 10.1007/978-3-319-64119-5_3.

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

[13] 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.

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

[15] 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.

[16] Y. Sirgabsou, C. Baron, C. Bonnard, L. Pahun, L. Grenier, and P. Esteban, “Investigating the use of a model-based approach to assess automotive embedded software safety,” presented at the 13th International Conference on Modeling, Optimization and Simulation (MOSIM20), Nov. 2020. Accessed: Feb. 01, 2022. [Online]. Available: https://hal.laas.fr/hal-02942695

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

[18] “Overview of Functional Safety Measures in AUTOSAR.” [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/43/AUTOSAR_EXP_FunctionalSafetyMeasures.pdf

[19] “Safety Use Case Example.” [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_SafetyUseCase.pdf

[20] M. Machin, L. Sagaspe, and X. de Bossoreille, “SimfiaNeo, Complex Systems, yet Simple Safety,” in *9th European Congress on Embedded Real Time Software and Systems*, 2018, p. 4.