Erschienen in: Formal Methods for Industrial Critical Systems : 23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3-4, 2018, Proceedings / Howar, Falk; Barnat, Jiří (Hrsg.). - Cham : Springer, 2018. - (Lecture Notes in Computer Science ; 11119). - S. 35-51
https://dx.doi.org/10.1007/978-3-030-00244-2_3

Automated Functional Safety Analysis of Automated Driving Systems

Martin Kölbl(✉) and Stefan Leue(✉)

University of Konstanz, Konstanz, Germany
{martin.koelbl,stefan.leue}@uni.kn

Abstract. In this paper, we present a method to assess functional safety of architectures for Automated Driving Systems (ADS). The ISO 26262 standard defines requirements and processes in support of achieving functional safety of passenger vehicles, but does not address in particular autonomous driving functions. Autonomous driving will bring with it a number of fundamental changes affecting functional safety. First, there will no longer be a driver capable of controlling the vehicle in case of a failure of the ADS. Second, the hardware and software architectures will become more complex and flexible than those used for conventional vehicles. We present an automated method to assert functional safety of ADS systems in the spirit of ISO 26262 in light of these changes. The approach is model-based and implemented in the QuantUM analysis tool. We illustrate its use in functional safety analysis using a proposed practical ADS architecture and address, in particular, architectural variant analysis.

1 Introduction

The functional safety of software-driven functions in passenger vehicles is currently the subject of the ISO 26262 [8] international standard. It specifies development processes and requirements ensuring functional safety of software defined safety-critical functions, also referred to as items, in automobiles. The ISO 26262 standard focuses primarily on the safety of the software-defined items in the presence of systematic software and random hardware faults.

The advent of assisted and autonomous driving is fundamentally changing the architecture of software-defined critical automotive systems. As a consequence the methodological foundations of asserting functional safety of such systems will have to be redeveloped. The current version of the ISO 26262 standard, as well as the current proposed revision on this standard [9], do not account for the functional safety of autonomous driving functions.

First, the development of autonomous driving systems (ADS) will at some point lead to vehicles in which a human driver will no longer be available to take over control of the vehicle. Following the classification in the SAE J3016 standard [20], this will be the case starting at level 4. Whereas classical functional safety approaches follow a *fail-safe* approach, which in case of a failure relies on a driver being able to take over control of the vehicle and bring it into a safe state, ADS systems have to be designed to operate in a *fail-operational* manner. This means that in the presence of the failure of some ADS function, the overall vehicle system will remain operational for a certain period of time, with a given probability, in order to navigate the vehicle automatically into a safe location, for instance the shoulder [24]. This is frequently also referred to as "limp-home" mode. For the analysis of functional safety properties this means that the availability of these limp-home mode functions in the presence of a system failure needs to be proven.

Second, the conventional approach to functional safety, as reflected by the current ISO 26262 standard, is highly "item-oriented". This means in particular that one driving function, or item, is implemented by one software component executing exclusively on one hardware unit, referred to as electronic control unit (ECU). Current systems already break with this strict concept and run a low number of functions on a single ECU. However, safety arguments largely rely on execution in isolation, with the exception that some degree of freedom of interference, including that caused by concurrency problems, at the level of the underlying execution platform has to be proven. This will not be the appropriate paradigm for ADS. In those systems, many sub-functions will co-operate and be highly interdependent in order to implement an overall system function, namely to drive safely from location A to location B [18]. Furthermore, for cost, performance, flexibility and dependability reasons, ADS will be implemented on networked computing platforms that encompass a low number of processors, connected by high bandwidth real-time networks, and potentially possessing multiple cores [13]. To increase reliability, redundant software functions can be mapped to different hardware components, both statically and possibly also dynamically. As a consequence, many functions will be mapped to a single or more hardware components, which means that a software-hardware mapping problem needs to be considered in the system and safety design. Again, current ISO 26262-type functional safety analyses do not account for this type of architectures.

Third, ADS will be highly concurrent, due to the parallel processing of sensor data and decision making to support different driving functions, leading to concurrency non-determinism. Another change with ADS is the application of non-linear machine-learning algorithms based on neural networks that are heavily used in environment perception. Non-determinism and non-linearity make it particularly difficult to use classical safety analysis techniques, such as Fault Tree Analysis (FTA) or Failure Mode and Effects Analysis (FMEA) proposed for piloted driving in ISO 26262, in a non-automated, manual fashion.

In this paper we propose a method to analyze functional safety of ADS "in the spirit" of ISO 26262, to the extent that it is applicable, and address some of the challenges pointed out above. The method is model-based and relies on SysML [19] models that describe the nominal and the failure behavior of components, as well as software-hardware mappings. We embed these models into the QuantUM method and tool [15,17] for analyzing causes of safety violations. QuantUM employs automated causality checking [16] in order to compute, depicted as a probabilistic fault tree, ordered sequences of events that are deemed to be causes for safety violations. The benefits of this approach include the following aspects.

- The algorithmic model analysis methods employed in QuantUM (model checking, causality checking) are well suited to deal with concurrency induced non-determinism. Dealing adequately with the non-linearity caused by using neural networks based machine learning is not addressed in this paper.
- The proposed analysis avails itself to an implementation in an automated software tool. Once the models and properties are defined, the analysis performed by QuantUM requires no further interaction with an engineer.
- The SysML models can easily be modified, for instance to analyze architectural alternatives as well as alternative software-hardware mappings during design space exploration. The functional safety analysis can then easily be repeated at little cost by invoking QuantUM on the modified model.
- The developed tools can be qualified according to, for instance, ISO 26262.

We evaluate our approach by applying it to a case study in which we perform a functional safety analysis for a practical ADS architecture [5] for which we analyze two mappings of ADS functions to hardware. The analyzed system failures can be used to assess the impact of single or multiple faults on the overall failure probability, as requested by ISO 26262. The analysis also enables an engineer to select efficient failure handling concepts and to evaluate different possible architectures while meeting safety goals as specified by ISO 26262.

Related Work. The most closely related work on automated model-based safety analysis for autonomous vehicles is [7]. It uses a block definition diagram and a manually created fault tree to compute probabilities for the purpose of safety analysis. In contrast to our work, no causal explanations for failures are automatically derived from the model.

Model-based techniques are applied to evaluate an automotive architecture in several papers. The approach of [6] is not automated, and it does not address the specifics of ADS. UML models, which are similar to SysML models, are also verified in [2,22], but both do not quantify system failures.

The paper [1] also addresses safety engineering for autonomous vehicles. It proposes an approach that differs from that of ISO 26262 by focussing on safety mechanism to detect all malfunctions.

Structure of the Paper. In Sect. 2 we present the foundations of our work which includes the demands of ISO 26262 on vehicles, the change in the architecture with the development of ADS and the QuantUM approach, which we will extend. In Sect. 3 we explain the analysis steps to verify an ADS architecture in the "spirit" of ISO 26262. In Sect. 4 illustrate our approach by applying the steps on two ADS architectures. In Sect. 5 we draw conclusions and suggest future developments.

2 Preliminaries

Functional Safety and Autonomous Driving. The ISO standard 26262 [8] as well as its recently proposed revision [9] define requirements on software development processes for safety-critical functions of an automotive passenger vehicle. This is to ensure that the functional safety of a passenger vehicle is challenged by no more than an acceptable residual risk. The standard is focused on mechanisms that ensure functional safety of critical software-driven functions in the presence of systematic faults and random hardware faults. It assumes that systematic faults can be eliminated by verification and validation techniques, in particular testing. The standard does not tackle random software failures, which happen non-deterministic, for instance, due to concurrency issues or special environment influences. Notice that the ISO 26262 standard does not address techniques to ensure "Safety of the Intended Function" (SOTIF), i.e., the safety of intended functionalities of the vehicle itself. A standard addressing this safety aspect is currently under development [10].

Two characteristics of the ISO 26262 standard are important in the context of this work. First, the standard is "item-oriented", which means that it addresses safety mechanisms for items, such as airbag control, steering, braking, light control, etc., in isolation. This approach is inappropriate for ADS since different vehicle functions will be interdependent by acting as backup functions for others. Further, the driver as the function integrator and coordinator in piloted driving is not available, which means that the software has to take over these integration functions. Second, neither the published version of the ISO 26262 standard nor the its proposed revision address assisted or autonomous driving per se [11]. To the contrary, the ISO 26262 standard allows safety mechanisms to rely on the driver taking over control of the vehicle in order to mitigate the impact of function failures, which in ADS at SAE level 3 is likely not to be practical [23], and at levels 4 and 5 is not even foreseen [20]. Nonetheless, we show how formal analysis techniques can be used to support the safety engineering of ADS in the spirit of ISO 26262. According to ISO 26262, different driving functions, referred to as "items", are assigned an Automotive Safety Integrity Level (ASIL), ranging from the least critical ASIL A to the most critical ASIL D. The determined ASIL implies the design methods that are to be used. As argued above, we will consider the ADS driving function as a unique "item" in the ISO 26262 sense and perform a safety analysis on this set of functions as a whole, including an assignment of an ASIL.

According to ISO 26262, safety goals need to be defined to ensure that the failure probability of software functions in the presence of random hardware faults lies at an acceptable minimum. For each ASIL, a maximum tolerable probability of failing a safety goal due to random hardware faults is specified. A system failure may be a result of a single fault (single-point failure) or a combination of faults (multiple-point failure). From the safety goals, functional safety requirements are derived. In safety analyses one will also have to consider fault rates of the underlying hardware, for instance sensor faults, as well as the hardware-specific fault detection rates. Those will later occur as parameters of our models.

Following ISO 26262, the system architecture design is derived from technical safety requirements. For ASILs A to D, the standard recommends documenting the architecture using a semi-formal notation, such as the SysML, for ASIL A and B, and strongly recommends this for ASIL C and D [9, Part 6]. The system architecture then needs to be verified against the safety requirements [8, Part 4]. The process mandated by ISO 26262 for this verification includes a system design analysis to identify possible effects of faults, the causes of possible failures and a quantification of failures. Applicable methods include Fault Tree Analysis (FTA) and Markov models. The use of formal verification techniques, including model checking [4], is recommended for software architecture verification for ASIL C and D [8, Part 6]. This includes a verification that certain safety goals are met by a given system design [8, Part 8]. We propose that an automated approach based on a formal analysis of the state space described by the system architecture helps to detect and explain safety goal violations at an early stage in the safety engineering process, thereby meeting the requirements of ISO 26262. It also enables automated, tool-supported architectural variant analysis during safety and system engineering, greatly contributing to reducing the related costs.

Model-Based Safety Analysis - The QuantUM Approach. Safety analysis relies on the establishment of cause-effect relationships between states or events in a system. Causality checking [16] is an automated, algorithmic approach to compute cause-effect relationships for events in a model of a system. It is based on model checking and systematic, complete state space exploration. Based on a counterfactual reasoning argument, it computes ordered sequences of events as being causal for the violation of a safety specification, defined as the (un-)reachability of a hazard state. In the context of the QuantUM toolset [15,17], causality checking is used to automatically compute sequences of sequentially ordered events with minimal length which are causal for violations of the reachability property representing the hazard. The SysML model is given by block definition diagrams (bdd) to depict units of the architecture, and state chart diagrams (stm) to specify their behavior. The SysML model contains both the nominal and the failure behavior of the architectural components. In QuantUM, the computed causal events are then depicted as a fault tree [17], with the considered hazard forming the top level event.

The causes for a model failure that QuantUM calculates are represented by a fault tree including the calculated probabilities. In the interpretation of the fault tree notation that QuantUM uses, the nodes in the graph do not all correspond to subsystem faults, but rather to events belonging to the causal process leading up to a hazard. The top level event is connected to an or gate. The or gate is connected to a number of ordered and gates, each one representing a causality class. A causality class is specified by a minimal ordered sequence of events that jointly, and in that order, cause the occurrence of the hazard. Notice that QuantUM can also determine the non-occurrence of single events as the cause of a hazard.

System Architectures for ADS. A functional architecture for autonomous driving is proposed in [5]. The authors extract, from several conceptual as well as practical implemented architectures, a layered architecture. The semantic understanding of the external world is calculated in the *perception layer*. It computes an external world model based on a fusion of the various forms of external sensor information that it receives. The external world model in conjunction with the internal state of the car, which is defined among others by the energy management and failure states of the platform, are used by the *decision and control layer* to make decisions about the execution of a trajectory. The trajectory is then used by the *vehicle platform manipulation* layer to drive the actors, like steering and braking, and keeping the platform overall stable. All three layers have a complex structure of interdependent, cooperating elements, each representing a specific function.

Functional Safety Goals for ADS. A predominant idea in ISO 26262 is that a system needs to reach a safe state in the event of a system failure, in other words, that it is *fail-safe*. When the driving is piloted, this can often be achieved by switching the defective subsystem off and leaving it up to the driver to deal with the situation. In autonomous driving, this option does not exist, as argued above. The objective here needs to be that in the presence of the failure of one function in an ADS, the overall system architecture needs to remain operational for a certain period of time so as to ensure that a safe state can be reached. This capability is often referred to as "fail-operational". The ISO 26262 standard states: "If a safe state cannot be reached by a transition within an acceptable time interval, an emergency operation shall be specified." [8, Part 3]. This means that designing safety mechanisms that ensure a limited backup capability for a defective functionality for a certain period of time is within the practices recommended by ISO 26262. A typical example would be that the braking system takes over functionalities of a failed steering control system by applying differential torque or braking for a limited period of time so as to "limp home" to a safe part of the road, such as the shoulder. The safety goals that we pursue in our analysis will, hence, have to reflect the probabilities of remaining *fail-operational* for a certain period of time.

3 Safety Analysis of an ADS Architecture

Safety Goals for an ADS. Following the earlier made argument we consider the driving function of an ADS to be one item, i.e., one driving function. Using this assumption we perform a safety analysis for this item in the spirit of ISO 26262.

As argued above, we need to consider a fail-operational architecture. When reaching a failure state, the ADS reacts by switching to an emergency mode that handles the failure situation. For a safety analysis of an ADS, we consider possible hazards of an architecture and derive appropriate safety goals to prevent the hazards:

1. When a vehicle is operating as an ADS it has to control the vehicle platform even if it is in an emergency mode. If the control is lost, then the vehicle will

crash. To prevent this hazard we derive the safety goal SG1: *Ensure that the ADS provides driving information to the vehicle platform at any time.*

2. The ADS can have an undetected failure. As a consequence, the emergency mode may not be activated. The detection of a failure ensured by the safety goal SG2: *Ensure that the emergency mode is enabled when a failure of the ADS occurs.*
3. If the system cannot enter or remain in the emergency operation mode for a specified period of time, a safe state may not be reachable. We assume the period of time necessary to reach a safety state to be t_1 seconds and derive the safety goal SG3: *Ensure that the emergency mode of the ADS is available on demand for at least t_1 seconds.*

The ASIL classification of a safety goal is determined according to the severity of a function failure caused by a hazard, the probability of exposure to a situation with a potential failure, and the controllability of the failure situation by the driver. We assume the severity of each hazard of the ADS to be potentially life-threatening (S3 according to [8, Part 3]). Since the ADS system will be active most of the time when the vehicle is in operation, certainly during more than 10% of the operation time, we assume the probability of exposure to be high (E4 according to [8, Part 3]). We also assume the controllability to be very low (C3 according to [8, Part 3]), since in the case of SAE level 3 driving the driver may be surprised by a failure situation, or unable to handle it due to the low occurrence rate of such a situation. These valuations hold for all three safety goals and consequently this implies, according to ISO 26262, an ASIL-D classification for each safety goal.

As argued above, ISO 26262 recommends the use of formal methods, including model checking, for the analysis of ASIL D safety goals.

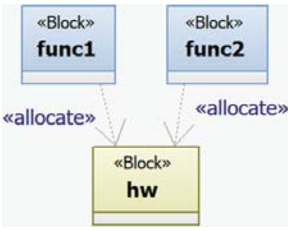

Fig. 1. Mapping 1 (Color figure online)

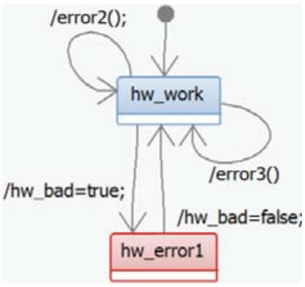

Fig. 2. Stm `hw` (Color figure online)

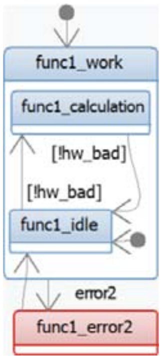

Fig. 3. Stm `func1` (Color figure online)

Automated Safety Analysis of an ADS Architecture. We now describe how an extended version of the QuantUM tool can be used to perform an automated safety analysis for a given ADS architecture with respect to safety goals SG1 - SG3.

Step 1: ADS Modeling. The system architecture of a vehicle consists of several software function units executing on a number of hardware units. Each unit is represented by a block in a SysML bdd, see the example in Fig. 1. It depicts two software function units, represented by blocks colored blue, executing on one hardware unit, colored yellow. Assigning a software function to a hardware unit on which the function is executed is referred to as software-hardware mapping. In our bdds, mappings are depicted using dashed arrows with the label `<<allocate>>`. The behavior of each unit is modeled by a SysML stm. The stms of different units execute concurrently. For the example in Fig. 1 the behavior of the blocks `hw` and `func1` are exemplified in Figs. 2 and 3, respectively.

In the stms, the blue states represent the nominal behavior of the units, and the red states represent its failure behavior. The state machines execute their normal behavior by staying in a "work" state. To reach a failure state, a fault event has to occur. The first type of fault directly leads to a failure of the unit. This can for instance be caused by a loss of power, or by a permanent error such as a broken hardware element. These faults are modeled by a transition to a failure state, such as `hw_error1` in Fig. 2. The unit remains in this state until it is repaired, represented by a repair transition to the work state. As a result of entering a failure state, a hardware unit stops executing and any software function, executing on this hardware unit, will cease to operate as well. To model this behavior, the Boolean variable `hw_bad` is set to true and all transitions of the function unit are disabled by a guard `!hw_bad` (see Fig. 3). The second type of fault leads to an error inside of the hardware unit, such as a bit flip, even though the unit continues to operate. The hardware unit is not corrupted, but an error is propagated to functions executing on that unit. In the SysML model, error propagation is modeled by message passing. In the example in Fig. 2, two errors are propagated by messages `error2` and `error3` to the respective software functions. With the receive of such an error the function 1 enters the `func1_error2` state and function 2 enters the `func2_error3` state upon receiving `error3`. From these states, the function can return to its normal behavior by a transition representing failure repair.

Step 2: System Failure Modeling. The ADS fails if one of the safety goals SG1-SG3 is violated. The violation of these system goals needs to be mapped to states that the different stms in the system are entering. QuantUM offers the possibility to tag states in the stms of different blocks as error states, and then permits to either use a logical *or* or a logical *and* between all tagged states in order to characterize a violation state of the system. To model the safety goals needed here we extend this rather inflexible scheme. An ADS has the structure of a set of channels. Sensory input data is processed by a function and the output data is forwarded to the next function, forming one channel called the primary channel. The emergency mode adds a second, partly redundant backup channel to the ADS. The ADS fails and violates SG1 if there is a function failure in each of the channels. We attach a Boolean variable `bad` to each function and permit forming logical expressions on these variables to express the failure of one channel. We combine the failure expressions of each channel with an "and"

and add the result to the property. For example, in order to check two redundant channels the resulting property has the form *it is never the case that step1 or step2 or ... of channel1 is bad and step1 or step2 or ... of channel2 is bad.*

Step 3: Analysis of Emergency Mode Failures. A violation of SG3 implies that the normal ADS behavior has a failure and either the emergency mode functionality is not available on demand, or it is not provided for at least a certain period of time and therefore constitutes a fundamental challenge to the safety of the vehicle. In the following we compute the probability P_{fail} for a violation of SG3. The analysis performed by QuantUM works on a global state graph obtained by interleaving the local behaviors of the concurrent system hardware and software components. A path in this graph, representing an execution of the ADS, constitutes a violation of SG3 if in a state the emergency mode is being activated but not going to be available for at least a period of time t_1. We characterize the set of all emergency activation states S in the global state graph using a Boolean expression e formed as described in Step 2. In accordance with the foundations of probabilistic model checking we will consider reaching a first state $s_i \in S$ as a stochastic event, with the path consisting of a stochastic experiment. The event of reaching a state s_i first, denoted by $reach\_s_i$, precludes the event of reaching another state $s_i' \in S$ first, which means that stochastic events we consider do not overlap. As a consequence we may partition the sample space, which consists of all possible paths in the global state graph, according to the events $reach\_s_i$. In a first probabilistic model checking step performed by QuantUM we compute the probability $P(reach\_s_i)$ to reach each state in S within a period of a driving cycle t_dc. In a second model checking step we compute the probability $P(fail_i|reach\_s_i)$ to reach a failure from state s_i within a time t_1. To enable the first model checking step we change the model in such a way that we conjoin $\neg e$ with all transition guards. This means that when the system enters a state in which e becomes true, this state is turned into an end state with no enabled exit transition. For each end state we calculate its probability. For the second model checking step we compute the probability $P(fail\_s_i|reach\_s_i)$ by starting in any state defined by expression e. The probability P_{fail} is computed by a summation over all products of $P(fail_i|reach\_s_i) \cdot P(reach\_s_i)$, which is justified by the memoryless nature of CTMCs. No causality checking will be performed and no fault trees will be computed by QuantUM during SG3 violation analysis.

Step 4: Probability Rates. Probability rates, in particular for hardware failures, repairs and failure detection, are difficult to determine and usually depend on a specific domain and the concrete hardware used. However, even if precise rates are not available, the comparison of the relative failure probabilities of architectural variants with identical and with different estimated or assumed rates can be of great importance. This can for instance answer the question how architectural variants will affect failure probabilities, or what error detection rates are required to achieve a desired level of failure probability. In QuantUM, the SysML model is labeled with probability rates, for instance for the probability of executing a failure or repair transition. QuantUM uses probabilistic model checking, in particular model checking for Continuous Time Markov Chains (CTMCs) [3], in order to compute the probabilities for the causes leading to a violation of safety properties. A fault event of the hardware may lead to different faults in a system. In this situation we distribute the fault rate over the different fault transitions. The portion of the fault rate that each transition receives relies on domain specific knowledge that the designer needs to provide. For example, in Fig. 2, a bit flip with a probability rate of 10^{-4} can cause an `error2` or an `error3`. Notice that throughout the paper, rates are assumed to be per hour. Assume that it is typical that 40% of the errors are of type `error2` and 60% are of type `error3`. This leads to a fault rate of $0.4 \cdot 10^{-4}$ for `error2` and a fault rate of $0.6 \cdot 10^{-4}$ for `error3`. To split the fault rate in this way is appropriate for CTMCs, cf. [3].

A potential threat to the validity of the failure probabilities computed by QuantUM and the probabilistic model checker Prism [12] that QuantUM uses, is the fact that the original SysML model mixes non-probabilistic and probabilistic transitions. For the non-probabilistic transitions Prism assumes a default rate of 1. Assuming that we consider one time step, based on the negative exponential distribution on which CTMCs are based this translates into a probability of less that 1 of taking this transition with which the accumulated path probability up to this step will be multiplied. However, we do not experience a negative effect on the total failure probability since the SysML model structure that we propose implies that the system will cycle through non-probabilistic normal behavior, for which the path probability is 1, until it performs one probabilistic failure transition to enter a failure state. For example, the state `func1_work` in Fig. 3 has non-probabilistic transitions between the states `func1_calculation` and `func1_idle` with a default rate of 1, remaining is state `func1_run` until the probabilistic transition `error2` is taken.

4 Case Study: A Comparison of Autonomous Driving Architectures

Step 1: ADS Modeling. [5] proposes a functional architecture generalized from real architectures. We use part of this functional model and add several hardware units. The resulting mapping problem leads to a number of architectures. The SysML bdd in Fig. 4 gives an overview of the structure of the first architectural variant that we consider. We model the perception layer by a block `Perception` and the motion and control layer by a block `Trajectory`. Since the functions represented by these two blocks are critical for the proper functioning of the ADS we add blocks `PerceptionSafe` and `TrajectorySafe` to provide redundant backup functionality. The function represented by the block `Trajectory_Selection` selects by default the trajectory of block `Trajectory`, but switches in case of a failure of these blocks to the alternative trajectory computed by block `TrajectorySafe`. The block `MotionControl` represents the interface with the vehicle platform manipulation layer by providing it with control parameters, such as steering angle, braking force or differential torque, that the vehicle platform will translate into commands for the actuators of the vehicle. Figure 4 also illustrates the software-hardware mapping that we propose for the first architectural variant. Notice that the primary functionalities for perception and trajectory computation are mapped to the hardware block ADS_primary, while the backup functionality ADS_backup is mapped to a separate hardware unit ADS_backup, thus increasing the probability that the backup functionality will be available even in the case of a failure of the primary hardware represented by block ADS_primary.

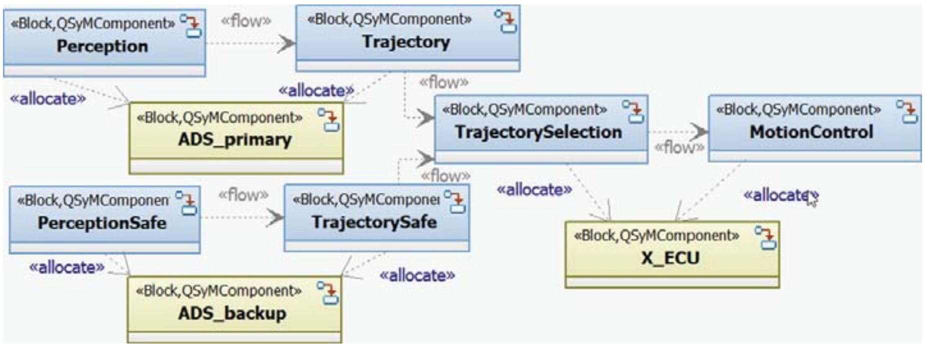

Fig. 4. Architectural variant 1 for ADS

The state machine modeling the behavior or block ADS_primary is given in Fig. 5. The hardware operates correctly in state run. In this state, the occurrence of a detected error inside the hardware, for instance a memory bit flip, is communicated to the Perception block using a perception_error message in case the perception function is currently executing on ADS_primary. In case a trajectory computation function is executing, the hardware error will be communicated using a trajectory_error message to the Trajectory block. If in the run state an undetected hardware error occurs, the impact on the hardware is unknown. We model this by a transition into state undetected along which we set the failure variable bad to true. In this state, no software function can be executed on the hardware.

The behavior of the block Perception is defined by the hierarchical state machine in the stm diagram in Fig. 6. The normal behavior is modeled in the nested state Normal. The function starts its computation in the state idle and cycles through states idle and calculate, which represents the processing of sensor information, as long as the variable ADS_primary.bad is false. When returning to idle it sends the message trajectory_input to the function trajectory in order to indicate that input data for the trajectory function is available. Upon receipt of a message perception_error, the perception function can decide either to handle this message and remain in the Normal state, or it can decide to enter the ErrorData state and set its bad value to true. Upon repair it can return to the idle state and resume execution. Deviating from the classical ISO 26262 viewpoint to only consider hardware failures, we also consider software failures. Such a failure in the block Perception is modeled by a non-deterministic group transition from the Normal state to the CalculationError state in the course of which the bad variable will be set to false. We assume that these errors can

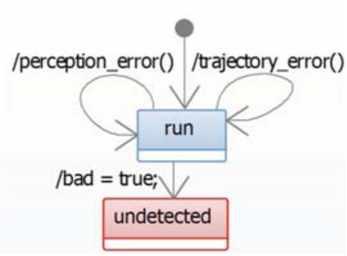

Fig. 5. Stm `ADS_primary`

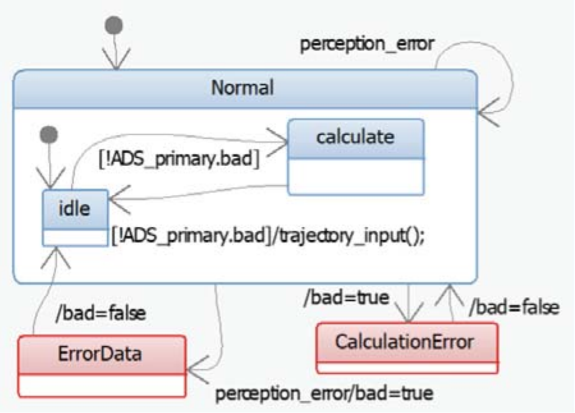

Fig. 6. Stm `Perception`

also be repaired, modeled by a return to the `Normal` state. Space limitations do not allow us to present the complete behavioral model of the ADS. The other blocks representing hardware and software functions have behaviors similar to the ones described above.

Step 2: System Failure Modeling. We characterize a system failure of the ADS violating SG1 or SG2 using different Boolean expressions for each architectural variant and safety goal. The Boolean propositions refer to the `bad` variables of the blocks in the SysML model. For architectural variant 1, a system failure violating SG1 happens when a software function or a hardware unit fails in both channels, or when at least one of the functions `TrajectorySelection` or `MotionControl` or the hardware `X_ECU` fails. This leads to the Boolean failure expression

$$\begin{aligned} &((ADS\_primary.bad \vee Perception.bad \vee Trajectory.bad) \\ \wedge&(ADS\_backup.bad \vee PerceptionSafe.bad \vee TrajectorySafe.bad)) \\ \vee&(TrajectorySelection.bad) \vee (MotionControl.bad \vee X\_ECU.bad). \end{aligned} \tag{1}$$

Architectural variant 2 differs from the first in that the block `TrajectorySelection` is mapped to the block `ADS_backup`. As a consequence, architectural variant 2 fails under the same failure condition as variant 1, and additionally by a failure of `ADS_backup`. This leads to the Boolean failure expression

$$\begin{aligned} ((ADS\_primary.bad \vee Perception.bad \vee Trajectory.bad) \\ \wedge(ADS\_backup.bad \vee PerceptionSafe.bad \vee TrajectorySafe.bad)) \\ \vee(ADS\_backup.bad \vee TrajectorySelection.bad) \\ \vee(MotionControl.bad \vee X\_ECU.bad). \end{aligned} \tag{2}$$

The function `TrajectorySelection` is responsible for selecting the emergency trajectory in case of a failure. The function fails if the function itself or the underlying hardware is in a failure state. This leads to a violation of SG2, expressed by the Boolean failure expression $\mathit{TrajectorySelection.bad} \vee \mathit{X\_ECU.bad}$ for architectural variant 1 and $\mathit{TrajectorySelection.bad} \vee \mathit{ADS\_backup.bad}$ for variant 2.

Step 3: Analysis of Emergency Mode Failures. For the computation of the failure probability of the emergency mode we need to determine the expected operation time of the emergency mode t_1, a Boolean expression representing a failure of the ADS and a Boolean expression characterizing the activation of the emergency mode. We assume t_1 to be 10 s. The failure states of the ADS for the two architectural variants are encoded by the Boolean expressions 1 or 2, respectively. The emergency mode of the ADS is activated in both architectural variants if a software function running on hardware `ADS_primary` or the hardware `ADS_primary` itself fails. We encode these states using the Boolean expression $(\mathit{ADS\_primary.bad} \ \vee \mathit{Perception.bad} \vee \mathit{Trajectory.bad})$.

Step 4: Probability Rates. In order to determine rates in the context of our case study we assume `ADS_primary` and `ADS_backup` to be implemented using "standard" hardware without hardware checks in order to meet the high computing power demands of the software functions executing on them. In such hardware components most faults happen because of memory errors [21], and we assume typical fault rates of 10^{-4}. Since there are no special computing power demands that apply to `X_ECU` and since we have not accounted for any redundancy here we assume safety hardware to be used with a fault rate of 10^{-8}. We further assume a hardware fault detection rate of 99%, i.e., 1% of the errors remain undetected. As described above, the probability of detected hardware faults are distributed evenly over all software functions running on the considered hardware unit. For instance, `perception_error` and `trajectory_error` are assumed to each have a probability of 49.5%, i.e., a rate of $0.495 \cdot 10^{-4}$. We assume that a software function affected by a detected hardware error handles the error with a probability of 90%, but will fail with a probability of 10%. For software failures of the perception function we assume a fault rate of 10^{-4}. Functions in a failure state can resume by a repair transition. We assume a repair rate of $4 \cdot 10^{-2}$ for software functions (cf. [7]).

Analysis using QuantUM. We assume a driving cycle duration t_{dc} of 1 h in all of the analyses. The result of the analysis for violating SG1 is a fault tree with the state representing the SG1 violation as top level event, and 19 disjunctive tree branches for architectural variant 1 and 16 disjunctive tree branches for architectural variant 2. We call the disjunctive tree branches *causes*. For both variants, space limitations do not permit us to present the full fault tree here. Architectural variant 1 has the probability of $1.31998187 \cdot 10^{-8}$ and architectural variant 2 the probability of $4.30677042 \cdot 10^{-6}$ to violate SG1. Due to the redundant structure of the architecture in both variants, analyzing SG2 in isolation leads to two single source failures already detected by SG1. One single source failure involves `trajectory_selection_error` for both variants and the other failure involves `X_ECU_undetected` for variant 1 and `ADAS_backup_undetected`

Table 1. Computational effort for SG violation analyses

	Architectural variant 1			Architectural variant 2		
	Memory	Time	States	Memory	Time	States
SG1	143.27 MB	63.79 min	235,765	124.84 MB	57.14 min	207,052
SG2	284.11 MB	4.93 min	321,133	310.35 MB	1.12 min	324,464
SG3	99.25 MB	6.59 min	349,937	160.04 MB	4.96 min	354,943

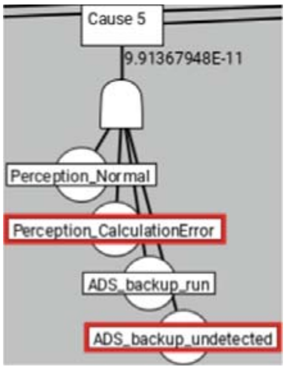

Fig. 7. Cause 5 of architectural variant 1 (Color figur online)

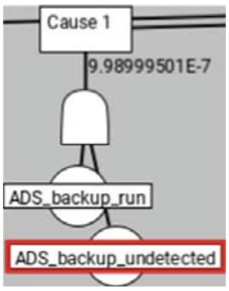

Fig. 8. Cause 1 of architectural variant 2

for variant 2. The probability of a violation of SG3 is $6.399474 \cdot 10^{-9}$ for variant 1 and $6.164128 \cdot 10^{-9}$ for variant 2.

The experiments were performed on a computer with an i7-6700K CPU (4.00 GHz), 60 GB of RAM and a Linux operation system. The computational efforts in memory, time and for the architectural variants are depicted in Table 1. The column *States* gives the number of states explored by QuantUM, in the case of SG3 this only comprises the number of states analyzed by Prism.

The memory effort for SG1 and SG2 is small in comparison to previous models [14]. The small memory effort is due to the fact, that the current implementation of QuantUM, does not compute duplicate state prefix matching as described in [14]. However, for the analyzed models, the current version of QuantUM computes all causes, since each failure state of the stms is only reached by a single trace. All other traces leading to a failure state are extensions of the single trace and so not minimal.

Result Interpretation. The architectural variant 1 has a lower probability of violating SG1. This result can be explained as follows. The fault trees for the two architectural variants differ mainly in the probability of the causes that contain the event `ADS_backup_undetected`, representing an undetected hardware failure in the hardware unit that is subject to the altered software-hardware mapping. The fault tree of architectural variant 1 contains four causes that contain the event `ADS_backup_undetected`, of which one cause is depicted in Fig. 7, with a probability of $9.91367948 \cdot 10^{-11}$ and thus not contributing significantly to the total SG1 violation probability. All other causes that contain `ADS_backup_undetected` have a similarly insignificant probability. Notice that failure events in the fault trees are marked in red.

The fault tree of architectural variant 2 contains one cause with this event, depicted in Fig. 8, with a probability of $0.99899950 \cdot 10^{-6}$, thus contributing significantly to the SG1 violation. While in architectural variant 1 the `ADS_backup_undetected` fault needs to coincide with a `Perception_CalculationError`, in architectural variant 2 the occurrence of `ADS_backup_undetected` suffices to lead to an SG1 violation. The difference in the probabilities of the two considered causes is due to the fact that the conditional occurrence of two failure events, such as in cause 5 of architectural variant 1, is less probable than the unconditional occurrence of a fault event as in cause 1 of architectural variant 2. Due to the software-hardware mapping in architectural variant 2, the fault event `ADS_backup_undetected` directly leads to an SG2 violation, and this scenario has a high probability. The difference in the probabilities of SG1 violations can hence be traced back to the difference in the hardware-software mappings used in both architectural variants.

In the following, we discuss the influence of the detection and error handling rates. We first increase the detection rate of `ADS_backup` for variant 2 from 99% to 99.99%. The higher detection rate decreases the failure probability of cause 2 from $9.98999501 \cdot 10^{-7}$ to $9.98999995 \cdot 10^{-9}$. This change has no significant effect since the probability of reaching error state `undetected` is decreased by the same amount that the error probability of the functions running on the hardware is increased. The probability of violating SG1 is now mainly due to reaching failure state `ErrorData` of function `TrajectorySelection`, which is $3.32799547 \cdot 10^{-6}$. In a second step we increase the error handling rate of function `TrajectorySelection` from 90% to 99%. This decreases the failure probability for `ErrorData` in function `TrajectorySelection` to $3.32805910 \cdot 10^{-8}$. As a consequence the overall failure probability of violating SG1 decreases from $4.30677042 \cdot 10^{-6}$ to $5.60069041 \cdot 10^{-8}$. We notice that detection and error handling rates have an essential influence on the failure probability of the ADS.

A violation of SG3 is less probable than 10^{-9} for both variants. The small probabilities are reasonable since the ADS remains in the emergency mode for only 10 s, which is much shorter than the assumed driving cycle of one hour. Unexpectedly, a violation of SG3 is more probable for variant 1 ($6.399474 \cdot 10^{-9}$) than for variant 2 ($6.164128 \cdot 10^{-9}$). The difference is due to the fact that variant 2 fails more probable without entering the emergency mode.

ISO 26262 requires an analysis of single and multiple point failures, and whether failures are detected or undetected. We extract this information from the causes in the fault trees. A cause representing a single point failure contains a single failure event, other causes are multiple point of failure. For example, Cause 1 of variant 2 is a single point failure since it contains the single failure event `ADS_backup_undetected`. An undetected failure is represented by a cause that contains at least one undetected failure event. Cause 5 of variant 1 represents such an undetected failure. With this information it is possible to perform further analyses on undetected failure rates and to relate them to single and multiple point faults, as required by ISO 26262.

5 Conclusion

We have presented an automated approach to support the design time functional safety analysis for architectures supporting ADS. The paper addresses the handling of the complexity of future ADS by analyzing a flexible mapping of hardware and software functions. We have applied the proposed approach to two variants of a practical ADS architecture and compared the two variants. We have shown that the proposed approach gives necessary information to perform functional safety analyses in the spirit of ISO 26262. The analysis included fail-operational behavior, software faults and interdependent driving functions which are so far not adequately addressed by ISO 26262. We see great potential in supporting ISO 26262 style functional safety analyses of innovative automotive architectures using the formal algorithmic analyses that QuantUM supports.

Future research will address an improved integration of the analysis into existing tools and methods, for instance by incorporating automated Failure Mode and Effects Analysis (FMEA), more flexible property specification, and an improved scalability of the method, in particular using symbolic analysis techniques.

Acknowledgements. We wish to thank Stephan Heidinger, Matthias Kuntz and Majdi Ghadhab for discussions at the early stages of this work.

References

1. Adler, R., Feth, P., Schneider, D.: Safety engineering for autonomous vehicles. In: DSN Workshops, pp. 200–205. IEEE Computer Society (2016)
2. Bahig, G.M., El-Kadi, A.: Formal verification of automotive design in compliance with ISO 26262 design verification guidelines. IEEE Access **5**, 4505–4516 (2017)
3. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Softw. Eng. **29**(6), 524–541 (2003)
4. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
5. Behere, S., Törngren, M.: A functional reference architecture for autonomous driving. Inf. Softw. Technol. **73**, 136–150 (2016)
6. Cuenot, P., Ainhauser, C., Adler, N., Otten, S., Meurville, F.: Applying model based techniques for early safety evaluation of an automotive architecture in compliance with the ISO 26262 standard. In: Proceedings of the 7th European Congress on Embedded Real Time Software and Systems (ERTS^2) (2014)
7. Ghadhab, M., Junges, S., Katoen, J.-P., Kuntz, M., Volk, M.: Model-based safety analysis for vehicle guidance systems. In: Tonetta, S., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2017. LNCS, vol. 10488, pp. 3–19. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66266-4_1
8. ISO: Road vehicles - functional safety. ISO 26262, International Organization for Standardization, Geneva, Switzerland (2011)

9. ISO: Draft international standard, road vehicles - functional safety. Technical report ISO/DIS 26262, International Organization for Standardization, Geneva, Switzerland (2016)
10. ISO: Road vehicles - safety of the intended functionality. Technical report ISO/WD PAS 21448, International Organization for Standardization, Geneva, Switzerland (2017)
11. Koopman P., Wagner, M.: Challenges in autonomous vehicle testing and validation (2016). Preprint https://users.ece.cmu.edu/~koopman/pubs/koopman16_sae_autonomous_validation.pdf
12. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
13. Leitner, A., Ochs, T., Bulwahn, L., Watzenig, D.: Open dependable power computing platform for automated driving. In: Watzenig, D., Horn, M. (eds.) Automated Driving, pp. 353–367. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-31895-0_14
14. Leitner-Fischer, F.: Causality checking of safety-critical software and systems. Ph.D. thesis, University of Konstanz, Germany (2015)
15. Leitner-Fischer, F., Leue, S.: QuantUM: quantitative safety analysis of UML models. In: QAPL. EPTCS, vol. 57, pp. 16–30 (2011)
16. Leitner-Fischer, F., Leue, S.: Causality checking for complex system models. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 248–267. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_16
17. Leitner-Fischer, F., Leue, S.: Probabilistic fault tree synthesis using causality computation. IJCCBS **4**(2), 119–143 (2013)
18. Martin, H., Tschabuschnig, K., Bridal, O., Watzenig, D.: Functional safety of automated driving systems: does ISO 26262 meet the challenges? In: Watzenig, D., Horn, M. (eds.) Automated Driving, pp. 387–416. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-31895-0_16
19. OMG: Systems Modeling Language (SysML), Version 1.5. Technical report, OMG (2017)
20. SAE: J3016_201609: Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles, September 2016. https://www.sae.org/standards/content/j3016_201609
21. Schroeder, B., Pinheiro, E., Weber, W.: DRAM errors in the wild: a large-scale field study. Commun. ACM **54**(2), 100–107 (2011)
22. ter Beek, M.H., Gnesi, S., Koch, N., Mazzanti, F.: Formal verification of an automotive scenario in service-oriented computing. In: ICSE, pp. 613–622. ACM (2008)
23. Watzenig, D., Horn, M.: Introduction to automated driving. In: Watzenig, D., Horn, M. (eds.) Automated Driving, pp. 3–16. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-31895-0_1
24. Weiss, G., Schleiss, P., Drabek, C., Ruiz, A., Radermacher, A.: Safe adaptation for reliable and energy-efficient E/E architectures. In: Watzenig, D., Brandstätter, B. (eds.) Comprehensive Energy Management - Safe Adaptation, Predictive Control and Thermal Management. SAST, pp. 1–18. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-57445-5_1