Iris Groher, Thomas Vogel (Hrsg.): Software Engineering 2023 Workshops,
66 Digital Library, Gesellschaft für Informatik, 2023

Providing Evidence for Correct and Timely Functioning of Software Safety Mechanisms[1]

Jan Steffen Becker[2], Björn Koopmann[2], Ingo Stierand[2], Lukas Westhofen[2]

Abstract: In many application domains, the development of safety-critical systems must follow standards that define process steps and artifacts to establish a comprehensive safety argumentation. Commonly, this involves the identification of hazards and risks as well as the formulation of a safety concept to mitigate these risks. The concept is decomposed into safety requirements, which are finally implemented in hardware and software. All steps must be covered by analyses to ensure that the concept is effective and correctly implemented. This work focuses on timing aspects of the safety concept, i. e., on how it can be ensured that risk mitigation occurs *in time*. Based on an industrial use case, we show how consistent timing specifications can be derived, decomposed, and implemented in a complete and sound way. The approach extends previous work on contract-based design and investigates on explicating failure modes and fault detection in contract specifications. Finally, we show how model checking can support the verification of safety concepts and their implementation.

Keywords: Safety Assurance; Timing Analysis; Contract-based Design; Model Checking; Safety Mechanisms; ISO 262626; Traceability

1 Introduction

The design of safety-critical systems calls for rigorous development processes and analysis methods that support engineers in constructing solid safety argumentations. In many application domains, standards exist that define process steps and artifacts required to achieve such argumentation. ISO 26262 [IS18], for example, is a standard that aims to ensure functional safety of automotive systems. It mandates the identification of potential hazards and risks imposed to users and their environment as part of a *hazard analysis and risk assessment* (HARA). Top-level safety requirements (safety goals) and safety integrity levels are derived as input for the definition of the functional safety concept to mitigate the identified risks. This includes performing structured cause analyses such as fault tree analysis (FTA) and failure mode and effect analysis (FMEA). Subsequently, a technical safety concept is defined for the hardware/software (HW/SW) development phase of the system [Tr16] that realizes the required safety mechanisms.

The ISO 26262 also mandates that all these steps must be covered by analyses to ensure that the system actually adheres to the safety goals. As the initial requirements are iteratively

[1] This work has been partially funded by the *Federal Ministry of Education and Research* (BMBF) as part of *PANORAMA* (reference no. 01IS18057G) and *MANNHEIM-AutoDevSafeOps* (reference no. 01IS22087C).

[2] Institute of Systems Engineering for Future Mobility, German Aerospace Center, Escherweg 2, 26121 Oldenburg, Germany, {jan.becker, bjoern.koopmann, ingo.stierand, lukas.westhofen}@dlr.de

doi:10.18420/se2023-ws-09

refined along with the system design, analyses ensure that the refined requirements can be traced back to the top-level requirements in a verifiable way to allow statements about their compliance. In this context, standards such as ISO 26262 provide clear guidance in the form of minimum requirements for the analyses to be performed (e.g. [IS18] Part 4, Tab. 1 & 2 and Part 9). By integrating the results into a *safety (assurance) case* [Bi13, Pa11], they can be used as evidence to prove compliance with safety goals or to substantiate (sub-)claims.

The purpose of a safety case is to demonstrate that all possible faults were identified during the safety-related design activities and are effectively mitigated by the system. This also includes to ensure timely functioning of safety mechanisms, i. e., the execution of failure avoidance strategies before a risk can develop into a hazard. The time span from the occurrence of a fault to a hazardous event, such as a collision of an automated vehicle with another road user, is referred to as *fault-tolerant time interval* (FTTI). Since a delayed response from a safety mechanism would inevitably have serious consequences, a careful consideration and analysis of the timing behavior of safety mechanisms under the presence of faults is of key importance. However, classical timing analyses for determining worst-case response times, e. g., from sensor input to controller output, do not consider faults. The present paper aims at closing this gap. The paper reports on a case study [St21a] that demonstrates the following:

1. Investigation of an approach for deriving real-time requirements based on early hazard analysis, including their subsequent handling along the design process up to the verification of correct implementation
2. Extension of pattern-based timing and safety contracts to enable explicit modeling of failure modes and safety mechanisms
3. Development of a prototypical tool support for fault-aware timing analysis to verify the correct and timely functioning of software safety mechanisms

The paper is structured as follows. Section 2 summarizes related work and places our contribution in its scientific context. In Sect. 3, a running example is introduced, for which timing requirements are derived from FTTI considerations in Sect. 4. Section 5 discusses the safety concept and its implementation. The contract formalization of the requirements is given in Sect. 6. Section 7 elaborates on the analysis. Finally, Section 8 concludes the paper.

2 Related Work

The ISO 26262-compliant development of safety-critical embedded systems constitutes a broad field of research. Related work deals with almost all design phases and artifacts from item definition and early hazard analysis to the implementation of safety mechanisms and the construction of safety cases. Frese et al. [Fr20], for example, refined the definition of FTTI and dealt with the terminology of timing properties of fault handling mechanisms. In [DHW19], an approach for deriving real-time requirements from FTTIs was presented. Kreiner et al. [PKK15] present a system of common design patterns for safety architectures.

The high complexity of safety-critical system design promotes the use of *model-based systems engineering* (MBSE) approaches. The combination of domain-specific languages, providing a clear semantic framework, with suitable tool support helps to reduce design effort and to avoid mistakes. Several domain-specific languages such as SysML are well-established in the automotive domain. An overview on MBSE is given in [Es08]. Eclipse APP4MC[3], which is based on the AMALTHEA meta-model, addresses a broad spectrum of the HW/SW design phase such as task partitioning and allocation, and allows detailed modeling of data and process flows. AUTOSAR mainly focuses on the technical realization, including detailed operating system and electronic control unit configurations. Various tools supporting AUTOSAR exist, partly combined with an ecosystem of backends for the deployment on hardware platforms. MBSE combines well with the application of formal methods. Pattern-based formalization of timing specifications has already been applied in industrial contexts and implemented in commercial tooling, as shown e.g. in [Be18].

The modeling of safety assurance cases is supported by the *Structured Assurance Case Metamodel* (SACM) [We19], which is part of the *Open Dependability Exchange* meta-model [Re20]. It provides a structured way of formulating safety argumentations. Moreover, prototypical tool support for creating safety cases [Ma18] and managing assurance evidence [Va22] exist. In [Bi18] a UML profile for safety and reliability analysis data is proposed that is also compatible to SACM. MBSE enables analysis automation to support seamless integration of different design phases. Besides industrial tools, academic analysis approaches exist. [MN20], for example, have presented an approach for model-based safety assessment using SysML and component fault trees. [Kr19] employed analytical timing analysis techniques to analyze AMALTHEA models.

ISO 26262 mandates the use of suitable methods to establish traceability between safety-related design artifacts. *Traceability information models* (TIMs) are a common tool in model-based design to model relations between artifacts. In [St21b], two example TIMs are contrasted and compared with others from literature. The TIM of the MobSTr dataset modeled in Eclipse Capra is an exemplar of such a supporting model [St21a].

3 Running Example

The methods developed in this work have been applied to the MobSTr (Model-based Safety Assurance and Traceability) dataset [St21b]. The dataset extends the use case of the WATERS 2019 Industrial Challenge [Re21]. MobSTr demonstrates the application of an ISO 26262-compliant design process and adds further safety and traceability artifacts.

The use case is a highly automated vehicle (in the following called *ego vehicle*) that shall perform two main functions: following a predefined route and avoiding collisions with other traffic participants or obstacles. The system, depicted in Figure 1, consists of *GPS*, *Lidar*, and

[3] `https://www.eclipse.org/app4mc/`

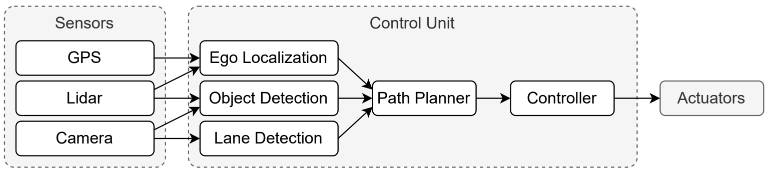

Fig. 1: System Functional Architecture

Camera sensors, a *Control Unit* as well as *Actuators*. The *Control Unit* implements functions to localize the ego vehicle (*Ego Localization*), detect, localize, and classify other traffic participants and obstacles (*Object Detection*), detect lane boundaries (*Lane Detection*), plan maneuvers (*Path Planner*), and calculate actuator commands (*Controller*).

During the HARA, hazardous events and potential causes thereof are identified. Thereafter, *safety mechanisms* must be defined that mitigate the effects of the causes of potential hazards. Generally, a safety mechanism may apply to more than one hazard and a variety of different situations. This paper exemplifies hazard *H-1: Omitted braking maneuver*. It describes the collision with another road user (or an obstacle) because a necessary braking maneuver of the ego vehicle is not performed in time. Possible causes might be failures of the sensors, the control unit, or the actuators. As a running example, a fault-tolerant architecture of the *Object Detection* component has been selected. It uses both *Lidar* and *Camera* to detect objects. We consider two failure modes, i. e., ways in which the component may fail:

1. One of *Lidar* or *Camera* fails permanently.
2. One or both of *Lidar* and *Camera* fail for a limited amount of time.

The latter is called a *transient failure*. Both failures will be mitigated by preventing an update of the maneuver plan if corrupted or missing data is detected. Hence, the crucial point in the safety concept is to prove correct timing of braking maneuvers in such situations.

An excerpt of the safety case is depicted in Figure 2. Providing evidence for the satisfaction of all stated safety goals is one of the main objectives in constructing a safety argumentation. Based on the identified hazards (G3, G8), safety goals are defined for the functional architecture (S4, see Section 5). In the course of decomposing the architecture and its realization by a HW/SW architecture, the safety goals are decomposed, safety mechanisms are added to cover the respective safety requirements, and timing requirements from the FTTI consideration are assigned to the individual tasks of the corresponding AMALTHEA model (G27). Section 6 details the resulting contract specifications. The final proof obligation is to show that all contract specifications are satisfied by the HW/SW architecture (G29). According to ISO 26262, these steps should be supported by means of verification (Sn14). Section 7 details an automated verification approach based on model checking.

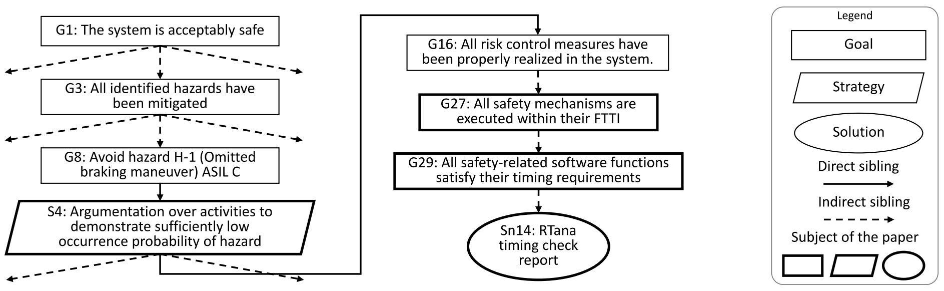
G1: The system is acceptably safe
G3: All identified hazards have been mitigated
G8: Avoid hazard H-1 (Omitted braking maneuver) ASIL C
S4: Argumentation over activities to demonstrate sufficiently low occurrence probability of hazard
G16: All risk control measures have been properly realized in the system.
G27: All safety mechanisms are executed within their FTTI
G29: All safety-related software functions satisfy their timing requirements
Sn14: RTana timing check report
Legend
Goal
Strategy
Solution
Direct sibling
Indirect sibling
Subject of the paper

Fig. 2: Excerpt of the MobSTr Safety Case

4 Fault-Tolerant Time Interval

According to ISO 26262, an FTTI has to be determined for every identified fault that may lead to a hazard. The FTTI refers to the time span between occurrence of the fault that, unless mitigated, causes the hazardous event, and the occurrence of the hazard. Within this period, both detection and mitigation of the fault must be completed. The overall approach for calculating the FTTI follows the results of [Fr20], with some modifications regarding the assumed safety concept[4]. Frese et al. are in line with the view of ISO 26262, which assumes that components can be equipped with safety mechanisms that observe the outputs, and would bring the system from an *unsafe state* after occurrence of a fault, back to a *safe state*. Figure 3 illustrates how the different time intervals relate to each other. Table 1 lists the constants used in the following calculations. Note that these are example values based on a rough guess and may differ from real world data.

Tab. 1: Constants

Description	Variable	Value
Reaction time	t_{react}	2.57 s
Braking time	t_{brake}	2.0 s
Sensor processing time	t_{sense}	50 ms
Actuator time	t_{act}	100 ms
Control unit WCRT	t_{chain}	< 800 ms

First, we determine the FTTI from a “vehicle-level” perspective. This is the time span between the fault and the hazardous event. Note that we exclude faults with zero-length FTTIs, as such faults must be proactively avoided and corresponding safety mechanisms deviate significantly from the non-zero case. The FTTI calculation depends on the considered driving function, the fault, the hazardous event, and the driving situation. In our running example, this process is simplified, as we only consider a single hazard and a simple single-point fault. The fault is a corrupted lidar frame, leading to an omitted braking maneuver, and in turn to a collision with an obstacle. Hence, the FTTI is below the time that is available from the first detection of the obstacle until the vehicle must have been stopped in front of the obstacle. This time frame can be split into the time t_{react} needed for reaction and t_{brake} needed for braking: $\mathsf{FTTI} < t_{\mathsf{react}} + t_{\mathsf{brake}} \approx 4.57\,\mathrm{s}$.

[4] In contrast to Freese et al. we consider a *safety mechanism implemented with emergency operation* [IS18]. This does not affect compliance to the standard.

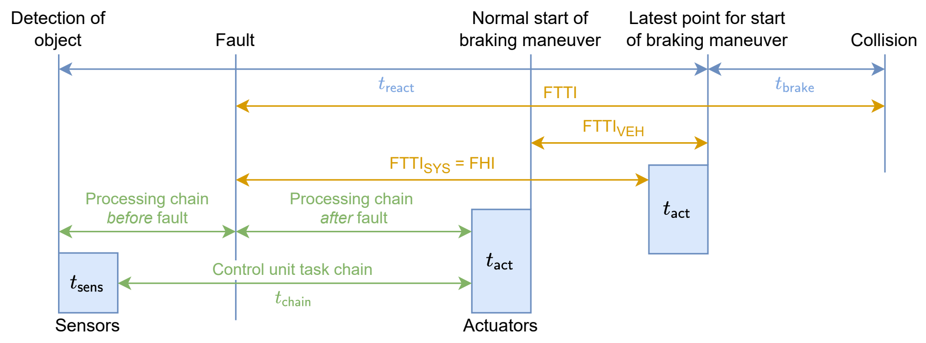

Fig. 3: Safety-Relevant Time Intervals

From an engineering perspective, the time interval $\mathsf{FTTI}_{\mathsf{SYS}}$ is the relevant one, as it specifies the time that the system has for fault handling. In terms of ISO 26262, this is the item-level *fault handling time interval* (FHI). Compared to $\mathsf{FTTI}_{\mathsf{VEH}}$, it starts where the fault can be detected within the system, and ends where output has to be set in order to avoid the hazard. Later on in the safety process, fault detection and mitigation are distinguished, and thus the FHI is split into *fault detection* (FDI) and *fault reaction time interval* (FRI). Because in our example the safe state cannot be reached by shutting down the system, we have to consider the actuator time instead of a deactivation time considered by Frese et al Considering the braking, actuator, and sensing times to be known, and assuming that the fault is first visible at the sensor output, this is $\mathsf{FTTI}_{\mathsf{SYS}} = \mathsf{FHI} \leq t_{\mathsf{react}} - t_{\mathsf{act}} - t_{\mathsf{sense}} \approx 2.43\,\mathrm{s}$.

5 Functional Decomposition and HW/SW-Level Design

After conducting the HARA, safety mechanisms must be defined that describe how the safety goals shall be implemented in order to avoid the identified hazards. According to ISO 26262, this is done by decomposing the safety goals into safety requirements that are allocated to components of the system. As a second step, these functional safety requirements are refined into technical safety requirements, which detail their implementation. In parallel, the subcomponents are mapped to HW/SW units.

Table 2 shows an excerpt of a possible breakdown of the safety goal SG-1 addressing the aforementioned hazard H-1. The requirements in the table consider failure of an optical sensor (e. g., *Lidar* or *Camera*) as a root cause of an omitted braking maneuver. Note that in a complete development process, all potential causes need to be considered, which shall be supported by systematic analyses methods such as FTA or FMEA. Because the paper focuses on verification of timing behavior, the requirements breakdown stops at the level reached in Table 2, except for SR-1-1.3, which we will make visible in the software model.

Safety requirement SR-1-1a states that there need to be two processing chains, one for the *Lidar* and one for the *Camera*. According to the remaining requirements, we need two

Tab. 2: Textual Safety Requirements

ID	Description	Time Bound
SG-1	The system shall prevent omitting required braking maneuvers.	2.43 s
SR-1-1a	The system shall use lidar and camera for object detection.	1006 ms
SR-1-1b	The system shall ensure that objects are detected if lidar or camera fails.	1006 ms
SR-1-1.1	The system shall identify sensor failures.	66 ms
SR-1-1.1.1	The system shall identify when the lidar sensor has failed.	60 ms
SR-1-1.1.2	The system shall identify when the camera has failed.	60 ms
SR-1-1.2	The system shall mitigate sensor failures.	940 ms
SR-1-1.2.1	The system shall detect objects using lidar.	795 ms
SR-1-1.2.2	The system shall detect objects using camera.	55 ms
SR-1-1.2.3	The system shall fuse the objects detected by lidar and camera.	25 ms
SR-1-1.3	The system shall use only information from working sensors.	n/a
SR-1-2	...	...

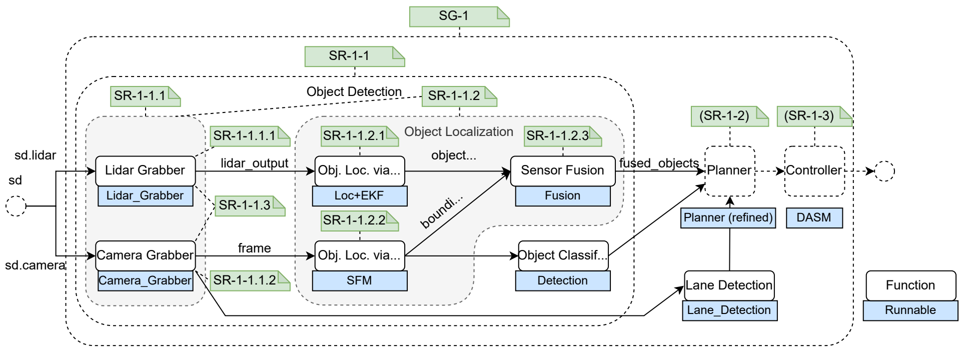

Fig. 4: Mapping of Safety Requirements to Components and Deployment to Runnables

separate components (*Lidar Grabber* and *Camera Grabber*) to monitor and preprocess the raw sensor data, components that use these data for locating objects, and a component that fuses the results (*Sensor Fusion*). Figure 4 shows a functional decomposition of the *Object Detection* component with the requirements from Table 2 mapped to the subcomponents. Note that the *Lidar* and *Camera* outputs have been merged into a single data source named `sd` (sensor data), which carries both the lidar signal (`sd.lidar`) and the camera signal (`sd.camera`). This simplifies the formalization of SR-1-1 later on in Section 6.

In order to ensure that the safety concept is also timewise correct, each safety requirement gets a time bound assigned (see Table 2). The FHI identified in Section 4 is assigned to the safety goal. Recall that the FHI is split into FDI and FRI. The FDI is assigned to SR-1-1.1.

Next, the components are deployed to software units, since all functionality is implemented in software. Figure 4 shows the deployment of components to so-called *runnables* in the AMALTHEA software model of the implementation. In AMALTHEA, a runnable is an abstraction of an executable entity, such as a C function. The runnables named in the

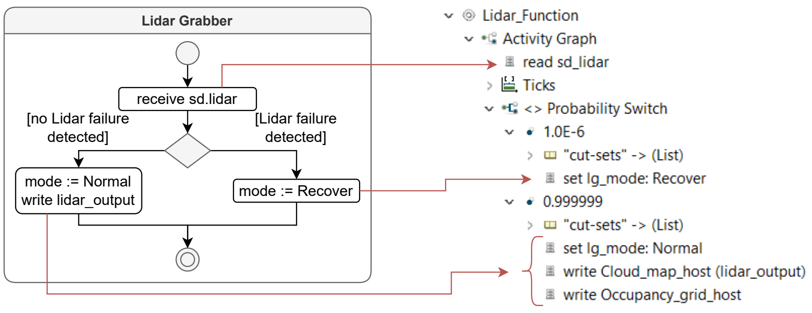

Fig. 5: Implementation of the *Lidar Grabber* Functionality (left) in Eclipse APP4MC (right)

figure are modified versions of the original runnables in the challenge model. Runnables in AMALTHEA contain information about the execution time, data access, and control flows.

Figure 5 shows the implementation of SR-1-1.3 in the *Lidar Grabber* component. The `lidar_input` from Figure 4 is mapped to the two labels `Cloud_map_host` and `Occupancy_grid_host` in the AMALTHEA model, which refer to memory addresses. The different branches in the left-hand part of Figure 5 are modeled as a probability switch. Each branch is annotated with a *cut set* that enumerates the failure mode combinations handled in that branch. The normal operation is annotated with an empty cut set, and the other branches model alternative execution paths for failure mitigation. Because AMALTHEA does not provide modeling elements for cut sets, *custom elements* are used for this purpose. The implementation of the *Camera Grabber* component is modeled analogously. Note that the complete AMALTHEA model of the example system is available online[5].

6 Timing Contracts

In order to perform verification of correct timing, the AMALTHEA model is transformed into a corresponding analysis model together with formalized versions of the specified timing behavior. Assume/guarantee contracts enable formalizing both the *assumptions* of a component about its environment, and the guarantees that a component provides in such an environment. For specifying contracts, we adopt the MTSL specification language [Bö19] extended with modes [Kr22]. The following two patterns are used:

$$E_{\texttt{IN}}\ \texttt{occurs every}\ I\ \texttt{[in mode}\ M\texttt{].} \tag{1}$$

$$\texttt{Reaction(}E_{\texttt{IN}}\texttt{,}\ E_{\texttt{OUT}}\texttt{) within}\ I\ \texttt{[in mode}\ M\texttt{].} \tag{2}$$

$E_{\texttt{IN}}$ and $E_{\texttt{OUT}}$ are events that occur at the component's input and output ports `IN` and `OUT`, and I is a time interval. The parts in square brackets are optional and limit the validity of the specification patterns to an operating mode M of the component in focus. Pattern (1) states

[5] `https://github.com/jansbecker/ase-2023`

Tab. 3: Contracts of the Lower-Level Components

Lidar Grabber	
A_{L1}:	`sd.lidar occurs every 66 ms.`
G_{L1}:	`Reaction(sd.lidar.OK,(set(mode,Normal),lidar_output)) within [5,60] ms.`
G_{L2}:	`Reaction(sd.lidar.NOT_OK,(set(mode, Recover)) within [5,60] ms.`
Camera Grabber	
A_{C1}:	`sd.camera occurs every 66 ms.`
G_{C1}:	`Reaction(sd.camera.OK,(set(mode,Normal),lidar_output)) within [7,60] ms.`
G_{C2}:	`Reaction(sd.camera.NOT_OK,(set(mode,Recover)) within [7,60] ms.`
Object Localization via Lidar	
G_{OL}:	`Reaction(lidar_output,object_poses) within [120,795] ms.`
Object Localization via Camera	
G_{OC}:	`Reaction(frame,bounding_boxes) within [20,55] ms.`
Sensor Fusion	
G_{F1}:	`Reaction(object_poses,fused_objects) within [5,25] ms.`
G_{F2}:	`Reaction(bounding_boxes,fused_objects) within [5,25] ms.`

that $E_{\mathtt{IN}}$ occurs periodically with a distance from the interval I. Pattern (2) expresses that every event $E_{\mathtt{IN}}$ is answered within I by an occurrence of $E_{\mathtt{OUT}}$. Subsequently, the considered failure modes will be mapped to the modes referenced in the specification patterns.

Table 3 shows the contracts of all lower-level components. For the sake of brevity, we omit assumptions about inner signals from the table. In the following, we focus on the *Lidar Grabber* and *Camera Grabber* components. The specification details the behavior already shown in Figure 5. We assume that the *Lidar* and *Camera* sensors have three different modes: `OK` denotes normal behavior, `transient_failure` is a failure mode where the component will finally recover from, and `permanent_failure` is a failure mode without recovery. The set {`transient_failure`, `permanent_failure`} is abbreviated as `NOT_OK`. Furthermore, we assume that the grabber components in Figure 4 can observe the failure modes of the *Lidar* and *Camera* together with receiving input events via the `sd.lidar` and `sd.camera` signals, respectively. We use $E_{\mathtt{IN}}.F$ to denote that an input event $E_{\mathtt{IN}}$ has been generated in mode $F \in \{\mathtt{OK}, \mathtt{NOT\_OK}\}$. As an abbreviation, we write `sd`F if F occurs on one of `sd.lidar` or `sd.camera`. Each grabber maintains a mode $M \in \{\mathtt{Normal}, \mathtt{Recover}\}$ that reflects the observed (failure) mode: `Normal` indicates fault-free behavior, and `Recover` is entered whenever a sensor failure is detected. A transition to mode M is expressed as a special event `set(mode,`M`)`. The combined modes of the grabber components determine the mode of the *Object Detection* component. Because the *Sensor Fusion* shall mitigate one missing input, we define that *Object Detection* is in mode `Recover` if both grabbers are in mode `Recover`, otherwise it is in mode `Normal`.

The contract of the *Object Detection* is shown in Table 4 and formalizes the safety concept at system level. As described above, single-point failures shall be handled transparently by

Tab. 4: Contract of the Top-Level Component *Object Detection*

A:	`sd occurs every 66 ms.`
G1:	`Reaction(sd.OK,fused_objects) within [100,1006] ms in mode Normal.`
G2:	`Reaction(sd.NOT_OK,set(mode,Recover)) within [5,66] ms in mode Normal.`
G3:	`Reaction(sd.OK,fused_objects) within [100,940] ms in mode Recover.`

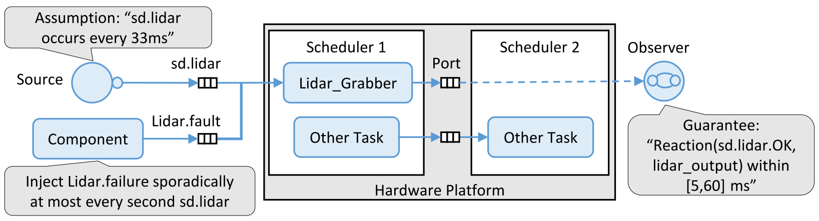
Assumption: "sd.lidar occurs every 33ms"
Source
sd.lidar
Component
Lidar.fault
Inject Lidar.failure sporadically at most every second sd.lidar
Scheduler 1
Lidar_Grabber
Other Task
Port
Scheduler 2
Other Task
Hardware Platform
Observer
Guarantee: "Reaction(sd.lidar.OK, lidar_output) within [5,60] ms"

Fig. 6: Structure and Overview of the RTANA$_{\text{2sim}}$ Model

the subcomponents. Guarantee G1 expresses that in this case as well as in the fault-free case, responses must be generated within the FHI. Recall that the FHI is split into FDI and FRI. Guarantee G2 formalizes fault detection by requiring a mode change within the FDI, and G3 formalizes the fault handling by requiring a response to the next input event within the FRI.

7 Providing Evidence by Verification

For the final step of the partial safety case in Figure 2, we now sketch how formal verification provides a solution (namely, Sn14). RTANA$_{\text{2sim}}$ is an explicit-state model checker dedicated to verifying task networks. It combines timed automata-like task networks with real-world scheduling strategies. As preemptive scheduling strategies are undecidable for most task models in dense time [Fe07], RTANA$_{\text{2sim}}$ approximates the continuous time model used in AMALTHEA by a discrete one. The main idea is to translate the given AMALTHEA software model into an RTANA$_{\text{2sim}}$ analysis model, which consists of a representation of all tasks and runnables as *execution automata*. Event generators (*sources*) simulate the assumptions of the contracts in focus, and a set of *observers* monitor the compliance with the contract guarantees. Figure 6 shows the structure of an RTANA$_{\text{2sim}}$ model.

The event generators drive the inputs of the tasks and model the environment as it is specified by the assumption of the annotated contracts. For the analysis of software safety mechanisms, specific event generators inject virtual faults into the system, which are also derived from the corresponding contract assumptions. We use the term *virtual* here, because faults are simply modeled as additional variables that indicate which faults are currently present. In the AMALTHEA model, runnables incorporate these variables in terms of (fault-annotated) probability switches, such that the control flow considers handling present faults. The observers monitor the output of both event generators and tasks to witness satisfaction of the guarantees. Technically, each observer enters its dedicated bad state whenever the guarantee under observation is violated by the system's behavior. In each analysis run, RTANA$_{2\mathrm{sim}}$ thus tries to verify that the bad states of all observer automata are unreachable. RTANA$_{2\mathrm{sim}}$ is executed on the input model discussed above and provides a verdict for all specified properties. It is able to find errors in an earlier version of the running example published as part of the MobSTr dataset [BKS21] as well as the unfeasibility of the original WATERS 2019 Industrial Challenge model reported already in [Kr19]. On the other hand, the contracts listed in Section 6 pass the satisfaction check described above.

8 Conclusion

The paper contributes to the ISO 26262-compliant development of safety-critical systems by investigating an approach for providing evidence in a safety argumentation. Such a process involves the identification of potential hazards and risks as well as the formulation of a safety concept to mitigate these risks, which is finally implemented in HW/SW. The timing aspect inevitably appears in this process because it must be ensured that risk mitigation occurs *in time*. This calls for complete and sound methods for elicitation, decomposition, and implementation of real-time requirements. This paper shows how consistent timing specifications can be derived, starting from the identified safety goals. The approach extends previous work on contract-based design by explicating failure modes and fault detection. While closing the gap of mode-dependent timing analysis, it is shown how model-checking can support the verification of safety concepts and their implementation.

Bibliography

[Be18] Becker, Jan Steffen; Bertram, Vincent; Bienmüller, Tom; Brockmeyer, Udo; Dörr, Heiko; Peikenkamp, Thomas; Teige, Tino: Interoperable Toolchain for Requirements-Driven Model-Based Development. In: ERTS 2018. 2018.

[Bi13] Birch, J.; Rivett, R.; Habli, I.; Bradshaw, B.; Botham, J.; Higham, D.; Jesty, P.; Monkhouse, H.; Palin, R.: Safety Cases and Their Role in ISO 26262 Functional Safety Assessment. In: Computer Safety, Reliability, and Security. Springer, pp. 154–165, 2013.

[Bi18] Biggs, G.; Juknevicius, T.; Armonas, A.; Post, K.: Integrating Safety and Reliability Analysis into MBSE: overview of the new proposed OMG standard. INCOSE International Symposium, 28:1322–1336, 07 2018.

[BKS21] Becker, J. S.; Koopmann, B.; Stierand, I.: Safety Relevant Time Intervals for MobSTr. Technical report, OFFIS e.V., published as part of the MobSTr dataset, 2021.

[Bö19] Böde, E.; Damm, W.; Ehmen, G.; Fränzle, M.; Grüttner, K.; Ittershagen, P.; Josko, B.; Koopmann, B.; Poppen, F.; Siegel, M.; Stierand, I.: MULTIC-Tooling. FAT Series, 316, 2019.

[DHW19] Denomme, D.; Hooson, S.; Winkelman, J.: A Fault Tolerant Time Interval Process for Functional Safety Development. In: SAE World Congress Experience 2019. SAE, 2019.

[Es08] Estefan, J.: Survey of Model-Based Systems Engineering (MBSE) Methodologies. INCOSE MBSE Focus Group, 25, 01 2008.

[Fe07] Fersman, E.; Krcal, P.; Pettersson, P.; Yi, W.: Task Automata: Schedulability, Decidability and Undecidability. Information and Computation, 205(8):1149–1172, 2007.

[Fr20] Frese, T.; Leonhardt, T.; Hatebur, D.; Côté, I.; Aryus, H.-J.; Heisel, M.: Fault Tolerance Time Interval. In: Neue Dimensionen der Mobilität. Springer Gabler, pp. 559–567, 2020.

[IS18] ISO: Road Vehicles – Functional Safety. Standard ISO 26262:2018, International Organization for Standardization, Geneva, Switzerland, 2018.

[Kr19] Krawczyk, L.; Bazzal, M.; Govindarajan, R. P.; Wolff, C.: Model-based Timing Analysis and Deployment Optimization for Heterogeneous Multi-Core Systems Using Eclipse APP4MC. In: Conference on MODELS-C. IEEE, 2019.

[Kr22] Kröger, J.; Koopmann, B.; Stierand, I.; Tabassam, N.; Fränzle, M.: Handling of Operating Modes in Contract-based Timing Specifications. In: Verification and Evaluation of Computer and Communication Systems. Springer, pp. 59–74, 2022.

[Ma18] Maksimov, M.; Fung, N. L. S.; Kokaly, S.; Chechik, M.: Two Decades of Assurance Case Tools: A Survey. In: Computer Safety, Reliability, and Security. Springer, 2018.

[MN20] Munk, P.; Nordmann, A.: Model-based Safety Assessment with SysML and Component Fault Trees: Application and Lessons Learned. Software and Systems Modeling, 2020.

[Pa11] Palin, R.; Ward, D.; Habli, I.; Rivett, R.: ISO 26262 Safety Cases: Compliance and Assurance. In: Conference on System Safety. IEEE, 2011.

[PKK15] Preschern, C.; Kajtazovic, N.; Kreiner, C.: Building a Safety Architecture Pattern System. In: European Conference on Pattern Languages of Program. EuroPLoP '13, Association for Computing Machinery, New York, NY, USA, 2015.

[Re20] Reich, J.; Frey, J.; Cioroaica, E.; Zeller, M.; Rothfelder, M.: Argument-Driven Safety Engineering of a Generic Infusion Pump with Digital Dependability Identities. In: Model-Based Safety and Assessment. Springer, pp. 19–33, 2020.

[Re21] Rehm, F.; Dasari, D.; Hamann, A.; Pressler, M.; Ziegenbein, D.; Seitter, J.; Sañudo, I.; Capodieci, N.; Burgio, P.; Bertogna, M.: Performance Modeling of Heterogeneous HW Platforms. Microprocessors and Microsystems, 87(C), 2021.

[St21a] Steghöfer, J.-P.; Koopmann, B.; Becker, J. S.; Stierand, I.; Zeller, M.; Bonner, M.; Schmelter, D.; Maro, S.: The MobSTr Dataset – An Exemplar for Traceability and Model-based Safety Assessment. In: Requirements Engineering Conference. IEEE, 2021.

[St21b] Steghöfer, J.-P.; Koopmann, B.; Becker, J. S.; Törnlund, M.; Ibrahim, Y.; Mohamad, M.: Design Decisions in the Construction of Traceability Information Models for Safe Automotive Systems. In: Requirements Engineering Conference. IEEE, 2021.

[Tr16] Trei, M.; Maro, S.; Steghöfer, J.-P.; Peikenkamp, T.: An ISO 26262 Compliant Design Flow and Tool for Automotive Multicore Systems. In: Product-Focused Software Process Improvement. Springer, pp. 163–180, 2016.

[Va22] de la Vara, J. L.; García, A. S.; Valero, J.; Ayora, C.: Model-Based Assurance Evidence Management for Safety-Critical Systems. Software and Systems Modeling, 2022.

[We19] Wei, R.; Kelly, T. P.; Dai, X.; Zhao, S.; Hawkins, R.: Model Based System Assurance Using the Structured Assurance Case Metamodel. Journal of Systems and Software, 154:211–233, 2019.